Improving the Java Code Generator Component of the SLCO Framework

van Nijnatten, Melroy

Award date:
2022

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Department of Mathematics and Computer Science
Software Engineering and Technology Research Group

Improving the Java Code Generator Component of the SLCO Framework

Master Thesis

Melroy van Nijnatten

Committee Members:

Dr. Ing. Anton J. Wijs
Dr. Savio Sciancalepore
Dr. Ir. Loek G.W.A. Cleophas

Version 1.1.0

Eindhoven, November 2022

Abstract

The development of complex, multi-component software is a time-consuming and error-prone task that is addressed by the Simple Language of Communicating Objects (Slco) framework. The objective of this thesis is to increase the performance, robustness and maintainability of the Java code generator component of the Slco framework and improve the quality of the code it generates, whilst also ensuring that the model to code conversion is correct through the application of formal verification techniques. Additionally, the effectiveness of said improvements is investigated through a thorough performance analysis. With that in mind, the purpose of this work can be summarized by the following question: given the metrics given above, which improvements can be conceived and implemented, and moreover, how do they fare when put into practice?

Contents

List of Figures

List of Tables

List of Algorithms

List of Code Listings

Chapter 1

Introduction

The development of complex, multi-component software is a time-consuming and error-prone task that is addressed by the Simple Language of Communicating Objects (Slco) framework. The Slco framework adheres to a model-driven software development methodology and uses model transformations to verify and generate code. The intended functionality of the system is specified through a Domain-Specific Language (DSL), which can be used to model systems consisting of concurrent and communicating components through state machines.

The objective of this thesis is to increase the performance, robustness and maintainability of the Java code generator component of the Slco framework, and more importantly, of the code it generates. Throughout this research, structural improvements will be introduced to the following aspects of the generated code. First, deterministic behavior will be added to the generated code through the application of SMT solving and other complementary techniques. Within the current implementation of the Java code conversion component in the Slco framework, the transitions are chosen completely non-deterministically, namely by picking an arbitrary transition from the list of available transitions. Consequently, it is possible that the chosen transition, if guarded, is not active, and therefore, it cannot be fired in the current state. Obviously, this behavior is inefficient and unreliable, and hence, it is deemed to be a good starting point for this research.

Additionally, it has been observed that the locking mechanism, which is a crucial component of the generated code that ensures the atomicity of statements and operations, cannot handle specific situations correctly or reliably. On top of that, the inclusion of deterministic structures will add an additional level of complexity to the locking process which will need to be addressed. Thus, the secondary objective of this project will be to improve the locking mechanism, whilst also ensuring that the latter synergizes appropriately with the concept of deterministic decisions.

Moreover, the formal verification of all model transformations is considered to be a cornerstone of the Slco framework, and therefore, the formal verification of the generated code is a top priority. The functional correctness of the generated code will be verified through the VerCors verification tool set. Nevertheless, it needs to be noted that the formal verification of the generated code is a complex endeavor. Given the limited time that is provided to complete the thesis, and the broad selection of topics that will be covered therein, it regrettably is infeasible to create a verification process that covers every single aspect of the generated code. Nevertheless, the aim of this work will be to consider as many angles as possible within the given time frame.

Finally, it is important to measure the effectiveness of the improvements that have been made to the code generator component. The latter will be achieved by applying the code generator to a collection of synthetic and practical test models. In turn, the generated code will be subjected to

a performance analysis, such that the effectiveness of the available code generator configurations can be measured and investigated. Particularly, the concurrency characteristics of the programs will be a primary target of interest during the analysis, given the framework's focus on the creation of parallel programs. Lastly, the performance analysis shall be used to discover additional shortcomings within the generated code that can be acted upon in the future.

1.1 Thesis Structure

The thesis will be structured as follows. To start with, the Slco framework and the accompanying language will be introduced in Chapter 2. Next, several structural improvements will be made to the parser of the Slco framework in Chapter 3. Afterwards, the concept of deterministic structures will be introduced in Chapter 4. On top of that, Chapter 5 will detail the enhancements made to the locking mechanism in an effort to improve its robustness. Subsequently, the original implementation of the code generator will be reviewed, improved and restructured in Chapter 6. The process of formally verifying the generated code will be elaborated upon in Chapter 7. Additionally, the effectiveness of the revised decision structure and locking mechanism will be investigated in Chapter 8 through the execution of several synthetic and practical test models. Finally, the overall conclusion of the thesis will be presented in Chapter 9.

Chapter 2

SLCO: The Simple Language of Communicating Objects

The Simple Language of Communicating Objects (SLCO) framework has been created to simplify the development of complex, parallel and multi-component software through a model-driven approach [13]. The primary building blocks of the language are classes, and the concurrent state machines and statements contained therein. The SLCO framework uses a similarly named Domain-Specific Language (DSL) to specify the software behavior in the shape of a model, which can then be converted to other artifacts through model-to-model transformations.

The SLCO framework provides several useful features through these transformations, such as the formal verification of desired functional properties for a given model, achieved through the conversion of the model to an MCRL2 equivalent. The MCRL2 toolset [9] can then be used to apply model checking [2] to the transformed model, after which the desired functional properties can be specified as μ-calculus formulas to perform formal verification. In addition, transformations exist within the SLCO framework that allow for a given SLCO model to be converted to a DOT file, which allows for a better understanding to be gained on the overall structure of the state machines expressed within the model. Another useful feature is the conversion of one SLCO model to another: one could, for instance, simplify a given SLCO model through the SLCO framework. The final major feature is the code generator component of the SLCO framework, which automatically transforms the given SLCO model to (optimized) multi-threaded Java code, effectively hiding all parallel programming challenges from the developer.

Furthermore, the scope formal verification within SLCO is not limited to validating functional qualities–the SLCO framework strives to perform formal verification during each and every step of the development process, without requiring expert knowledge or direct involvement from the developer, which the framework aims to achieve by verifying the correctness of the model transformations directly, instead of the result thereof. The benefit of this approach is that the correctness of the transformations can be determined once-and-for-all: hence, the correctness of the attained result can always be guaranteed for the transformation in question, because the process of attaining said result through the transformation is, by itself, verified to be mechanically correct. Similar code generation and formal verification features are provided by other frameworks such as SCADE 6 [16], SIMULINK [15] and EVENT-B [1]. Both SIMULINK and EVENT-B do support formal verification of generated code [25, 17], but it is unclear whether their code generators are verified at a mechanical level. SCADE 6 on the other hand does provide a mechanically verified code generator through the use of Lustre's verified compiler [6], but is disadvantaged by its limited expressiveness, due to its primarily focus being on the sampling-actuating model of control engineering. Thus, the SLCO framework will strive to provide the desired consistency validation for all transformations, while also providing a more expressive language to define the desired functionality with.

The SLCO framework has been implemented in Python[1,2]. In addition, the SLCO language has been defined through the TEXTX [14] meta-language, with the model-to-model transformation from SLCO to Java being performed through the JINJA2[3] template language. The chapter will be structured as follows. First, an introduction to TEXTX's grammar will be given in Section 2.1. Next, the SLCO 2.0 language will be introduced in Section 2.2. Finally, Section 2.3 concludes the chapter and contains several SLCO models through which the language is demonstrated.

2.1 The TextX Grammar

The syntax of SLCO's DSL, called the SLCO 2.0 language, is defined through TEXTX, which is a meta-language tool for specifying DSLs in Python. The syntax definition is clear and unambiguous, and hence, it will be referred to extensively during the introduction of the SLCO 2.0 language to provide a concise overview of the supported structures. However, note that a full understanding of the TEXTX language will not be required–a natural language description will be given of each construct, independently from the syntax definition itself. Nevertheless, a basic understanding of the TEXTX grammar is considered helpful, since it allows for the concrete syntax and the textual description thereof to be cross-referenced if further clarification if required. As such, the remainder of this section will be dedicated to giving an introduction to the TEXTX grammar.

```
<name>: <body>;
```

Listing 2.1: The general structure of a TEXTX rule definition.

In TEXTX[4], a language is defined through rules. The general structure of a rule is provided in Listing 2.1. Each rule starts with a name, which is followed by a colon that leads into the rule's body, with the rule declaration itself being concluded with a semicolon. The rule's body is defined to be an expression that contains the desired parsing behavior of the rule. The following sections will discuss the majority of basic expressions that are provided by the TEXTX grammar.

2.1.1 Base Types

First, several base type rules are provided by the TEXTX language, including but not limited to the `ID`, `INT` and `BOOL` rules. An `ID` matches a common identifier that consists of letters, digits and underscores and is hence generally used as the type of a name or identity, an `INT` rule matches to integer values, and a `BOOL` rule matches to the Boolean values `true` and `false`.

2.1.2 Matching

```
StringMatch: 'A';
RegexMatch: \[a-z]+\;
```

Listing 2.2: A demonstration of matching expressions in the TEXTX grammar.

The base type rules are followed by expressions that match to a given string or regular expression. The matching expressions, demonstrated in Listing 2.2, are the basic building blocks of the language, and allow for more complex expressions to be created. String matches are defined through quoted strings and will match a literal string in the input, and are often used to match for keywords in the target language. Regular expressions, to be enclosed by backlashes, are defined using the syntax of regular expressions as used in Python.

[1]The original Git repository can be found at https://gitlab.tue.nl/SLCO/SLCO.git
[2]The revised implementation can be found at https://github.com/melroy999/2IMC00-SLCO.git
[3]The documentation of JINJA2 is given at http://jinja.pocoo.org
[4]The official documentation of the grammar can be found at https://textx.github.io/textX/3.0/grammar/

2.1.3 Sequence

```
Sequence: E1 E2 E3;
```

Listing 2.3: A demonstration of a sequence in the TextX grammar.

The sequence operator, as demonstrated in Listing 2.3, is an n-ary operator that is used to chain sub-expressions together. Expressions contained within a sequence will be matched in the given order. By default, whitespace characters are skipped when parsing a sequence, and hence, sub-expressions may be separated by an arbitrary number of white spaces.

2.1.4 Ordered Choice

```
OrderedChoice: E1 | E2 | E3;
```

Listing 2.4: A demonstration of an ordered choice in the TextX grammar.

The n-ary vertical bar operator (`|`), as demonstrated in Listing 2.4, denotes an ordered choice between expressions. During parsing, the matching of the listed expressions will be performed from left to right, which guarantees that the first successful match will be chosen. The benefit of ordered choices is that the parsing always yields the same parse tree, and hence, the output will always be unambiguous. Furthermore, it allows for certain expressions to be prioritized.

2.1.5 Optional

```
Optional: E1 E2?;
```

Listing 2.5: A demonstration of optional expressions in the TextX grammar.

As demonstrated in Listing 2.5, an expression can be made optional during matching by appending it with the unary question mark operator (`?`). The rule `Optional` requires a match with `E1`, which is sequenced with an optional expression `E2`. In other words, the rule will match to both expression `E1` and the sequence of expressions `E1 E2`. The optional operator relatively has a high order of precedence–as shown through the example, the operator solely targets expression `E2`, and not the sequence `E1 E2` in its entirety. More complex optional behavior can be expressed by surrounding the target expression by a set of parentheses prior to adding the optional operator.

2.1.6 Repetitions

```
ZeroOrMore: E1*;
OneOrMore: E1+;
```

Listing 2.6: A demonstration of repetitions in the TextX grammar.

Repetition operators are provided that allow for an expression to be repeated an undefined number of times. Zero or more repetitions of an expression are specified by appending the expression with the asterisk (`*`) operator. Similarly, one or more repetitions are specified by appending the plus (`+`) operator. Both types of repetition operators are demonstrated in Listing 2.6. Note that, similarly to the optional operator, repetitions have a high order of precedence–more complex behavior can be expressed through the combination of parentheses and repetition operators.

```
UnorderedGroupS: ((E1 E2) E3)#;
UnorderedGroupC: (E1 E2 | E3)#;
```

Listing 2.7: A demonstration of unordered groupings in the TextX grammar.

An additional repetition type operator provided by TextX is the unordered grouping operator as presented in Listing 2.7. As shown in the listing, an unordered group is expressed by appending a sequence or ordered choice with the number sign (`#`). An unordered grouping operator matches any input that repeats a permutation of the sub-expressions contained within the given sequence or ordered choice. Note that nested sequence and ordered choice sub-expressions are not considered separate elements in the group–due to this, rules `UnorderedGroupS` and `UnorderedGroupC` are considered to be equivalent. Hence, the sequence `E1 E2 E3 E3 E1 E2` is a match to both rules in the given example, but `E3`, `E1 E3 E2` and `E3 E3 E1 E2` are not.

2.1.7 Assignments

```
PlainAssignment: a=E1;
```

Listing 2.8: A demonstration of a plain assignment in the TextX grammar.

All of the expressions and operators discussed up to this point have in common that they allow for a rule to be created that can match a certain input, but they do, thus far, not provide the means to extract information and data contained therein. The plain assignment operator (`=`), as demonstrated in Listing 2.8, allows for a target match or rule value to be saved as an attribute within the generated Python object graph. The left-hand side of the assignment contains the attribute name, with the right hand side holding a reference to another rule or match expression.

```
BooleanAssignment: a?=E1;
ZeroOrMoreAssignment: a*=E1;
OneOrMoreAssignment: a+=E1;
```

Listing 2.9: A demonstration of additional assignment types in the TextX grammar.

Note that the plain assignment is just one of four assignment types provided by the TextX grammar. The remaining types of assignments are presented in Listing 2.9. The first rule uses the Boolean assignment operator (`?=`), which sets the target attribute to `true` if the target match can be made, `false` otherwise. The second rule uses the zero or more assignment operator (`*=`), and assigns a list that contains all matches of the rule to the target attribute. If no matches can be made, the attribute will be assigned an empty list. Finally, the third rule uses the one or more assignment operator (`+=`), which behaves virtually the same as the zero or more assignment operator, but will fail if no matches can be made instead of returning an empty list.

2.1.8 Repetition Modifiers

```
RepetitionModifier: a*=E1[','];
```

Listing 2.10: A demonstration of a repetition modifier in the TextX grammar.

The repetition modifier expression demonstrated in Listing 2.10 allows for modifications to be made to expressions with repetition (`*`, `+`, `#`, `*=`, `+=`). The modifications are defined at the end of an expression between square brackets, with each entry separated by a comma. TextX defines two modifier types: separator modifiers and end-of-line termination modifiers (`eolterm`). The former is used to set a simple string or regular expression as the required separator between elements in the list, while the latter will terminate the repetition on an end-of-line character.

2.1.9 References

```
MatchRuleReference: a=E1;
LinkRuleReference: a=[E1];
```

Listing 2.11: A demonstration of references in the TextX grammar.

A reference to a rule can be included within another rule to create a nested structure of rules or refer by name to instances of rules that have been instantiated elsewhere. The two types of references are demonstrated in Listing 2.11, being the match rule and link rule references respectively, with the latter being defined by surrounding the rule name with square brackets. A match rule reference simply executes the target rule, while the link rule reference matches to an identifier for an existing instance of the given rule, with the reference itself being resolved automatically by TextX. By default, the identifier is equivalent to the `name` attribute of the target rule.

2.1.10 Syntactic Predicates

```
NegativeLookahead: !Keyword ID;
PositiveLookahead: Keyword &ID;
```

Listing 2.12: A demonstration of syntactic predicates in the TextX grammar.

The final basic expressions that remain to be discussed are syntactic predicates, which focus on implementing lookahead functionality for the parser. Two types of syntactic predicates are provided by the TextX grammar, namely the negative lookahead (`!`) and positive lookahead (`&`). Both types of lookaheads are demonstrated in Listing 2.12. Observe that the syntactic predicate operators are placed before their target expression. The rule `NegativeLookahead`, using a negative lookahead, matches to any `ID` that is not a match to `Keyword`, and can hence be used to ensure that the sets of available identities and keywords are disjoint. The rule `PositiveLookahead` uses a negative lookahead, and will only match an `ID` if it is preceded by a `Keyword`.

2.2 The SLCO 2.0 Language

The Slco 2.0 language has been designed to model systems that consist of a collection of concurrent and communicating objects at a convenient level of abstraction [13]. An object-oriented approach is taken for the overall design of the language. The constructs used within the language can be summarized as follows. The definition of an Slco model consists of a finite number of classes, together with a collection of instantiations of these classes as objects, communication channels, and user-defined actions. Each class contains a finite number of state machines that run concurrently, together with a collection of communication ports and member variables associated with the class. State machines consist of a finite collection of states, the guarded transitions between those states, and a set of local variables. The transitions express the desired behavior of the program through a list of atomic statements that need to be executed upon activation.

The language allows for two forms of communication, namely message-passing, and interaction through variables. The objects in the Slco language are designed to be loosely coupled, i.e., objects are primarily designed to function independently from one another, and hence do not have access to members or components of other objects. Hence, the only allowed form of communication between objects is message-passing, which is performed through channels. Similarly, state machines are not allowed to access each other's members either. However, contrarily to classes, the state machines and their parent objects maintain a tight coupling, which facilitates a different type of communication: state machines contained within the same object are allowed to interact with each other through the parent object's member variables.

The remainder of this section will focus on giving an in-depth description for all of the components provided by the Slco 2.0 language. Each description given will start with the syntax definition of the target component as defined through the TextX grammar, followed by a textual description of the syntax and, if appropriate, the functionality of the construct itself. Alternatively, one may refer to Section 2.3 for a collection of concrete examples. On top of that, it is important to note that several changes have been made to the syntax due to the presence of inconsistencies: see Section 3.2 for further details.

2.2.1 Model

```
SLCOModel:
  'model' name=NID '{'
    ('actions' actions+=Action)?
    ('classes' classes*=Class)?
    ('objects' objects*=Object[','])?
    ('channels' channels*=Channel)?
  '}'
;

NID: !Keyword ID;
```

Listing 2.13: The SLCO model declaration syntax defined through the TEXTX grammar language.

The syntax definition of the root component of the SLCO model is given in Listing 2.13. A model declaration starts with the `model` keyword that is succeeded by the model's name, followed by curly brackets that contain a finite number of actions, classes, objects and channels in the given order, preceded by their respective keyword. The aforementioned components within the brackets will be elaborated further in the sections 2.2.1.1, 2.2.2, 2.2.1.2 and 2.2.1.3 respectively.

All components are defined to be optional, with the exception of the model name: the latter is defined to be a `NID`, which is a value of built-in type `ID` that is not equivalent to any of the keywords, with the exclusion of keywords being achieved through a negative lookahead. Hence, the name can be any common identifier consisting of letters, digits and underscores, under the condition that the chosen identifier does not match one of the keywords. Finally, it is important to note that the syntax dictates that at least one action should be declared when the actions keyword is included in the model definition, and that, contrarily to the other components, the list of instantiated objects needs to be provided as a comma separated list.

2.2.1.1 Actions

```
Action: name=NID;
```

Listing 2.14: The action declaration syntax defined through the TEXTX grammar language.

User-defined actions are used to define behavior that has not yet been fully specified, with the intention being that the code associated to the action can be implemented later, either by manually adding the code or by defining the desired behavior using the already existing language constructs. The syntax of an user-defined action is given in Listing 2.14, and consists solely of a name acting as the action's label. Observe that the action name adheres to the same constraints set to the model's name–this limitation will hold for any component using the `NID` terminal, and hence, further repetition of a name's definition is deemed unnecessary.

2.2.1.2 Objects

```
Object:
  name=NID ':' type=[Class] '(' assignments*=Initialisation[','] ')'
;

Initialisation:
  left=[Variable] ':=' (right=INT | right=BOOL | ('[' (rights+=INT[','] | rights+=BOOL[',']) ']'))
;
```

Listing 2.15: The object declaration syntax defined through the TEXTX grammar language.

In a practical application, it can be useful to have multiple (configurable) instantiations of the same class within a model. In SLCO, the aforementioned behavior is implemented through the concept of instantiated objects. The syntax of an object is given in Listing 2.15 and consists of

two parts: the definition of an object, and the definition of variable instantiations for variables included within the target class of the object.

First, the definition of an object will be discussed. Each object definition consists of an unique name for the object itself and the name of the class that is being instantiated. Additionally, the initial values of the variables used within the class can be defined between the brackets following the class name in a comma separated list. The inclusion of initial values for variables is optional– variables that retain their default value can be excluded from the list. On top of that, initial values given during the variable declaration itself are overwritten by initial values given in the object initialization.

The syntax for initializing variables within the object definition is defined as follows. An initialization is depicted as an assignment, with the left hand-side referring to the target variable by name, and the right-hand side depicting the desired target value for the variable in question. The assigned value needs to be a singular `Integer`, `Boolean`, or a comma separated list of the aforementioned types of the appropriate length enclosed by square brackets.

2.2.1.3 Channels

```
Channel:
  name=NID '(' type*=Type[','] ')' (
    (
      synctype='async' ('[' size=INT ']')? (losstype='lossless' | losstype='lossy')
      'from' source=[Object] '.' port0=[Port] 'to' target=[Object] '.' port1=[Port]
    ) | (
      synctype='sync'
      'between' source=[Object] '.' port0=[Port] 'and' target=[Object] '.' port1=[Port]
    )
  )
;
```

Listing 2.16: The channel declaration syntax defined through the TextX grammar language.

A central aspect of the Slco framework is the communication between objects and their components. Two types of communication are defined within the Slco framework: message-passing, which is used for communication between different objects, and interaction through shared variables, which facilitates communication between components of the same object. In this section, the message-passing variety of communication will be introduced through the concept channels.

The syntax of a channel is given in Listing 2.16. Each channel is given an unique name, an optional comma separated list of parameters of a given data type (to be described further in Section 2.2.4), a channel type that denotes whether the communication is synchronous or asynchronous, and the names of the two ports and their respective target objects that are connected through the channel in question. The port and target object pairs are to be paired through a punctuation mark. Additionally, it is required that each port can be attached to at most one channel.

The syntax, keywords, and additional configuration options differ per channel type. For an asynchronous channel, an optional buffer size can be specified by specifying the value between square brackets after the message type declaration. Moreover, a keyword needs to be included that specifies whether the asynchronous channel is lossless or lossy (with the latter specifying that messages may be lost at any time). The relation between two ports is specified with the `from` and `to` keywords. Synchronous channels do not provide any additional options, and instead specify a relation between two ports with the `between` and `and` keywords.

2.2.2 Classes

```
Class:
  name=NID '{'
    ('variables' variables*=Variable)?
    ('ports' ports*=Port)?
    ('state machines' statemachines*=StateMachine)?
  '}'
;

Port: name=NID;
```

Listing 2.17: The class declaration syntax defined through the TextX grammar language.

Next, the definition of a class within the Slco framework will be discussed. A class is at its core a grouping of concurrently running state machines that have access to the same set of member variables. Classes can communicate with other classes through message-passing with the channels described in the previous section, but do not have access to each other's member variables–only the contained state machines have access to the variables of their parent class. Henceforth, variables at the class level will be referenced to as class variables. Through the introduction of classes, there is a clear divide between the two modes of communication, which serves to reduce the ambiguity of the language's structure.

The syntax of a class is given in Listing 2.17. Note that there is no keyword for defining a class. The class's name is immediately followed by curly brackets, which contains a finite number of variables, ports and concurrent state machines in the given order, with each member being optional. Note that all members needs to be preceded by their associated keyword. Variable and state machine declarations will be described in Sections 2.2.4 and 2.2.3 respectively. As described in Section 2.2.1.3, the communication between classes requires ports to function. The ports themselves are declared within the class body, with the port being represented by their name. The names of classes, and the variable and port names used within the scope of a target class, are all required to be unique.

2.2.3 State Machines

```
StateMachine:
  name=NID '{'
    ('variables' variables*=Variable)?
    'initial' initialstate=State
    ('states' states*=State)?
    ('transitions' transitions*=Transition)?
  '}'
;

State: name=NID;
```

Listing 2.18: The state machine declaration syntax defined through the TextX grammar language.

State machines within the Slco framework are non-deterministic finite state machines using guarded and prioritized transitions between states, i.e., each state can have zero, one or more transitions to an arbitrary target state. A transition is considered active if its source state is equal to the current state of the state machine. In the original implementation, an active transition of the lowest priority is selected for activation. The selection process uses a non-deterministic approach if multiple candidates are present. The addition of a deterministic approach for selecting transitions will be discussed in Chapter 4. State machines communicate with each other through the class variables provided by the parent object–accessing class variables of other classes or objects is not allowed and needs to be achieved through message-passing instead. The latter is achieved by adding the appropriate signal constructs to the transitions list of statements. Signals will be described further in Section 2.2.6.4.

The syntax of a state machine is given in Listing 2.18. Similarly to a class definition, the declaration of a state machine is not preceded by a keyword. A state machine declaration starts with an unique name within the scope of the parent class. The name is followed by curly brackets containing the state machine's local variables, an initial state, the additional states, and transitions between states in the described order. All members are optional except for the initial state. Note that the initial state does not need to be added to the list of additional states. Variable and transition declarations will be described in Sections 2.2.4 and 2.2.5 respectively. The representation of a state is the name associated with the state in question. The variable and state names are all required to be unique within the scope of the state machine.

2.2.4 Variables

```
Variable:
  (type=Type?) name=NID
  (':=' (
    defvalue=INT | defvalue=BOOL | ('['(defvalues+=INT[','] | defvalues+=BOOL[','])']')
  ))?
;

Type:
  (base='Integer' | base='Boolean' | base='Byte') ('[' size=INT ']')?
;
```

Listing 2.19: The variable declaration syntax defined through the TextX grammar language.

The syntax of a variable is given in Listing 2.19. A variable is defined to be of a certain type followed by the variable's name, with the supported types being `Integer`, `Boolean`, unsigned `Byte` and arrays of the aforementioned types. The variable can be turned into an array by stating the length of the array between brackets after declaring the base type. The initial value of a variable can be declared through an assignment and needs to correspond with the given type and array length if applicable. Initial values of an array variable need to be comma separated and enclosed by square brackets. Assigning an initial value to a variable is optional–appropriate default values are assigned automatically to variables without a given initial value. Moreover, the initial values passed to an object instantiation take precedence over the initial values assigned during the variable declarations.

2.2.5 Transitions

```
Transition:
  (priority=INT ':')?
  ((source=[State] '->' target=[State]) | ('from' source=[State] 'to' target=[State]))
  ('{' statements*=Statement[';'] (';')? '}')?
;
```

Listing 2.20: The transition declaration syntax defined through the TextX grammar language.

The transitions in the Slco framework and the statements contained therein are used to describe the desired behavior of the target program. The syntax of a transition is given in Listing 2.20. The transition definition starts with a priority followed by a colon, where the priority is defined to be an `Integer` value. A lower number indicates a higher priority. Defining a priority is optional–if no priority is given, the transition is automatically given a priority of zero. Next, the names of the source and target states of the transition are defined. Two notations are supported to define the relation between the source and target states: one option is to connect the two states with an arrow (`->`), and the other uses the `from` and `to` keywords.

The optional body of the transition is enclosed by curly brackets, and contains one or more statements separated by, and optionally trailed by, a semicolon. The available types of statements will be introduced in Section 2.2.6. When a transition is fired, the statements contained within the

transition's body are executed sequentially in the described order. The execution of a transition aborts if a blocked statement, like a conditional statement evaluating to `false`, is encountered during its execution–any alterations or actions performed by the transition prior to reaching the blocking statement will remain, but the state machine will not transition to the target state. Oppositely, the state machine will transition to the target state if and only if all statements have been executed successfully. Additionally, depending on certain conditions, signals may be blocked statements too–see Section 2.2.6.4 for further details on these conditions.

Finally, the parallel execution of transitions is formalised using interleaving semantics, in which the SLCO framework's statements are atomic, i.e., a transition with a sequence of statements is equivalent to a sequence of transitions that each execute one of the statements in the same order. No finer-grained interleaving is allowed [13].

2.2.6 Statements

```
Statement:
  (Composite | ReceiveSignal | SendSignal | Delay | Assignment | Expression | DoAction)
;

DoAction: 'do' act=[Action];

Delay: 'after' length=INT 'ms';
```

Listing 2.21: The statement declaration syntax, including the syntax of action and delay calls due to their overall simplicity, defined through the TEXTX grammar language.

The SLCO framework offers several types of statements, namely (Boolean) expressions, assignments, composites, send and receive signals, action calls, and delays. A brief summary will be given for each of these components, after which they will be elaborated further within their own respective sections (except for action calls and delays, which will be discussed within this section due to their simplicity). To start with, (Boolean) expressions are statements that can be evaluated, assignments are used to alter variable values, and composites group an optional Boolean expression and one or more assignments into an atomic operation. The communication between objects is accomplished through send and receive signal calls. The aforementioned constructs will be discussed in Sections 2.2.6.1, 2.2.6.2, 2.2.6.3 and 2.2.6.4 respectively.

Next, the syntax of action calls and delays are given in Listing 2.21. An action call executes the action that has been defined previously within the model's root component–the `do` keyword needs to be provided together with the name of the action in question. A delay statement causes the state machine that initiated the transition to pause for the specified length of time. A delay is declared by the `after` keyword, followed by an integer number representing the pause length in milliseconds, with the delay statement itself being concluded with the `ms` keyword.

2.2.6.1 (Boolean) Expressions

```
Expression:
  left=ExprPrec4
  ((op='or' | op='xor' | op='and' | op='&&' | op='||') right=Expression)?
;

ExprPrec4:
  left=ExprPrec3
  ((op='!=' | op='=' | op='<>' | op='<=' | op='>=' | op='<' | op='>') right=ExprPrec4)?
;

ExprPrec3:
  left=ExprPrec2
  (( op='+' | op='-') right=ExprPrec3)?
;
```

```
ExprPrec2:
  left=ExprPrec1
  ((op='*' | op='/' | op='%') right=ExprPrec2)?
;

ExprPrec1:
  left=Primary
  (op='**' right=ExprPrec1)?
;

Primary:
  (sign='+' | sign='-' | sign='not')?
  (value=INT | value=BOOL | '(' body=Expression ')' | ref=ExpressionRef)
;

ExpressionRef:
  ref=NID ('[' index=Expression ']')?
;
```

Listing 2.22: The expression declaration syntax defined through the TextX grammar language.

Expressions can be used to group one or more constant values, variables and operators into numerical or Boolean statement of which the value can be evaluated. The supported constant values are of the types integer and Boolean. The scope of the variables does not matter–both class and local variables, including ones of the array variety, are valid targets to be referenced within an expression. The available unary operators are the logical negation (`not`), numerical negation (`-`) and parentheses (`()`) operators. Additionally, the following binary operators are included: addition (`+`), subtraction (`-`), multiplication (`*`), integer division (`/`), integer modulo (`%`), exponentiation (`**`), equality (`==`, `!=`, `<>`), relation (`<=`, `<`, `>`, `>=`), logical conjunction (`and`, `&&`), logical disjunction (`or`, `||`) and logical exclusive disjunction (`xor`). Expressions are used as a sub-component within assignments, composites and signals–these components will be discussed further in Sections 2.2.6.2, 2.2.6.3 and 2.2.6.4 respectively.

The syntax of an expression is given in Listing 2.22. As given, the syntax appears daunting, but the behavior expressed is simple: an expression is broken into multiple components to ensure that the order of precedence for the operators is respected. The order of precedence for the operators available within the Slco framework is given in Table 2.1. The components `Expression`, `ExprPrec4`, `ExprPrec3`, `ExprPrec2` and `ExprPrec1` all follow the same structure–the statement consists of a left component, an operator, and a right component, with the latter two components being an option combination. Additionally, the type of the right component is always equivalent to the type of the rule itself, which is done to ensure that the order of precedence will not be violated. The left component always refers to the component that contains the operators that are the next highest order of precedence. Note that the order of precedence enforced by the syntax is different to that of operators within Java: for the former to conform with the latter, logical conjunctions should take precedence over disjunctions, and relational operators (`<`, `<=`, `>=`, `>`) should take precedence over equality operators (`=`, `!=`, `<>`).

Base values within the expression's hierarchy are represented by primary objects. A primary object starts with an optional sign (`+`, `-`, `not`), with the target object itself being either an `Integer` or `Boolean` value, another expression surrounded by brackets, or an expression reference to a variable value. The latter is represented by the name of the target variable followed by an optional target expression that serves as the target index of an element within an array variable. If included, the index expression is required to be enclosed by square brackets.

Finally, the crucial observation needs to be made that the Slco 2.0 language adheres to a right-associative approach to parsing (operators are assessed from right to left), which is in conflict with the left-associative approach taken by Java (operators are assessed from left to right). Having a difference in associativity is counter-intuitive, confusing and makes the code generator difficult to

maintain. Hence, a change to the parsing definition will be suggested in Chapter 3 that ensures that the SLCO 2.0 language adheres to the left-associative approach used by Java. To avoid further confusion, the order of precedence of operators will be adjusted too to maintain consistency.

Operator	Description
`[]`	Array subscriptions
`()`	Parentheses
`not`, `+`, `-`	Negation, Unary Plus, Unary Minus
`**`	Exponentiation
`*`, `/`, `%`	Multiplication, Division, Modulo
`+`, `-`	Addition, Subtraction
`=` , `!=`, `<>`, `<`, `<=`, `>=`, `>`	Equality, Relational
`or`, `||`, `xor`, `and`, `&&`	(Exclusive) Disjunction, Conjunction

Table 2.1: The order of precedence of operators used within the SLCO framework. An operator precedes another if it is placed higher up in the table. Operators located within the same table row have the same precedence level and will hence have the same significance.

2.2.6.2 Assignments

```
Assignment:
  left=VariableRef ':=' right=Expression
;

VariableRef:
  var=[Variable] ('[' index=Expression ']')?
;
```

Listing 2.23: The assignment declaration syntax defined through the TEXTX grammar language.

Assignments can be used to alter the values associated with to variables and are used as a subcomponent of composites, which will be described in the following section. The syntax of an assignment is given in Listing 2.23. The left-hand side component of an assignment is a reference to the target variable by name (including the index if the variable is an array, enclosed by square brackets), followed by the assignment operator (`:=`), and finalized by an expression as the right-hand side component describing the desired value to be assigned to the target variable. Note that both local and class variables are valid targets within an assignment.

2.2.6.3 Composites

```
Composite:
  '[' (guard=Expression ';')? assignments*=Assignment[';'] ']'
;
```

Listing 2.24: The composite declaration syntax defined through the TEXTX grammar language.

Within the SLCO framework, the statements within a transition are defined to be atomic entities. However, there exist use cases in which atomicity at a statement level is not sufficient to attain the desired result. Take for example a transition that assigns a value `v` to an array variable `A` at index `i`, but only if `i` is within the bounds of the array. To achieve this, two statements are required–a guard expression that checks if `i` is within the array bounds, and an assignment that assigns `v` to `A[i]`. Yet, there are no guarantees that `i` remained unaltered between the evaluation of the guard and execution of the assignment. Recall that the transitions of different state machines are running in parallel using interleaving semantics. Due to this, another assignment to `i` can take place between the range check and the execution of the assignment. Thus, the desired behavior cannot be attained unless the entire operation is made atomic.

The concept of composite statements has been introduced to allow for a collection of statements to be grouped into one overarching atomic entity. A composite statement consists of an optional (Boolean) guard expression that is followed by zero or more assignments. The syntax of a composite statement is given in Listing 2.24. The body of the composite is to be enclosed by square brackets and contains all of the statements that are part of the structure, each of them separated by a semicolon. A trailing semicolon needs to be included if only a guard expression is given.

2.2.6.4 Signals

```
SendSignal:
  'send' signal=NID
  '(' params*=Expression[','] ')' 'to' target=[Port]
;

ReceiveSignal:
  'receive' signal=NID
  '(' params*=VariableRef[','] ('|' guard=Expression)? ')' 'from' target=[Port]
;
```

Listing 2.25: The receive and send signal syntax defined through the TextX grammar language.

The communication through message-passing channels is controlled by the signal statements. There exist two types of signal statements: ones that send data, and ones that receive data, with the syntax of both types being presented in Listing 2.25. A send signal opens with the `send` keyword, followed by the name given to the signal message. The assigned name is followed by a list of zero or more comma separated expressions describing the values that need to be sent over the target channel. The number of statements provided needs to be equal to the number of parameters defined by the channel. Moreover, the resulting types of the expressions need to match with the types of the target parameters. The send signal statement is concluded with the `to` keyword followed by the name of the port to be used for the communication.

Similarly, A receive signal starts with the `receive` keyword, followed by the name of the signal that is accepted as an input by the signal. The signal name is succeeded by brackets containing a comma separated list of zero or more references to variables in which the channel's parameters should be stored. The list of references can be appended by an optional guard expression, which needs to be preceded by a vertical bar (`|`). If the given expression evaluates to `false` under the received parameter values, the communication will fail. The receive signal statement is concluded with the `from` keyword followed by the name of the port to be used for the communication.

A signal statement may block under certain circumstances. Send signals are blocked if the buffer associated with the signal is full. Oppositely, receive signals are blocked when the associated buffer is empty or if the received message is not of the expected format. In addition, a receive signal will block if the communication failed due to the guard expression evaluating to `false`.

2.3 Concrete Examples

The Slco 2.0 language syntax will be demonstrated through a set of examples. The first model, given in Section 2.3.1, will focus on demonstrating the syntax of the Slco language. Secondly, a model will be discussed in Section 2.3.2 that depicts the behavior of an elevator. Lastly, a model will be introduced in Section 2.3.3 that depicts the puzzle game *Toads and Frogs*.

2.3.1 Syntax Demonstration

The first example model, given in Listing 2.26, is used as an official demonstration of the Slco 2.0 language [13] in terms of general concepts, structures and notations used to declare a model. Note that, due to the obscured parts of the model, it cannot be parsed in the given form–it is an incomplete model made for demonstration purposes only. The model is named `Test` (line 1), and

```
1  model Test {
2    actions init
3    classes
4      P { ... }
5      Q {
6        variables Integer x y
7        ports Out1 Out2 InOut
8        state machines
9          SM1 {
10           variables Boolean started:=false
11           initial Com0 states Com1 Com2
12           transitions
13           Com0 -> Com1 {
14             send M(false, 0) to Out1;
15             started:=true
16           }
17           Com1 -> Com1 {
18             [x>0; x:=x-1; y:=y+1];
19             send N(y) to Out2
20           }
21           1: Com1 -> Com2 { receive S() from InOut }
22           Com2 -> Com0 { do init }
23         }
24         SM2 { ... }
25     }
26   objects p: P(), q: Q(x:=10, y:=0)
27   channels
28     c1(Boolean, Integer) async[2] lossless from q.Out1 to p.In1
29     c2(Integer) async lossy from q.Out2 to p.In2
30     c3() sync between p.InOut and q.InOut
31 }
```

Listing 2.26: A concrete (partial) model declared with the SLCO 2.0 language syntax.

contains an action `init` (line 2), the classes `P` and `Q` (lines 4-5), the objects `p` and `q` as instances of `P` and `Q` respectively with the latter object having defined the initial values `x:=10` and `y:=0` (line 26), and the three channels `c1`, `c2` and `c3` connected to their target object ports (lines 27-30).

Within class `Q`, two class variables `x` and `y` of type `Integer` are defined (line 6), together with the available ports `Out1`, `Out2` and `InOut` (line 7). In addition, the class `Q` defines the two state machines `SM1` and `SM2` (lines 9-24). The state machine `SM2` defines a local variable `started` having the initial value of `false` (line 10), the initial state `Com0`, additional states `Com1` and `Com2` (line 11), and four transitions (lines 13-22).

The first transition starts in state `Com0` and ends in state `Com1` (line 13). The first statement is a signal that sends a message with name `M` and parameter values `false` and `0` to port `Out1` (line 14). The signal statement is followed by an assignment setting the local variable `started` to `true` (line 15). The second transition, being a self-loop from state `Com1` to `Com1`, starts with a composite containing the guard expression `x > 0` and assignments `x := x - 1` and `y := y + 1` (line 18). The next statement sends a message named `N` through port `Out2` with `y` as the parameter value. The third transition, having a priority of `1` and going from state `Com1` to `Com2`, receives a message with name `S` from port `InOut` (line 21). Due to the assigned priority that is higher than the default, it is ensured that the third transition will only be considered if the second transition blocks, i.e., the second transition will always be checked before the third when the state machine resides in state `Com1`. Finally, the fourth transition, moving from state `Com1` to `Com2`, executes the user-defined action named `init` (line 22), which for this particular model, is described as an action that performs some unspecified initialisation procedure.

Observe that channel `c1` is a lossless, asynchronous channel with a buffer size of two. The channel only accepts messages with a `Boolean` and a `Integer` parameter, and has the source port `q.Out1`

and destination port `p.In1` (line 28). Similarly, channel `c2` is an asynchronous lossy channel with no buffer that only accepts messages with one `Integer` parameter, having the ports `q.Out2` and `p.In2` as the source and target ports respectively (line 29). Channel `c3` is a synchronous channel accepting messages with no input parameters having `p.InOut` as a source port and `q.InOut` as a target port (line 30).

2.3.2 Elevator

```
1  model Elevator {
2    classes
3      GlobalClass {
4        variables Byte[4] req Integer t Integer p Byte v
5        state machines
6          cabin {
7            initial idle states mov open
8            transitions
9              from idle to mov { v>0 }
10             from mov to open { t=p }
11             from mov to mov { [t<p; p:=p-1] }
12             from mov to mov { [t>p; p:=p+1] }
13             from open to idle { [req[p]:=0; v:=0] }
14         }
15         environment {
16           initial read states
17           transitions
18             from read to read { [req[0]=0; req[0]:=1] }
19             from read to read { [req[1]=0; req[1]:=1] }
20             from read to read { [req[2]=0; req[2]:=1] }
21             from read to read { [req[3]=0; req[3]:=1] }
22         }
23         controller {
24           variables Byte ldir
25           initial wait states work done
26           transitions
27             from wait to work { [v=0; t:=t+(2*ldir)-1] }
28             from work to wait { [t<0 or t=4; ldir:=1-ldir] }
29             from work to done { t>=0 and t<4 and req[t]=1 }
30             from work to work { [t>=0 and t<4 and req[t]=0; t:=t+(2*ldir)-1] }
31             from done to wait { [v:=1] }
32         }
33     }
34   objects
35     globalObject: GlobalClass()
36 }
```

Listing 2.27: A concrete model declared with the SLCO 2.0 language syntax depicting an elevator.

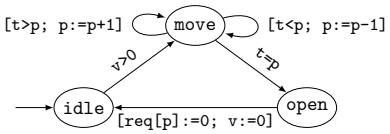
In this section, the `Elevator` model will be introduced, which depicts an elevator system servicing four floors. The model given in Listing 2.27, as included within the SLCO repository of example models under the name `elevator2.1.slco`[5], consists of the state machines `cabin`, `environment` and `controller`, with the four class variables `req`, `t`, `p` and `v` (line 4). In the model, the array variable `req` of length `n` is used to track which floors have requested the cabin, `t` is the floor number the controller directs the cabin to, `p` is the floor the cabin is currently at, and lastly, `v` is a byte that tracks if the cabin is currently executing the controller's movement instruction. The state machines contained within the model have been visualized in Figure 2.1. Each state machine fulfills a specific task:

- The `cabin` state machine (lines 6-14) has the states `idle`, `move` and `open`, with the model itself depicting the movement of the elevator cabin. The cabin starts in an `idle` state, and

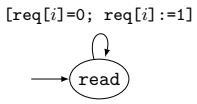
[5]Accessable at `https://github.com/melroy999/2IMC00-SLCO/tree/master/SLCO2.0models/elevator2.1.slco`

remains there until the elevator is instructed to move. In the `move` state, the cabin moves to the correct floor by moving up or down one step at a time. Once the right floor is reached, the cabin enters the `open` state, after which it proceeds to reset both `v` and `req[p]`, followed by a transition back to the `idle`. Observe that the `cabin` only has deterministic behavior in state `mov`–only one transition can be active at any time due to the chosen guard statements.

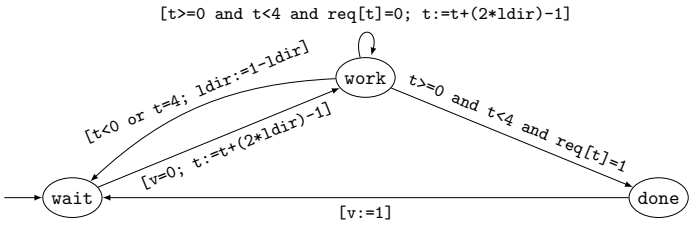
- The `environment` state machine (lines 15-22) only has a single state `read`, and has the task of modeling the act of calling an elevator to a certain position. Internally, state `read` has a separate transition for each floor $i \in [0, 4)$ back to itself, in which the associated value of `req[`i`]` is set to one if not already enabled. The `environment` state machine randomly picks a floor to call the cabin to, and is hence behaviorally completely non-deterministic.
- The `controller` state machine (lines 23-32) has the states `wait`, `work` and `done`, and instructs the elevator cabin's movements. The controller starts in the `wait` state and remains there until `v` becomes zero, after which it moves to the `work` state. In the `work` state, the controller increments or decrements the value of `t` until a floor is encountered that has a cabin request. The direction of the search is dictated by the `ldir`, which is a local variable (line 24) that is zero when searching up, and one when searching down. The search direction is swapped whenever the cabin reaches the top or bottom floor. The state machine proceeds to the `done` state after finding an open cabin request, and subsequently moves back to the `wait` state after enabling `v`. Observe that, similarly to state `mov` in state machine `cabin`, the `controller` exclusively displays deterministic behavior in state `work` due to the chosen guard statements.



(a) State Machine `cabin`.



(b) State Machine `environment`, with a separate transition for all $i \in [0, 4)$.



(c) State Machine `controller`.

Figure 2.1: Visual depiction of the `Elevator` SLCO model consisting of the three state machines `cabin`, `environment` and `controller` with the class variables `req`, `t`, `p` and `v`. Additionally, the `controller` state machine has a local variable `ldir`.

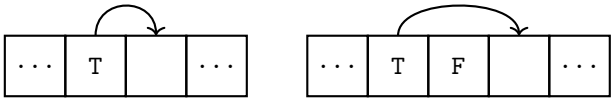
2.3.3 Toads and Frogs

The final example depicts a single player variant of the puzzle game named *Toads and Frogs* [22]. A brief introduction on the game and associated rules will be provided in Section 2.3.3.1, followed by a simulation of the game in the shape of an SLCO model in Section 2.3.3.2.

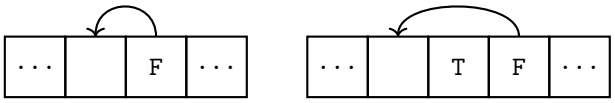
2.3.3.1 Game Description

Figure 2.2: The initial state of the *Toads and Frogs* playing board ($n = 4$). The board has an empty cell at the center flanked to the left by four toads (`T`) and to the right by four frogs (`F`).

The game is played on a playing board represented by an array of $n \cdot 2 + 1$ cells. There are n toads (depicted by the value `T`), n frogs (depicted by the value `F`), and one empty cell on the board. The initial state of the board is depicted in Figure 2.2: all toads and frogs are at the left and right side of the board respectively, with the empty cell being at the center position $n + 1$.



(a) Legal moves for toads (`T`).



(b) Legal moves for frogs (`F`).

Figure 2.3: The legal moves for toads and frogs in the *Toads and Frogs* puzzle game.

The movement of the toads and frogs is restricted to the moves presented in Figure 2.3. Toads are forced to hop to the right, while frogs are allowed only to hop to the left. A toad may hop to an empty cell that is located immediately to its right. Additionally, a toad may hop over a frog that is directly to its right if the target frog's neighbor is an empty cell–however, it is not allowed for a toad to hop over another toad. Analogously, a frog may hop to an open spot that is located directly to its left. Similarly, a frog can hop over a toad that is directly to its left if the target toad's neighbor is an empty cell. Any moves not mentioned above are deemed illegal.

Figure 2.4: The state that needs to be reached for a game of *Toads and Frogs* to be won ($n = 4$).

The end goal of the game is to mirror the playing field, i.e., the game is won if all toads are on the right side, and all frogs on the left side of the board, as depicted in Figure 2.4. Conversely, a game is considered lost if both toads and frogs have no legal moves remaining. In the single player variant of the game, the turns of the toads and frogs do not have to alternate–instead, the toads and frogs need to essentially work together to achieve a successful outcome.

2.3.3.2 Model Representation

The SLCO model `ToadsAndFrogs` depicting the *Toads and Frogs* puzzle game for $n = 4$ is given in Listing 2.28. The model has been designed in such a manner that the program is continuous, i.e., once the game is over, the board will be reset such that the game can be restarted. The model consists of the state machines `toad`, `frog` and `control`, with the four class variables `y`, `tmin`, `tmax`

```
1  model ToadsAndFrogs {
2    classes
3      GlobalClass {
4        variables
5          Integer y:=4 Integer tmin:=0 Integer fmax:=8
6          Integer[9] a:=[1,1,1,1,0,2,2,2,2]
7        state machines
8          toad {
9            initial q
10           transitions
11             from q to q { [y>0 and tmin!=y-1 and a[y-1]=1; a[y]:=1; y:=y-1; a[y]:=0] }
12             from q to q { [y>0 and tmin=y-1 and a[y-1]=1; a[y]:=1; tmin:=y; y:=y-1; a[y]:=0] }
13             from q to q { [y>1 and tmin!=y-2 and a[y-2]=1 and a[y-1]=2; a[y]:=1; y:=y-2; a[y]:=0] }
14             from q to q { [y>1 and tmin=y-2 and a[y-2]=1 and a[y-1]=2; a[y]:=1; tmin:=y; y:=y-2; a[y]:=0] }
15         }
16         frog {
17           initial q
18           transitions
19             from q to q { [y<8 and fmax!=y+1 and a[y+1]=2; a[y]:=2; y:=y+1; a[y]:=0] }
20             from q to q { [y<8 and fmax=y+1 and a[y+1]=2; a[y]:=2; fmax:=y; y:=y+1; a[y]:=0] }
21             from q to q { [y<7 and fmax!=y+2 and a[y+1]=1 and a[y+2]=2; a[y]:=2; y:=y+2; a[y]:=0] }
22             from q to q { [y<7 and fmax=y+2 and a[y+1]=1 and a[y+2]=2; a[y]:=2; fmax:=y; y:=y+2; a[y]:=0] }
23         }
24         control {
25           initial running states done success failure reset
26           transitions
27             from running to done { y=0 and a[y+1]=1 and a[y+2]=1 }
28             from running to done { y=1 and a[y-1]=2 and a[y+1]=1 and a[y+2]=1 }
29             from running to done { y=7 and a[y-2]=2 and a[y-1]=2 and a[y+1]=1 }
30             from running to done { y=8 and a[y-2]=2 and a[y-1]=2 }
31             from running to done { y>1 and y<7 and a[y-2]=2 and a[y-1]=2 and a[y+1]=1 and a[y+2]=1 }
32             from done to success { tmin>y and fmax<y }
33             from done to failure { not (tmin>y and fmax<y) }
34             from success to reset
35             from failure to reset
36             from reset to running { [
37               y:=4; tmin:=0; fmax:=8; a[4]:=0;
38               a[0]:=1; a[1]:=1; a[2]:=1; a[3]:=1;
39               a[5]:=2; a[6]:=2; a[7]:=2; a[8]:=2
40             ] }
41         }
42     }
43   objects
44     globalObject : GlobalClass()
45 }
```

Listing 2.28: A concrete model declared with the SLCO 2.0 language syntax depicting the single-player variant of the puzzle game named *Toads and Frogs*.

and `a` (lines 5-6). The variable `y` contains the index of the empty cell on the game board. The variables `tmin` and `fmax` track the indices of the leftmost and rightmost toad and frog respectively, such that the win condition can be evaluated efficiently. Lastly, the array `a` is a representation of the playing board, and encodes the empty cell as `0`, a frog as `1`, and a toad as `2`. The initial values assigned to the variables conform to the situation sketched in Figure 2.2. In the model, each state machine fulfills a specific role within the game:

- The state machine `toad` (lines 8-15) performs the movement actions that can be taken by a toad, and only contains the state `q`, which is marked as the initial state. Additionally, the state machine has four transitions. The first two transitions move a toad to an neighboring empty cell to the left, with the latter two executing a hop to the left over a frog into an empty cell. Each type of movement is represented by two transitions, with the difference being in the target subjects–the first occurrence (lines 11, 13) targets toads that are not the leftmost specimen, and vice versa for the last occurrence (lines 12, 14). The latter two transitions maintain the variable `tmin` in addition to executing the move itself, which ensures that `tmin` always points to the index of the leftmost toad.

- The state machine `frog` (lines 16-23) is analogous to the state machine `toad`, with the difference being that it focuses on the movements performed by the frogs instead. The first

two transitions move a frog to an neighboring empty cell to the right, with the latter two executing a hop to the right into an empty cell over a toad. On top of that, the second and fourth transitions (lines 20, 22) update the variable `fmax` such that it is always ensured that its value points to the index of the rightmost frog.

- The state machine `control` (lines 24-41) focuses on managing the state of the game. The state machine has the initial state `running` and is supplemented by the states `done`, `success`, `failure` and `reset`. In the `running` state, it is checked whether the game has reached a point where neither the toad or the frog is able to execute any moves. In terms of logic, this implies that the two slots to the left of the empty cell are frogs and two slots to the right are toads (lines 27-31). If the condition holds, the state machine proceeds to the `done` state, in which it is verified if the game has been completed successfully or ended in a failure. A game is successful if all toads are to the right and all frogs are to the left of the empty cell. Given that `tmin` and `fmax` represent the leftmost and rightmost toad and frog respectively, it must hence hold that `tmin > y and fmax < y` for the state machine to transition to the `success` state (line 32). Otherwise, a transition will be made to state `failure` (line 33). States `success` and `failure` make an empty transition to the state `reset` (lines 34-35). Lastly, the sole transition in the `reset` state will reset all the variables to their original value, such that the game can start over, signified by the transition returning to state `running` (lines 36-40).

Finally, it needs to be noted that the SLCO example model repository[6] does contain similar models (named `frogs.1` through `frogs.5`) that target the same single player variant of the *Toads and Frogs* puzzle game, barring the continuous nature of the model discussed in this section. However, through experimentation with the aforementioned models, it was observed by analyzing the generated code that the models do not behave as desired–in particular, it was discovered that most of the transitions are never enabled and can thus never be activated. Hence, the `ToadsAndFrogs` model has been redesigned from scratch to guarantee that the model behaves as expected.

[6] Accessable at `https://github.com/melroy999/2IMC00-SLCO/tree/master/SLCO2.0models`

Chapter 3

Preliminary Changes

The presence of a strong foundation is important, and hence, several changes will be made to the existing code used by the original code generator, including code of other modules. In this chapter, the focus will be on the latter, i.e., the improvements made to the code generator will be discussed in Chapter 6 instead. To start with, it needs to be noted that the Slco framework is used in ongoing research, and as such, all of the envisioned changes in the scope of this work will be implemented in a stand-alone project as to not affect the integrity of said research.

Two areas of interest have been identified that fall outside of the scope of the code generator component. First, several improvements can be made to the input of the code generator through the introduction of a preprocessing step that strives improve the structural consistency of the resulting syntax tree. On top of that, a number of structural inconsistencies between the Slco 2.0 language and the framework's target language Java have been identified in Section 2.2.6.1 that may lead to confusion and inadvertent (erroneous) behavior. The aforementioned issues and their solutions will be discussed further in Section 3.1 and 3.2 respectively.

3.1 Model Preprocessing

The code generator component of the Slco framework accepts any model deemed valid by the parser, but before the model can be used, it needs to be preprocessed. First, the model needs to be converted to a simple Slco model, which is defined to be a model in which each transition contains up to one statement. The Slco framework provides a transformation that converts an arbitrary model to a simple model that is semantically equivalent to the original model. The model transformation transforms a transition t consisting of statements S into a collection of transitions $t_1, ..., t_{|S|}$, such that each statement in S is associated with exactly one transition. Subsequently, the transitions are chained together in the given order through a collection of intermediary states. The conversion to a simple model, albeit not strictly necessary, allows for simplifications to be made to the code generator: for example, the procedures within the framework no longer need to account for situations where a transition fails halfway through its execution, since there can only exist a single point of failure in a transition, i.e., the transition's guard statement.

Next, several improvements are made to the syntax tree of the model to make the structure more maintainable and straightforward to work with. The given syntax tree is converted to a more elaborate data structure that extends the syntax tree with a collection of helper functions and support fields. Furthermore, the data structure automatically tracks parent and child relationships of objects within the tree, such that fields can be provided that reference to an object's parent state machine and class. Similarly, variable and expression reference objects are given a direct reference to the target variable instance, and classes are given references to their object instantiations.

On top of that, several structural improvements are made that focus upon the homogenization and simplification of objects within the parse tree. Through the homogenization, it is ensured that every transition has an expression that acts as its guard statement, i.e., it is ensured that every transition starts with an expression: as such, a true expression needs to be prepended to the list of statements for transitions that start with an assignment. Moreover, all composites are required to have a guard statement, with a default guard expression evaluating to true being assigned if absent, and default values of class and state machine variables are added if absent. Subsequently, several structural simplifications are performed. First, superfluous expression nodes are eliminated, i.e., expression objects that do not have an operator, with the aim of improving the readability of the structure. If applicable, expressions are simplified further by applying negations directly to the operators contained therein, and any double negations, if present, are removed. Next, superfluous assignments, namely, assignments that assign a variable to itself, are removed from the model. In addition, an attempt is made to replace composites by a single expression or assignment, depending on the guard statement and assignments contained therein. Composites having a guard statement that never holds true are replaced by the guard statement, given that the assignments are unreachable. Oppositely, a composite consisting of a single assignment is replaced by the latter if its guard statement holds invariably true. If a composite contains no assignments, it is replaced by its guard statement. Finally, note that for tractability and verification purposes, it is ensured that each statement maintains a reference to its original definition.

3.2 Inter-Language Consistency

In Section 2.2.6.1, it has been observed that there exist inconsistencies between the definition of the Slco 2.0 language and that of the target language Java: the order of precedence for the operators does not match, and the Slco 2.0 language is defined to be right-associative, instead of Java's left-associative approach. Said issues can be resolved by revising the language definition, but doing so is undesirable, due to the fact that the parser is not part of the code generator: instead, the parser is separate component of the framework that is used by other modules as well. Consequently, any changes made to the language definition may lead to unforeseen issues or consequences within other modules of the framework, which is problematic, since the Slco framework is used in ongoing research. As such, any changes made to the parser need to be considered carefully–revisions to the parser should only be implemented when deemed truly necessary.

First, observe that alternatively, both the order of precedence and associativity issues can be resolved by adding additional parenthesis to the target expression. Parentheses have an order of precedence that is higher than any other operator, and as such, they can be used to override the precedence conventions. Similarly, parenthesis can be used to make the desired expression syntactically unambiguous, such that both the left and right-associative approaches result in the same parse tree. However, the requirement of this approach is that the developer needs to formulate the expressions in a specific manner, which may or may not be counter-intuitive to their expectations, due to the fact that the majority of modern programming languages are left-associative and share the same operator precedence. Therefore, said inconsistencies can easily lead to confusion, which in turn makes the process of modeling behavior more error-prone. On top of that, the presence of discrepancies between the two languages has not been mentioned within the framework's documentation, and hence, it is presumed that the differences are unintended behavior. As such, several improvements have been made to the definition of the Slco 2.0 language to resolve the aforementioned consistency issues and improve the user-friendliness of the syntax. Furthermore, the required overrides are made only within the scope of the code generator component, which in turn ensures that ongoing research is not disrupted by the changes made.

The revised rules within the language definition are given in Listing 3.1, with the changes made being threefold. First, additional rules have been added to the language definition, such that the number of expression rules is equivalent to the number of levels in the desired order of precedence. Secondly, the operators have been redistributed over the expression rules, such that the operators

contained within each level match the operator precedence used by Java. The revised order of operator precedence in the SLCO framework is presented in Table 3.1. Lastly, the rules have been restructured, such that all operators, with the exception of the power operator, become left-associative operators. However, the latter is not a straightforward conversion, due to the fact that TEXTX uses a top-down parsing strategy: as such, it is incapable of handling left-recursive rules, since the inclusion thereof will cause the parser to end up in a state of infinite recursion. Said state can be avoided by redefining the rules such that they do not contain a recursive reference to itself. Alternatively, another parser using a bottom-down parsing strategy can be used, but given the concerns brought forward in the previous paragraph, this is not deemed an appropriate solution. Therefore, the rules have been redefined, such that it only contains references to the rule depicting the next higher level of precedence. Additionally, each level of precedence is treated as an n-ary operator over the operators contained within, such that recursion to itself is no longer required: instead of having a left and right side of an expression, each expression now consists of a collection of `terms` that are all of the same order of precedence, which does include the operators themselves. Consequently, a conversion of the `terms` attribute to a left-associative binary relation is required, since all modules using the parser rely upon a binary tree structure to function properly. The aforementioned restructuring is trivial, and can be implemented through a simple loop or recursive procedure that pairs the terms around the target operator from left to right, using the previously processed term as the left side of the relation.

```
Expression:
 terms=ExprPrec7 ((terms='or' | terms='||') terms=ExprPrec7)*
;

ExprPrec7:
 terms=ExprPrec6 ((terms='and' | terms='&&') terms=ExprPrec6)*
;

ExprPrec6:
 terms=ExprPrec5 ((terms='xor') terms=ExprPrec5)*
;

ExprPrec5:
 terms=ExprPrec4 ((terms='!=' | terms='=' | terms='<>') terms=ExprPrec4)*
;

ExprPrec4:
 terms=ExprPrec3 ((terms='<=' | terms='>=' | terms='<' | terms='>') terms=ExprPrec3)*
;

ExprPrec3:
 terms=ExprPrec2 ((terms='+' | terms='-') terms=ExprPrec2)*
;

ExprPrec2:
 terms=ExprPrec1 ((terms='*' | terms='/' | terms='%') terms=ExprPrec1)*
;

ExprPrec1:
 left=Primary (op='**' right=ExprPrec1)?
;

Primary:
 (sign='+' | sign='-' | sign='not')?
 (value=INT | value=BOOL | '(' body=Expression ')' | ref=ExpressionRef)
;

ExpressionRef:
 ref=NID ('[' index=Expression ']')?
;
```

Listing 3.1: The improved syntax of an expression as expressed through the TextX grammar language. The appropriate operators have been made left-associative. On top of that, the operator precedence has been adjusted to be in accordance with that of Java.

Operator	Description
`[]`	Array subscriptions
`()`	Parentheses
`not`, `+`, `-`	Negation, Unary Plus, Unary Minus
`**`	Exponentiation
`*`, `/`, `%`	Multiplication, Division, Modulo
`+`, `-`	Addition, Subtraction
`<`, `<=`, `>=`, `>`	Relational
`=` , `!=`, `<>`	Equality
`xor`	Exclusive Disjunction
`and`, `&&`	Conjunction
`or`, `||`	Disjunction

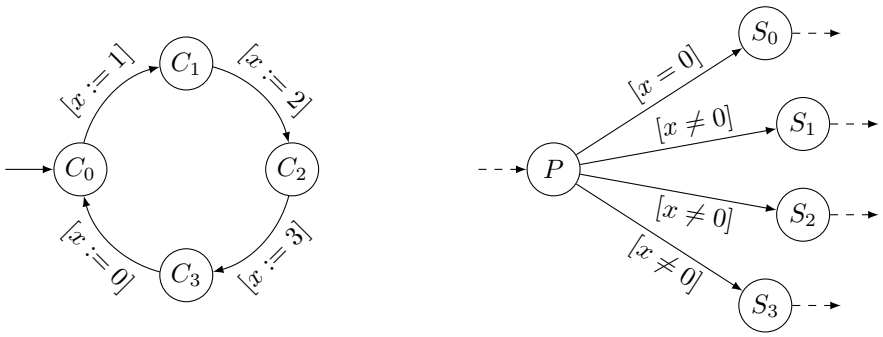
Table 3.1: The revised order of precedence of operators used within the Slco framework, adhering to the operator precedence as used in Java. An operator precedes another if it is placed higher up in the table. Operators located within the same table row have the same precedence level and will hence have the same significance.

Chapter 4

Deterministic Structures

In real-world applications, state machines often have a deterministic flavor, i.e., the transitions are only active under specific circumstances, in which the guard expressions could potentially be mutually exclusive. As such, an efficient decision structure can be constructed that avoids the execution of transitions that are inactive within the current context.

As it stands, the original implementation of the Slco code generator does not utilize a decision structure–instead, it picks a random transition out of the list of transitions that start in the current state, without filtering out the inactive transitions prior to the selection. The chosen non-deterministic approach is not only inefficient, but may also lead to situations with unexpected behavior. Take for example the situation sketched in Figure 4.1, which has the two state machines `Counter` and `Distributor`. The `Counter` state machine in Figure 4.1a counts up to three, after which it starts over at zero again, i.e. $x \leftarrow (x + 1) \mod 4$. The `Distributor` state machine in Figure 4.1b makes a decision in state P, based on the value of x. If $x = 0$, the transition $P \to S_0$ is the only active transition, otherwise, all transitions but $P \to S_0$ are active. Intuitively, one would expect that the transition $P \to S_0$ is taken whenever $x = 0$. Given that $x = 0$ only holds in the state C_0, and that `Counter` contains four states, it holds that $P \to S_0$ occurs with a probability of $p = \frac{1}{4}$ when using a deterministic procedure for the decision in state P of the `Distributor` state machine. However, the original implementation makes the decision non-deterministically, which means that the probability of executing $P \to S_0$ has been reduced to $p = \frac{1}{4} \cdot \frac{1}{4} = \frac{1}{16}$. As such, the transition $P \to S_0$ might be taken significantly less frequently than expected, which could lead to a discrepancy between the intended behavior and the executed behavior.



(a) State Machine `Counter` (b) State Machine `Distributor`

Figure 4.1: A parallel program consisting of the two state machines `Counter` and `Distributor` with a shared variable x, the latter of which has the initial value $x := 0$.

An effort has been made to solve the issue described above by introducing a priority system for the transitions. Each transition is given an (optional) priority such that higher priority transitions can be considered for firing before lower priority ones, e.g., the transition $P \rightarrow S_0$ in state machine `Distributor` could be given a higher priority, such that it is considered before the alternatives. However, the priority system introduces other issues and concerns, such as performance issues introduced by potential human errors in the assigned priorities. First of all, the priority system may lead to inefficient behavior that will not be encountered by a deterministic decision procedure. Take for example the parallel program described in Figure 4.1, and let all transitions with the guard $x \neq 0$ have a priority that is higher than that of transition $P \rightarrow S_0$. Under $x = 0$, it follows that transition $P \rightarrow S_0$ can only be fired after needlessly checking three inactive transitions with a higher priority. The aforementioned situation could have been avoided through the use of a simple decision structure. Additionally, it is important to keep in mind that the environment does not remain constant during the execution of the priority based decision procedure, i.e., the values of the shared variables might have changed between transition evaluations. As such, it might occur that a higher priority transition may have become active during the checking of a lower priority one. Thus, it is not guaranteed that the program strictly respects the assigned priorities, which might conflict with the developer's expectations.

4.1 Problem Statement and Approach

The inclusion of priorities does not solve the underlying issue, an as such, a more robust solution is needed that properly adds deterministic behavior to the model. An algorithm for the construction of decision structures is proposed in Section 4.3 that aims to reduce the non-deterministic behavior of the generated code through the concept of decision problems, more specifically the Satisfiability Modulo Theories (SMT) problem, to construct collections of transitions that can be decided between deterministically. The design of the algorithm is modular, such that several alternative implementations can be provided for certain components of the procedure. Said implementations have different strengths and weaknesses, and therefore, the right option needs to be chosen for the task at hand to attain the desired behavior. The modular design is achieved by representing the decision node construction process as a class: selective parts of the algorithm can be replaced or added through inheritance, which in turn keeps the code simple and maintainable.

The structure of this chapter will be as follows. To start with, the concept of decision problems and the associated theory solver tools used in the implementation will be introduced in Section 4.2. The decision structure construction algorithm will be discussed in Section 4.3, with its sub-procedures being discussed within Sections 4.4, 4.5 and 4.6 respectively, after which the available solver configurations are presented in Section 4.7. Finally, the chapter is concluded with Section 4.8, in which all of the available configurations are demonstrated on a concrete example. The conversion of the decision structure to code is outside of the scope of this chapter: instead, the integration of the decision structures into the code generator will be discussed in Chapter 6.

4.2 Satisfiability Modulo Theories (SMT)

In this chapter, several decision problems will be encountered that need to be solved. The concept of a decision problem is simple and straightforward: given an input model, an algorithm needs to be constructed that can give an answer to a yes or no question pertaining to the given model. An Satisfiability Modulo Theories (SMT) problem is a specific instance of a decision problem with a large amount of expressive power. Unlike the better known Boolean Satisfiability (SAT) problem for binary variables, SMT is capable of determining the satisfiability of expressions and formulas containing constructs and variables of other types as well, including real numbers, integers and more complex data structures such as collections [11].

The basic terminology used within SMT requires a brief introduction. The problem to be solved

is called an SMT instance or model, which consists of several binary-valued predicates over a set of variables of a specified type. The predicates are defined in first-order logic and they are used to put binary constraints upon the model that need to be satisfied for the instance to be considered solvable. The instance is called satisfiable if at least one configuration of variables exists for which the model is valid, i.e., all the predicates hold. Oppositely, if no such solution exists, the model is unsatisfiable, which implies that the model has contradictions and is hence unsolvable.

The expressions used for the transition guards in Slco consist primarily of classical inequalities, and as such, they can be modeled in SMT with ease. The implementation of the decision structures will use SMT to construct groups of transitions containing guard expressions with non-intersecting solution spaces. Generally, this can be achieved through simple logic statements using existential quantifiers. For example, two expressions have overlapping solution spaces if there exists a satisfiable solution for the conjunction of the two expressions. Furthermore, SMT can be used for several other small miscellaneous optimizations, including but not limited to the elimination and the simplification of given expressions.

On top of that, the SMT solvers themselves often provide even more possibilities, since they will not only output whether a model is satisfiable or not, but also provide a solution if the model is deemed satisfiable. In addition, several of the SMT solver implementations allow for the definition of not only decision problems, but also optimization problems in SMT through the inclusion of non-binary optimization predicates [23]. Hence, the greatest strength of SMT solving lies within the possibility to model an entire problem as a SMT instance, without having to define or implement an algorithm that can find the solution to the problem at hand. The use of SMT will greatly reduce the overall complexity of the problem, and due to the fact that no complex model conversions are needed, the implementation will become less error-prone and hence easier to maintain as well. Moreover, the readability and provability of the implementation will increase drastically, since the problem will be defined solely through a set of simple and verifiable mathematical equations: a constraint cannot be circumvented or revoked once it has been introduced, and as such, each of the introduced constraints can be analyzed independently for correctness.

Unfortunately, SMT solving also has several shortcomings and disadvantages. For one, an SMT model cannot be solved in polynomial time, since SMT solving is an NP-hard problem. Thus, it is unlikely that an SMT solution will outperform a purpose-made algorithm fulfilling the same purpose. However, due to a drastic increase of computational power of computers over the last few decades, the time needed to solve an SMT problem has reduced considerably too. In any event, the overall performance of SMT solving are not a major concern in this project, since the solver will be used exclusively in the code generation and not in the produced code itself. An additional disadvantage is that certain mathematical constructs cannot be modeled in SMT, such as exponentiation of a number to a non-constant value. However, given that these constructs are used sporadically in practice, it has been decided that it is better to mark incompatible constructs as exceptional cases that need to be processed outside of the decision structures.

The SMT solver that has been selected for this project is called Z3 [10], or more specifically Z3Py, which is its implementation in Python. The Z3 solver is known to be an efficient and reliable SMT solver that provides all the functionality described above. On top of that, the author has used Z3 during several courses in the past, and as such, is familiar with Z3 and its syntax. Hence, the use of Z3Py as the SMT solver for the project is deemed a safe and logical decision.

4.3 Constructing the Decision Structure

The overarching procedure for the construction of the decision structure is given in Algorithm 1, taking a collection of transitions T as the only input, with the requirement being that all transitions in T start in the same state, i.e., the transitions are grouped by the starting state prior to the extraction of the decision structures. Recall that a transition's activity is dictated by the combination of its source state and guard expression: the source state of a transition remains

constant, and hence, it does not make sense to make it part of a transition's conditional. Instead, it is more efficient to narrow down the list of target transitions beforehand, based on the current state of the state machine. The list of transitions T is converted to a tree structure of decision nodes $\langle d : G' \rangle$, where d is the type of choice made, namely DET for deterministic and NDET for non-deterministic, and G' is a list of candidates the decision is made over, where G' can be a mixture of both transitions and other decision nodes. To simplify the code generator implementation, it is required that the top level of the generated decision structure is a non-deterministic node. Additionally, the levels within the decision structure need to be of alternating decision types, i.e., non-deterministic decision nodes only contain deterministic decision nodes and vice versa.

Algorithm 1: GENERATEDECISIONSTRUCTURE(T)

```
1 T_false ← {t | t ∈ T : ¬∃_v(t_v)}
2 T_remaining ← T \ T_false
3 d ← CREATENONDETERMINISTICNODE(T_remaining)
4 d ← SIMPLIFY(d)
5 return d
```

The GENERATEDECISIONSTRUCTURE(T) algorithm does the following. First, the transitions T are divided into two groups. The superfluous transitions, i.e., transitions that have a guard expression with an empty solution space and thus never hold true, regardless of the chosen variable values, are assigned to T_{false} (line 1), where the notation t_v represents the evaluation of the guard expression of transition t under variable configuration v. Subsequently, the remaining transitions are assigned to $T_{\text{remaining}}$ (line 2). Note that all of the transitions in T_{false} are excluded from the constructed decision structure, due to the fact that they are, by definition, never in an active state: hence, the inclusion thereof would introduce unreachable code segments, which is detrimental to the quality of the generated code. Next, the remaining transitions $T_{\text{remaining}}$ are converted to a non-deterministic decision node d, using the function CREATENONDETERMINISTICNODE(T), which will be introduced in the following section (line 3). Finally, an attempt is made to simplify the generated decision structure d (line 4).

4.4 Extracting Non-Deterministic Nodes

A procedure is given in Algorithm 2 that converts a list of transitions T to a non-deterministic decision node $\langle \text{NDET} : G \rangle$ with the options G, where the latter is defined to be a collection of deterministic nodes and transitions that the decision is made over. A non-deterministic decision node makes an arbitrary choice between the options in G, and as such, its primary function is to handle decisions in which there is no strictly deterministic behavior to rely on, i.e., it is preferable that all pairs of choices $o_1, o_2 \in G$ have intersecting solution spaces. On top of that, it is required that each option $o \in G$ consists of transitions that are of the same priority, since otherwise, it cannot be guaranteed that the priorities set by the developer are respected.

Algorithm 2: CREATENONDETERMINISTICNODE(T)

```
1 T_conflicting ← {t | t ∈ T : ∀_{s∈T}(∃_v(s_v ∧ t_v))}
2 G_transitions ← GETNONDETERMINISTICNODEGROUPS(T \ T_conflicting)
3 G_nodes ← {CREATEDETERMINISTICNODE(T') | T' ∈ G_transitions}
4 G_nodes ← G_nodes ∪ T_conflicting
5 Sort G_nodes on the priority of the contained members in increasing order.
6 return ⟨NDET : G_nodes⟩
```

The CREATENONDETERMINISTICNODE(T) procedure is characterized as follows. First, the procedure filters out the transitions $T_{\text{conflicting}}$ that have intersecting solution spaces with every other

transition in T (line 1), due to the fact that the transitions $t \in T_{\text{conflicting}}$ are strictly deterministic if and only if they are considered in isolation, i.e., a combination of t with any other transition will invariably result in both transitions being possible active at the same time. Next, the GetNonDeterministicNodeGroups(T) procedure is called for $T \setminus T_{\text{conflicting}}$ (line 2), which selects the groups of transitions $G_{\text{transitions}}$ that the non-deterministic decision is made over. Note that the preemptive exclusion of the transitions $T_{\text{conflicting}}$ is an optimization measure, and as such, said exclusion is not strictly necessary. However, substantial performance benefits have been observed due to the exclusion thereof: the aforementioned procedure, of which two variants will be introduced in Section 6, uses SMT solving during the group selection process, and therefore, the performance of the procedure is highly dependant on the size of T. Afterwards, each group of transitions $T' \in G_{\text{transitions}}$ is converted to a deterministic decision node through the use of procedure CreateDeterministicNode(T) given in Section 4.5 (line 3). Subsequently, the previously excluded transitions $T_{\text{conflicting}}$ are added to G_{nodes} (line 4), under the observation that any transition $t \in T_{\text{conflicting}}$ can be part of a deterministic choice if and only if it is the only option within said choice, after which the groups in G_{nodes} are sorted based on the assigned priorities (line 5), such that options with a lower priority are always considered first. Finally, the procedure returns a non-deterministic decision node with the choices G_{nodes} (line 6).

4.4.1 Grouping Approaches

In this section, two implementations will be discussed for the procedure GetNonDeterministicNodeGroups(T). The procedure takes a list of transitions T as the input and gives a collection of transition groups G as the output. For each transition grouping $T' \in G$, it must hold that a deterministic decision can be taken between the transitions in T', i.e., a relation should exist between the transitions $s, t \in T'$ that can give an unambiguous indication on the active state of transition t based solely on that of transition s or vice versa. For example, the activity of transition s could imply that t cannot be, due to the observation that s and t cannot be active simultaneously. Oppositely, the active region of s may be a subset of that of t, which implies that transition t is guaranteed to be active when s is.

As described in the problem statement, the aim is to reduce the non-deterministic behavior of the generated code, which the proposed approach will achieve by minimizing the number of choices in each non-deterministic node. Therefore, the size of G needs to be minimized to ensure that the sizes of the collections $G_{\text{transitions}}$ and G_{nodes} used in Algorithm 2 are optimal. As such, the group selection procedure is considered to be an optimization problem for which an solution can be found through the application of an SMT solver. For this purpose, two implementations have been created using SMT solving. The first option, given in Section 4.4.1.1, uses a greedy approach that repeatedly extracts groups from T until no transitions remain. Oppositely, the second approach, given in Section 4.4.1.2, models the entirety of the problem as one comprehensive SMT model instance, which in turn ensures that the acquired result is optimal. The advantages and drawbacks of the proposed implementations will be detailed in their own respective sections.

4.4.1.1 Greedy

Algorithm 3: GetNonDeterministicNodeGroups(T) (Greedy)

```
1 G ← ∅
2 while |T| > 0 do
3 |   M ← CreateGroupingModel(T)
4 |   T' ← The transitions t ∈ T assigned to group slot 0 within the solution of model M.
5 |   G ← G ∪ {T'}
6 |   T ← T \ T'
7 return G
```

The first group selection approach repeatedly extracts groups of transitions $T' \subseteq T$ from T until there are no more transitions that remain to be processed. The greedy selection procedure is presented in Algorithm 3. The procedure takes a collection of transitions T as the sole input, and outputs a list of groupings G that meets the requirements set in Section 4.4.1. The procedure starts by defining an empty set G (line 1), after which it proceeds to repeatedly extract maximum-sized transition groups adhering to the set constraints until all transitions have been processed (lines 2-6). Next, the SMT model M is created through the procedure CreateGroupingModel(T) (line 3), which will be introduced shortly. The model M is subsequently solved, after which the solution is used to construct the list of target transitions T' (line 4), where a transition is part of T' if and only if the assigned group is 0. Finally, the transitions T' are added as a group to G (line 5), after which all transitions in T' are removed from T (line 6).

Algorithm 4: CreateGroupingModel(T) (Greedy)

1 $M \leftarrow$ Create an empty SMT model instance.
2 Add all support variables, and the associated value constraints, to the model M.
3 Add constraints to M that restrict solution space overlaps within group slot 0.
4 Add constraints to M that ensure equal priorities within group slot 0.
5 Add an optimization constraint to M that minimizes the sum of assigned group slots.
6 **return** M

The procedure CreateGroupingModel(T), given in Algorithm 4, introduces the variables and constraints that are required to represent the selection process as an SMT optimization problem, and takes a list of transitions T as the only input. First, an empty model M is created (line 1), after which all of the support variables and their associated value constraints are added to M (line 2). The support variables and their associated constraints are given in Sections 4.4.2 and 4.4.3 respectively. Next, constraints are introduced to M that limit the types of solution space overlaps that may occur within group 0 of the solution (line 3). The overlap constraints, and the three implemented variants thereof, are discussed in Section 4.4.4. Furthermore, several additional constraints are added to M that ensure that all of the transition $t \in T'$, i.e., the transitions selected to be part of group 0, are of the same priority (line 4). The priority constraints are given in Section 4.4.5. Finally, the definition of model M is concluded through the introduction of the optimization constraint, which is elaborated further in Section 4.4.6.

The advantage of the greedy approach is its simplicity and reliability: a transition is either part of the solution, or it is not. Thus, the approach makes it more straightforward to find potential errors within the implementation of the model. Moreover, the greedy approach is more efficient, since fewer constraints are needed in the SMT model M to attain a valid result, due to the fact that the transitions in T are only split into two groups. Consequently, the approach has better performance characteristics, since the running time of SMT solvers depends heavily on the number of constraints contained within a model. The algorithm aims to minimize the size of G through the intuition that the size of G can be reduced by maximizing the size of the extracted transition groups T'. As such, the proposed procedure is a greedy algorithm, since an optimal solution is chosen during each loop iteration. Therefore, the disadvantage of the proposed procedure is that the solution of the greedy approach is sub-optimal: it cannot be guaranteed that the result G is a minimal solution to the group selection problem.

4.4.1.2 Optimized

Next, an implementation of the GetNonDeterministicNodeGroups(T) procedure is proposed that uses a more elaborate SMT model that guarantees an optimal group assignment. The procedure presented in Algorithm 5 takes a list of transitions T as the input, and returns a list of transition groups G that meet the requirements set in Section 4.4.1. The procedure starts by constructing SMT model M that assigns every transition $t \in T$ to a group slot $0 \leq i < |T|$ in ac-

Algorithm 5: GetNonDeterministicNodeGroups(T) (Optimal)

```
1 M ← CreateGroupingModel(T)
2 G ← The transitions t ∈ T grouped by the slots assigned within the solution of model M.
3 return G
```

cordance with the given constraints (line 1). Subsequently, the model M is solved, after which the transitions are grouped based on the group slots assigned within the model solution (line 2).

Algorithm 6: CreateGroupingModel(T) (Optimal)

```
1 M ← Create an empty SMT model instance.
2 Add all support variables, and the associated value constraints, to the model M.
3 Add constraints to M that restrict solution space overlaps within group slots 0 ≤ i < |T|.
4 Add constraints to M that ensure equal priorities within group slots 0 ≤ i < |T|.
5 Add an optimization constraint to M that minimizes the sum of assigned group slots.
6 return M
```

The optimal version of the procedure CreateGroupingModel(T), given in Algorithm 6, introduces the variables and constraints that are required to represent the group selection process of transitions in T as an SMT optimization problem. The described procedure is analogous to the CreateGroupingModel(T) procedure described in Section 4.4.1.1, but with one crucial difference: the overlap and priority constraints are introduced for all groups $0 \leq i < |T|$, instead of just group 0. Consequently, all of the group slots are subjected to the target constraints, and as such, the transitions can be grouped directly by the assigned group slots, since the constraints ensure that the required characteristics hold for all group slots within the solution.

The optimal approach has the primary benefit that the selected grouping G is, as the name suggests, optimal: G is of the minimum attainable size with the given constraints, which implies that the non-deterministic behavior within the generated code is minimal. However, a more elaborate SMT model needs to be constructed to find the solution, which leads to a measurable performance impact. On top of that, the optimal procedure is difficult to maintain, since there are no intermediate steps that can be analyzed during the implementation process. As such, the optimal approach is deemed more prone to error than its greedy counterpart.

4.4.2 Variables

A series of (support) variables need to be introduced that can be used to construct constraints upon the model. The use of support variables simplifies the overall notation of the constraints, and hence improves the readability thereof. Furthermore, the inclusion of these supportive variables is considered to be a necessary workaround, since Z3Py does not support the use of existential and universal quantifiers in optimization problems. The available variables are as follows:

$group[t] \in \mathbb{Z}$ As a starting point, a number of variables need to be defined that hold the result of the model. Let $group[t]$ be an integer value that represents the group a transition $t \in T$ is assigned to. The allowed range of values for variable $group[t]$ depends on the approach chosen in Section 4.4.1. If the selection process is greedy, then $0 \leq group[t] < 2$ needs to hold, where a value of zero signifies that t is part of the selected group. Otherwise, the allowed range of values is $0 \leq group[t] < |T|$, in which case the assigned number is equivalent to the group number. Observe that the number of available group slots for the optimal grouping approach is equivalent to the length of T, which ensures that each transition can be assigned to an unique group if necessary.

$priority[t] \in \mathbb{Z}$ In addition, variables need to be included that hold the priority of the transitions, such that the latter can be used by the model to create groups in which all members have equal priorities. Let $priority[t] = t_p$ be an integer constant that holds the priority of transition $t \in T$, with the notation t_p representing the priority of the given transition t.

$and[s][t] \in \mathbb{B}$ On top of that, several variables need to be introduced that indicate whether the guard statements of two transitions have intersecting solution spaces. Let $and[s][t] = \exists_v(s_v \wedge t_v)$ be a Boolean constant that denotes whether pairs of transitions $s, t \in T$ have a common solution within their solution space, with the notation t_v representing the evaluation of the guard expression of transition t under variable configuration v.

$equal[s][t] \in \mathbb{B}$ The next set of variables indicate whether the solution spaces of two transition guard expressions are equivalent. Let $equal[s][t] = \forall_v(s_v = t_v)$ be a Boolean constant that denotes whether a pair of transitions $s, t \in T$ have equivalent solution spaces, i.e., there does not exist a variable configuration v for which the evaluations s_v and t_v differ.

$subset[s][t] \in \mathbb{B}$ The final collection of variables is used to identify a subset relationship between the solution spaces of two transition guards. Let $subset[s][t] = \forall_v(s_v \implies t_v)$ be a Boolean constant that denotes whether a transition $s \in T$ has a solution space that is a subset of that of transition $t \in T$. In other words, verify that there does not exist a variable configuration v for which the implication $s_v \implies t_v$ holds.

4.4.3 Variable Constraints

Several constraints need to be introduced that enforce the limitations set upon the variables as described in Section 4.4.2, i.e., the value of $group[t]$ for $t \in T$ needs to be limited to the desired range, with all other variables needing to be equivalent to the desired constant value.

$$\bigwedge_{t \in T} \Big(0 \leq group[t] < 2 \Big) \tag{1a}$$

$$\bigwedge_{t \in T} \Big(0 \leq group[t] < |T| \Big) \tag{1b}$$

First, it needs to be ensured that the value of $group[t]$ for $t \in T$ is always within the described range. As discussed in the previous section, the allowed range of values depends on the approach chosen in Section 4.4.1. The constraint depicted in Equation 1a, which forces the value of $group[t]$ to be either 0 or 1, is used for solutions using a greedy approach. Oppositely, Equation 1b is used by the optimal approach, and depicts the requirement that $group[t]$ needs to refer to one of the available group slots 0 to $|T| - 1$, such that every transition can be assigned to a group.

$$\bigwedge_{t \in T} \Big(priority[t] = t_p \Big) \tag{2}$$

Next, it needs to be ensured that the $priority[t]$ constant is equivalent to the priority of the targeted transition. The constraint accomplishing this is given in Equation 2, which equates $priority[t]$ for all $t \in T$ to the priority of transition t represented by the value t_p.

$$\bigwedge_{s \in T} \bigwedge_{t \in T} \Big(and[s][t] = \exists_v(s_v \wedge t_v) \Big) \tag{3}$$

The constants $and[s][t]$, for all transition pairs $s, t \in T$, are constrained by Equation 3. The given constraint ensures that $and[s][t]$ holds true if and only if there exists a variable configuration v for which both s_v and t_v hold true, which signifies that s and t have intersecting solution spaces.

$$\bigwedge_{s \in T} \bigwedge_{t \in T} \Big(equal[s][t] = \forall_v (s_v = t_v) \Big) \tag{4}$$

The constraint given in Equation 4 requires that the constant $equal[s][t]$, for all transition pairs $s, t \in T$, depicts the equality relation between the guard statements of transitions s and t. The given constraint ensures that $equal[s][t]$ holds true if and only if the solution spaces for transitions s and t are equivalent, i.e., for all variable configurations v, it needs to hold that s_v equals t_v.

$$\bigwedge_{s \in T} \bigwedge_{t \in T} \Big(subset[s][t] = \forall_v (s_v \implies t_v) \Big) \tag{5}$$

Lastly, the constants pertaining to the subset relation between transitions is constrained through Equation 5. The given constraint dictates that, for all transition pairs $s, t \in T$, it must hold that $subset[s][t]$ holds if and only if the solution space of the guard of s is a subset of that of t, i.e., for all variable configurations v, it needs to hold that s_v implies t_v.

4.4.4 Overlap Constraints

The next category of model constraints focuses on introducing rules that restrict the presence of solution space overlaps between transitions assigned to a target group i, with the possible values of i being determined by the chosen solving approach. If the greedy approach is chosen, the constraints are introduced for $i = 0$ only. Otherwise, overlap constraints will be introduced for all $0 \leq i < |T|$, such that it can be assured that all groups in the solution meet the requirements set by said constraint. Additionally, the constraints are required to accept any group that consists of at most one transition–the presence of the aforementioned property is crucial, since it ensures that both solving approaches work appropriately. The property is required by the greedy approach, since it guarantees that the size of the pool of unprocessed transitions will shrink during every iteration. In other words, it guarantees that the selection process of the greedy approach will terminate. For the optimal solution, the property ensures that there is always a satisfiable solution to the model, i.e., a solution in which each transition is assigned to an individual group.

Three different overlap constraints have been defined and implemented. The first variant, given in Section 4.4.4.1, is a basic rule that does not allow any overlaps to be present within selected groups. The second variant, given in Section 4.4.4.2, allows for two transitions to overlap if and only if their solution spaces are equivalent. Lastly, the third variant, given in Section 4.4.4.3, allows for two transitions to overlap if and only if their solution spaces satisfy a subset relation.

4.4.4.1 Basic

The first and most straightforward overlap constraint strictly prohibits the presence of overlapping solution spaces within a target group i. As such, when a transition $t \in T$ is selected to be part of group i, it needs to hold that no other transition $s \in T$, with exception of t itself, has been assigned to group i while having intersecting solution spaces with t.

$$\bigwedge_{t \in T} \left(group[t] = i \implies \neg \bigvee_{s \in T \setminus \{t\}} \Big(group[s] = i \wedge and[s][t] \Big) \right) \tag{6a}$$

The aforementioned behavior is expressed by the constraint given in Equation 6a through the use of an implication, i.e., for $group[t] = i$ to hold, it must be implied that the observations $group[s] = i$

and $and[s][t]$ are mutually exclusive for all transitions $s \in T \setminus \{t\}$. Note that the exclusion of t within the exclusivity check is strictly necessary, due to the fact that $and[t][t]$ invariably holds true, and as such, it is impossible to assign transition t to group i, since $group[t] = i$ and $and[t][t]$ are not mutually exclusive. Thus, t needs to be excluded for the constraint to yield usable results.

Proof of Correctness First, it needs to be proven that the constraint ensures that group i does not contain overlapping solution spaces. Suppose that the SMT solver assigns the transitions T' to group slot i, and that T' contains a pair of transitions $s, t \in T'$, with $s \neq t$, for which $\exists_v(s_v \wedge t_v)$ holds true, i.e., transitions s and t have solution spaces that do intersect. As such, the value of $and[s][t]$ is true, since by definition, $and[s][t] = \exists_v(s_v \wedge t_v)$ for all pairs $(s, t) \in T$. Moreover, for s and t to be part assigned to group slot i, it must hold that $group[s] = i$ and $group[t] = i$. Given this observation, it follows that the left side of the implication in Equation 6a evaluates to true, i.e., $group[t] = 1$, and as such, $\neg \bigvee_{s \in T \setminus \{t\}}(group[s] = i \wedge and[s][t])$ must hold for both s and t to be in T'. Given that $s \in T \setminus \{t\}$, it must thus follow that $group[s] = 1 \wedge and[t][s]$ evaluates to false. However, this is a contradiction, since $group[s] = 1 \wedge and[t][s]$ with the given values evaluates to true, which violates the proposed constraint. As such, the constraint will not allow for s and t to the assigned to the same group slot i, and hence, it is guaranteed that all pairs of transitions $s, t \in T'$ that have been assigned to group i have no overlapping solution spaces.

Furthermore, it is clear to see that Equation 6a does not reject a configuration in which t is the only element assigned to group i, due to the fact that $group[s] \neq i$ for all $s \in T \setminus \{t\}$. As such, the disjunction on the right-hand side will always evaluate to false, which in turn will make the implication hold true. Thus, it follows that the proposed constraint meets the requirement that groups consisting of a single transition are always satisfactory.

4.4.4.2 Equality

The approach taken in the previous constraint satisfies the basic requirements, but the suggested rule proves to be sub-optimal in situations where multiple transitions share the same solution space. Let $s \in T$ and $t \in T$ be two distinct transitions for which the solution spaces are equivalent. Additionally, let $u \in T$ be another transition that does not overlap with either s or t. In the current approach, two hypothetical deterministic groups are created: one group containing u and s, and the other consisting solely of t. Intuitively, all three transitions can be contained in one deterministic group through the introduction of a nested non-deterministic group, namely a deterministic group consisting of the transition u and the nested group $\langle \text{NDET} : \{s, t\}\rangle$. Moreover, the nested node can be given an encapsulating guard statement, which, given the fact that s and t have equivalent solution spaces, is equal to either of the two guard statements.

The benefit of the proposed revision is that it improves the effectiveness and efficiency of the generated code by increasing the degree of deterministic behavior therein: the choice between s and t remains non-deterministic, but the choice in question can only be reached by passing through a deterministic choice beforehand. As such, transitions s and t will not be visited if u is active and vice versa, which in turn reduces the overall frequency of failed transition activations.

The desired behavior can be defined as follows: overlaps between two transitions is allowed if and only if their solution spaces are equivalent. In other words, when a transition $t \in T$ is selected to be part of group i, it needs to hold that no other transition $s \in T$, with the exception of t itself, has been assigned to group i while having intersecting solution spaces with t, with the exception that s and t may be part of the same group if they have equivalent solution spaces.

$$\bigwedge_{t \in T} \left(group[t] = i \implies \neg \bigvee_{s \in T \setminus \{t\}} \Big(group[s] = i \wedge and[s][t] \wedge \neg equal[s][t] \Big) \right) \tag{6b}$$

The aforementioned behavior is modeled by the constraint given in Equation 6b. The proposed

rule is a refinement of the basic overlap constraint given in Equation 6a. The conjunction within the clause $group[s] = i \wedge and[s][t]$ has been extended with a negated reference to the variable $equal[s][t]$, such that the clause will evaluate to false when two transitions are equal, regardless of the value of $and[s][t]$. As such, the implication will not reject groupings in which all distinct solution spaces are non-overlapping, which in turn fulfills the desired behavior.

Proof of Correctness To start with, it needs to be proven that the given constraint allows overlaps in i if and only if the overlapping transitions have equivalent solution spaces. Suppose that the SMT solver assigns the transitions T' to group slot i, and that T' contains a pair of transitions $s, t \in T'$, with $s \neq t$, for which $and[s][t]$ holds true, i.e., s and t do have overlapping solution spaces. Additionally, suppose that $\exists_v(s_v \neq t_v)$ holds true, i.e., the solution spaces of transitions s and t are not equivalent. As such, $\forall_v(s_v = t_v)$ does not hold, which implies that $equal[s][t]$ evaluates to false due to the constraint that $equal[s][t] = \forall_v(s_v = t_v)$. Given that s and t are in group i, it follows that $group[s] = i$ and $group[t] = i$, and hence, the left side of the implication in Equation 6b evaluates to true. Consequently, it follows that the negated disjunction $\neg \bigvee_{s \in T \setminus \{t\}}(group[s] = i \wedge and[s][t] \wedge \neg equal[s][t])$ must hold true for both s and t to be in T'. Given that $s \in T \setminus \{t\}$, it must follow that $group[s] = i \wedge and[s][t] \wedge \neg equal[s][t]$ evaluates to false, but yet, this is a contradiction: under the given values, the aforementioned conjunction evaluates to true. As such, the constraint will not allow for s and t to be assigned to the same group slot i, and hence, it is guaranteed that all transition pairs $s, t \in T'$ assigned to i have solution space overlaps if and only if they have equivalent solution spaces.

On top of that, Equation 6b does not reject a configuration in which t is the sole element assigned to group i. In Section 4.4.4.1, it has been shown that the basic overlap constraint meets the set requirement, based on the fact that $group[s] \neq i$ for all $s \in T \setminus \{t\}$ holds. As such, the conjunction $group[s] = i \wedge and[s][t] \wedge \neg equal[s][t]$ will never hold, which in turn assures that the negated disjunction it is used within evaluates to true. Thus, it follows that the revised constraint meets the requirement that groups consisting of a single transition are always considered valid.

4.4.4.3 Subset

The equality-based overlap constraint given in Section 4.4.4.2 can be improved further through the use of a subset relation instead of equality relation. Let $s, t \in T$ be two distinct transitions for which the solution space of s is a subset of that of t. Moreover, let $u \in T$ be another transition that has no overlap with s and t. Both of the approaches discussed so far will force s and t to be in different deterministic groups, on the basis that the solution spaces of s and t overlap and are not equivalent. The aforementioned behavior is not optimal: the subset relation indicates that transition s only needs to be considered if t is active, which is as of yet untapped deterministic behavior. Hence, s and t should be combined into a nested non-deterministic group, in which the guard statement of t can be used as the encapsulating guard statement of the decision node.

The advantage of the proposed approach is twofold. First of all, the frequency of failed transition activations is reduced, since it can be deduced from the active state of t whether s can be active or not. As such, the generated code becomes more efficient, since the activity of a transition will only be checked if truly necessary. Secondly, the approach can be applied in a recursive manner, since it is possible that the nested non-deterministic groups can be dissected and processed further. Therefore, a more elaborate decision structure can be created that has as many decision node levels as required, which in turn increases the flexibility and effectiveness of the generated code.

The desired behavior can be defined as follows: overlaps between two transitions is allowed if and only if their solution spaces have a subset relation, i.e., when a transition $t \in T$ is selected to be part of group i, it needs to hold that no other transition $s \in T$, with exception of t itself, has been assigned to group i while having intersecting solution spaces with t, with the exception that s and t may be part of the same group if one of the solution spaces is a subset of the other.

$$\bigwedge_{t \in T} \left(group[t] = i \implies \neg \bigvee_{s \in T \setminus \{t\}} \Big(group[s] = i \wedge and[s][t] \wedge \neg (subset[s][t] \vee subset[t][s]) \Big) \right) \quad (6c)$$

The constraint given in Equation 6c models the aforementioned behavior and improves the basic overlap constraint given in Equation 6a. The conjunction within the clause $group[s] = i \wedge and[s][t]$ has been extended with the clause $\neg(subset[s][t] \vee subset[t][s])$, such that the conjunction evaluates to false when transition $s \in T$ is a subset of $t \in T$ or vice versa, regardless of the value of $and[s][t]$. Therefore, the implication will not reject groupings that contain solution spaces with subset relations, which is the desired behavior. Observe that both the variables $subset[s][t]$ and $subset[t][s]$ need to be included in the negated clause, since subsets are an asymmetrical relation: as such, both directions need to be checked to identify a subset relation.

Proof of Correctness Next, it needs to be proven that the proposed constraint allows transitions in group i to overlap if and only if the solution space of one is the subset of the other. Suppose that the SMT solver assigns the transitions T' to group i, and that T' contains a pair of transitions $s, t \in T'$, with $s \neq t$, for which $and[s][t]$ holds true, i.e., the solution spaces of s and t intersect. Additionally, suppose that $\forall_v(s_v \implies t_v)$ and $\forall_v(t_v \implies s_v)$ both evaluate to false, i.e., s and t do not have a subset relation. Hence, by the constraint $subset[s][t] = \forall_v(s_v \implies t_v)$, it follows that the constants $subset[s][t]$ and $subset[t][s]$ have the value false. Given that s and t are in group i, it holds that $group[s] = i$ and $group[t] = i$, and as such, the left-hand side of the implication in Equation 6c holds true. Therefore, $\neg \bigvee_{s \in T \setminus \{t\}} (group[s] = i \wedge and[s][t] \wedge \neg(subset[s][t] \vee subset[t][s]))$ must hold true for both transitions s and t to be in T'. Given that $s \in T \setminus \{t\}$, it must follow that $group[s] = i \wedge and[s][t] \wedge \neg(subset[s][t] \vee subset[t][s])$ evaluates to false, but yet, this is not the case: under the given values, the conjunction holds. Thus, by contradiction, it can be concluded that, for all transition pairs $s, t \in T'$ assigned to i, it holds that s and t are allowed to overlap if and only if their solution spaces share a subset relation.

Moreover, it needs to be shown that Equation 6c does not reject a configuration in which t is the sole transition assigned to group i. Given that t is the only element in group i, it holds that $group[s] \neq i$ for all $s \in T \setminus \{t\}$. As such, the negated disjunction on the right side of the implication will hold true, due to the fact that it is over all $s \in T \setminus \{t\}$ and that the conjunction contained within will always evaluate to false, since $group[s] \neq i$ holds for all $s \in T \setminus \{t\}$. Thus, the implication for group i holds true, and hence, it follows that the proposed constraint meets the requirement that groups consisting of a single transition are considered a valid solution.

4.4.5 Priority Constraints

The final type of constraint focuses on the creation of rules that ensure that group i consists of transitions that all have the same priority. Observe that priorities within groups cannot be mixed, due to the fact that other decision groups may contain members of a lower priority: as a result, the transition priorities set by the developer would not be enforceable. Hence, it must hold that all selected transitions within the group have the same priority. The value range of i depends on the chosen solving approach: if greedy, $i = 0$, otherwise, the constraints are generated for all $0 \leq i < |T|$. On top of that, it needs to be ensured that group selections containing only a single transition always satisfy the constraint, such that the proper functionality of the solving approaches can be assured. Consult Section 4.4.4 for further details.

$$\bigwedge_{t \in T} \left(group[t] = i \implies \neg \bigvee_{s \in T \setminus \{t\}} \Big(group[s] = i \wedge priority[s] \neq priority[t] \Big) \right) \quad (7)$$

The aforementioned constraint is given in Equation 7, and expresses that, for transition t to be part of group i, it must be implied that no other transition $s \in T$, with exception of t itself, is in group i while having a different priority. I.e., for $group[t] = i$ to hold true, it must be implied that the observations $group[s] = i$ and $priority[s] \neq priority[t]$ are mutually exclusive for all $s \in T \setminus \{t\}$. As discussed in Section 4.4.4.1, the transition t needs to be excluded from the exclusivity check, since the inclusion thereof will cause t to be rejected by all groups.

Proof of Correctness First, it needs to be proven that the constraint ensures that group i consists of transitions having the same priority. Suppose that the SMT solver assigns the transitions T' to group slot i, and that T' contains a pair of transitions $s, t \in T'$, with $s \neq t$, for which $priority[s] \neq priority[t]$ holds, i.e., the priorities s_p and t_p are not equivalent. For both s and t to be part of group i, it must hold that $group[s] = i$ and $group[t] = i$. Due to this, the left side of the implication in Equation 7 evaluates to true, and as such, $\neg \bigvee_{s \in T \setminus \{t\}} (group[s] = 1 \wedge priority[s] \neq priority[t])$ must hold for both s and t to have been selected to be part of group i. Given that $s \in T \setminus \{t\}$, it must follow that $group[s] = 1 \wedge priority[s] \neq priority[t]$ evaluates to false. Yet, $group[s] = 1$ and $priority[s] \neq priority[t]$ both hold true, which leads to a contradiction: s and t could not have been assigned to the same group i. As such, the constraint guarantees that all pairs of transitions $s, t \in T'$ that have been assigned to group i have equal priorities.

Furthermore, it is clear to see that Equation 7 does not reject any configuration in which t is the only element assigned to group i, due to the fact that $group[s] \neq i$ for all $s \in T \setminus \{t\}$. As such, the disjunction on the right-hand side will always evaluate to false, which in turn will make the implication hold true. Thus, it follows that the proposed constraint meets the requirement that groups consisting of a single transition are always satisfactory.

4.4.6 Optimization Constraint

$$\textbf{minimize} \sum_{t \in T} group[t] \tag{8}$$

The SMT optimization problem is finalized by the inclusion of the minimization constraint given in Equation 8. The given constraint minimizes the sum of $group[t]$ for all $t \in T$, which effectively maximizes the size of each selected group, regardless of the chosen solving approach.

Proof of Correctness The desired behavior differs for each of the two solving approaches. Thus, it needs to be shown for both approaches that the desired characteristics are met.

The greedy approach requires that a maximum sized group is selected that meets all of the set constraints, where a transition is considered part of the selected group if and only if t is assigned to group 0, i.e., $group[t] = 0$. Transitions that are not part of the solution are instead assigned to the unrestricted group 1, due to the fact that $0 \leq group[t] < 2$ needs to hold. As such, it is trivial to see that the selection size is maximal if the sum of $group[t]$ for $t \in T$ is minimal.

On the other hand, the optimal approach requires that the number of groups in the solution is minimal, i.e., a new group should only be introduced if the set constraints cannot be satisfied otherwise. Suppose that G is the optimal grouping under the set constraints, and recall that $group[t]$ for $t \in T$ is constrained by the rule $0 \leq group[t] < |T|$. First, it needs to be observed that the groupings G and G' are considered equivalent as long as they consist of the same groups. As such, the assumption is made that the groups within G are ordered by their size in decreasing order, such that the sum of the assigned group slots $\sum_{t \in T} group[t]$ is minimal. For the sum to be minimal, it also needs to hold that the first group is assigned to slot 0, with all active groups i in the range $0 \leq i < |G|$ containing at least one transition, i.e., the sum will be minimal for the given group G if and only if $0 \leq group[t] < |G|$ holds for all $t \in T$.

Next, observe that G can be transformed into any another valid solution G' through a series of swaps and moves of transitions between groups. The value of $\sum_{t \in T} group[t]$ does not change if

two transitions get swapped, since the summation still consists of the same collection of values, but merely given in a different order. As such, the sum of assigned group can only be influenced by moving transitions from one group to another. Hence, suppose that a transition t in G is moved from slot i to slot j. If $i < j$, the sum value will increase, since $group[t] = i$ is smaller than $group[t] = j$. Oppositely, if $j < i$ holds, the value decrease. The grouping G is optimal, and hence, a direct move to slot $j < i$ will result in an invalid solution, due to the fact that one or more of the constraints set upon the model would be violated, i.e., t would have been assigned to group slot j in G by the model if the inclusion thereof would have been valid, since it would result in a lower sum value. Alternatively, one may move a transition s out of group j to group k prior to moving t from group i to group j. If $k < i$, it follows that the attained result has a sum of assigned groups that is lower than that of G: this is a contradiction, since G is defined to be optimal under the set constraints, and as such, it can only be concluded that suggested move violates said constraints. For $k = i$, the two move operations are behaviorally equivalent to a swap, and hence, the sum value remains unchanged. Finally, for $i > k$, it follows that the sum will increase, which, like the preceding case $i < j$, leads to a less optimal solution.

As demonstrated above, any valid solution G' originating from G through moves and swaps will, in the best case, result in the same sum of assigned groups. Otherwise, the sum value is bound to increase. Moreover, observe that the size of G' can only be increased if a transition t is moved from a group slot $0 \leq i < |G|$ to a slot $|G| \leq j < |T|$. Given that $j > i$, it follows that the sum value increases. As such, the minimization constraint given in Equation 8 will ensure that additional groups are introduced if and only if the set constraints cannot be satisfied otherwise, and thus, it can be concluded that the size of G is minimized by the given optimization constraint.

4.5 Extracting Deterministic Nodes

The deterministic decision node extraction procedure converts a collection of transitions T to a deterministic decision node $\langle \textsc{Det} : G \rangle$ with the options G, with the latter being a mixture of non-deterministic nodes and transitions that the decision is made over. Oppositely to the non-deterministic variant, a deterministic decision node chooses between options in G based entirely on the current state of variables, i.e., no randomness is involved in the selection procedure. Therefore, it is required that a relation exists between all transition pairs $s, t \in T$ that can give an unambiguous indication on the active state of transition t based solely on that of transition s or vice versa: all of the given overlap constraint variants given in Section 4.4.4 utilize relations that meet the aforementioned requirement, and as such, the groups $T' \in G$ selected by the SMT solutions are deemed valid inputs for the extraction procedure. Furthermore, all transitions in T need to be of the same priority, since otherwise, the priority set by the developer cannot be enforced.

Algorithm 7: CreateDeterministicNode(T)

```
1 R ← The reachability matrix of the and relation between transition pairs (s,t) ∈ T × T.
2 G_transitions ← Group transitions in T based on the clusters within the reachability matrix R.
3 G_nodes ← ∅
4 foreach T' ∈ G_transitions do
5     if |T'| ≤ 1 then
6         G_nodes ← G_nodes ∪ T'
7     else
8         G_nodes ← G_nodes ∪ {CreateNonDeterministicNode(T')}
9 return ⟨Det : G_nodes⟩
```

The procedure CreateDeterministicNode(T), given in Algorithm 7, is used to construct deterministic decision nodes. The procedure starts by creating a reachability matrix R, in which transition $s \in T$ is reachable by a transition $t \in T$ if and only if there exists a path from s to

t in the graph of solution space intersection relations (line 1). Afterwards, the transitions in T are grouped by the clusters within the reachability matrix R (line 2). Observe that two elements $s, t \in T$ within different clusters cannot have overlapping solution spaces, due to the fact that the reachability matrix R is over the intersection relation. Therefore, at any time, only a single group $g \in G_{\text{transitions}}$ can contain elements that are active, and as such, it will not occur that an arbitrary choice needs to be made between two or more active groups. Consequently, it is guaranteed that the generated decision node can choose a target group through a process that is completely deterministic. Next, G_{nodes} is defined to be an empty set (line 3), after which each transition group $T' \in G_{\text{transitions}}$ is processed further based on the size of T' (lines 4-8). The elements in T' are added directly to G_{nodes} if T' contains at most one transition (line 6). Otherwise, the procedure CreateNonDeterministicNode(T) is called to convert the transitions T' to a nested non-deterministic group instead, which in turn is added to G_{nodes} (line 8). As such, it follows that the decision structure construction procedure is recursive. Note that the use of a recursive approach is imperative, since it enables the procedure to create decision structures that consists of more than two decision node levels in depth: the equality and subset-based overlap constraints have been envisioned with multiple nested levels in mind, and hence, the recursive call is required to attain the desired functionality and performance benefits.

Proof of Correctness The described process is recursive, and therefore, care must be taken to ensure that the algorithm will terminate eventually. First, observe that the transitions in T are subjected to the requirements set by the overlap constraints discussed in Section 4.4.4. If the basic constraint is used, no overlaps are allowed in T: hence, it follows that each transition in R will end up in its own cluster, and as such, no recursion is performed since $|T'| \leq 1$ holds for all $T' \in G_{\text{transitions}}$. Oppositely, overlaps can occur when using the equality and subset-based overlap constraints, and consequently, there could exist $T' \in G_{\text{transitions}}$ for which $|T'| > 1$, and therefore, a recursive call may need to be made to Algorithm 7. The transitions T' will always end up in the same cluster, and hence, infinite recursion will occur if and only if procedure CreateNonDeterministicNode(T) is incapable of reducing the size of T'.

Recall that, in Algorithm 2, the procedure starts by gathering all transitions $T_{\text{conflicting}}$ that have overlapping solution spaces with all other transitions in T and that said transitions are excluded from the recursive call. As such, for T' to remain unaltered, it must follow that there do not exist any transitions $t \in T'$ that meet the requirements to be moved to $T_{\text{conflicting}}$. The transitions in T' are a cluster within the reachability matrix R, and as such, T' contains at least $|T'| - 1$ transition pairs that have guard statements with intersecting solution spaces. Given that T' are part of the same group, it follows that said intersections satisfy the requirements set by the overlap constraints. Hence, for the equality constraint, it must follow that all transitions T' are equal, which implies that all T' are moved to $T_{\text{conflicting}}$, since all $t \in T'$ overlap due to being equal. As such, T' is empty, and no further recursion occurs. On the other hand, the subset constraint requires that all intersecting transition pairs $s, t \in T'$ share a subset relation, i.e., the solution space of s needs to be contained fully within that of t or vice versa. Therefore, there exists at least one transition $u \in T'$ of which the solution space contains those of all transitions in T', since the constraint cannot be satisfied otherwise. Given that u overlaps with all other transitions in T', it follows that $T_{\text{conflicting}}$ is guaranteed to be non-empty, and as such, the size of T' is bound to decrease. Consequently, it can be concluded that the recursion within the proposed procedure has a finite depth, and as such, the algorithm is guaranteed to terminate.

4.6 Simplification

The procedure CreateNonDeterministicNode(T) given in Algorithm 7 creates valid decision structures, but in certain situations, the structure ends up to be more complex than necessary. As such, several simplifications can be made to the decision structure d, without affecting the functionality and validity thereof. For one, d could contain superfluous decision nodes that only

have one single option to choose from, and consequently, d may contain subtrees which can be flattened to a single decision node while still retaining the same target behavior.

Algorithm 8: SIMPLIFY($\langle t : G \rangle$)

```
1  G_new ← ∅
2  foreach o ∈ G do
3      if o is a decision node then
4          ⟨t' : G'⟩ ← SIMPLIFY(o)
5          if |G'| = 1 ∨ t = t' then
6              G_new ← G_new ∪ G'
7          else
8              G_new ← G_new ∪ {⟨t' : G'⟩}
9      else
10         G_new ← G_new ∪ {o}
11 return ⟨t : G_new⟩
```

The simplification procedure, given in Algorithm 8, processes the decision structure and performs the desired structural optimizations. Procedure SIMPLIFY($\langle t : G \rangle$) takes a decision node $\langle t : G \rangle$ as the input, and returns a simplified version of the node. The algorithm starts by defining G_{new} (line 1), which will contain the simplifications of all decisions in G. The procedure proceeds by simplifying all of the options $o \in G$ (lines 2-10) based on the type of o. If o is a decision node, then the procedure starts by recursively calling the simplification procedure on node o (line 4), yielding the decision node $\langle t' : G' \rangle$. Subsequently, it is checked whether the simplification of o is superfluous (line 5): a decision node is considered superfluous if it contains only one option, or if the child node is of the same type as the parent node. If the node is superfluous, the options G' are added directly to G_{new} (line 6). Otherwise, the simplification of o is added to G_{new} (line 8). If o is a transition, o is added back to G_{new}, since transitions cannot be simplified (line 10).

In Section 4.3, it has been specified that the top level of the generated decision structure needs to be a non-deterministic node such that simplifications can be made to the implementation of the code generator. The aforementioned requirement is fulfilled by the procedure SIMPLIFY($\langle t : G \rangle$), since the proposed algorithm always returns a node that is of the same type as the input node. Therefore, it is guaranteed that the type of the non-deterministic root node remains unaltered, regardless of the number of decisions contained therein.

Proof of Correctness For the SIMPLIFY($\langle t : G \rangle$) procedure to be correct, it needs to follow that the simplified decision structure is behaviorally analogous to the original. First, observe that in every branch of the algorithm, all of the decisions $o \in G$, or the simplifications thereof, are added back to G_{new}. Thus, it follows that the behavior of every decision $o \in G$ is added to the simplified instance of the target decision node. As such, it needs to be shown that the removal of a superfluous node does not change the depicted behavior. To start with, the case $|G'| = 1$ will be considered, i.e. the child node contains only a single option. The validity of this simplification is considered trivial, since a decision over one choice will inevitably lead to said choice being selected and evaluated: therefore, the sole element in G' can be added to G_{new} without a change of behavior. Next, it needs to be demonstrated that two nodes of the same type can be merged by adding the options of the child node $\langle t' : G' \rangle$ to the simplified parent node $\langle t : G_{\text{new}} \rangle$. By definition, the decisions within a deterministic decision node have non-overlapping solution spaces. Given that $\langle t' : G' \rangle \in G$, it must hence follow that the decisions in G' do not have intersecting solution spaces with G, and as such, all options G' can be added to G_{new} without violating the semantics of a deterministic decision node. Oppositely, non-deterministic decision nodes do not require the options to have overlapping solution spaces–their presence is merely preferred. Therefore, G' can

be added to G_{new} without violating the semantics of the non-deterministic decision node. However, due to the merge operation, it will hold that $|G| \leq |G_{\text{new}}|$, and as such, the probability distribution of the decision structure is not equivalent to that of the original. Nevertheless, the structure is considered behaviorally analogous to the original, since the SLCO framework merely requires that an arbitrary transition is picked out of the pool of active transitions.

4.7 Available Solver Configurations

In the course of this chapter, several options have been introduced for the implementation of the grouping approach and overlap constraints. The desired combination can be passed on to the code generator through the `-decision_structure_solver_id=?` argument, to be described further in Section 6.3. The question mark needs to be replaced with the identity of the target solver, with the identity of each solver being given in Table 4.1. If no argument is given, the code generator will default to a greedy grouping approach in combination with the equality overlap constraint.

Id	Approach	Constraint
0	Greedy	Basic
1	Greedy	Equality
2	Greedy	Subset
3	Optimized	Basic
4	Optimized	Equality
5	Optimized	Subset

Table 4.1: The implemented solver techniques, including their respective ids. The available grouping approaches and overlap constraints are described in Section 4.4.1 and 4.4.5 respectively.

4.8 Concrete Example

To conclude with, a concrete example will be discussed that demonstrates the difference between the options provided by the decision structure construction algorithm. The `Nesting` model, given in Listing 4.1, depicts a program consisting of transitions of which the guard statements are related through a hierarchical structure, i.e., the majority of the guard statements are related though a subset relation. Additionally, several transitions have been duplicated, such that the model contains guard statements that occur more than once. All of the transitions have the same priority of zero, except for the last transition, which has a priority of two.

The decision structures generated through each solver configuration listed in Table 4.1 are given in Figure 4.2. The decision structures are depicted as a tree of objects, where the nodes are represented by their type and the transitions by their priority and guard statement. Note that the chosen representation for the transitions does not allow for a specific transition to be identified if multiple transitions share the same guard statement and equal priority: the latter is not considered an issue, since transitions with the same guard statement can be interchanged safely, as long as they have the same priority and each guard is associated with exactly one transition.

First, observe that the results achieved by the greedy and optimized grouping approaches are alike. The results of the basic overlap constraint are equivalent in structure, with each decision node in the result of one approach having a partner with the same type and number of decisions in the other. On top of that, the resulting decision structures are completely equivalent for the equality and subset overlap constraints. Furthermore, it is clear that all of the given decision trees are structurally sound: for each deterministic structure, it holds that the options contained therein do not contain overlapping solution spaces.

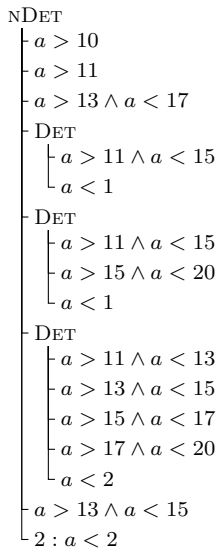
In addition, the overlap constraints are behaving as expected. For the basic overlap constraint,

```
model Nesting {
  classes
  P {
    state machines
    SM1 {
      variables Integer a
      initial SMC0
      transitions
      from SMC0 to SMC0 { a > 10 }
      from SMC0 to SMC0 { a > 11 }
      from SMC0 to SMC0 { a > 13 && a < 17 }
      from SMC0 to SMC0 { a > 11 && a < 15 }
      from SMC0 to SMC0 { a > 11 && a < 15 }
      from SMC0 to SMC0 { a > 11 && a < 13 }
      from SMC0 to SMC0 { a > 13 && a < 15 }
      from SMC0 to SMC0 { a > 13 && a < 15 }
      from SMC0 to SMC0 { a > 15 && a < 20 }
      from SMC0 to SMC0 { a > 15 && a < 17 }
      from SMC0 to SMC0 { a > 17 && a < 20 }
      from SMC0 to SMC0 { a < 1 }
      from SMC0 to SMC0 { a < 1 }
      from SMC0 to SMC0 { a < 2 }
      2: from SMC0 to SMC0 { a < 2 }
    }
  }
  objects p: P()
}
```

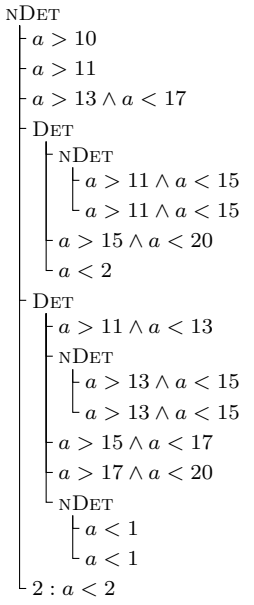
Listing 4.1: An SLCO model consisting of transitions that follow a hierarchical decision structure.

the results thereof given in Figures 4.2a and 4.2d, decision structures with two levels of decision nodes are generated: the root node is a non-deterministic decision, and all of its child nodes are deterministic decisions. All of the transitions contained within the deterministic decisions have non-overlapping solution spaces, which coincides with the intended behavior of the constraint. The decision structures that are the result of the equality-based overlap constraints, as listed in Figures 4.2b and 4.2e respectively, contain three decision node levels, where each level alternates in type. The third level of the structure consists exclusively of transitions and non-deterministic decision nodes, with the latter only containing transitions in which the guard statements are equivalent. Therefore, it holds that transitions with overlapping solution spaces may be part of the same deterministic group if and only if their solution spaces are equivalent. The decision structures generated using the subset-based overlap constraints, given in Figures 4.2c and 4.2f, show that the algorithm does not have a set depth limit: instead, as many levels are created as required. Moreover, for all deterministic decision nodes within the resulting decision structure, it holds that the solution spaces of the transitions contained therein share a subset relation, and hence, the subset-based overlap constraint are satisfied. Lastly, observe that the priorities of transitions are handled appropriately by all of the configurations: for all deterministic groups, it holds that the transitions contained therein all are of the same priority.

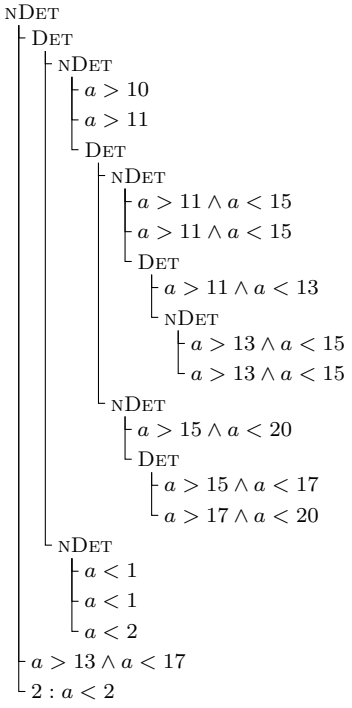
The decision structures generated by the three overlap constraint types show a different degree of deterministic behavior. For the basic overlap constraint, the root node contains a non-deterministic decision node with eight options, with the non-deterministic root of the equality and subset-based approaches having merely six and three options to choose from respectively. Consequently, the likelihood of picking an active transition is higher for the latter constraints, since it is more probable that a branch will be picked that can resolve the decision (semi-)deterministically instead of through an arbitrary choice. Nevertheless, the additional decision levels may introduce a performance overhead, and as such, the available configurations will need to be tested in practice.



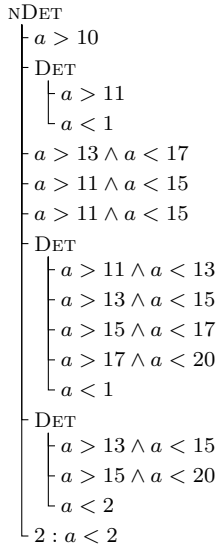
(a) Greedy + Basic (0)



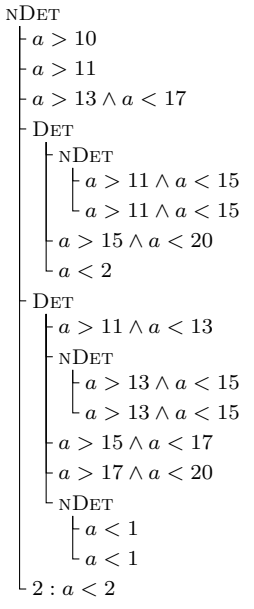
(b) Greedy + Equality (1)



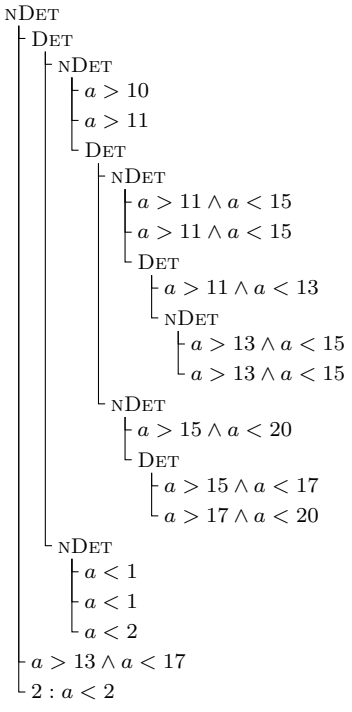
(c) Greedy + Subset (2)



(d) Optimized + Basic (3)



(e) Optimized + Equality (4)



(f) Optimized + Subset (5)

Figure 4.2: Visualizations of the decision structures constructed for the `Nesting` model defined in Listing 4.1, where each subfigure holds the decision structure that is the result of the associated grouping approach and overlap constraint combination.

Chapter 5

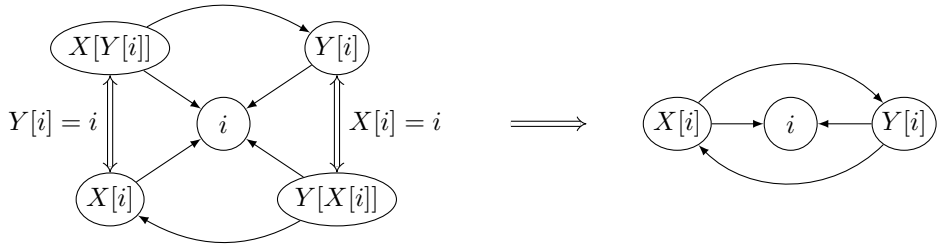
Statement Atomicity

The SLCO framework focuses heavily upon the concept of automatically generating code of concurrent programs, which is achieved by letting the state machines defined within the SLCO model run in parallel. These state machines have access to two types of variables: local variables that are part of the state machine themselves, and class variables that are part of the parent class. As the name suggests, local variables can only be accessed by the state machine in question, while the class variables can be accessed by all state machines within the same class. Hence, class variables are considered to be a shared resource. By design, SLCO has defined statements to be atomic in nature, since without atomicity, it is possible for race conditions to occur within the code, which may in turn result in incorrect or undesirable behavior. The atomicity of statements is not guaranteed in Java and as such, additional mechanisms need to be introduced to enforce it. In Java, this is achieved by locking the variables that are part of the statements, to ensure that only a single thread has read and write access to the variable at any time. Note that only the class variables need to be locked to achieve atomicity, since by construction, it is ensured that the local variables can only be accessed by the thread running the state machine itself. Moreover, it needs to be ensured that the locking mechanism does not affect the liveliness of the code by preventing the occurrence of lock-deadlocks. In the implementation, a general and well-known technique will be used to ensure that lock-deadlocks cannot occur, namely by enforcing a fixed ordering of locks. Section 5.2 will go into further detail on the concept of (lock-)deadlocks and the chosen technique. The locking mechanism included within the original implementation of the code generator does adhere to the fixed lock ordering requirement by assigning every variable–including all of its positions if it is an array type variable–an unique numeric identity following an incremental order. The predetermined locking order is then enforced by sorting the lock requests prior to the actual acquisition of the locks. Nevertheless, the current implementation is not perfect: several situations exist in which the current implementation cannot guarantee the atomicity of statements. Hence, it is the focus of this chapter to suggest and implement improvements to the locking system to ensure the atomicity of statements in all situations.

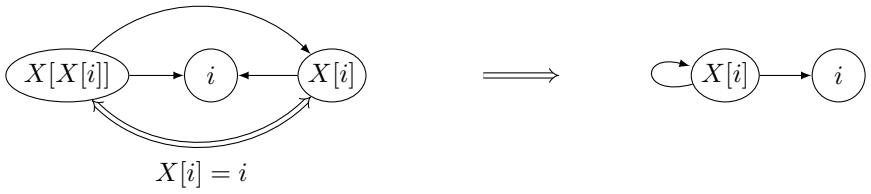
The first issue identified in the original implementation is the improper treatment of inter-lock-dependencies within the locking mechanism, where an inter-lock-dependency is defined as a relation between two class variables. An array type class variable X has an inter-lock-dependency with class variable i if the latter is used to determine which instance of X needs to be locked, i.e., i is used as or is part of the index $X[i]$. Inter-lock-dependencies add an additional layer of complexity to the locking mechanism, and if handled incorrectly, they may potentially lead to deadlocks, even in situations where the strict ordering is enforced correctly. The problem at hand will be clarified through a simple example. Suppose that the statement $X[i]$ needs to be made atomic, and let X and i be variables that are part of the same class. First, it is important to observe that the values of i and $X[i]$ both remain volatile until the locks have been successfully acquired, i.e., the values of both i and $X[i]$ can still be changed by other threads even during the locking process

itself. Given that i remains volatile, $X[i]$ cannot be locked reliably unless i is locked prior to the locking request of $X[i]$. Take for example a situation where the value of i is changed to i'. If i is not locked before requesting $X[i]$, it is possible that $X[i]$ will be requested erroneously, while $X[i']$ should be requested instead for proper atomicity. Moreover, exceptions and lock-deadlocks may occur due to the fact that there is a potential mismatch between the identity of the locks that need to be requested and released, which is obviously undesirable. Hence, the concept of phased locking will be introduced to handle inter-lock-dependencies appropriately.

The presence of inter-lock-dependencies and the need for phased locking introduce additional challenges that need to be addressed. As mentioned before, locks need to be acquired in a fixed order to ensure that lock-deadlocks cannot occur, and as such, inter-lock-dependencies will need to adhere to this paradigm as well. Therefore, it is required that the index within an inter-lock-dependency is a predecessor in the lock ordering. However, due to this requirement, it is not always possible to have such an ordering, since it is possible that circular dependencies exist within the inter-lock-dependencies that cause conflicts. Take for example the statement $X[Y[i]] = Y[X[i]]$, using the class variables X, Y and i, and as such, the inter-lock-dependencies present within the statement are $X[Y[i]] \to Y[i]$, $X[Y[i]] \to i$, $Y[i] \to i$, $Y[X[i]] \to X[i]$, $Y[X[i]] \to i$ and $X[i] \to i$. Thus, it is required that $Y[i]$ is locked before $X[Y[i]]$, and $X[i]$ is locked before $Y[X[i]]$. Next, consider the situation where both $X[i]$ and $Y[i]$ have the value i. When substituting these values, it is now the requirement that $Y[i]$ is locked before $X[i]$, and $X[i]$ is locked before $Y[i]$. However, this situation results in a contradiction: only one of the requirements can hold at any time, and hence, a strict ordering is not attainable. For convenience, the described situation has been sketched in Figure 5.1a, where the inter-lock-dependencies are depicted as a directed graph. On the left-hand side, the original graph is shown, containing all of the dependencies listed above. The right hand-side contains a reduced graph in which both the values $X[i]$ and $Y[i]$ have been substituted with i, resulting in a circular dependency between $X[i]$ and $Y[i]$. Moreover, the same issue will occur when the statement contains a variable that has a dependency with itself. Figure 5.1b depicts the statement $X[X[i]]$ having the class variables X and i, and inter-lock-dependencies $X[X[i]] \to X[i]$, $X[X[i]] \to i$ and $X[i] \to i$, resulting in a self-dependency. If $X[i]$ has the value i, it follows that $X[i]$ needs to be locked before $X[i]$, which is impossible.



(a) A graph for statement $X[Y[i]] + Y[X[i]]$ with the self-loop $X[i] \leftrightarrow Y[i]$ in the reduction.



(b) A graph for statement $X[X[i]]$ with the self-loop $X[i] \leftrightarrow X[i]$ in the reduction.

Figure 5.1: Graph visualizations of the inter-lock-dependencies for two statements, wherein the class variables are depicted by the nodes and the corresponding inter-lock-dependencies are represented by the edges. In the figure, the left-hand side contains the full graph, with the right-hand side showing a reduced graph that is the product of the given value substitutions.

This aforementioned problem is exacerbated by the fact that the identities used to attain the strict ordering of locks are allocated as continuous blocks. Each variable is assigned an unique identity, which is then offset by the given index to lock the appropriate position within the array. Because of this fact, the presence of a circular dependency between two distinct variables will always violate one of the two lock ordering requirements, since by construction, one of the variables will always need to be locked before the other, no matter the target position. This implies that the target indices of inter-lock-dependencies are superfluous, and hence, the dependency graphs shown in Figure 5.1 can be simplified further by obfuscating the indices within the nodes, which results in the situation sketched in Figure 5.2. Moreover, due to this crucial detail, it also follows that circular dependencies will always violate the lock ordering. Thus, a robust mechanism needs to be created that is capable of resolving circular dependencies.

(a) $X[Y[i]] + Y[X[i]]$ (b) $X[X[i]]$

Figure 5.2: Graph visualizations of inter-lock-dependencies for several statements. In this figure, the indices that accompany the variables have been obfuscated due to them being superfluous.

Furthermore, the locking mechanism does not possess the means to deal with short-circuit evaluations adequately. Short-circuit evaluation denotes the semantics of propositional connectives in which the second argument is evaluated only if the first argument does not suffice to determine the value of the expression [4]. Suppose that X is a class variable and that $|X| = 10$. In addition, consider the expression $i \geq 0 \wedge i < 10 \wedge X[i]$, in which the first two terms of the conjunction verify whether i is within the bounds of X. Within Java, conjunctions and disjunctions are defined to be short-circuiting operators, and hence, $X[i]$ will only be evaluated within the generated code if the bound check on variable i succeeds. Consequently, the locking mechanism needs to adhere to the same principles when determining which locks to request, since otherwise, unexpected behavior such as out of index exceptions may occur for statements that are otherwise completely valid, which is undesirable. Unfortunately, within the original implementation, the occurrence of short-circuit evaluations is not considered when generating the locks to be requested, and hence, it cannot be guaranteed that the generated code is exception free, even if the model is sound. Therefore, improvements need to be made to the locking mechanism, such that it can be ensured that the locking mechanism will not trigger exceptions that would otherwise not have occurred.

Nevertheless, the proper treatment of inter-lock-dependencies and short-circuit evaluations is only part of the challenge. The locking system has a major impact upon the performance of the generated code, and as such, the assignment of lock identities used to enforce the strict ordering of locking needs to be handled with great care. The assigned lock identities need to be made compatible with inter-lock-dependencies and phased locking, while also ensuring that the overall impact of resolving circular references and short-circuit evaluations remains minimal. Moreover, other exceptional situations exist that may inadvertently cause lock-deadlocks. Hence, in this chapter, the focus will be on introducing several techniques that tackle and resolve these challenges.

5.1 Problem Statement and Approach

Several issues have been identified in the original implementation of the locking mechanism that need to be addressed. Within this chapter, an improved locking mechanism will be introduced that aims to tackle all of the described problems in a coherent and structural manner. To achieve this, a locking data structure will be created that facilitates the creation of a strict lock ordering, while also

simplifying the resolution of inter-lock-dependencies and short-circuit evaluations. Furthermore, a collection of procedures will be provided that can be used to resolve circular dependencies and issues pertaining to short-circuit evaluations. Analogously to the original implementation, lock-deadlock freedom will be attained by adhering to a strict lock ordering, where each variable and index combination is assigned an unique identity that defines the ordering.

The overall structure of the chapter will be as follows. First, a brief introduction of the concept of deadlock freedom will be given in Section 5.2, including a more detailed discussion on how lock-deadlocks may occur and how they can be avoided in practice. An introduction to the locking data structure will be given in Section 5.3, followed by an elaboration of the initialization process of said structure in Section 5.4. Next, the concept of lock propagation will be discussed in Section 5.5, which will relocate the acquisition and release of the target locks to attain the desired scope of atomicity. The process that assigns lock identities to class variables is introduced in Section 5.6. The description of the solution is concluded by Section 5.7, which describes the finalization of the locking data structure. Finally, several optimization will be discussed in Section 5.8 that can be or have been made to the proposed solution.

5.2 (Lock-)Deadlock Freedom

The concept of deadlock freedom is an important aspect for any concurrent program, since the potential presence of a deadlock heavily affects both the effectiveness and overall quality of the written code. Hence, it is important that the automatically generated code of the SLCO framework remains deadlock free. A deadlock is defined to be a state in which several components of a program are waiting indefinitely for each other to proceed. Deadlocks can occur due to many reasons, but in this chapter, the focus will be fully upon a subgroup of deadlocks aptly called lock-deadlocks. A lock-deadlock occurs when two or more threads all require resources that the other threads have already acquired. The concept of a lock-deadlock can be demonstrated through a simple example. Suppose that the program contains two threads A and B and let $\mathcal{L}_{aq}(A)$ and $\mathcal{L}_{aq}(B)$ be the locks acquired by threads A and B respectively. Moreover, let $\mathcal{L}(A)$ and $\mathcal{L}(B)$ be the locks requested by threads A and B. The program enters a lock-deadlock state when $\mathcal{L}(A) \cap \mathcal{L}_{aq}(B)$ and $\mathcal{L}(B) \cap \mathcal{L}_{aq}(A)$ are both non-empty simultaneously.

Fortunately, lock-deadlocks can be prevented effectively by simply ensuring that the locks for the class variables are always acquired through the same strict ordering [18]. In the implementation, the strict ordering of lock acquisitions is achieved by first assigning unique lock identities to all the class variables within the class, including all of the positions within the arrays. Afterwards, the locking requests can be sorted on the associated lock identities at runtime, which will enforce the strict lock ordering. Hence, lock-deadlocks will not occur.

5.3 Locking Data Structure

For the locking mechanism to be robust, a locking data structure needs to be created that facilitates the creation of a strict lock ordering, while also simplifying the resolution of inter-lock-dependencies and short-circuit evaluations. The locking mechanism aims to maximize the concurrency between threads, and as such, it would be beneficial to acquire or release a lock upon a variable as close to its use within a statement as possible, under the condition that statement atomicity is ensured. Through the encapsulation of a statement by a method, locks can be acquired or released at virtually any position within the model's syntax tree, and as such, the proposed approach is viable. The localized acquisition of locks is especially important when short-circuit evaluations are performed, since failing to do so may cause exceptions to occur, even if the statement is technically sound. Therefore, the locking data structure should strive to follow the control flow of the given model, such that locks will only be acquired if and only if the code has reached a branch in which the conditions set upon accessing the target variable are met.

The envisioned locking data structure consists of two interconnected structures–a atomic node tree, which adheres to the syntax tree of the model, and a locking node graph, which follows the control flow of the to-be-generated code. The atomic nodes within atomic node tree provide the code positions at which locks can be acquired and released, and function as the building blocks of the data structure, while the locking nodes within the locking node graph dictate the locks that should be acquired and/or released at a given position. For more information on atomic node trees and locking node graphs, consult Sections 5.3.1 and 5.3.2 respectively.

5.3.1 Atomic Node Tree

Within the locking structure, the atomic node tree acts as the top level of the structure. Each atomic node is associated with exactly one object within the model's syntax tree, and dictates the lock actions that have to be executed upon the entering and exiting of the associated statement. The opposite relation does not hold true, i.e., it does not hold that every object within the syntax tree is associated with an atomic node: therefore, the presence of an atomic node is optional. Each atomic node maintains a list of child nodes that are considered to be sub-statements of the statement associated with the atomic node in question. Each atomic node is associated with three locking nodes: one entry node that is placed at the entry point of the statement, and two exit points that are located at the statement's success and failure exit points. Directed edges are added between the locking nodes of the atomic node and those of its direct children to model the control flow of the program. Circular references in the control flow are not allowed. Given the child-parent relation described earlier, it hence follows that the atomic node tree follows the same structure as the model's syntax tree, and therefore, the model's control flow will be respected as long as the entry and exit locking nodes contained by the atomic nodes are connected in the appropriate order. The atomic node tree and its underlying locking node graph are constructed through the application of a collection of patterns which will be discussed in Section 5.4.1.

5.3.2 Locking Node Graph

The locking node graph dictates which locks should be acquired or released at a given point in the program. The locking node graph is a directed acyclic graph (DAG) and the relations therein adhere to the control flow of the model and generated program. Each locking node within the graph is associated to exactly one atomic node and contain references to the lock objects that need to be acquired or released at the locking node's position within the atomic node's partner statement. Lock objects will be discussed further in Section 5.3.2.1. Additionally, each locking node is associated with a locking instruction object, which contains the finalized locking targets that need to be acquired and released by the locking node. The locking instruction also provides instructions on how to resolve locks that are in conflict with inter-lock-dependencies and/or short-circuit evaluations. Consult Section 5.3.2.2 for more information on locking instruction objects, and Section 5.7.2 for the process through which the object is constructed.

5.3.2.1 Locks

Class variables are a shared resource between threads, and as such, they need to be locked before use. The purpose of lock objects is to target and encapsulate said class variable references, such that they can be incorporated into the locking structure. Each lock object contains a variable reference that references to the target class variable instance and rewrite rules can be added to a lock such that the effect of assignments can be applied to the index of the target variable. Furthermore, locks keep a reference to the locking node in which they have originally been acquired, such that the original location of a lock in the locking node graph can always be recovered. The latter is necessary, since certain locks are location sensitive, i.e., an exception may occur if the lock is requested at another position in the control flow, which may occur when the lock is requested before a statement that performs a bound check on one or more variables used within the index of the atomic node's target statement. Location sensitive locks are marked, such that they can easily

be identified, and the lock object maintains a list of bound checked locks that it relies upon. Lastly, a lock may be marked as dirty if it violates the strict ordering or location requirements.

5.3.2.2 Locking Instructions

The locking instructions contain the finalized lock acquisition and release sets, and as such, the data contained within the locking instructions will be used during the rendering of the code, instead of the lock acquisitions and releases contained within the locking node. Furthermore, the locks within the locking instructions are represented by lock requests instead of lock objects. Similarly to lock objects, lock requests contain a reference to the class variable instance to be targeted. However, the difference between the two objects is that lock requests over the same class variable instance, i.e., same variable name and index combination, will yield the same lock request instance, and as such, all lock requests over the same target object are represented by the same object instance. Note that this is not possible for locks, since lock objects are not static–the target variable instance may change depending on the rewrite rules. As such, lock requests can only be generated once the acquisition position of a lock is guaranteed to no longer change.

Furthermore, the structure of the locking instruction provides the additional attributes that are required to render the lock management routine for a given locking node. For one, the locking instruction has a field that contains a list of locking phases that are required to acquire a set of locks without violating the requirements set by inter-lock-dependencies. The concept of phased locking will be discussed in Section 5.7.2.2. In addition, the locking instruction maintains a list of order-violating lock requests that are required to perform the weak unpacking operation, to be introduced in Section 5.7.2.1 of this chapter.

5.4 Initializing the Locking Structure

Next, the steps required for the initialization of the locking data structure will be introduced. The initialization process will create the earlier described atomic node tree and the locking node graph that form the backbone of the locking mechanism. In the scope of this section, the focus will be on the initial construction of the locking structure, and not the processing thereof to attain a valid structure: instead, the operations required to make the structure valid in terms of atomicity and ordering requirements will be discussed in Section 5.5 and 5.7.

The structure of this section will be as follows. First, the locking structure will be initialized through a collection of patterns that represent the building blocks of the atomic node tree and locking node graph. Said patterns are introduced in Section 5.4.1. Once the locking structure has been constructed, all locking nodes contained therein will subsequently receive the appropriate locks in their acquisition and release lists. For experimentation purposes, several implementations are provided for this process. The introduction of locks to the structure and the available approaches will be discussed in Sections 5.4.2 and 5.4.2.1 respectively. Finally, the necessary flags and markings are added to structure, such that issues pertaining to short-circuit evaluations can be tracked: the process thereof will be discussed in Section 5.4.3.

5.4.1 Atomic Node Structural Patterns

The control flow of each object within the Slco framework differs per instance, and as such, the pattern to be used depends upon the object in question and configuration thereof. Patterns will be provided for (Boolean) expressions, assignments, composites, transitions and decision nodes within their own respective sections. The primary purpose of the locking structure is to resolve the challenges introduced by short-circuit evaluations, and therefore, it has been decided that the patterns should focus solely on modelling the control flow behavior that pertains to short-circuit evaluations. Consequently, the locking structure might be incomplete in terms of the control flow of the object in question, which can be advantageous: short-circuit evaluations cannot occur in

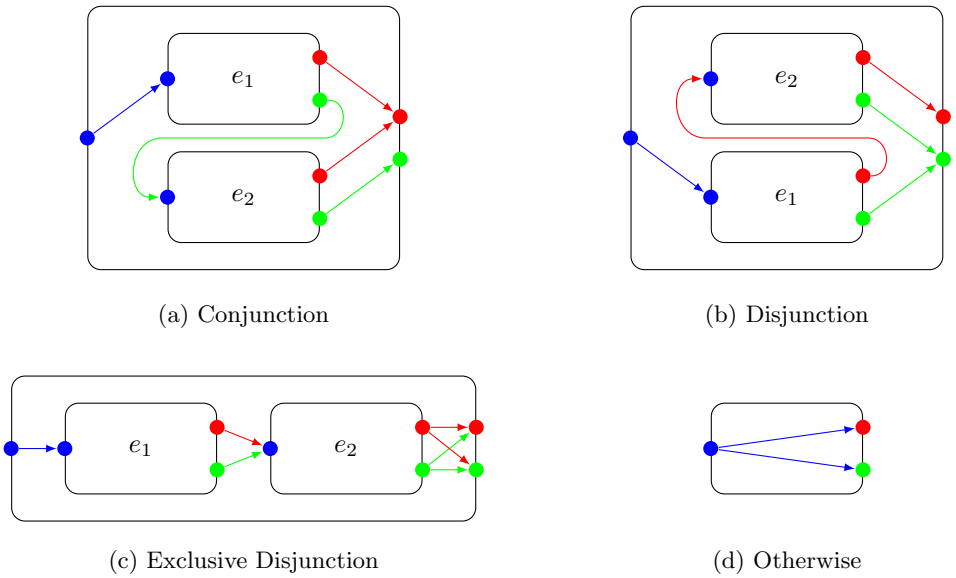
the excluded parts of the control flow, and hence, locks can be requested for an entire expression instead of individual sub-expressions, which is more efficient due to the potential performance overhead the locking structure may introduce.

Within the following sections, the atomic node patterns will be visualized for each type of object. The atomic nodes within the figures are represented by rectangular shapes and the locking nodes associated with an atomic node are represented by nodes upon the latter's border. The visualization of an atomic node may contain a reference to the object it belongs to. The child-parent relationship of atomic nodes is depicted through the enclosure of a set of child atomic nodes by the parent node. At several occasions, atomic nodes are drawn that have a dashed outline, which means one of the following. If the atomic node is a named node, it follows that the node is optional, where the inclusion thereof depends upon the existence of object being referenced to. If the atomic node contains dots, it is instead implied that the pattern is repeated a finite number of times for the objects within a given range. The locking nodes are color coded depending on their location in the control flow: the entry node of an atomic node is blue, and the success and failure exit nodes are green and red. The entry node is always drawn at the left side of the atomic node, and the failure and success nodes are drawn on the top and bottom right respectively. Lastly, the color of the edges drawn within the patterns are based on the color of its source for clarity.

5.4.1.1 (Boolean) Expressions

The characterization of the control flow of a (Boolean) expression depends entirely upon the operator that is being used by the expression. To start with, a differentiation will be made between expression and primary objects, with the former being discussed first. For expression objects, two categories have been identified: operators that contain nested Boolean statements, and operators that cannot. Conjunctions, disjunctions and exclusive disjunctions are part of the former category, with any remaining operators being part of the latter. The former category requires the creation of atomic nodes for all values contained therein, since said Boolean expressions may be subjected to short-circuit evaluations. Oppositely, operators within the latter category cannot short-circuit, and as such, the creation of atomic nodes for its sub-expressions is not necessary. Under certain circumstances, the outcome of a conjunction or disjunction can be predicted prior to its full evaluation, and as such, they may short-circuit. For conjunctions, a short-circuit occurs when one of its members evaluates to false. Similarly, a disjunction short-circuits when one of its members evaluates to true. The outcome of an exclusive disjunction cannot be predicted, and as such, no short-circuits will occur therein. For all types of expressions, the expression on the left-hand side is visited first, followed by the expression on the right-hand side if present.

The atomic node patterns for expression objects are given in Figure 5.3. The pattern for conjunctions, given in Figure 5.3a, connects the entry point of the atomic node to the entry point of the left-hand side expression e_1. The success exit of e_1 is connected to the entry point of the right-hand side expression e_2, with the failure exit short-circuiting to the failure exit of the expression's atomic node. The success and failure exit nodes of expression e_2 are connected to the conjunction's exit nodes respectively. The pattern for disjunctions, given in Figure 5.3b, follows a similar pattern, but with the crucial difference being that the roles of the success and failure exit points have been swapped. Exclusive disjunctions adhere to the pattern given in Figure 5.3c. The pattern connects the atomic node's entry point to the entry point of the left-hand side expression e_1. Subsequently, expression e_1 is chained to the right-hand side expression e_2, with no consideration for the outcome of e_1. Similarly, the exit nodes of e_2 are connected to both of the parent node's exit points, since the outcome of a exclusive disjunction cannot be predicted statically. Finally, Figure 5.3d presents the pattern used by all remaining types of expressions: the entry point of the atomic node is connected to both of the exit nodes, due to the fact that the outcome cannot be predicted without evaluating the expressions contained therein. By construction of the locking graph, it is ensured that atomic nodes are generated only for Boolean expressions, and therefore, connecting both exits will not lead to incorrect behavior.



(a) Conjunction

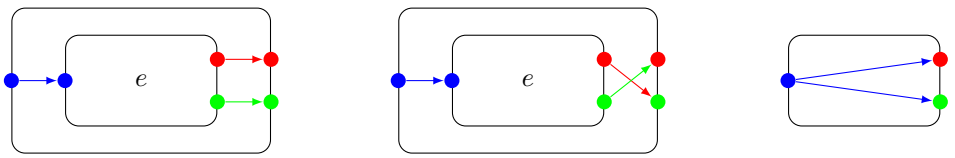
(b) Disjunction

(c) Exclusive Disjunction

(d) Otherwise

Figure 5.3: The patterns used to construct an atomic node for an expression object.

Next, the control flow of primary objects will be discussed. First, it needs to be observed that the control flow of a primary object depends upon its structure. An atomic node needs to be generated for expressions used within the body of a primary statement, since the expression contained therein could be subjected to short-circuit evaluations. Boolean expressions have well defined exit points, and as such, the signs of primaries need to be handled with care: if the sign is a negation, the exit points should be swapped to ensure structural integrity. Primary objects referring to a constant or variable do not require the creation of an atomic node for the target value, since it is impossible for the target value to perform a short-circuit evaluation.



(a) Expression reference

(b) Expression reference (negated)

(c) Otherwise

Figure 5.4: The patterns used to construct an atomic node for a primary object.

The atomic node patterns for primary objects are given in Figure 5.4 and are subdivided into three categories. The first category, of which the pattern is given in Figure 5.4a, targets expressions without negations. The pattern connects the entry point to the entry of the Boolean expression e contained within the primary. By construction of the locking structure, atomic nodes will only be generated for expressions and primaries having a Boolean result, and as such, both exits of e can safely be connected to their respective counterparts in the atomic node of the primary without creating unexpected behavior. The second category covers expressions that are negated by the primary. The pattern given in Figure 5.4b is structurally equivalent to the one given in Figure 5.4a, but with one crucial difference: the connections to the exit node of the parent node have been swapped to account for the negation. Lastly, all remaining cases are covered by the pattern presented in Figure 5.4c, which connects the entry point of the primary to both exit points, without creating an atomic node for the objects contained therein.

5.4.1.2 Assignments

The control flow of an assignment is characterized as follows. Upon the execution of a statement, the left-hand side of the statement, being the target variable, is visited first. Observe that the left-hand side of an assignment cannot be converted to a method, since this would lead to syntax that is invalid in Java. Therefore, no atomic node is generated for the left-hand side of an assignment– instead, the target value is covered by the atomic node of the assignment itself. Subsequently, the expression at the right-hand side of the assignment is visited. The primary purpose of the locking structure is to ensure the proper treatment of short-circuit evaluations, which will only only occur within Boolean expressions. As such, the control flow depends upon the return type of the expression, which is equivalent to the type of the variable that the expression is being assigned to. In the patterns, an atomic node is created for the right-hand expression if and only if the target variable is of a Boolean type. Observe that, on top of that, assignments cannot fail, and hence, only the success exit receives a connection, with the failure exit remaining unconnected.

(a) Boolean target variable

(b) Integer/Byte target variable

Figure 5.5: The patterns used to construct an atomic node for an assignment object.

The atomic node patterns have been visualized in Figure 5.5. The pattern for assignments to Boolean variables, given in Figure 5.5a, connects the entry point of the atomic node to the entry point of the right-hand Boolean expression e. The success of the assignment is not determined by the expression e, and as such, both the exits of the atomic node of e are connected to the success exit of the assignment. The pattern for assignments to integer or byte variables, given in Figure 5.5b, simply connects the entry point of the atomic node to the node's success exit point, without creating an atomic node for the expression at the right-hand side of the assignment.

5.4.1.3 Composites

The control flow of a composite has the following characterization. Statements within a composite have to be considered atomic as a collective, and as such, atomic nodes need to be made for all statements contained therein, and all statements need to be chained together in the defined order to achieve the desired behavior. The assignments within a composite can only be executed if the guard statement hold true, and as such, the guard statement needs to be visited first. If the guard statement fails, the composite should fail. Otherwise, the composite should proceed by sequentially visiting all assignments contained within the composite.

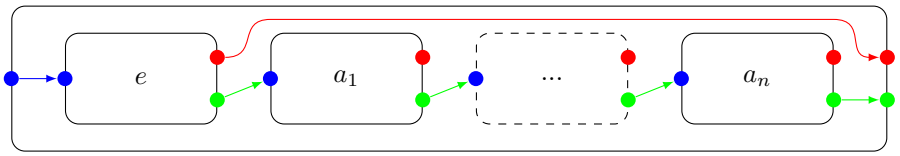


Figure 5.6: The pattern used to construct an atomic node for a composite object.

The atomic node pattern of a composite is presented in Figure 5.6. Let e be the guard statement of the composite, and a_1 to a_n be the assignments contained therein in the defined order. The composite starts with the guard statement, and as such, the entry point of the composite is connected to the entry point of the guard expression. Assignments within the composite are only executed when the guard evaluates to true: as such, the failure exit of the guard statement needs to

be connected to the failure exit of the composite, and the success exit needs to be connected to the entry point of the first assignment. Assignments can only succeed, and as such, only the success exit of a statement is connected to the entry of the next statement in the composite. The success exit of the last statement is connected to the success exit of the composite. Therefore, a composite will fail if the guard expression fails, and succeed once all statements have been executed.

5.4.1.4 Transitions

The control flow of a transition is characterized in the following way. First, it needs to be observed that, in the SLCO language, atomicity is achieved at the statement level: as such, each statement in a transition needs to be atomic. The code generator uses simple SLCO models, and therefore, a transition will at most contain two statements, namely the transition's mandatory guard statement and an optional composite or assignment statement added through the preprocessing of the structure. Consequently, the outcome of a transition is determined by its guard statement, since the secondary statement will always succeed by construction, given that assignments always succeed and a composite will only become a secondary statement if its guard statement is true. The guard statement of a transition is part of the atomic decision structure, and as such, an atomic node needs to be created for the former, which in turn needs to be incorporated into the control flow of the latter. In addition, atomic nodes need to be created for the secondary statements as well, but unlike the guard statements, these are not to be connected to the main structure–instead, they are an isolated atomic entity, since otherwise, the entire transition becomes atomic, and not just the target statement, which is not the desired behavior.

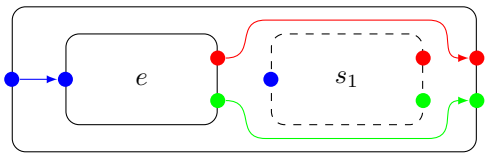


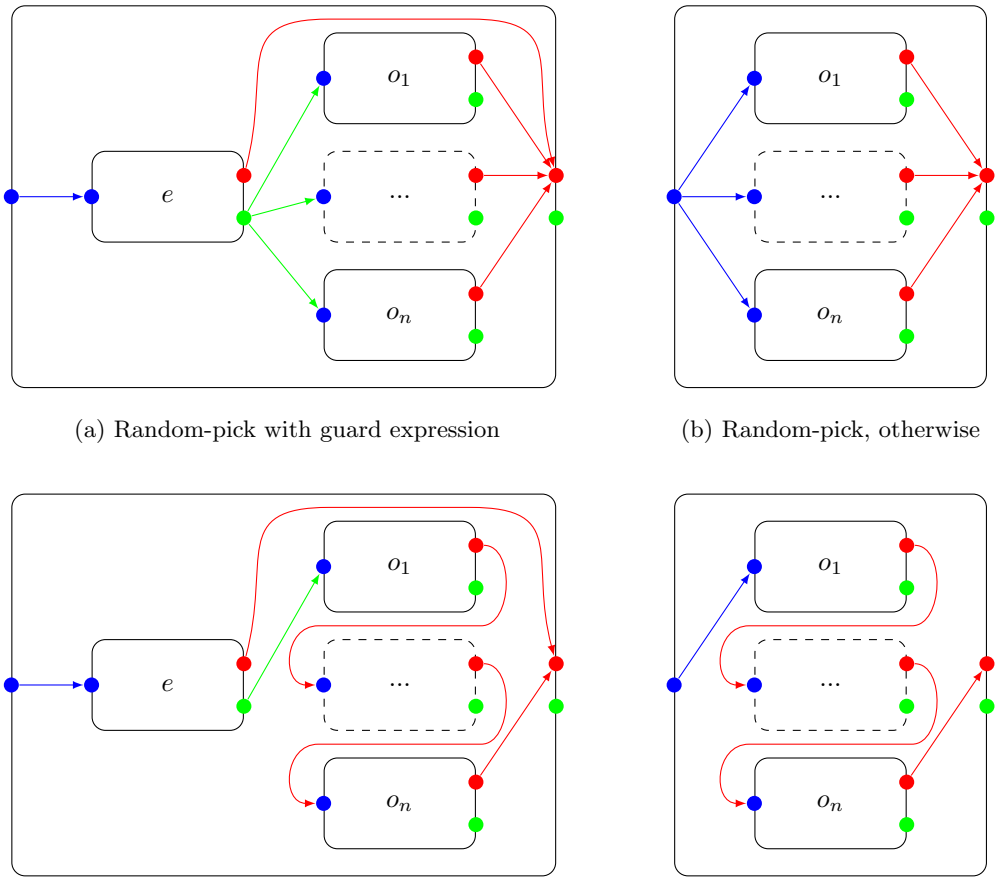
Figure 5.7: The pattern used to construct an atomic node for a transition object.

The atomic node pattern of a transition is given in Figure 5.7. Let e be the transition's guard statement and s_1 be the optional secondary statement. The outcome of the transition is determined by the transition's guard statement e, and as such, the entry point and exit points of a transition are connected to their respective counterparts in the atomic node of e. Furthermore, the atomic node of s_1 is not attached to the entry and exit points of the parent node, since the statement should be atomic in isolation. Note that the border of the atomic node associated to statement s_1 is dashed, due to the fact that the node should only be created when its counterpart exists.

5.4.1.5 Decision Nodes

Finally, a characterization will be given for the control flow of decision nodes. The implementation of the decision structure in Java code has not been discussed yet, but the required control flows can already be categorized independently of the implementation. Two control flow categories exist, with the selection thereof being based on the type of node and program settings. The first category handles decisions that are resolved by randomly picking an option, and as such, only one of the options will be visited. Two subtypes exists of this category–a random pick with and without a guard expression. The second category handles decisions that visit every option sequentially, until an option succeeds or all options have been exhausted. Similarly, a subtype is present for the presence and absence of a guard expression. Therefore, for both types, it holds that no further options need to be visited once one of the options completes successfully. Lastly, the control flow of a decision node will only proceed if all visited options fail to execute, which in turn signifies that the current branch of the decision structure has failed to make a selection.

The atomic node patterns of the decision nodes are visualized in Figure 5.8. Let o_1 through o_2 be all the options contained within the decision node, and let e denote its guard statement if present.



(a) Random-pick with guard expression

(b) Random-pick, otherwise

(c) Sequential with guard expression

(d) Sequential, otherwise

Figure 5.8: The patterns used to construct an atomic node for decision node objects.

The first pattern, given in Figure 5.8a, focuses on the control flow of a random pick operation that, prior to execution, is subjected to a guard statement. The guard statement determines whether an option should be picked, and as such, the entry node of the decision node is connected to the entry point of guard expression e, with the failure exit of e being in turn connected to the failure exit of the decision node. Subsequently, the success exit of e is connected to the entry point of all options o_1 to o_n, since it cannot be determined prior to execution which of the options will be picked. Finally, the exit points of all options o_1 to o_n are connected to the failure exit of the decision node. The second pattern, given in Figure 5.8b, contains the control flow of random pick operations that do not have an encompassing guard expression. Structurally, the control flow is similar to that of the guarded variant, with the difference being that the atomic node of the guard statement has been omitted. As such, the entry points of all options are connected directly to the entry point of the decision structure. The third pattern, presented in Figure 5.8c, covers a guarded sequence of decisions where all options are visited in order until an option succeeds. The options should not be visited if the guard statement e does not hold true: as such, the entry node of the atomic node is connected to the entry point of e, and the failure exit of e is connected to the failure exit of the decision node. The success exit of e is connected to the entry point of o_1, such that the options can be visited in sequence. The decision is made sequential by connecting the failure exit of one option to the entry of the next option, with the failure exit of the last option being connected to the failure exit of the decision node. The last pattern, given in Figure 5.8d, focuses on sequential decisions that are not encompassed by a guard expression. Structurally, the

pattern is nearly identical to the third pattern, with the difference being that the atomic node for guard statement e has been omitted: instead, the entry node of the decision node is connected directly to the entry node of option o_1.

5.4.2 Adding Lock Entries

Once the atomic node tree and locking node graphs have been initialized, lock entries can be added to the locking structure. The lock entries dictate which locks are to be acquired and released by a target locking node, such that atomicity can be achieved. Note however that the mere introduction of locks to the graph is not sufficient to achieve atomicity–a collection of follow up procedures is required to move the locks throughout the locking structure, which will be achieved by the lock propagation process described in Section 5.5.

Algorithm 9: AddLockEntries(n)

```
1 L ← ∅
2 if |n.children| > 0 then
3     for m ∈ n.children do
4         AddLockEntries(m)
5     if n.partner is an assignment then
6         L ← ℒ(n.partner.left)
7 else
8     L ← ℒ(n.partner)
9 Add all locks L to the appropriate acquire and release fields within the locking nodes of n.
```

The procedure that adds locks to the locking structure is presented in Algorithm 9, and takes a root locking node n as the sole input. The algorithm starts by defining a list of locks L that will eventually hold all of the locks required by the target node n (line 1). Locks only need to be acquired by the leaf nodes in the atomic node tree, and as such, it first needs to be determined if the atomic node n has child nodes or not (line 2). Assignments are an exception to this rule, due to the fact that the left-hand side, that being the reference to the variable being assigned to, cannot be encapsulated by an atomic node: for assignments, a lock needs to be generated for the left-hand side, even if not a leaf node. If n has child nodes, a recursive call to AddLockEntries(n) will be made for all child nodes m of atomic node n (lines 3-4) such that the leaf nodes of n can be found. Next, the procedure checks if n is associated with an assignment: if so, the locks required by the assignment's left-hand side are added to L (lines 5-6). If n does not have child nodes, all locks required by the statement associated to n are assigned to L instead (line 8). The algorithm concludes by assigning the locks L to the appropriate fields in the locking nodes in n (line 9). The locks L need to be acquired by the entry node, and released by the exit nodes of n.

The list of locks is generated through the lock generation function $\mathcal{L}$, which takes a target statement s as an input. One of the aims of this report is to discover whether a fine-grained locking approach outperforms a broader alternative, and therefore, for the sake of experimentation, several implementations of the lock generation function $\mathcal{L}$ have been implemented that acquire locks at different levels of broadness. The locking modes made available to the function $\mathcal{L}$ will be discussed in Section 5.4.2.1. Once all locks have been added, two passes of the lock acquisition restructuring algorithm will be executed: see Section 5.5.1 for further details.

5.4.2.1 Locking Modes

The locking mechanism provides four implementations for the lock generation function $\mathcal{L}$, where each implementation operates at a different level of broadness. The function $\mathcal{L}$ takes a statement s as an input, and returns a set of lock objects that are required to make the behavior of s atomic.

First, let $\mathcal{R}_C(s)$ denote the set of class variable references that are contained within statement s. The different implementations of $\mathcal{L}$ are defined as follows.

Element The element-scoped implementation creates a lock object $\langle v, i \rangle$ for every class variable reference $r \in \mathcal{R}_C(s)$, where v is equivalent to the target variable of r and i is equivalent to the index used in r. The array indices are part of the locks, and as such, each lock targets used objects at the element level. The advantage of this implementation is that the locking will be performed at the highest attainable fine-grained level. Therefore, the expectation is that the element-scoped implementation will incur the fewest number of halts due to lock acquisitions, which consequently results in optimal concurrency within the generated program.

Variable The variable-scoped implementation creates a lock object $\langle v, i \rangle$ for every class variable reference $r \in \mathcal{R}_C(s)$, where v is equivalent to the target variable of r. Oppositely to the element-scoped implementation, the value of i is not equivalent to the index defined within r: instead, i is always assigned the value zero for array variables. Therefore, all accesses to an element in array variable v will target the same lock object, which effectively implies that the target locks are based solely on the target variable. The benefit of having locks be variable-scoped is that the length of an array will no longer have a profound impact upon the performance of the locking system. The disadvantage of the suggested approach is a potential reduction of concurrency within the generated program.

Statement The statement-scoped implementation creates a lock object $\langle v, i \rangle$ if the list of class variable references $\mathcal{R}_C(s)$ is non-empty, where the target lock v is chosen to be the first variable within the current class. Consequently, any statement within a class referencing a class variable will lock the same variable, which in turn ensures that locking is achieved at a statement level. The advantage of statement-scoped locks is that statement atomicity is achieved with the fewest locks possible, namely a single lock per statement invocation. However, the proposed approach does not allow statements to run in parallel, even if they use different variables: as such, the concurrent behavior allowed within the generated program will be minimal.

No locks The last implementation of the function $\mathcal{L}$ does not generate any locks, and should thus not be used in a practical application, since atomicity of statements cannot be guaranteed. Therefore, this implementation should only be used for experimental purposes, with the aim of measuring the overall impact of the locking mechanism.

Currently, the element-scoped approach is the default implementation. The remaining options are activated by providing the arguments `-lock_array`, `-statement_level_locking` and `-no_locks` respectively, with each option being considered mutually exclusive. The available command line arguments are elaborated further in Section 6.3. Furthermore, the experiments conducted with the available lock generation functions will be discussed in Chapter 8.

5.4.3 Control Flags and Markings

The initialization of the locking structure is concluded through the introduction of the control flags and markings that are required to track and resolve issues caused by short-circuit evaluations. For each lock in the locking structure, it is verified whether the lock is reliant upon its position in the locking node graph for correct functioning, i.e., the lock in question presumably has an index that is bound checked by one or more of the earlier statements within the conjunction or disjunction. Each lock tracks the locking nodes in which the aforementioned bound checked variables are acquired, such that violations of the required ordering can be detected with ease. Furthermore, each locking node is flagged with a number through which the ordering of locking nodes can be tracked: by construction, it is guaranteed that all predecessors have a number that is lower than the target node. Consequently, it becomes trivial to verify whether a lock has moved past the

atomic node in which the bound checking is performed, since the ordering can be verified through a simple and reliable numerical comparison.

In certain situations, it is impossible to meet the ordering requirements set by short-circuit evaluations. Take for example a statement for which two locks i and j need to be acquired that are over the same array variable. Furthermore, i is acquired by a locking node that is a predecessor of the node that acquires j, and between the two nodes, there exists another locking node in which a variable used in j are subjected to a bound check. The lock propagation process, to be discussed in Section 5.5, will have no choice but to move the acquisition of lock j to the locking node acquiring i, and as such, the location requirements cannot be met under any condition. Several optimizations are made to avoid location sensitivity violations, and locks with a greater risk of violations are given a higher priority. Therefore, unsolvable violations are marked, such that the effect thereof can be excluded from the optimization efforts to attain better results.

5.5 Lock Propagation

The lock propagation process consists of two parts, where each part focuses on moving locks in one direction in the locking node graph, with the goal of broadening the active region of a lock to achieve atomicity over the entire statement or decision structure. Therefore, the procedure needs to ensure that locks will not be released and subsequently attained again in a path through the locking node graph. The first algorithm, to be given in Section 5.5.1, moves the appropriate lock acquisitions upstream in the locking node graph. Similarly, the second algorithm, to be defined in Section 5.5.2, moves violating lock releases downstream in the locking node graph. Observe that the locking node graph is an acyclic directed graph, and as such, the two algorithms guarantee that the active period of a lock is widened, i.e., it is impossible for a lock to be acquired for a shorter period than previously defined in the initial state of the locking node graph.

5.5.1 Restructure Lock Acquisitions

The goal of the lock acquisition restructuring algorithm is threefold. First, the procedure needs to ensure that the strict lock ordering can be enforced for every path through the locking node graph. As such, it needs to hold that locks over variables with a lower lock identity are locked prior to or in the same locking node as those with a higher identity. Secondly, the lock release restructuring algorithm requires that each unique target lock is requested at most once on each path through the locking node graph. Consequently, for the sake of simplicity and efficiency, it is required that locks over the same variable instance are acquired within the same locking node, such that it can be ensured by the locking instruction construction process that only a single lock acquisition needs to be performed for each unique lock target: the uniqueness of a lock target is determined in a static manner, i.e., two lock acquisitions over $x[0]$ and $x[0 \cdot i]$ are considered to be over the same lock target, but $x[0]$ and $x[i]$ are not. Lastly, the restructuring algorithm needs to maintain the structural integrity of the locking node graph, i.e., when a lock acquisition is moved upstream to a locking node v, it needs to be ensured that the target lock is released on all paths through v. Similarly, if a lock is moved upstream from a locking node v, the lock in question needs to be acquired on all paths through v prior to reaching node v.

The proposed lock acquisition restructuring algorithm will be used before lock identities are assigned to the class variables, and as such, the procedure has two different modes of operation, namely one that considers the lock identities and one that restructures based solely on the variables used by the locks. The current mode of the algorithm is represented by the heuristic $\mathcal{H}$, which is a function that will indicate whether the acquisition of the given target lock needs to be moved upstream to meet the requirements set upon the structure of the locking node graph. On top of that, let $\mathcal{L}_P(v)$ be an aggregate function that returns all of the locks that have been acquired before entering locking node v. Furthermore, note that due to a shortcoming in the implementation, it is required that the heuristic $\mathcal{H}$ ensures that the acquisition of locks with non-constant

indices are guaranteed to arrive at their final position in a single pass of the algorithm.

The heuristic without lock identities, which will be referred to as the heuristic function $\mathcal{H}_V$, will be the first heuristic to be considered. The lock identities are assigned in continuous sequential blocks, i.e., the lock identity of a specific element in the array can be accessed by taking the sum of the lock identity and the value of the target index. Therefore, the ordering requirements set between locks targeting the same array variable can be processed prior to having assigned lock identities to the variables, which in turn may lead to a more optimal lock identity assignment, given that the assignment of lock identities is based partially upon the lock acquisition ordering as observed within the tentative locking node graph. The heuristic function $\mathcal{H}_V$ adheres the following criteria. The acquisition of a lock i in locking node v should be moved to all predecessors of v if and only if there exists a lock $j \in \mathcal{L}_P(v)$ which targets the same base variable and for which one of the following criteria holds: the index that is referenced to by i or j is non-constant, or the constant index of i is less than or equal to that of j. Observe that the former criteria ensures that the acquisition of locks with non-constant indices are guaranteed to arrive at their final position in a single pass of the algorithm, since the value of the index does not influence the position– instead, the lock is moved to the earliest occurrence of the variable. The heuristic that considers lock identities, referred to as the heuristic function $\mathcal{H}_I$, takes a similar approach and adheres the following criteria. The acquisition of a lock i in locking node v should be moved to all predecessors of v if and only if there exists a lock $j \in \mathcal{L}_P(v)$ for which one of the following conditions hold: lock j is over a variable with a lock identity that is greater than that of i, or i and j are over the same variable and the index that is referenced to by i or j is non-constant, or the constant index of i is less than or equal to that of j. In other words, the lock should be moved if it risks violating at least one of the first two requirements set upon the acquisition restructuring process.

Algorithm 10: RestructureLockAcquisitions$(G, \mathcal{H})$

```
1  L_P ← a mapping from locking nodes to locks.
2  V ← nodes in G in topological order.
3  foreach v ∈ V do
4      foreach p ∈ v.predecessors do
5          Add all locks in p.acquire and L_P(p) to L_P(v).
6  foreach v ∈ reverse(V) do
7      L_V ← {i | i ∈ v.acquire : H(L_P(v), i)}
8      v.acquire ← v.acquire \ L_V
9      if v.partner is an assignment and v is an exit node then
10         Apply the assignment as a rewrite rule to all locks in L_V.
11     foreach p ∈ v.predecessors do
12         p.acquire ← p.acquire ∪ L_V
13         foreach q ∈ p.successors do
14             q.release ← q.release ∪ L_V
```

The lock acquisition restructuring process is given in Algorithm 10 and takes the locking node graph G and mode of operation heuristic $\mathcal{H}$ as inputs. The procedure starts by defining the aggregate function $\mathcal{L}_P$, where the chosen representation thereof is a mapping between locking nodes and the associated collections of acquired locks (line 1). Next, all of the required data is added to $\mathcal{L}_P$, i.e., for each locking node v, a list of locks is maintained that have been acquired prior to reaching v. During the population process of $\mathcal{L}_P$, the mapping data of predecessor nodes is utilized, and as such, a topological ordering V needs to be created over the locking nodes in G (line 2), such that the required data can be generated in an order that ensures the presence of the prerequisite data (lines 3-5). For each node $v \in V$, the value $\mathcal{L}_P(v)$ is defined to be the union of p.acquire and $\mathcal{L}_P(p)$ for all $p \in v$.predecessors, where p.acquire are the locks to be acquired in p

and v.predecessors is defined to be the predecessors of v in the graph G.

The algorithm proceeds by moving the eligible lock acquisitions to the correct position. The primary loop in the movement process iterates over all nodes v in G in a reverse topological order (line 6), such that it can be ensured that all successors of the target node v have been processed beforehand. Given a locking node v, the procedure starts by gathering all lock $i \in v$.acquire that need to be moved according to the heuristic $\mathcal{H}$ (line 7). The locks $\mathcal{L}_V$ that need to be moved are subsequently removed from v.acquire (line 8), after which a check is performed to see whether a rewrite rule need to be applied to all locks in $\mathcal{L}_V$ (line 9). A rewrite rule needs to be applied to the locks if the locking node v is the exit node of an assignment: the value of a variable used in the locks could have been altered by the assignment, which may result in the wrong lock being requested if the alteration is not accounted for through the application of the appropriate rewrite rule (line 10). With the rewrite rule applied, the algorithm proceeds by moving the acquisition of the locks $\mathcal{L}_V$ to the predecessor nodes $p \in v$.predecessors of locking node v (line 12). Additionally, it is ensured that the locks $\mathcal{L}_V$ are released by all successors $q \in p$.successors of predecessor node p (line 14), which in turn guarantees that the moved locks are released on all paths through locking node p, and therefore, the structural integrity of the locking node graph is assured.

The chosen reverse topological iteration order for the primary loop is crucial to attain a correct result. Certain locks may need to be moved multiple times, and hence, it is required that the locks in question have been moved to node v before the latter gets processed. The successor nodes of v will move lock acquisitions to v, and as such, the ordering requirements are satisfied by a reverse topological ordering. Furthermore, two passes of the algorithm are required for the lock acquisition positions to be final. The aggregate function $\mathcal{L}_P$ is defined as a precalculated mapping, and as such, changes made through the movement of locks are not reflected therein, since the mapping is not dynamic. Consequently, locks with constant index references are not guaranteed to end up at their final position during the first pass, since the first pass may have moved locks over the same variable with non-constant indices further upstream, which is not accounted for in the aggregate function $\mathcal{L}_P$. Therefore, a second pass of the algorithm is required to attain a correct result, such that the locks with constant indices can catch up with their non-constant counterparts.

The need for two passes is inconvenient, and hence, it would be prudent to revise the implementation in the future to properly compensate for the effect of moving locks. On top of that, it would allow for the definitions of the heuristics $\mathcal{H}_V$ and $\mathcal{H}_I$ to be made more flexible, since it would no longer be required that locks with non-constant indices are guaranteed to arrive at their final position in one pass. As such, the following adjusted heuristics $\mathcal{H}_{V'}$ and $\mathcal{H}_{I'}$ can be created. The heuristic function $\mathcal{H}_{V'}$ adheres the following criteria: the acquisition of a lock i in locking node v should be moved to all predecessors of v if and only if there exists a lock $j \in \mathcal{L}_P(v)$ which targets the same base variable and has an index that may possibly be greater than or equal to the index targeted by lock i, i.e., to adhere to the strict ordering, it may occur that i needs to be acquired before or at the same time as j. Note that the given inequality can be evaluated through SMT solving. The heuristic that considers lock identities, referred to as the heuristic function $\mathcal{H}_{I'}$, takes a similar approach and adheres the following criteria. The acquisition of a lock i in locking node v should be moved to all predecessors of v if and only if there exists a lock $j \in \mathcal{L}_P(v)$ for which one of the following conditions hold: lock j is over a variable with a lock identity that is greater than that of i, or i and j are over the same variable and j refers to an index that may possibly be greater than or equal to the index targeted by lock i.

5.5.2 Restructuring Lock Releases

The goal of the lock release restructuring algorithm is to ensure that, on each path through the locking node graph G, each unique lock target that has been acquired is only released once. The lock release restructuring procedure follows a similar approach as taken by the lock acquisition restructuring procedure, with several minor but crucial differences. First of all, the restructuring is performed on lock request objects instead of lock objects, which are contained within the locking

instructions of the locking nodes. The semantics of the lock request objects differ, in the sense that lock requests share instances if the target variable instance is equivalent. Given that the acquisition and release collections are defined to be sets, it hence follows that each unique locking target is guaranteed to be acquired or released only once per locking instruction. The goal of the restructuring algorithm depends upon this property, and therefore, the restructuring of lock releases should only be performed once the lock acquisitions in the locking node graph have been finalized, which includes the conflict resolutions performed during the construction process of the locking instructions. Furthermore, no strict lock ordering is required for the release of locks, and as such, no ordering needs to be enforced by the restructuring algorithm. Hence, the decision on when a lock request release should be moved depends solely on whether an instance of the same lock request object exists downstream in the locking node graph. Hence, the algorithm needs to be aware of the lock requests that are released by a locking node's successors. Let v be the current locking node and let $\mathcal{L}_S(v)$ be an aggregate function that returns all of the locks that are released by all direct and indirect successors of node v. Consequently, the release of a lock request i by v needs to be moved to the successors nodes of v if $i \in \mathcal{L}_S(v)$ holds.

Algorithm 11: RestructureLockReleases(G)

```
1  L_S ← a mapping from locking nodes to lock requests.
2  V ← nodes in G in topological order.
3  foreach v ∈ reverse(V) do
4      foreach s ∈ v.successors do
5          Add all locks in s.instruction.release and L_S(s) to L_S(v).
6  foreach v ∈ V do
7      L_V ← v.instruction.release ∩ L_S(v)
8      v.instruction.release ← v.instruction.release \ L_V
9      foreach s ∈ v.successors do
10         s.instruction.release ← s.instruction.release ∪ L_V
```

The lock release restructuring process is given in Algorithm 11 and takes as input the locking node graph G. The procedure starts by defining the aggregate function $\mathcal{L}_S$, where the chosen representation thereof is a mapping between locking nodes and the associated collections of released lock requests (line 1). Next, all of the requisite data is added to $\mathcal{L}_S$, i.e., for each locking node v, a list of lock requests is maintained that are released by direct and indirect successor nodes of node v. During the population process of $\mathcal{L}_S$, the mapping data of successor nodes is utilized, and as such, a reverse topological ordering V needs to be created over the locking nodes in G (line 2), such that the required data can be generated in an order that ensures the presence of the prerequisite data (lines 3-5). For each node $v \in V$, the value $\mathcal{L}_S(v)$ is defined to be the union of s.instruction.release and $\mathcal{L}_S(s)$ for all $s \in v$.successors, where s.instruction.release is the collection of lock requests that are to be released by the locking instruction of w and v.successors is defined to be the list of successor nodes of v in the graph G.

The algorithm proceeds by moving the eligible lock request releases to the correct position. The primary loop in the movement process iterates over all nodes v in G in a topological order (line 6), such that it can be ensured that all predecessors of the target node v have been processed beforehand. Given a locking node v, the procedure starts by gathering all lock requests $i \in v$.instruction.release of which the release needs to be moved to the successor nodes (line 7), which is defined to be the lock requests in the intersection of $i \in v$.instruction.release and $\mathcal{L}_S(v)$. The locks $\mathcal{L}_V$ that need to be moved are subsequently removed from v.instruction.release (line 8). Rewrite rules do not need to be applied, since each lock request is given an unique identity through which the original lock target can be recovered at runtime, i.e., the locking identity and the offset used by the lock request are stored for later use upon the acquisition of the lock request. Finally, the algorithm moves the release of the lock requests $\mathcal{L}_V$ to the successor nodes $s \in v$.successors of

locking node v (line 10). Note that it does not need to be ensured that the lock requests in $\mathcal{L}_V$ are acquired by by the predecessors of v, since the acquisition restructuring algorithm already ensures the structural integrity of the locking node graph, i.e., it is guaranteed by the lock acquisition restructuring procedure that, if a lock is moved upstream from a locking node v, that the lock in question is acquired on all paths through v prior to reaching node v.

Analogously to the acquisition restructuring algorithm, the chosen topological iteration order for the primary loop is crucial to attain a correct result. Certain locks may need to be moved multiple times, and hence, it is required that the locks in question have been moved to node v before the latter gets processed. The predecessors nodes of v will move lock releases to v, and as such, the ordering requirements are satisfied by a topological ordering. The algorithm has no ordering requirements, and as such, one pass of the algorithm is sufficient to attain a final result.

5.6 Lock Identities

The next order of business is to assign order-defining lock identities that ensure that inter-lock-dependencies and phased locking can be attained not only effectively, but also efficiently. The locking of variables has a considerable impact on the performance of the overall program, and as such, locks should be used as sparingly as possible. The assignment of proper lock identities will be crucial in this endeavor, since both the inter-lock-dependencies and phased locking rely heavily upon the ordering of the lock identities to limit the number of requested locks.

5.6.1 Variable Dependency Graph

First, a variable dependency graph needs to be constructed that subjects the class variables contained therein to the ordering requirements set by the inter-lock-dependencies and the locking node graph, such that this graph can be used to allocate a fitting lock ordering. Within the dependency graph, the variables are represented by the vertices, and the ordering requirements are depicted by the directed edges contained therein. By construction, a variable v depends upon variable w if one or both of the following conditions holds true. Firstly, inter-lock-dependencies are considered, i.e., a dependency between the variables v and w is added if w is referenced to within the index of v, since w needs to be acquired before the index of v can be evaluated. Secondly, the locking node graph structure is enforced, i.e., a variable v depends on w if the latter has been acquired before v within the current branch of the locking node graph: superfluous self-loops need to be avoided, and as such, no relation is added if v and w refer to the same base variable.

The process of constructing the dependency graph is given in the Algorithm 12, which takes a class c as the sole input. First, a locking node graph $G_{\mathcal{L}}$ is constructed that covers all of the statements and structures contained within the class c (line 1). Afterwards, an empty directed graph G is created (line 2) that will hold the resulting dependency graph. Subsequently, an empty mapping $\mathcal{V}_P$ is defined which, after construction, will map a given locking node to the list of variables that have been acquired by the node and its predecessors so far (line 3), after which the algorithm proceeds by iterating over all locking nodes $n \in G_{\mathcal{L}}$ in topological order (lines 4-14).

Each iteration starts by constructing the list of previously acquired variables T (line 6), being the union of the values $\mathcal{V}_P(w)$ of all direct predecessors p of locking node n. Next, the algorithm iterates over all locks $i \in n$.acquire that need to be acquired in node n (lines 7-13), after which edges are added to graph G based on the inter-lock-dependencies and the structure of the locking graph (lines 10-13 and 8-9 respectively). Finally, the value of $\mathcal{V}_P(n)$ is updated upon exiting the nested loop such that it contains the appropriate data for subsequent iterations (line 14).

5.6.2 Assigning Lock Identities

The variable dependency graphs that were introduced in the previous section will be used to assign lock identities to all class variables within the model. The allocation of lock identities will

Algorithm 12: ConstructDependencyGraph(c)

```
1  Let G_L be the locking node graph covering all objects contained within class c.
2  G ← an empty directed graph.
3  V_P ← a mapping from locking nodes to variables.
4  N ← nodes in G_L in topological order.
5  foreach n ∈ N do
6      T ← the union of variables V_P(w) for all w ∈ n.predecessors.
7      foreach i ∈ n.acquire do
8          foreach v ∈ T \ {i.variable} do
9              Add an edge from i.variable to v in the dependency graph G.
10         if lock i has a target index then
11             S ← all class variables used within the index of i.
12             for v ∈ S do
13                 Add an edge from i.variable to v in the dependency graph G.
14     V_P(n) ← the union of variables T and all class variables present in n.acquire.
15 return G
```

be performed on a class by class basis, since classes do not share access to their variables, and as such, the lock identities can be local to the classes. Using Algorithm 12, a dependency graph G can be constructed that contains all variable relations within the target class. For a class variable to be locked prior to another, its lock identity needs to be lower. Thus, it is trivial to see that for an inter-lock-dependency to be resolved correctly and efficiently, variables used in the indices will need to be assigned a lower lock identity. On top of that, the assigned locking identities should respect the order of the lock acquisitions within the intermediate locking node graph whenever possible, such that locks stay active as short as possible and are requested at the latest opportunity. As discussed in Section 5.5.1, lock acquisitions can only move upstream in the locking node graph, and as such, any further moves incurred due to the assigned locking identities will be detrimental to the measure of concurrency taking place within the generated program. Therefore, it is desirable that class variables occurring earlier on in the topological ordering of the intermediate locking node graph are assigned a lower lock identity. By construction, the dependency graph G includes both of these relations: if a variable v depends upon another variable w, it follows that w should be assigned a lower lock identity than v. The envisioned algorithm assigns incremental identities, and as such, it follows that variables with no dependencies should be processed first.

For the sake of simplicity, array lock identities are defined to be sequential continuous blocks, i.e., the lock identity of a specific element in the array can be accessed by offsetting the identity by the value of the target index. The concept of phased locking becomes far more complex with non-continuous blocks, since due to the potential presence of non-constant indices, the phases can no longer be constructed in a preprocessing step. Therefore, the phases would need to be generated at runtime instead, which leads to additional complications, i.e., the value of the used indices could change during the evaluation, which in turn may invalidate the earlier constructed locking phases. Given the added complexity and performance overhead, it has been concluded that sequential continuous lock identity blocks are a more workable option.

The overall process of allocating lock identities is given in Algorithm 13, which takes a singular class c as an input. The AssignLockIdentities(c) algorithm starts by constructing the variable dependency graph G (line 1), after which it proceeds with finding the nodes in G that do not have any successors (lines 5-10). The selected nodes are assigned an incremental lock identity based on the size of the variable, after which the node itself and all of its relations are consequently removed from G. The removal of the node ensures that previously dependant nodes in the tree are turned into leaf nodes once all their dependencies are satisfied, after which they can be processed

Algorithm 13: AssignLockIdentities(c)

```
1  G ← ConstructDependencyGraph(c)
2  i ← 0
3  while |G.nodes| > 0 do
4      do
5          V ← {v | v ∈ G.nodes : |v.successors| = 0}
6          Order V such that variables having location sensitive locks are processed first.
7          foreach v ∈ V do
8              v.lockid ← i
9              i ← i + v.size
10             Remove node v from the graph G.
11     while |V| > 0
12     if |G.nodes| > 0 then
13         v ← a node in G picked by the selection heuristic.
14         v.lockid ← i
15         i ← i + v.size
16         Remove node v from the graph G.
```

in the same manner in the next iteration. The selection and processing of leaf nodes continues until the supply of leaf nodes in G has been fully exhausted. Note that at this point, G might not necessarily be empty due to circular references in the dependency graph. In such a situation, it is impossible to create an ordering in which it is assured that variables used in the indices have lower lock identities. As such, not all complications can be resolved solely through the assignment of lock identities, and certain improvements will need to be made to the locking mechanism to tackle this issue. Said improvements will be accomplished through the concept of lock request unpacking, to be discussed further in Section 5.7.2.1. For the remainder of this section, the focus will remain on the assignment of lock identities.

In the case of circular references, a node need to be selected for processing and removal based on several heuristics (lines 12-16). The decision of which node to pick is not trivial, since it might have a profound impact upon the overall performance and efficiency of the program. The selection heuristics will be discussed further in Section 5.6.3. Once the chosen variable node has been processed and removed from G, all of the above steps are repeated until the graph is empty and all variables have been assigned a lock identity.

5.6.3 Selection Heuristic

The resolution of circular references in lock requests has a heavy impact upon the performance of the generated code, and as such, it is important to take the effect of unpacking into account when assigning the lock identities. To achieve this, a node selection heuristic will need to be created that enables Algorithm 13 to assign a lock identity configuration that handles circular references as efficiently as possible. Several criteria for the heuristic have been identified that will mitigate the impact of unpacking on the overall performance of the generated code.

1. *Variable size*: the size of the variable dictates the number of additional locks that have to be acquired due to the unpacking. The locking of variables is an expensive operation, and as such, it is preferable to unpack variables that are limited in size.

2. *Frequency of use*: class variables that are used more frequently than others are less viable for unpacking, since the unpacking operation would need to be executed more frequently as well. Thus, it is preferable to unpack infrequently used variables.

3. *Number of dependencies*: the number of dependencies that a class variable has upon selection is equivalent to the number of variable combinations that would require unpacking, i.e., if a node gets selected, all of its remaining dependencies will invariably be assigned a higher lock identity. As such, the selection heuristic should aim to minimize the number of dependencies of the selected variable node. Note that the initial step of Algorithm 13 already uses this criteria to freely assign lock identities to class variables that have no remaining dependencies.

4. *The number of affected state machines*: in the SLCO language, state machines run in parallel. State machines need to halt upon encountering a locked class variable instance, and hence, it is important to minimize the number of situations in which state machines have to halt, since it will maximize the measure of concurrency in the generated program. Thus, it is preferable to select class variables that affect a limited number of state machines.

5. *Presence of location sensitive locks*: in Section 5.7.2.1, it will be shown that the unpacking of locks due to location sensitivity violations is more costly than resolving violations caused by the strict ordering requirement. Therefore, it is advisable that bound-checked variables are given a higher lock identity than their dependencies.

Generally, the aforementioned criteria are in different and mutually distinctive domains, and as such, there unfortunately does not exist a one-criteria-fits-best solution. Thus, the node selection heuristic will need to find a balance between the proposed criteria, which in turn introduces additional challenges. Currently, weights are assigned to each of the proposed criteria, such that a weighted average of normalized criteria measurements can be used to select the best candidate. On top of that, multiple measurements are taken for the *frequency of use* measurement: besides providing a total number, counts are also provided that consider the number of occurrences per category, i.e., the number of statements or locking nodes that use the variable.

However, due to a lack of time, the selection heuristic has not been optimized. In the current implementation, measurements have been added for all but one of the criteria, with the number of affected state machines being excluded. Additionally, no weights have been described to each of the criteria: each criteria measurement has an equal contribution instead, which implies that criteria with multiple measurements are weighted more heavily by the heuristic. In practice, the current configuration performs adequately, but nevertheless, further research and experimentation is required to construct a more optimal and robust selection heuristic.

5.7 Finalizing the Locking Structure

Finally, the locking structure will be finalized to ensure that all lock acquisitions and releases are at the correct position within the locking node graph, with no structural violations. First, two passes of lock propagation are performed to restructure the locks to be acquired: in these passes, the lock identities assigned to variables will be considered, and as such, the lock acquisition positions will be deemed final once the lock propagation passes are completed. Subsequently, violations within the structure pertaining to the strict lock ordering or location sensitivity of locks will be marked: the process thereof is elaborated further in Section 5.7.1. Once all violations have been marked, locking instruction objects will be generated for all locking nodes within the locking node graph, which will contain the finalized collection of lock targets to acquire and release at the given position in the graph. Furthermore, the target variable instances of all locks that are marked due to a violation of the lock ordering or location sensitivity requirements will be adjusted, such that the violations are resolved. Refer to Section 5.7.2 for further details on the construction of locking instructions and the resolution of violations. Finally, lock propagation is used to restructure the lock request releases within the locking node graph, such that it is ensured that every lock request is only released once on every path through the locking node graph. Once the finalization of the structure is completed, the locking data structure will be sound: the locking node graph contained therein will guarantee both the atomicity of statements and strict lock ordering requirements.

5.7.1 Marking Violations

After two passes of the lock acquisition propagation, all lock acquisitions are guaranteed to be at their final position, and as such, the dirty markings for violating locks can be added. Two types of violations exist. Firstly, a lock object within the locking structure is marked as dirty if its inter-lock-dependencies violate the strict lock ordering, i.e., the lock for array variable v uses a variable w within its index with a lock identity that is higher or equivalent to that of v, which implies that the strict ordering cannot be adhered to. In other words, let $\mathcal{L}$ be the lock generation function as defined in Section 5.4.2.1: a lock i over an array variable v needs to be marked as dirty if there exists a lock $j \in \mathcal{L}(i.\text{index})$ over variable w for which $v.\text{id} \leq w.\text{id}$ holds.

Secondly, a lock needs to be marked as dirty if the latter causes a location sensitivity violation to occur: it can no longer be guaranteed that the acquisition of the lock object in question is subjected to bound check operations within the model definition, i.e., the acquisition of the lock in question has moved to a locking node of which the position in the control flow precedes the bound check performed upon the variables used within its index. Let i be a location sensitive lock over an array variable v and let L be the entry locking nodes of the atomic nodes that perform the bound checking on the target index of v: lock object i is marked dirty if the acquisition of i is performed in a locking node m that precedes or is equivalent to any node in L.

5.7.2 Generating Locking Instructions

With the lock acquisitions finalized and violations marked, the next step will focus on generating the required locking instruction objects for all nodes within the locking node graph. The locking instructions contain the finalized lock acquisition and release sets, and as such, the data contained within the locking instructions will be used during the rendering of the code. During the construction of a locking instruction, all of the lock objects contained within the associated locking node are converted to lock request objects through a lock request provider function: the latter ensures that all lock requests over the same variable instance are represented by a single lock request object instance. The function achieves this by mapping each target variable instance to its associated lock request object. If a match exists within the function's mapping, the same instance will be returned, otherwise, a new lock request is created for the target variable instance, which subsequently is added to the mapping. The variables to be locked are class-scoped, and as such, the lock request provider will create lock requests that are unique for the target class.

The locking instruction object construction process for a locking node n starts by converting all non-dirty locks acquisitions in n to lock requests through the use of the lock request provider function $\mathcal{R}_P$. Subsequently, the generated lock requests are added to the locks to be acquired set of the locking instruction. Next, all dirty lock objects to be acquired in n, i.e., the lock requests that violate the integrity of the locking structure, are resolved through a process called unpacking. During unpacking, the entirety of an array is locked by creating lock requests for every index in the target variable. If the weak variant of unpacking is used, the violating lock request is acquired after all locks within the locking node have been acquired. Afterwards, the supplementary locks used in the unpacking are released, with the location of the release depending on the type of unpacking: for weak unpacking, the supplementary lock requests are released within the node they have been acquired in, and as such, they acted as a placeholder to safely acquire the target lock request; for strict unpacking, the supplementary locks are released within the locking nodes that are to release the unpacked lock, and therefore, the violating lock request is replaced in its entirety. Next, all non-weakly unpacked lock releases in n are converted to lock requests, after which they are subsequently added to the locks to be released set of the locking instruction. The concept of unpacking will be described in further detail in Section 5.7.2.1.

Lastly, the requirements set by the inter-lock-dependencies are dealt with by subdividing the set of lock requests to be acquired in n into a list of locking phases. The generated list of phases ensures that the lock requests that are depended upon in an inter-lock-dependency are fully acquired before the associated variables are accessed to evaluate the index evaluation. See Section 5.7.2.2

for more information on locking phases and the algorithm used for the creation thereof. Once all of the locking instructions have been constructed, lock propagation is used to restructure the lock releases within the locking node graph. The latter finalizes the locking node graph, which in turn ensures that the atomicity of statements is guaranteed.

5.7.2.1 Unpacking

Circular dependencies in array variable indices cannot be resolved solely through the lock identities, i.e., additional mechanisms are needed to tackle situations where the strict lock ordering cannot be attained due to conflicting inter-lock-dependencies or location sensitivity violations. As mentioned previously, circular dependencies will lead to situations where the array variable's index contains a variable that has a lock identity that is higher than the lock identity of the array variable itself, and consequently, the index itself cannot be evaluated without violating the strict ordering. Given that the value of the index at the time of locking cannot be determined, the lock requests for the violating array variable will need to be decomposed, or, as it will be referred to within the remainder of this document, unpacked. The unpacking of a dirty array variable ensures that all indices of the targeted array variable are locked prior to the evaluation of the index. Therefore, the array index requested by the violating lock request is guaranteed to be locked prior to the evaluation of the index value itself, and hence, the original request becomes superfluous.

Two forms of unpacking are utilized by the locking structure, namely weak and strict unpacking, where each form of unpacking tackle a different type of violation in the ordering. Order violations caused by inter-lock-dependencies are resolved through weak unpacking, which is a form of unpacking that activates the required additional locks for as short a time as possible. During weak unpacking, all supplementary locks are requested first to ensure that the strict lock ordering is enforced. Subsequently, the original lock is requested, after which all supplementary locks used during the unpacking operation are released, all within the same locking node. Observe that the original lock can be requested without violating the lock ordering, since all indices of the variable in question are already held by the thread in question: the locking mechanism uses reentrant locks, and therefore, the target lock will stay active even after releasing the supplementary locks.

Strict unpacking is performed when a lock is marked dirty due to a location sensitivity violation. For the latter category, it cannot be assured that the index of the lock in question meets the bounds set by the model. Consequently, the use of the index may result in exceptions related to bound checking, even when the model checks for the appropriate bounds. The locking mechanism should not cause exceptions unless the model itself is faulty, and as such, it needs to be ensured that exceptions will not occur in the aforementioned circumstances. Strict unpacking accomplishes this by completely replacing the target lock by its unpacked components: hence, if the index is within the bounds of the array, it is assured that the lock to the appropriate element of the array is acquired, since the array is guaranteed to be locked in its entirety. Unfortunately, the impact of strict unpacking upon the concurrency within a program is substantial as a result. Alternatively, it may be possible to release the supplementary locks after the given bounds have been verified. Oppositely, one may opt to not request the lock if the given index is outside of the array. However, for the latter, it needs to be noted that inadvertent exceptions may occur during said range check operation if the index makes use of a nested array variable of which the index is out of range. Unfortunately, the proposed alternatives could not be investigated due to a lack of time.

Observe that, during the unpacking process, it is not necessary to filter out lock requests that are acquired or released at different positions in the locking node graph. Nevertheless, the correctness of the previous statement depends heavily upon the behavior of the lock propagation process. For one, the lock release restructuring procedure given in Algorithm 11, which is to be executed after all nodes are created, ensures that lock requests are released only once per control flow path by pushing the supplemental lock requests downstream in the locking node graph. Therefore, a lock request is guaranteed to only be released once. However, the lock acquisition positions are already finalized, and as such, lock propagation cannot be relied upon to guarantee that lock requests are

only acquired once per control flow path: instead, the locking instruction construction relies upon the fact that it is guaranteed by the preceding two passes of lock acquisition propagation that array variables with non-constant indices are acquired within the same locking node as their constant counterparts. Given that the collection of locks to be acquired within the locking instruction is defined to be a set, it hence follows that a lock request is guaranteed to only be acquired once, and consequently, no filtering of lock requests is required during the unpacking process.

Nevertheless, it can be concluded that unpacking is an expensive operation: unpacking drastically increase the number of locks that need to be acquired, and on top of that, the acquisition of certain often-used locks can heavily impact the concurrency of the program, especially if the associated class variables are used by multiple state machines. Therefore, the unpacking of locks should be avoided whenever possible, and as such, lock identities need to be assigned in a manner that minimizes the impact of lock request unpacking, which this work aims to achieve through the selection heuristics previously discussed in Section 5.6.3.

5.7.2.2 Locking Phases

Inter-lock-dependencies cannot be solved solely through the assignment of lock identities, since lock-deadlocks may still occur when class variables used in the indices are not locked prior to the creation of the lock request. Thus, an additional mechanism needs to be introduced to tackle this issue, namely phased locking. The idea of phased locking is simple–divide the list of lock requests into smaller groups, and only proceed to the next locking phase after all the locks requested in the previous phase have been acquired successfully.

Algorithm 14: ConstructLockingPhases($G, \mathcal{L}$)

```
1  Sort all ⟨v, i⟩ ∈ 𝓛 on the lock identity of v.
2  𝒫, 𝓛′, V ← ∅
3  foreach ⟨v, i⟩ ∈ 𝓛 do
4      if |v.successors| > 0 then
5          Append 𝓛′ to 𝒫 and empty out 𝓛′.
6          Remove all nodes v′ ∈ V from graph G.
7      𝓛′ ← 𝓛′ ∪ {⟨v, i⟩}
8      V ← V ∪ {v}
9  Append 𝓛′ to 𝒫
10 return 𝒫
```

Algorithm 14 divides the given list of lock requests $\mathcal{L}$ into locking phases, based on the structure of the provided variable dependency graph G over the target class. Note that the dependency graph G used as an input is not allowed to have circular references, and as such, G should only be generated after the violating locks have been unpacked. The ConstructLockingPhases($G, \mathcal{L}$) algorithm starts by sorting $\mathcal{L}$ on the lock identity of the associated key nodes (line 1), such that $\mathcal{L}$ adheres to the strict lock ordering of the base variables. Note that it is not necessary to consider the offset of the lock request during the sorting, since all lock requests over the same variable will, by construction, be part of the same locking phase. The presence of self-referencing class variables will not lead to complications either, due to the unpacking operation that has taken place beforehand. Observe that the final ordering of the lock requests within a phase can only be determined at runtime, since the provided offsets can be a non-constant value.

Next, several support variables are introduced (line 2). Let $\mathcal{P}$ be a list of phases that have been generated so far, and let $\mathcal{L}'$ be the current phase that is being constructed. Additionally, let V be a list of nodes that have been encountered in the phase. The algorithm proceeds by iterating over the lock requests in $\mathcal{L}$ (line 3), whilst adding all encountered lock requests and nodes to $\mathcal{L}'$ and V respectively (lines 7-8), until a node v is encountered that has one or more dependencies (line 4).

Once such a node v has been encountered, the current phase $\mathcal{L}'$ is flushed into $\mathcal{P}$ (line 5), after which all nodes $v' \in V$ are to be removed from G (line 6). The process on lines 4-8 is repeated until all lock requests in $\mathcal{L}$ have been processed, after which the last phase $\mathcal{L}'$ is flushed into $\mathcal{P}$ again if non-empty (line 9).

Proof of Correctness Throughout the length of this report, it has been stressed extensively that the class variables need to be locked in a strict predetermined order to avoid lock-deadlocks, and as such, it is important to prove that the concept of phased locking, and the implementation thereof in Algorithm 14, does not violate this central paradigm.

First, it is important to observe that the phases themselves can only be sorted at runtime, since the indices used in the lock requests might be a non-constant value. Thus, the algorithm only needs to ensure that the phases have non-intersecting lock identity ranges. Suppose that the algorithm generates a list of locking phases $\mathcal{P}$, containing the phases p_1 and p_2 with intersecting lock identity ranges. Furthermore, assume that p_2 is a successor of p_1 in $\mathcal{P}$, such that it holds that $\max(v.\text{lockid} \mid \langle v, i\rangle \in p_1) \geq \min(v.\text{lockid} \mid \langle v, i\rangle \in p_2)$. Next, observe that the lock requests $\langle v, i\rangle \in \mathcal{L}$ are sorted on the lock identity of the node key v. Given that the lock requests are sorted, it is trivial to see that the lock requests in $\mathcal{L}$ are grouped by the associated variable, with every consecutive group having a higher base lock identity than the predecessor. Moreover, recall that array type class variables have lock identities assigned sequentially in a continuous block the size of the array variable itself. Thus, it follows that $\max(v.\text{lockid} \mid \langle v, i\rangle \in p_1) \geq \min(v.\text{lockid} \mid \langle v, i\rangle \in p_2)$ cannot occur, unless the split is made between requests over the same key variable.

However, by construction, it is impossible for Algorithm 14 to divide lock requests having the same variable as a key over two or more phases. The conditional on line 4 of the algorithm, which is checked prior to the addition of the target lock request to the current phase, can only evaluate to true for the first lock request using the specified key. Consecutive requests sharing the same key v will never trigger a phase flush, since the number of successors of v will become zero due to the removal of all nodes $v' \in V$ from G. Note that this observation holds true, since circular references are removed from G through the unpacking operation. Thus, the phases will never have intersecting lock identity ranges, and as such, the phased locking will not create situations where the strict lock ordering cannot be enforced.

5.8 Further Optimizations

During the implementation of the locking mechanism described in this chapter, several elements have been identified that may be improved upon further. Firstly, it has been observed that the ordering of options within a decision node may have a significant impact upon the number of unpack operations that need to be performed. The possibility of reordering options within a decision node will be explored further in Section 5.8.1. Furthermore, a critical look will be taken in Section 5.8.2 at the location sensitive locks and the issues caused by the criteria chosen for the violation markings thereof. Lastly, the importance of proper exception handling within the code will be highlighted, since an exception in one of the state machines in the critical section may potentially lead to lock-deadlocks for the other state machines due to improper cleanup of the program state. Several suggestions will be made in Section 5.8.3 to tackle this issue.

5.8.1 Reordering Decision Node Options

The performance of the locking mechanism can be improved by rearranging the options in a decision node such that a more optimal locking node graph can be created. To maximize concurrency within the program, it is important that the active period of a lock to a variable remains as short as possible, such that other state machines in the model will have more opportunities to use said variable. Given the chosen patterns of decision nodes and the sequential variant thereof, it can thus be beneficial to reorder the decisions $o \in G$ with this observation in mind, since the entire

decision structure is treated as an atomic entity, i.e., the decision structure is seen as a single statement. Furthermore, the effect that the ordering of the options in a decision node has upon the number of unpacking operations that need to be performed is significant, even more so for the strict variant thereof: strict unpacking operations occur when a lock of a bound checked variable needs to be acquired before the bound check operation is performed, and therefore, if the bound checked variable is used in a previous option, a strict unpacking operation becomes unavoidable when a sequential decision is to be performed. Performing a strict unpacking is a costly operation, and as such, said operation should be avoided whenever possible.

During the scope of the thesis, several experiments have been conducted with the reordering of options within a decision node to optimize the performance of the locking mechanism. To start with, a solution has been envisioned that models the problem as an SMT problem. Within the solution, a table is created in which each row represents one of the available options and each column depicts one of the variables used within the options of the decision nodes–a cell within the table has the value one if the variable is used within the given option and false otherwise. On top of that, the new ordering of the options within the decision node is represented by a mapping that associates each option to its index within the constructed ordering. Two optimization targets have been defined for the SMT model with each focusing on one of the challenges defined above. The first optimization target minimizes the sum of differences between the mapped indices of the first and last occurrence of each variable within the model, which in turn ensures that collectively, the variables used within the decision node remain locked for as short a duration as possible. The second optimization target aims to minimizes the number of variables which are subjected to an unavoidable location sensitivity violation due to the given ordering. The optimization targets remain to be merged into a single optimization statement, which can be achieved by taking a weighted sum between the two factors–as of yet, appropriate weights for the factors has not been determined yet, and therefore, more experimentation is required in the future. Furthermore, an alternative implementation has been created that reorders the options using a greedy approach. The latter uses a selection heuristic to pick the most optimal option for the current index. In the selection heuristic, options are given a score based on several factors that pertain to the current state of the variables and location sensitivity markings present within the available options. The latter solution is not optimal, but has proven to be more efficient than the SMT solution.

Unfortunately, the aforementioned implementations could not be integrated into the final product due to structural revisions to other components. Nevertheless, the associated code remains within the SMT module of the project repository for the sake of transparency and completeness, such that the envisioned optimizations may be incorporated in the future.

5.8.2 Improving Location Sensitivity Markings

The current approach chosen for dealing with short-circuit evaluations is not optimal–within the current implementation, a location sensitivity violation may occur even if the variables used within the index are subjected to the appropriate bound checks. Take for example a decision node with an encompassing guard statement that has location sensitive locks: by construction, the encompassing guard statement is equivalent to the guard statement of one of the transitions contained therein. Therefore, an unavoidable location sensitivity violation will occur, due to the fact that the acquisition of the location sensitive lock needs to be moved upstream to the encompassing guard statement, despite the observation that the bound checks upon the target variables are guaranteed to have been performed already by the encompassing guard statement.

Therefore, several improvements need to be made to the marking of location sensitivity violations to circumvent the aforementioned issue. First, a location sensitive lock needs to be made aware of the state that all prior bound checks are in, i.e., it needs to be known whether the lock is contained within the bound check's success or failure branch, such that state thereof can be considered. Next, a bound check should be marked as superfluous if the latter is guaranteed to hold true under the context of the bound checks performed priorly within the program's control

flow. The aforementioned solution remains to be implemented and tested.

5.8.3 Exception Handling

The improvements made in the scope of this chapter improve the quality of not only the generated code, but also the locking system itself. However, several external factors exist, such as erroneous input models, that still have the potential to cause lock-deadlocks through the triggering of exceptions in one or more of the state machines. Moreover, the exceptions are often times caused by the inner workings of the locking management system itself, which is in a sense logical. After all, the locks need to be acquired before the desired functionality in the model can be executed. The locking mechanism is especially exposed to exceptions pertaining to range checking, such as an `IndexOutOfBoundsException`. The proper handling of exceptions is an important aspect of designing a robust and reliable code generator, but unfortunately, the original implementation does not mitigate the effect of exceptions on the locking mechanism. In this work, several improvements have been made to the implementation to ensure that the occurrence of an exception in one or more of the threads running the state machines cannot cause class-wide lock-deadlocks.

First, a brief explanation will be given on how an exception can cause a lock-deadlock. Recall that, for the locking system to work correctly, it is required that all locks that have been acquired for an operation will eventually need to be released after use. Under normal circumstances, the generated lock management code automatically ensure that this requirement is met, but unfortunately, this is not the case when an exception is thrown. Without proper exception handling, the thread the state machine is running in stops outright, leaving the program in an unclean state in which the faulty thread still retains ownership over the acquired locks. Thus, given that the lock manager is a shared resource between all state machines in the same class, the other state machines have no choice but to wait indefinitely to attain their requested locks, resulting in a lock-deadlock.

Several approaches have been considered to tackle the issue of exceptions and their resulting lock-deadlocks. One options is to create a module that preemptively verifies the integrity of the Slco models through the concept of range checking, such that it can be ensured that exceptions cannot occur. However, exceptions might occur by other means, and hence, exception handling must be done regardless. Thus, a more reactive approach has been taken instead. The currently implemented approach uses the exception handling mechanisms available in Java to catch exceptions in the threads, after which the state of the locking mechanism is restored by automatically releasing all the locks owned by the current thread. Releasing the acquired locks in the described manner is an expensive operation, since every lock needs to be checked individually, with each lock potentially having been acquired multiple times due to the use of reentrant locks. However, the performance of the aforementioned mechanism is not a primary concern, given that exceptions only occur in exceptional situations. For the sake of simplicity, the current implementation does not interrupt the state machines sharing the same class instance.

The simplification made above is often times impractical, since generally, the state machines within a class depend upon one another for the overall program to function properly. Consequently, the implementation will need to ensure the property of dependency safety [19]: when an operation fails, all operations depending upon the success of the aforementioned operation also cease executing. Dependency safety can be achieved in a straightforward manner through the language mechanism called failboxes [19]. Considerable progress has been made on the implementation of a language extension providing failboxes in Java [28], and it is the intention that failboxes will eventually be integrated into Slco's code generator.

Chapter 6

Code Generation

Within the preceding chapters, several additions and revisions to the mechanisms and data used within the code generator have been discussed. However, up to this point, the integration thereof into the code generator project has not been touched upon. Therefore, this chapter will focus on the implementation of the code generator, and the challenges faced therein. The code generator project has been rewritten from the ground up, with the aim of improving the overall code quality of the project. Code within the original implementation has been re-used with the necessary improvements when appropriate. The decision to rewrite the majority of the Slco to Java transformation project, albeit inefficient and time-consuming, is considered to be justifiable, since the process thereof resulted in a greater understanding of the project and its less obvious underlying challenges and intricacies. Furthermore, the rewriting process has facilitated the integration of the deterministic structures and the overhauled locking mechanism, making the latter an integral part of the revised code generator, instead of an afterthought.

During the rewriting process, the Slco code generator project has been divided into multiple modules: each module focuses on a key aspect of the code generation process, where the key aspects are defined to be the model definitions, preprocessing procedure, decision and locking structure generation, and rendering of the code. The use of separate modules ensures that the codebase stays clean and well-structured, which in turn improves the maintainability and quality of the project. Furthermore, the modules are subdivided further when deemed appropriate, such that each component focuses on solving a different aspect of the problem. On top of that, several additional structural and implementational issues have been identified that will be touched upon in the remainder of this chapter, the latter of which will be structured as follows. First, the restructuring of the code generator component of the Slco framework will be elaborated upon in Section 6.1, with the focus being predominantly upon the rendering module thereof. The improvements made to the generated Java code, including the integration of the deterministic structures and improved locking mechanism, are discussed in Section 6.2. Lastly, the revised command-line arguments for the Slco to Java transformation project will be detailed in Section 6.3.

6.1 Restructuring the Code Generator

During the code rewriting process, several structural problems have been identified that warrant attention. In the current implementation of the code generator, the entirety of the rendering process is performed through the use of a single code template, which is considered to be unconventional and against common practice. As such, several improvements have be made to the code template, which will be discussed further in Section 6.1.1. Furthermore, several extensions to the code generator will be made in the following chapters, such that the code can be formally verified and analyzed in terms of performance. To facilitate the extension process, the concept of class

inheritance will be applied to the code generator in Section 6.1.2.

6.1.1 Code Templates

Several structural improvements have been made in an effort to improve the overall quality of the code generator itself. First of all, the sole code template used for the code generator has been restructured thoroughly by splitting the original template into several smaller templates, which are all limited to the scope of a single object within the SLCO language. Java is an object-oriented language, and hence, it stands to reason that the code generator templates should follow the same principle. Through this change, the code templates have become smaller and more manageable, which considerably improves the readability and maintainability of the code. Moreover, the use of object-scoped templates allows for a more modular approach to be taken, which in turn improves the flexibility and reusability of the procedures used within the code generator.

Additionally, it has been observed that the code template itself contains logic, which is not considered to be best practice. Instead, comprehensive view models should be used that already contain all the preprocessed data, such that the logic is completely decoupled from the rendering itself. Furthermore, the use of recursive calls within the code templates is avoided. The JINJA2 template language supports the use of recursive functions through filters, which allows for a template to make a call to a Python function and include the attained result. The disadvantage of the aforementioned approach is that the hierarchical structure of the code generator is spread over multiple files, i.e., both the renderer module and code templates are required to get a full picture on the code's program flow, which results in code being unclear and cluttered. Therefore, the code generator has been restructured, such that all recursive calls are performed within the renderer module prior to rendering the affected code template. The code generated by the recursive calls is added to the object's view model, such that it can be included through the code template.

6.1.2 Utilizing Inheritance

Within Chapters 7 and 8, additional code generators will be defined in which supplementary code constructs are added to formally verify and measure the performance of the code respectively. Therefore, it would be beneficial to reuse the code defined by the base code generator as a foundation for the aforementioned extensions, such that changes in the former are automatically applied to the latter. To achieve this, the base code generator has been converted to a class object, such that the behavior defined therein can be extended or overwritten through inheritance.

The code rendering class is structured as follows. For each object in the SLCO language, class functions are defined through which the appropriate code is rendered. Furthermore, functions are provided that provide the code that needs to be rendered at a specific location relative to the object's body, i.e., through the methods, code can be included before or after the object if applicable. One may for example extend the code generator with a performance metric by adding a statement in a transition's closing body that increments a counter after successful completion. The aforementioned approach ensures that virtually any behavior in the rendering class can be extended or overwritten with relative ease, while keeping the code simple, maintainable and easy to experiment with. Furthermore, contract functions are provided for objects if the latter is rendered as a Java method, through which the formal verification statements to be defined in Chapter 7 can be included. Similarly, supportive methods are included for objects represented by Java classes, such that the contents of the class' preamble and constructor can be altered.

6.2 Improving the Generated Code

During the rewrite process, a wide range of structural improvements have been made to the generated code. For one, several inconsistencies between SLCO's semantics and the implemented behavior have been resolved. The latter will be elaborated upon in Section 6.2.1. Furthermore,

the locking mechanism has been redesigned, such that the latter bases the lock operations on the data contained within the new locking data structure. The aforementioned revisions will be discussed further in Section 6.2.2. On top of that, the decision structures have been integrated. The implementation thereof will be introduced Section 6.2.3. Next, the concept of memory consistency will be discussed, including the effect the current lack thereof has upon the validity and reliability of the program itself. Additionally, several recommendations will be made on how memory consistency can be achieved by the use of the proper Java keywords and data structures. Further details will be given in Section 6.2.4. Finally, the use of the busy-waiting anti-pattern within a state machine's transition selection loop is reconsidered in Section 6.2.5.

6.2.1 Resolving Semantical Discrepancies

Several discrepancies were observed between the semantics of the SLCO model and the generated code that may lead to unexpected behavior, or in the worst case, erroneous code that cannot be compiled. In the SLCO language, every class may define its own set of variables which are scoped to the class itself and its underlying state machines. Moreover, the language allows for the definition of multiple instantiations of the same class, and if applicable, different initial values can be assigned to each class instantiation as well. However, in the original implementation, SLCO classes do not have a direct code representation: instead, all the classes defined in the model are merged into a single overarching class object. Consequently, the scope of the class variables is no longer class local, which leads to complications when two or more classes, including all other instantiations of the same class, define variables with the same name. Hence, it is possible that multiple classes share the same variable instance between SLCO classes, which does not only violate the requirement that class variables can only be seen by the class itself, but may also cause compile errors when two variables share the same name but have different types. On top of that, the lack of proper scoping makes it impossible to instantiate multiple standalone instances of the same class, which is in conflict with the capabilities defined by the SLCO language itself. The revised implementation resolves said discrepancies by strictly following the semantics of the SLCO model and treating the SLCO classes as separate objects that can be instantiated multiple times.

Furthermore, a nested class structure is used for the objects representing SLCO classes and state machines, such that the latter receives full access to the variables of the class without requiring a reference to the parent class. Additionally, state machines are now an extension of an enumeration type containing all of its states, granting the state machine direct access to its states in a short and simple notation. Said restructuring ensures that the defined states are defined within the scope of the state machine. Observe that the former is the opposite of the approach taken in the original implementation, where all states are stored within a single global enumeration.

Lastly, it needs to be noted that several components have not been implemented yet. Within the original code generator, no implementation is given for the conversion of communication channels, ports, delays and actions to Java code. Unfortunately, the aforementioned objects are likewise not supported by the revised implementation of the code generator, due to the fact that these features are considered to be of lower priority. Nevertheless, the implementation of a delay should be straightforward, and no implementation needs to be provided for actions since they are used primarily for verification or placeholder purposes. Furthermore, a verified implementation of the communication channels and ports exists within a previous implementation of the SLCO framework. Said implementation will be added to the code generator at a later date.

6.2.2 Locking Mechanism

The locking mechanism has been generalized such that a new independent instance thereof can be instantiated for each of the classes. The class variables that need to be locked are defined to be class local, and hence, it stands to reason that the locking mechanism needs to be class local as well. Furthermore, by making the locking mechanism class and instance local, it is possible to have multiple instances of the same class run completely independently from one another, since there

will be no interference of the locks requested by other instances of the same class. In addition, the code that sorts the locks to be acquired has been merged into the locking mechanism itself, since the sorting of locks is a mandatory step that ensures the absence of lock-deadlocks.

On top of that, several adjustments have been made to the locking mechanism, such that the latter bases the lock operations on the data contained within the new locking data structure. First, the use of control node methods, which are required to perform the locking operations at the required level of granularity and flexibility, will be elaborated upon in Section 6.2.2.1. Next, the rendering of the locking routine will be discussed in Section 6.2.2.2. Lastly, the atomicity of decision structures and the benefits and drawbacks thereof will be touched upon in Section 6.2.2.3.

6.2.2.1 Control Node Methods

Within the locking data structure, each statement has three positions at which lock operations can be performed, namely when entering and exiting the statement, where the latter is subdivided further into a success and failure branch exit respectively. The locking instructions associated with the aforementioned positions can be accessed through the locking nodes contained within the statement's atomic node, where the former dictates which locks are to be acquired and released at the given position. The atomic nodes that act as the foundation of the locking data structure form a nested structure, and as such, certain locking positions may possibly be located in the middle of a statement's execution. Therefore, locks need to be requested in-line, such that the locking mechanism follows the control flow of the program. Unfortunately, Java does not provide such an in-line notation, and as such, wrapper functions need to be introduced in which the lock operations can be executed prior to and after the execution of the target statement.

```
1  <contract>
2  private boolean <name>() {
3      <opening body>
4      if(<conditional>) {
5          <success closing body>
6          return true;
7      }
8      <failure closing body>
9      return false;
10 }
```

Listing 6.1: The structural pattern used for the creation of a control node method.

The control node methods are wrapper functions that replace a given statement by a method call. The structure of a control node method is given in Listing 6.1. Each control node is given a descriptive `name`, which is based upon the state, transition and statement that the control node method is associated with, followed by a numeric suffix to ensure uniqueness. The locking operations that need to be executed before entering the target statement are performed within the `opening body` of the method. Next, the value of the statement is evaluated in an if statement, where the `conditional` is equivalent to a text representation of the statement, in which sub-statements with locking operations are replaced by calls to their respective control node methods. The locking operations to be performed when exiting the statement are executed in the `success closing body` and `failure closing body` of the method, with each branch exit being dealt with by their respective closing body. Finally, the control node method is preceded by a `contract`, which contains human-readable information on the statement associated with the method.

The structural pattern for the creation of control node methods uses a conditional, and hence, it should only be used as a wrapper function for Boolean expressions. Atomic nodes are not generated for numerical expressions, and as such, the locking data structure guarantees said requirement by construction. On top of that, a control node method wrapper function is generated only when the associated statement has lock operations to perform, i.e., at least one of the acquisition and release fields of the locking instructions contained within the statement's atomic node should be

non-empty. Otherwise, the statement is included in-line, wherein further sub-statements have been replaced by calls to their respective control node methods if appropriate.

6.2.2.2 Rendering a Locking Routine

The locking operations to be performed during a locking routine are dictated by a locking node's locking instructions, and as such, a template has been created to render the latter. The locking process consists of three steps. First, a series of locks are requested during one or more locking phases. To attain correct atomic behavior, it is required that certain variables, i.e., those used within the index of an array variable, are locked prior to usage in the evaluation of the latter. Therefore, all locks in a phase need to be acquired prior to moving on to the next phase. Furthermore, it is not guaranteed that the locks in a locking phase adhere to the strict lock ordering, due to the fact that some of the indices used in the phase may be non-constant. As such, the list of lock identities targeted by a phase needs to be sorted prior to acquiring the locks.

The process of acquiring a series of locks is characterized as follows. To start with, the appropriate lock identities are calculated and added to a target array, where the lock identity of a target is defined to be the sum of the lock identity of the target value and target index. Subsequently, the target array is sorted on value in increasing order to enforce the strict lock ordering, after which the locks are acquired in the resulting order. Internally, the atomicity of a target variable instance is achieved through the use of reentrant locks, where each locking identity is assigned its own individual lock instance. Reentrant locks ensure that a thread halts if it does currently not possess the ownership over the target lock object. On top of that, the ownership over a reentrant lock can be acquired multiple times by a thread. The latter is a necessary property, since it may occur that two distinct lock targets evaluate to the same lock identity, i.e., $x[0]$ and $x[i]$ refer to the same target variable instance if i is equal to zero.

Furthermore, the values of variables may be altered by the defined statements, and as such, the target lock identity of each lock request is stored in a traceability array upon the acquisition of a lock to a variable, such that the original lock identity target can be recovered reliably upon its release. Within the aforementioned traceability array, the index is equivalent to the lock requests identity, and the associated value is the lock identity that has been used to acquire the variable. For the sake of efficiency, both the target and traceability array are reused, i.e., the arrays used are only defined once at the scope of the state machine, with the values therein being rewritten during every locking operation. Consequently, Java's garbage collection has less of an impact upon the program, which in turn translates into a more consistent level of performance.

The locking routine will proceed to the second step once all locking phases have been completed. During the second step, locks are acquired over target objects that have been subjected to weak unpacking. Through the unpacking operation, it is ensured that the appropriate locks are already held by the thread, i.e., during the first step of the locking routine, the array associated to the unpacked lock is guaranteed to have been locked in its entirety. Therefore, lock-deadlocks are no longer a risk, and as such, no strict ordering needs to be enforced. Similarly, the use of phased locking is no longer required, and as such, the list of target locks can be acquired as-is.

The locking routine is concluded by the third step, in which the state machine relinquishes ownership over a given set of variables. As stated previously, the state of variables might have been altered by the executed statements. The recovery of the original value of a variable by computational means is infeasible, and as such, the list of locks to relinquish is based entirely on the values stored for the given targets within the traceability array. Consequently, it is guaranteed that the acquisition and relinquishment of an object's ownership is performed through the same reentrant lock, and as such, the integrity of the locking mechanism is ensured. Lock-deadlocks cannot occur when releasing locks, and as such, no strict unlock ordering needs to be adhered to.

Optionally, the locking routine may include statements that verify whether the target state machine has ownership over the variables used within a statement before they are read of written to.

The ownership check is included within the opening body of the statement's control node method, and as such, it is assured that the check is performed prior to evaluating the conditional or any control node methods contained therein. The check ensures that an exception is thrown if the thread does not own the necessary locks to access the variable in question.

To conclude with, several alternative constructs have been considered to achieve synchronization between threads by alternative means. For one, it may be feasible to use Java's synchronization blocks and monitor objects instead of reentrant locks when the locking is performed at a base variable or statement level scope. In addition, the use of atomic types in Java may be considered to guarantee atomicity for less complex operations. Another promising option is to use separate locks for read and write access to achieve a higher degree of concurrency between threads, given that it is safe for multiple threads to read a variable at the same time, as long as the value of the variable remains unaltered. Regrettably, the aforementioned options could not be explored further due to a lack of time, and as such, they are remain topics for future research.

6.2.2.3 Atomicity of Decision Structures

Within the code generator, the decision structures are treated as atomic operations. The atomic node patterns for decision nodes have been defined within Section 5.4.1.5, where a decision may be taken through a sequential or random-pick approach respectively. However, observe that the sequential or random-pick approach classifications are not an exact match to the decision types defined in Section 4.3, where a decision over a set of options is defined to be deterministic and non-deterministic. The classifications differ in the sense that Chapter 4 focuses on defining the available types of decisions, while Chapter 5 focuses on defining the manner in which a decision may be performed. Consequently, a mapping needs to be created between the two definitions, such that the appropriate atomic pattern can be applied for a given decision node.

During a deterministic decision, the invalidity of an option implies that the remaining options in the decision node should be visited. Therefore, a sequential approach is required when making a deterministic decision. On the other hand, no requirements are set upon the process of making a non-deterministic decision, besides the fact that an arbitrary element should be chosen out of the list of available options–specifically, observe that randomness is not a requirement. As a result, both the sequential and random-pick approach to decision making are applicable for nodes of a non-deterministic type, and as such, the atomic pattern to be applied within the locking data structure construction depends entirely upon the expected behavior. The latter can be configured through the command-line arguments that will be discussed further in Section 6.3.

Furthermore, it needs to be noted that at times, it may be more optimal to only partially treat decision structures as atomic operations. In particular, it is possible to use a sequential approach for decision making within a non-deterministic root node of the decision structure without the requirement that the ownership of a variable need to be carried over between decisions. By construction, the root node of a decision structure is defined to be a decision node that is non-deterministic. In addition, the non-deterministic root node does not have a guard statement, and as such, it is guaranteed that there are no lock operations to perform at the entry and exit locking nodes of the root node. As such, the application of a random-pick atomic node pattern to a sequential approach is valid in the root node, since each option in the root decision node is ensured to be atomic in isolation, i.e., all locks required for the execution of the option are acquired and released within the scope of the latter's own atomic node.

The primary benefit of the approach described above is that it facilitates a higher degree of concurrency between threads, since the state machine will relinquish ownership over variables between checking options within its root decision node, which in turn ensures that the other state machines gain more opportunities to access said variables. The goal of a non-deterministic decision is to pick an arbitrary option, and as such, it can be argued that the decision should be insensitive to the change of variable values between options. Observe that the latter is not applicable to deterministic decisions, since by definition, the execution of an option fully depends upon the

evaluation of its predecessor, i.e., an option should not be visited if its predecessor found and executed an active transition: the aforementioned relation can only be guaranteed if the variables remain unaltered. Additionally, it needs to be stressed that the approach discussed above cannot be applied to nested non-deterministic nodes, since it cannot be guaranteed that its options are atomic in isolation, i.e., the options may depend upon the acquisition and release of locks prior to or after its execution. As such, it is strictly necessary to use the sequential atomic node pattern, since otherwise, the integrity of the locking data structure cannot be guaranteed.

On the other hand, it could be desirable for the root decision to remain atomic in its entirety. If the entire structure is atomic, the priorities set on transitions are guaranteed to be respected, since the values of the variables used within the decision cannot change while the state machine is picking a transition. For the sake of consistency and simplicity, the latter approach has been incorporated into the final implementation. Nevertheless, the ideas brought forward within the scope of this section may warrant further investigation.

6.2.3 Decision Structures

In the previous section, a clarification has been given on how the decision structures described in Chapter 4 relate to the decision making approaches introduced in Chapter 5. Next, the implementation of the aforementioned approaches will be elaborated upon. First, it needs to be noted that the atomic node patterns provided for the approaches dictate the expected behavior that the implementations should adhere to, i.e., the control flow of the implementations needs to match the behavior portrayed within the given patterns. Furthermore, within Section 6.2.2.3, an optimization has been discussed that allows for only part of the decision structure to be treated as an atomic operation, with the latter is to be achieved by applying the random-pick atomic node pattern to a sequential approach to decision making. As such, it would be beneficial if the implementation of the sequential approach is designed to be compatible with the random-pick atomic node pattern ahead of time, with the aim to facilitate the potential application thereof.

The chosen implementations for the decision making approaches are as follows. The sequential approach is implemented through a series of if-statements, where each option is visited in succession until an active transition has been found and executed. Note that said if-statements are not generated by the sequential approach: instead, it the options contained therein are required to render their own guard statement. The latter improves the maintainability of the code, since it allows for the conditional to be part of the code template of the associated object. Furthermore, the random-pick approach is implemented through a switch statement, in which the option to be executed is selected through the use of a random number generator. Note that the case distinction within the random-pick approach is omitted if a decision is made over a single option.

Observe that for both approaches, it may occur that the active transition are part of a nested decision. According to the atomic node patterns of the decision nodes, the decision process should finish once an active transition has been found, and therefore, a mechanism is required that gracefully terminates the selection process, including the situations where nested decision nodes are present within the decision structure. The return statement of a method fulfills the aforementioned purpose–as such, the decision structure is to be contained within a single method. In addition, the return function is to be called if a transition is active and completed its execution. Consequently, the success branch of a transition is bound to terminate the decision process, and as such, the presence of else statements between options is no longer mandatory.

Lastly, it may occur that a conditional is rendered more than once within the same branch of the decision structure: for example, it may occur that the guard of a transition is equivalent to the encompassing guard of a decision node that precedes it. As such, optimizations are required that mark a guard statement as superfluous if the statement is bound to hold within the context created by its preceding conditionals. Unfortunately, the necessary adjustments needed to achieve the latter remain to be implemented at this time.

6.2.3.1 Alternative Code Templates

In addition to the included if-then statements, several other options exist to express deterministic decision structures in Java, such as case distinctions and polymorphism. Case distinctions can be used in decision structures where the majority of options are equalities with the same left-hand side expressions, while polymorphism can be used in mappings to express more complex decision behavior. On top of that, the use of if-then statements can be refined further by utilizing the else branch within the construct to cover an option that, given the state of the other options in the decision, is bound to hold true. Consequently, the conditional of one of the decisions in the decision structure does not need to be evaluated, which in turn makes the code more efficient.

Unfortunately, the aforementioned alternatives are not included in the implementation because of several reasons, prime amongst them being a lack of time. For the inclusion of if-then-else statements, it needs to be verified whether the disjunction of the guards of all options always holds true, which can be achieved through the application of SMT solving. Case distinctions are similar to if-then(-else) statements in overall structure, but more complex logic is required to determine whether the given decision structure fits the case distinction model or not. A proof of concept for if-then-else statements and case distinctions has been included in a draft implementation, but due to changes made in the locking mechanism, the prototype regrettably had to be scrapped. Furthermore, the implementation of polymorphism introduces similar issues, since the key used in the mapping will also have to satisfy several constraints. However, polymorphism may also have a detrimental effect upon the overall performance of the program, given that by construction, mappings are not as efficient as conventional conditionals. Moreover, the construction of keys for the mappings requires a great deal of care, since the creation of new objects can lead to performance issues caused by the unpredictable behavior of the Java garbage collection.

6.2.4 Concurrency and Memory Consistency

In the original implementation of the code generator, no effort has been made to deal with the concept of memory consistency between threads. Due to several performance optimizations made in the Java Virtual Machine (JVM) compiler in Java, it is not guaranteed by default that a change to a variable in one thread is immediately visible to another thread. Hence, the values of class variables seen by the state machine threads are not guaranteed to be consistent with one another, which may potentially result in a memory consistency error [24]. The lack of proper memory consistency is undesirable, since it may lead to unexpected behavior.

Generally, memory consistency errors can be prevented by adding the `volatile` keyword to the variable's declaration, which will inform the compiler that the variable should not be optimized. Hence, the code generator needs to ensure that all of the class variables are marked as volatile. However, the volatile keyword will not always be sufficient to attain full memory consistency, since the keyword only covers the actual reference pointing to the variable and not the variable's value itself. As such, complex data types like lists and arrays require additional mechanisms to attain memory consistency. Several options exist to make the values in a collection type volatile. An often suggested approach is to use the `AtomicIntegerArray` and `AtomicReferenceArray` classes instead, which both guarantee that individual fields are accessed with volatile and atomicity semantics, with the disadvantage that the atomicity is superfluous due to the code generator's internal locking mechanism. Alternatively, a wrapper class can be created that internally marks the target variable as volatile. However, note that this alternative need to be handled with care, since wrapper classes will only work appropriately when the reference pointer to the wrapper class remain unchanged after initialization.

In the current iteration of the revised implementation, all primitive type class variables are properly marked as volatile. Unfortunately, non-primitive types like arrays have not yet been updated to a construct that ensures memory consistency due to time constraints. Moreover, additional testing is required, since it is currently unknown which of the two suggested solutions will perform best

in regards to performance and efficiency. Nevertheless, it is believed that the integration of either of the two options will be straightforward.

6.2.5 Busy-Waiting Primary Loop

In computer science and software engineering, busy-waiting is a technique in which a process waits for a condition to hold true by repeatedly checking the value of said condition. Within the original implementation of the code generator, a busy-waiting loop is used to pick transitions, even though none may be active at the given moment in time. Busy-waiting is a well known anti-pattern in programming, and therefore, it is preferable that the use thereof is avoided if possible.

The resource inefficiency of the busy-waiting technique can be mitigated through a delay function, which ensures that the thread in question enters a sleep state for a predetermined amount of time after every iteration performed by the busy-waiting loop. Alternatively, a wait-notify system can be used, in which a thread halts its own execution until it receives a signal that it may proceed. During the exploratory stage of the project, a small-scale isolated experiment has been conducted with a wait-notify system that wakes up a thread if a variable it relies upon in its current state changes value. The implementation achieved this by creating a central management thread that, upon receiving a value change message, signals all threads subscribed to the affected variable, indicating that their execution may be resumed. However, note that great care needs to be taken with the management of said signals, since the improper timing thereof may result in a program-wide deadlock due to all threads ending up in a sleep state.

On the other hand, the use of busy-waiting in the transition selection loop is beneficial to the state machine's responsiveness to value changes committed by other state machines in the parallel program. The operations performed by the state machines are single statements, which given due their simplicity happen in quick succession. Therefore, for every transition to have a fair chance to fire, they need to be equally reactive to changes made to the state of the program. Having state machine threads go to sleep while waiting for variable changes, which in turn makes the program less reactive, may unfairly hinder the transitions contained therein from firing, since the brief scope the transition needs to react in may have passed before the thread firing said transition has had a chance to wake up. Furthermore, it may occur that certain state machines in the program always have active transitions, and as such, will not go to sleep. Consequently, the aforementioned state machines may inadvertently consume a disproportionate amount of computational resources compared to other state machines, which may be undesirable. Given the concerns that have been described above, it is clear that fairness is a key factor to consider when seeking for an alternative for the busy-waiting technique, such that the responsiveness of state machines is guaranteed.

Unfortunately, the exact approach to be taken to tackle the busy-waiting problem remains undecided, and as such, no changes have been made in the revised implementation of the code generator pertaining to the mitigation of busy-waiting behavior. Nevertheless, the inefficiency of the latter is undesirable, and as such, the issue merits further investigation in the future.

6.3 Command-Line Arguments

Several command-line arguments have been defined within the Slco 2.0 to Java code transformation project through which certain components of the code generator can be configured. All of the command-line arguments are considered optional, except for the argument that points the program to the file that contains the model that is to be transformed to a Java program.

model_path The path to the Slco 2.0 model that needs to be transformed to a Java program.

The optional command-line arguments have been subdivided into several categories that match the structure of this document, namely the categories that pertain to the decision structures, locking mechanism, formal verification and the performance analysis components of the code generator. Each of the aforementioned categories will be discussed within their own section.

6.3.1 Decision Structures

The first category of command-line arguments to be discussed are those related to the deterministic structures introduced in Chapter 4. Throughout this work, several options are provided that have an influence upon the decision structure construction and rendering procedures, including but not limited to the solver configurations described in Section 4.7. The command-line arguments provided for the decision structures are the following.

`-use_random_pick` Indicate that a random-pick approach should be used for the resolution of non-deterministic decisions (Section 6.2.3). Within the code, the non-deterministic decision will be taken through a case distinction that selects an option based on a random value.

`-no_deterministic_structures` Disable the construction of the deterministic decision structures. Instead, all transitions will be contained within the root non-deterministic decision node, which forces the state machine to always select an arbitrary transition.

`-decision_structure_solver_id=`i Select the decision structure solver configuration that is to be used (Section 4.4). The configuration is a combination of a solver strategy (greedy or optimal) and an overlap constraint (basic, equality or subset). The value of $i \in [0, 5]$ is the identity of the combination to be used, with the available options being the following:

0. A greedy basic solver meeting minimum requirements (Equations 1a, 6a)
1. A greedy solver that merges equal transitions (Equations 1a, 6b) (default)
2. A greedy solver that creates a subset-based structure (Equations 1a, 6c)
3. An optimal basic solver meeting minimum requirements (Equations 1b, 6a)
4. An optimal solver that merges equal transitions (Equations 1b, 6b)
5. An optimal solver that creates a subset-based structure (Equations 1b, 6c)

6.3.2 Locking Mechanism

Furthermore, several configuration options have been included for the locking mechanism introduced in Chapter 5. One may for example control the locking mode, which dictates the scope of the locks that will be generated. Note that several of the given command-line arguments are mutually exclusive, i.e., only one should be given at any time.

`-lock_array`[1] Activate the locking mode in which locks are variable-scoped, i.e., array variables are locked in their entirety through a single lock object instead of their individual indices (Section 5.4.2.1).

`-statement_level_locking`[1] Activate the locking mode in which locks are statement-scoped, i.e., all statements that use a class variable are to use a single lock object, which ensures that at most one statement has access to class variables at any moment in time (Section 5.4.2.1).

`-no_locks`[1] Activate the locking mode in which no locking is performed. Note that this option should only be included for experimental purposes, since it generates faulty code. The generated code will not guarantee the atomicity of statements requirement (Section 5.4.2.1).

`-verify_locks` Include statements within the code that verify whether the target state machine has ownership over the variables used within a statement before they are read of written to (Section 6.2.2).

`-visualize_locking_graph` Show a visualization of the finalized locking node graph, which contains the structure of the graph and a list of lock objects that are to be requested and released within the given nodes.

`-fair_locking` Use reentrant locks that adhere to a fairness policy.

[1] The arguments activating locking modes are mutually exclusive.

6.3.3 Formal Verification

Next, the command-line arguments associated with the formal verification of the generated code will be introduced. The formal verification process, to be discussed in Chapter 7, does not have any settings that need to be configured, and as such, only a single command-line argument is provided which indicates that formal verification annotations need to be included.

`-vercors_verification` Generate a Java program that includes VerCors statements, such that the generated code can be formally verified through the VerCors tool (Chapter 7).

6.3.4 Performance Analysis

Lastly, command-line arguments are provided that enable and control the inclusion of performance measurements which will be discussed in Chapter 8. During the performance testing, it is important that the behavior of the program is influenced as little as possible. The majority of the arguments are used to configure the logger used for the logging driven performance measurements, such that the logger can be tuned to achieve the aforementioned goal.

`-performance_measurements` Generate a Java program that includes performance measurements through which the performance of the generated code can be analyzed (Chapter 8).

`-iteration_limit=`i[2] Set a limit on the number of iterations that the main loop of each state machine may make. The value of i will be used as the limit, where the default value of i is defined to be 10000. The limit is applied only when the argument is included.

`-running_time=`i[2] Set a time limit on the execution of each state machine. The value of i, representing a given number seconds, will be used as the time limit, where the default value of i is defined to be 60 seconds. The limit is applied only when the argument is included.

`-log_file_size=`s Define the rollover size for log files generated by the logging driven performance measurements. The value s is a string defining the number of megabytes, with the default size being 100MB. A larger value results in a higher memory usage, while lower values have a detrimental impact upon the overall performance of the logging routine.

`-log_buffer_size=`i Define the ring buffer size to be used by log4j framework during the gathering of the logging driven performance measurements. The value i determines the number of slots in the ring buffer, with the default number of slots being $2048 \cdot 2048 = 4194304$. The value needs to be tuned in such a manner that the logger can keep up with the number of logging entries without having the ring buffer fill up completely.

`-compression_level=`i Define the compression level that should be used by the log4j framework during the gathering of the logging driven performance measurements. The value $i \in [0, 9]$ defines the compression level to be used during the compression of rollover files, where i has a default value of 3. A compression level of 0 is no compression, 1 is the fastest compression, and 9 is the most space efficient compression. The use of compression causes the program execution to pause, with the length of the pause being longer for higher levels of compression.

`-package_name=`s Define the name of the Java package that the test files should be part of, such that multiple classes with the same name can be included within a project without complications. The default value of s is defined to be the empty string.

[2] The arguments setting limits on the primary loop are mutually exclusive.

Chapter 7

Formal Verification

The formal verification of all model transformations is considered to be a fundamental aspect of the Slco framework, and as such, a formal verification of the conversion from an Slco model to Java code is required as well. Within the Slco framework, several techniques are used to perform the formal verification of an Slco model of transformation thereof. For instance, the desired functional properties of an Slco model can be verified by transforming the model to an mCRL2 model. Subsequently, the desired properties can be verified through the mCRL2 tool set [9] and its model checking [2] capabilities. Furthermore, it is possible to verify specific user-defined transformations through a novel transformation verification technique [12, 26] implemented in the Refiner tool [27]. Lastly, the preservation of properties for other Slco model transformation applications can be achieved by verifying the resulting Slco model through mCRL2.

The code generator component cannot be verified through mCRL2, and as such, a different approach is required for the formal verification thereof. Instead, the functional correctness of the generated Java code is verified through the use of the VerCors [5] verification tool set. Note that the formal verification process described within the scope of this chapter will not be complete– regrettably, a full verification is not possible within the scope of the thesis due to the associated time limitations. Instead, the chapter will focus on exploring manners in which certain aspects of the generated code can be verified, such as its structure and the correctness of the locking mechanism rendered therein. Moreover, it needs to be stressed that the code generator will not be mechanically verified by the described process, i.e., the correctness of the generated code is determined based solely upon the input and the result of the code generating process, and not on the verified correctness of the code used within the code generator itself.

This chapter will be structured as follows. First, the VerCors tool set will be introduced in Section 7.1, which is the verification tool set used to validate the generated Java code. Next, the process of verifying the generated code will be discussed in Section 7.2, which focuses on the verification of the code's structure and locking mechanism. Lastly, the issues and challenges faced during the implementation of the verification process will be elaborated upon in Section 7.3.

7.1 VerCors Tool Set

The Java code generated by the Slco framework will be verified through VerCors [5], which is a verification tool set that uses static analysis to prove partial correctness of a given piece of code. The expected behavior of verification target is expressed through a series of annotations that are to be included within the generated code: the syntax to be used within the annotations will be discussed briefly in Section 7.1.1. The VerCors tool set is a particularly effective for the verification of parallel and concurrent programs, due to the use of permission-based Concurrent Separation Logic (PBSL) as its logical foundation. The latter is a program logic that restricts a thread's

ability to read from, or write to, shared memory, based on the set of (fractional) permissions that the thread has in its possession. The benefit of PBSL is that the properties of data-race freedom and memory safety are simple to verify, since they are a consequence of the soundness argument of the logic. Therefore, the use of VerCors tool set seems to be appropriate for the verification of the Java code generated by the Slco framework, given the latter's primary goal of simplifying the development of formally verified concurrent programs. Additionally, the PBSL program logic may potentially be used to formally verify the locking component of the generated code. However, it needs to be noted that the use of the PBSL program logic does incur a notational overhead, since all of the required permissions need to be managed explicitly by the included annotations.

The Vercors tool set verifies each method in isolation, i.e., the context in which the method in question is called is not considered: instead, the validity of a method is determined entirely upon its contract and body. The contract specifies the pre-conditions that must be met prior to calling the method, and the post-conditions that are to hold upon the method finishing execution. Subsequently, it is verified whether the body guarantees that the post-condition holds if an input is given that satisfies the defined pre-condition. As stated previously, VerCors checks for partial correctness, and as such, the post-conditions are guaranteed to hold if the program terminates. However, observe that VerCors does not prove whether the program terminates.

Several alternative static verification tool sets exist that are annotation based, including but not limited to tools such as VeriFast [20], Dafny [21], OpenJML [8], KeY [3] and VCC [7]. However, the majority of the aforementioned tools have their limitations. First of all, tools such as Dafny, OpenJML and KeY are limited to the verification of sequential programs, and therefore, cannot be used to effectively verify parallel and concurrent language constructs. Secondly, VCC is language-dependent, and hence, it can only be used to verify programs written in the C programming language. Conversely, VerCors and VeriFast are both capable of verifying multithreaded programs written in Java or C, and therefore, both are considered to be fitting candidates. Nevertheless, VerCors has already been used to verify the original implementation of the code generator, and as such, VerCors has been chosen over VeriFast for the sake of consistency and familiarity.

7.1.1 Specification Syntax

Within the scope of this section, a brief overview will be given on the syntax that should be used to express specifications within the annotations. To start with, the target specifications are embedded into the target code through special comments, namely inline comments starting with `//@` or block comments wrapped by `/*@` and `@*/`. Furthermore, several keywords are provided through which the desired behavior can be described, which are subdivided into two categories: keywords pertaining to the method contract, and keywords to be used within the method body.

The keywords to be used within the scope of a method contract will be discussed first. As stated previously, the pre-conditions and post-conditions of a method are specified within its contract. The desired specifications are expressed through the `requires` and `ensures` keywords respectively, followed by the condition that is required to hold. The contract may define multiple pre-conditions and post-conditions. The `context` keyword can be used as a short-hand notation if the given pre-condition and post-condition are equivalent, i.e., the `context` keyword includes the given condition as both a pre-condition and post-condition of the method. The return value of a (non-void) method can be referred to through the `\result` keyword. Additionally, the original value of an expression `expr` prior to the execution of the method can be accessed using the expression `\old(expr)`. Lastly, additional parameters and return values can be added to a method through the `given` and `yield` keywords respectively, which in turn can be accessed through the keywords `with` and `then` upon making a call to the target method.

Next, the keywords associated with the body of a method are introduced. A loop invariant can be defined through the `loop_invariant` keyword followed by the condition that needs to hold prior to entering and after each iteration of the target loop. The loop invariant is to be placed above the loop header. The previously discussed `context` keyword can be replaced by the keyword

`context_everywhere` if the given condition is to be used as a loop invariant for all loops within the method body. Assertions can be made through the `assert` keyword followed by the expression that needs to be asserted: assertions serve as an intermediate step with the aim to guide the verification process in the event where VerCors cannot derive the post-conditions based solely on the given code and established facts, such as pre-conditions and assumptions. An assumption can be made through the `assume` keyword followed by the expression that needs to be taken for granted for the remainder of the method body, regardless of the state the program is in, i.e., the assumption will be treated as a fact even if it is contradictory to the state of the program. Lastly, note that assumptions cannot be revoked, and as such, they need to be handled with care.

The expressions that can be used to define specifications are an extension of the expressions available in the target language, i.e., an expression that is valid in Java can be used in a specification, under the condition that the expression does not incur side-effects on the program state. On top of that, VerCors supports the use of implications, existential quantifiers and universal quantifiers in specifications, where the quantifiers simplify the process of reasoning about array variables.

Finally, the notation used to describe and reason with permissions will be elaborated upon. A permission is referred to through the predicate `Perm(`v`, `q`)`, where v is the target value and $0 < q \leq 1$ is the fractional permission over v. The ownership system in VerCors enforces that the sum of permissions fractions for all active permissions over a memory location v cannot exceed one. As such, if a thread has a fractional permission of one over v at a given point in time, it is implied that no other thread can have any active permissions over v, since otherwise the sum of fractional permissions would exceed one. Consequently, a fractional permission of $q = 1$ denotes write access, while $q < 1$ indicates read access. Any permission with a fractional permission greater than one will result in a verification error. Two permissions `Perm(`v`, `q_1`)` and `Perm(`v`, `q_2`)` can be merged into a permission `Perm(`v`, `q_1+q_2`)` using the separating conjunction operator (`**`), i.e., the formula `Perm(`v`, `q_1`)` `**` `Perm(`v`, `q_2`)` represents the permission `Perm(`v`, `$q_1 + q_2$`)`. Similarly, a permission predicate `Perm(`v`, `q`)` can be split into the formula `Perm(`v`, `$q\backslash 2$`)` `**` `Perm(`v`, `$q\backslash 2$`)`. Specifically, observe the use of a backslash in the permission fractional instead of a forward slash. The permission predicates required to execute a method are to be listed within the pre-conditions and post-conditions of the method's contract. Lastly, the concept of permission leaks needs to be discussed. A permission leak occurs when the set of permissions included within a method's pre-condition is not equivalent to those included in its post-condition, i.e., the method gives away or is granted certain permissions within the scope of its execution. Permission leaks need to be handled with care, since the presence thereof may prevent the successful verification of a model.

7.2 Verifying the Generated Java Code

The formal verification of the generated code is characterized as follows. Ideally, the verification process should be built upon the code that is rendered during the normal mode of operation, i.e., the existing code should not be overwritten when rendering the verification model, since otherwise, it cannot be guaranteed that the baseline code is valid. Instead, the verification model should only extend upon the original model by adding additional code to the latter in a manner that the original behavior of the program is not affected. Unfortunately, due to several tool set limitations, the latter has deemed to be impossible to achieve in practice, and therefore, overwrites need to be made to create a verifiable model. Consequently, the aim should be to overwrite the original behavior as sparingly as possible, to ensure that the baseline code and verification model are as closely related to each other as possible. The latter is facilitated by having the verification model rendering module(s) be a subclass of the original generator, such that it can be ensured that both versions are based upon the same code foundation.

Originally, it was the intention to create a single verification model that validates all of the required characteristics, which includes structural properties and the requirements related to the atomicity of statements. However, the latter proved to be infeasible to implement due to the strictness

of the VerCors permission system: the locking mechanism releases locks as early as possible, and as such, the post-condition of a method cannot be verified, because the method may have relinquished ownership over the variables used therein within its body. Consequently, several verification models will need to be created that focus upon verifying different aspects of the generated code. To start with, the structural characteristics of the generated code will be formally verified, such that it is ensured that the rendered code executes the behavior expressed by the input model appropriately. The structural verification of the model will be detailed further in Section 7.2.1. Next, the properties associated with the atomicity of statements requirement are verified, such that it can be ensured that the revised locking mechanism functions properly. The verification of the locking mechanism will be discussed in Section 7.2.2.

Once again, it needs to be stressed that the revised code generator is and will not be mechanically verified, since the latter is too complex of an endeavor to achieve within the scope of a thesis project. In addition, the formal verification process described within the following sections will by no means be perfect, which the collection of issues and challenges faced during the implementation thereof, as described in Section 7.3, will attest to. Nevertheless, the provided verification models may provide valuable insights that can be acted upon in the future.

7.2.1 Structural Verification

Within the first step of the formal verification of the code generator, it is verified whether the statements described within the input model are translated correctly into Java code. In other words, it is verified whether the execution of a given target in the input model and its counterpart in the generated Java code lead to the same outcome. The correctness of each target is verified in isolation, i.e., the effect of parallelism and the atomicity of statements requirements are not verified within this step of the verification process. As such, the locking instructions are excluded from the verification model–the atomicity requirements and the verification of the locking mechanism will be performed in Section 7.2.2 instead.

Consequently, the structural verification is performed under the requirement that the verification target has write permissions over all variables of its parent class. The latter is achieved by including a context statement that requires and ensures the given set of class variable permissions. On top of that, the verification target has full ownership over all variables of its parent state machine, since the latter is considered to have a local scope. For array variables, it needs to hold that all target indices are within the bounds of the target array variable, since otherwise, the model will fail to verify on the ground of having insufficient permissions. Therefore, the inclusion of explicit bound checks on indices is required to successfully verify the generated code. The location and execution of the bound check within the verification model depends on the type of object that is being validated. For expressions, the range check is included as a requirement within the contract of the associated method, i.e., the model will fail to verify if the method is called with an input that does not satisfy the required bound checks. The aforementioned range checks are included within the contracts of control node methods and transitions, where the latter pertains only to variables used within the guard statement thereof. For assignments, the range checks are treated as assumptions, i.e., it is assumed prior to the execution of an assignment that the values of all indices used therein are within the required range. Analogously, the necessary range check assumptions are included when calling a transition within a decision structure. In VerCors, assumptions need to be handled with great care, since the conditions expressed within an assumption hold for the remainder of the method's body, i.e., the condition will remain to hold true even if the values of the variables contained therein are altered by subsequent statements. As such, the assumptions made for the range checking are performed within separate methods, with the aim of limiting the scope in which the assumption holds: the assumption is made within the method's body, with the outcome thereof being included as a post-condition within the method's contract. An example of a range check method with the appropriate VerCors annotations is given in Listing 7.1.

The inclusion of the aforementioned range checks is counter-intuitive, since the presence thereof

```
/*@
// Require and ensure full access to the target class.
context Perm(c, 1);

// Require and ensure that the state machine has full access to its own array variables.
context Perm(y, 1);

// Require and ensure that the state machine variable arrays are not null and of the right size.
context y != null && y.length == 2;

// Require and ensure the permission of writing to all state machine variables.
context Perm(y[*], 1);
context Perm(j, 1);

// Require and ensure that the state machine has full access to the arrays within the target class.
context Perm(c.x, 1);

// Require and ensure that the class variable arrays are not null and of the appropriate size.
context c.x != null && c.x.length == 2;

// Require and ensure the permission of writing to all class variables.
context Perm(c.x[*], 1);
context Perm(c.i, 1);

// Require and ensure that all of the accessed indices are within range.
ensures 0 <= c.i && c.i < 2;

// Ensure that all state machine variable values remain unchanged.
ensures (\forall* int _i; 0 <= _i && _i < y.length; y[_i] == \old(y[_i]));
ensures j == \old(j);

// Ensure that all class variable values remain unchanged.
ensures (\forall* int _i; 0 <= _i && _i < c.x.length; c.x[_i] == \old(c.x[_i]));
ensures c.i == \old(c.i);
@*/
private void range_check_assumption_t_0_s_2() {
    // Assume that all of the accessed indices are within range.
    //@ assume 0 <= c.i && c.i < 2;
}
```

Listing 7.1: An example of a range check method that contains all of the appropriate VerCors annotations. The method ensures that the value of class variable i is within the bounds of array variable x with $|x| = 2$, i.e., it is ensured by the method that $0 \leq i < 2$ holds.

may influence the validity of the verification, i.e., the range checks may create a context in which a given Boolean expression is bound to hold true, while under normal circumstances, the latter may evaluate to false as well. Unfortunately, the range checks are necessary, since it cannot be guaranteed that the input model performs all of the range checks explicitly. On top of that, the input model is not guaranteed to be sound, i.e., the model may express behavior in which an index exceeds the bounds of an array variable–the input model is not validated by the code generator, and it is not the aim of the structural verification to do so. Consequently, range checks need to be included, since otherwise, the generated code cannot be verified at all due to the occurrence of insufficient permissions violations. Furthermore, the listing of range check requirements within the method contracts is necessary, since the validation of the resulting state of the latter is subjected to the same permission related complications. Due to the latter, it is also impossible to perform all of the range checks as localized assumptions affecting only the target variable, since the VerCors tool set does not allow for assumptions to be made in method contracts. Regrettably, the inclusion of range checks is hence deemed to be the only workable solution.

The structural verification model verifies the following set of targets. For each control node method, it is verified whether the return value of the latter is equivalent to the Boolean expression

```
/*@
// Require and ensure full access to the target class.
context Perm(c, 1);

// Require and ensure that the state machine has full access to its own array variables.
context Perm(y, 1);

// Require and ensure that the state machine variable arrays are not null and of the right size.
context y != null && y.length == 2;

// Require and ensure the permission of writing to all state machine variables.
context Perm(y[*], 1);
context Perm(j, 1);

// Require and ensure that the state machine has full access to the arrays within the target class.
context Perm(c.x, 1);

// Require and ensure that the class variable arrays are not null and of the appropriate size.
context c.x != null && c.x.length == 2;

// Require and ensure the permission of writing to all class variables.
context Perm(c.x[*], 1);
context Perm(c.i, 1);

// Ensure that the result of the function is equivalent to the target statement.
ensures \result == (c.i >= 0);

// Ensure that all state machine variable values remain unchanged.
ensures (\forall* int _i; 0 <= _i && _i < y.length; y[_i] == \old(y[_i]));
ensures j == \old(j);

// Ensure that all class variable values remain unchanged.
ensures (\forall* int _i; 0 <= _i && _i < c.x.length; c.x[_i] == \old(c.x[_i]));
ensures c.i == \old(c.i);
@*/
private boolean t_SMC0_0_s_0_n_0() {
    return c.i >= 0;
}
```

Listing 7.2: An example of the structural verification of the control node method of expression $x[i] \geq 0$ that provides all of the appropriate VerCors annotations.

contained therein, i.e., it needs to be ensured that the result of the method is equivalent to expression that has been encapsulated by the method. Furthermore, expressions are not allowed to alter the values of variables, and as such, the verification model asserts whether the value of all variables within the associated class and state machine remain unchanged. An example of the structural verification of a control node method is provided in Listing 7.2. The verification of a transition is more involved. For one, the method needs to return preemptively if the guard expression of the transition does not hold true, with the desired return value being false in this particular case. Oppositely, the method should return true if all statements contained within the transition have been executed successfully. As such, it needs to be verified that the return value of the method is equivalent to the value of the guard expression. The values of variables used within the guard expression may be altered by subsequent assignments, and as such, the evaluation of the guard statement is stored within a temporary value for later use.

Furthermore, the assignments within a transition may not be executed if the guard expression does not hold. Consequently, if the result is false, it needs to hold that the values of all variables within the target's parent class and state machine remain unaltered. The latter can be expressed through an implication. If the guard expression holds true, it needs to follow that all assignments within the transition have been executed successfully. For an assignment to be translated correctly, it must hold that the value of the target variable is equivalent to the assignment's right-hand side

expression after the latter's execution. The expected value and the optional target index are stored within temporary variables prior to executing the assignment to compensate for the fact that the values used within its evaluation may be altered by subsequent assignments. Additionally, an assertion is performed after an assignment's execution to verify whether the new value of the target variable is equivalent to the expected value.

On top of that, the values changed by the assignments will need to be verified within the post-condition of the transition method. If the transition method returns true, it needs to hold that all of the affected variables have been assigned the appropriate values. Furthermore, all of the unaffected variables need to remain unchanged. For non-array variables, the latter verification is straightforward, i.e., it needs to hold that the value of the target variable is equivalent to the value assigned by the last assignment affecting said variable. For array variables, the verification is not as simple: the values used within the target indices may be changed by subsequent assignments, and as such, the stored indices need to be used instead. For each array variable, a function is generated that has all of the stored index and target value pairs of assignments as its arguments. In addition, the list of arguments is tailed by the target index and the variable's original value. The function contains a nested if-then-else structure, in which the target index is equated to the stored index: if the values are equal, then the expected value is returned by the function; otherwise, the function proceeds to the index of the next assignment until all have been exhausted. If the latter is the case, the target variable at the given index has not been altered, and as such, the old value is returned. Within the nested if-then-else structure, the assignments are visited in the reverse order of occurrence, i.e., it is ensured that the last assignment to a variable dictates the expected value. Finally, an assertion is performed which verifies whether the current value of the target variable at the given index is equivalent to the value returned by the function. An example of the structural verification of a transition with assignments is presented in Listing 7.3.

Ideally, the evaluation of and the changes made to the state of the state machine should be verified too: a transition should only be able to fire if it source state is equivalent to the current state of the state machine; the successful execution of a transition should move the state machine to its target state. Unfortunately, the VerCors tool set is incapable of verifying code that contains enumerations. Consequently, all references to the enumeration data type have to be removed during the verification process of the generated code, which in turn makes it impossible to verify the state transitions as rendered by the base instance of the code generator. Lastly, the structural integrity of the decision structures should be verified as well, but due to a lack of time, the formal verification of the structures has not been investigated, and as such, no formal verification is performed for the decision structures within the structural verification model.

7.2.2 Locking Mechanism Verification

Several approaches have been envisioned through which the locking mechanism can be verified, but so far, only one has proven to be successful in a practical setting. Within the original idea, the issue of representing reentrant locks through permissions is resolved through the use of an array that dictates which permissions are active at a given moment in time, i.e., a value that is larger than zero in a slot implies that the thread has write permissions over the variable instance that is associated with the given slot in the array. Observe that the array acts as a direct replacement of the collection of reentrant locks used in the original implementation: upon acquisition of a lock, the value in the associated slot is incremented; similarly, the value is decremented when the target lock is released. Next, the values in the array are constrained based on the locks that are supposed to be active. If a lock has been acquired, then it needs to hold that the value within the associated slot of the array is greater than zero. Furthermore, permission leaks are exploited to give and revoke access to a variable within the body based on the locking operations performed therein, which in turn ensures that the appropriate post-conditions are attained by the contents of the method body. Lastly, consistency constraints and supportive variables are added through which contextual information can be communicated to the code that calls the method in question using the provided `given` and `yields` syntax. The aforementioned approach has been investigated, but

```
1  /*@
2  pure int value_SMC0_0_x(int _i, int _i_0, int _rhs_0, int v_old) = (_i == _i_0) ? _rhs_0 : v_old;
3  @*/
4  /*@
5  // Require and ensure full access to the target class.
6  context Perm(c, 1);
7
8  // Require and ensure that the state machine has full access to its own array variables.
9  context Perm(y, 1);
10
11 // Require and ensure that the state machine variable arrays are not null and of the right size.
12 context y != null && y.length == 2;
13
14 // Require and ensure the permission of writing to all state machine variables.
15 context Perm(y[*], 1);
16 context Perm(j, 1);
17
18 // Require and ensure that the state machine has full access to the arrays within the target class.
19 context Perm(c.x, 1);
20
21 // Require and ensure that the class variable arrays are not null and of the appropriate size.
22 context c.x != null && c.x.length == 2;
23
24 // Require and ensure the permission of writing to all class variables.
25 context Perm(c.x[*], 1);
26 context Perm(c.i, 1);
27
28 // Require and ensure that all of the accessed indices are within range.
29 requires 0 <= c.i && c.i < 2;
30
31 // Ensure that the result of the function is equivalent to the target statement.
32 ensures \result == \old(c.i >= 0 && c.i < 2 && c.x[c.i] != 0);
33
34 // Declare the support variables.
35 yields boolean _guard;
36 yields int _rhs_0;
37 yields int _index_0;
38
39 // Ensure that the transition's return value is equivalent to the value of the guard.
40 ensures \result == _guard;
41
42 // Ensure that the appropriate values are changed, and if so, only when the guard holds true.
43 ensures _guard ==> (\forall* int _i; 0 <= _i && _i < c.x.length;
44     c.x[_i] == value_SMC0_0_x(_i, _index_0, _rhs_0, \old(c.x[_i])));
45 ensures !_guard ==> (\forall* int _i; 0 <= _i && _i < c.x.length; c.x[_i] == \old(c.x[_i]));
46 ensures c.i == \old(c.i);
47 ensures (\forall* int _i; 0 <= _i && _i < y.length; y[_i] == \old(y[_i]));
48 ensures j == \old(j);
49 @*/
50 private boolean execute_transition_SMC0_0() {
51     //@ ghost _guard = c.i >= 0 && c.i < 2 && c.x[c.i] != 0;
52     if(!(t_SMC0_0_s_0_n_4())) {
53         //@ assert !(c.i >= 0 && c.i < 2 && c.x[c.i] != 0);
54         return false;
55     }
56     //@ assert c.i >= 0 && c.i < 2 && c.x[c.i] != 0;
57     range_check_assumption_t_0_s_2();
58     //@ ghost _rhs_0 = y[c.i];
59     //@ ghost _index_0 = c.i;
60     c.x[c.i] = y[c.i];
61     //@ assert c.x[_index_0] == _rhs_0;
62     return true;
63 }
```

Listing 7.3: An example of the structural verification of a transition consisting of the composite $[i \geq 0 \wedge i < 2 \wedge x[i] \neq 0; x[i] := y[i]]$ that includes all of the required VerCors annotations.

unfortunately, the verification model has become too complex to verify, with even the simplest of input models taking an unreasonable amount of time to validate.

In response, an alternative approach has been investigated that models reentrant locks by merging fractional permissions. The expected benefit of the latter is that a permission over a variable can be acquired multiple times without causing issues in the verification process, under the condition that the sum of fractions for a target variable is never equal to or exceeds one, which can be achieved by having the denominator be larger than the maximum number of locks. Within said approach, a supplementary array is no longer required to assign permissions: instead, all permissions are chained together using a separating conjunction (**), which automatically merges permissions over the same variable. Observe that only read permissions can be attained through this approach–assignments require write permissions, and as such, an additional mechanism is required as a compensatory measure that temporarily elevates an existing read permission to a write permission during the evaluation of an assignment. Regrettably, the idea described above has proven to be more challenging to implement than originally anticipated due to the effect that the assignments have on variable values, i.e., the evaluation of an index expression may evolve over the execution of a transition due to the variables used therein changing value. Therefore, contextual information is required when listing the required and ensured permissions within the method's contract, i.e., the contract needs to have access to the original index with which the lock in question has been requested, such that it can be ensured that the reference to the lock remains a constant. The latter is infeasible, since VerCors analyses methods in isolation: the context in which a method is called is not considered during its evaluation–the validity of a method is based solely on the method's contract and body. Therefore, an alternative approach to the problem is required.

The final approach to the locking mechanism verification has been split into three phases, where each phase focuses on a different aspect of the verification process. The first phase, to be discussed in Section 7.2.2.1, validates the integrity of the locking structure by verifying whether each unique lock is only opened and closed once on each path through a transition selection procedure. Within the second phase, it is verified whether each class variable access is covered by the set of locks that is currently held by the thread. The second phase will be introduced in Section 7.2.2.2. The third and last phase, given in Section 7.2.2.3, validates the application of the rewrite rules to the indices used within the target locks, such that it can be verified that the effect of the value alterations incurred by the execution of assignments is compensated for appropriately. Lastly, note that the decision structures have been excluded from the verification process due to time limitations.

7.2.2.1 Phase 1: Structure

During the first phase of the locking mechanism validation process, the structural integrity of the locking mechanism is verified such that it can be guaranteed that it meets the set requirements. For one, the locking mechanism needs to ensure that threads do not hold the locks indefinitely, i.e., a lock that is acquired eventually needs to be released on every path through the program. Secondly, the lock that is released must be held by the thread for the operation to be valid. Furthermore, upon accessing a class variable, the lock associated with the variable needs to be held by the thread that is accessing said variable. Lastly, for the sake of efficiency, it is required that only one instance of each unique lock target is active at any time during the execution, i.e., it may not occur that a lock over $x[i]$ is acquired in the event where the aforementioned lock over $x[i]$ is already held by the thread. However, it is allowed for a lock $x[i]$ to be acquired when $x[0]$ is active, even if $i = 0$, since $x[i]$ and $x[0]$ are not considered to be the same target.

Within phase one, the aim is to ensure that the locking structure is structurally sound. As such, the atomicity requirements are not covered within the first phase: the question whether the locks the locks that are being held by a thread are sufficient to provide atomicity will be verified in the second phase instead. As such, the verification model of the first phase will, similarly to the structural verification performed in Section 7.2.1, require full write permissions over all state machine and class variables that are associated with the target thread. The process of attaining the

aforementioned permissions is analogous to the approach taken for the structural verification of the generated code, and therefore, will not be reiterated upon within the scope of this section.

Next, a mechanism needs to be created through which the acquisition and release of locks can be tracked and subsequently validated. To achieve this, a supplementary array variable is introduced which maps each unique locking target to a counter that tracks the number of active instances thereof at any given point in the program's execution. Within the code generator, the rendering of locking instructions is overwritten to increment and decrement the slot associated with the lock when acquiring and releasing the lock respectively. The counter slot associated with a lock can be accessed by using the latter's key as the index within the array: each unique lock target is assigned an unique number, and as such, it is assured that each lock has its own individual slot within the array. Specifically, note that a target lock's key should not be confused with its locking identity, since the latter is not unique or guaranteed to be a constant value. Additionally, assertions are added to the rendered code for locking instructions that verify whether the target lock has transitions to the expected state after performing locking operations or accessing the lock in question at its point of creation. By the requirements, each unique locking target should be acquired at most once at any point within the program's execution. Therefore, if a lock is active, the value contained within the array should be equal to one. Likewise, if a lock is in an inactive state, the associated value within the array should be equivalent to zero. Consequently, the model will be deemed invalid if the aforementioned conditions are not met.

The remainder of the verification targets are performed within the pre-conditions and post-conditions of the rendered methods. Within the scope of the method contract, the pre-conditions and post-conditions will verify whether the appropriate locks are active prior to and after executing the statement(s) contained therein respectively. Consequently, nested calls that have the wrong set of locks active at the given point in time will cause the verification to fail on the basis that the pre-conditions are not met, which portrays the desired behavior. The correctness of the structure depends upon the outcome of the expression that is evaluated within the method, since the latter determines whether the program passes through the failure or success exit within the locking data structure. Therefore, the inclusion of the evaluated expression within the rendered code remains a necessity. Lastly, it is verified within the decision structure's pre-condition and post-condition that no locks reside within an active state at the start and end of execution, with the aim of ensuring that no left-over locks remain active after the execution of a transition selection round. The verification of the locking structure of a control node method is exemplified in Listing 7.4.

7.2.2.2 Phase 2: Coverage

The second phase of the locking mechanism validation process focuses on verifying whether the locks that are associated with a given target statement are sufficient to attain full atomicity over the latter, i.e., it needs to be verified whether the locks created for a given statement cover all variables used therein. The verification process focuses solely on the coverage of class variables, and as such, the assertions and permissions required to access state machine variables are included within the method contract. The process of attaining the aforementioned permissions is analogous to the approach taken in Section 7.2.2, and therefore, will not be discussed here. The permissions required to attain ownership over class variables are not included within the method's contract: instead, the permissions are acquired through the application of assumptions within the method body. Said assumptions activate the appropriate permissions and ensure that the associated range checks are satisfied prior to referencing to a value contained within an array variable.

The use of assumptions is convenient, since all of the operations related to class variable permissions are performed locally. As such, the challenges introduced by the non-constant nature of variables are circumvented, due to the fact that the rewrite rules do not need to be accounted for: instead, the current values of variables can be used to attain the appropriate permissions, and as such, the application of rewrite rules and access to the context in which a method is called is no longer necessary. Specifically, observe that the lock coverage verification is performed at the

```
/*@
// Require and ensure full access to the target class.
context Perm(c, 1);

// Require and ensure that the state machine has full access to the arrays within the target class.
context Perm(c.x, 1);

// Require and ensure that the class variable arrays are not null and of the appropriate size.
context c.x != null && c.x.length == 2;

// Require and ensure the permission of writing to all class variables.
context Perm(c.x[*], 1);
context Perm(c.i, 1);

// Ensure that the result of the function is equivalent to the target statement.
ensures \result == (c.i >= 0);

// Ensure that all class variable values remain unchanged.
ensures (\forall* int _i; 0 <= _i && _i < c.x.length; c.x[_i] == \old(c.x[_i]));
ensures c.i == \old(c.i);

// Require and ensure full permission over the lock request variable.
context Perm(lock_requests, 1);

// Require and ensure that the lock request array is of the correct length.
context lock_requests != null && lock_requests.length == 2;

// Require and ensure full permission over all lock request variable indices.
context Perm(lock_requests[*], 1);

// Require that that no lock requests are active prior to calling the function.
requires (\forall* int _i; 0 <= _i && _i < lock_requests.length; lock_requests[_i] == 0);

// Ensure that that the following locks are active in the success exit of the function:
// - [0: i]
ensures \result ==> (\forall* int _i; 0 <= _i && _i < lock_requests.length;
    (_i == 0) ? lock_requests[_i] == 1 : lock_requests[_i] == 0);

// Ensure that that no lock requests are active in the failure exit of the function.
ensures !(\result) ==> (\forall* int _i; 0 <= _i && _i < lock_requests.length;
    lock_requests[_i] == 0);
@*/
private boolean t_SMC0_0_s_0_n_0() {
    lock_requests[0] = lock_requests[0] + 1; // Acquire c.i
    //@ assert lock_requests[0] == 1; // Verify lock activity.
    //@ assert lock_requests[0] == 1; // Check c.i.
    if(c.i >= 0) {
        return true;
    }
    lock_requests[0] = lock_requests[0] - 1; // Release c.i
    //@ assert lock_requests[0] == 0; // Verify lock activity.
    return false;
}
```

Listing 7.4: An example of Java code with VerCors annotations that formally verifies the structural integrity of the locking instruction associated with a control node method.

```
/*@
// Require and ensure full access to the target class.
context Perm(c, 1);

// Require and ensure that the state machine has full access to the arrays within the target class.
context Perm(c.x, 1);

// Require and ensure that the class variable arrays are not null and of the appropriate size.
context c.x != null && c.x.length == 2;
@*/
private boolean t_SMC0_0_s_0_n_3() {
    //@ assume Perm(c.i, 1); // Attain c.i.
    //@ assume 0 <= c.i && c.i < 2;
    //@ assume Perm(c.x[c.i], 1); // Attain x.c[c.i].
    return c.x[c.i] == 0;
}
```

Listing 7.5: An example of Java code with VerCors annotations that formally verifies the lock coverage within a control node method.

position in the program flow where the lock in question is needed to access the associated variable, and not at the position of the latter's acquisition in the locking node graph. Consequently, it remains to be verified that the acquired lock is equivalent to the lock used during the coverage verification, i.e., it needs to be asserted that the former and the latter point to the same element when subjected to the rewrite rules. The latter requires additional permissions, and as such, the verification of the rewrite rules will be performed within the next phase of the locking mechanism validation process, the latter of which will be discussed in Section 7.2.2.3.

The aforementioned assumptions are performed prior to reaching the target statement, such that it can be ensured that all of the required permissions are active prior to evaluating the latter. The permission assumptions that are rendered for a statement are based on the lock objects that have been created within the latter's atomic node. On top of that, the permissions are rendered in a topological order, such that it can be ensured that the ownership over a variable used within the index of another lock is acquired before accessing the latter. The verification of the model will fail in the event that a class variable is not covered by a lock, since insufficient permissions will be present to execute the statement in question. Additionally, each assignment in the transition is encapsulated by a wrapper method, such that it can be ensured that the permission and range check assumptions made therein are active only for the assignment in question. An example of the lock coverage verification is provided in Listing 7.5.

Unfortunately, the aforementioned verification process has several limitations. First of all, the verification model only supports code that is generated using the default locking mode, i.e., the lock objects need to be element-scoped for the evaluation to be viable, since the other locking modes generate locks that are not guaranteed to be equivalent to the class variable they are targeting. The latter can be compensated for by mapping the lock in question to the range of permissions covered therein, i.e., variable-scoped locking objects provide ownership over the entirety of a variable, including all of it indices. Regrettably, the aforementioned mapping has not been integrated into the verification progress, and as such, will have to be added in the future.

Furthermore, it needs to be noted that the second phase does not guarantee that the locking mechanism attains statement atomicity, i.e., it is not verified whether a lock stays active from the first use of the associated variable to the latter's last use in the target statement. As discussed within the previous sections, the verification performed by the VerCors tool set does not consider the context in which a method is called, and therefore, the independent verification of statement atomicity through the generated code becomes infeasible. Consequently, the atomicity of statements requirement will not be verified by the locking mechanism verification process. Lastly, the verification process does not verify whether a lock object generated for a statement in question is

```
1  /*@
2  // Require and ensure full access to the target class.
3  context Perm(c, 1);
4
5  // Require and ensure that the state machine has full access to arrays within the target class.
6  context Perm(c.a, 1);
7
8  // Require and ensure that the class variable arrays are not null and of the appropriate size.
9  context c.a != null && c.a.length == 9;
10
11 // Require and ensure the permission of writing to all class variables.
12 context Perm(c.y, 1);
13 context Perm(c.tmin, 1);
14 context Perm(c.fmax, 1);
15 context Perm(c.a[*], 1);
16 @*/
17 private boolean t_q_3_s_4_lock_rewrite_check_0() {
18     //@ ghost int _index = (c.y - 2); // Lock a[(y - 2)](a[y])
19     //@ assume 0 <= c.y && c.y < 9;
20     c.a[c.y] = 1;
21     c.tmin = c.y;
22     c.y = c.y - 2;
23     //@ assert _index == c.y;
24 }
```

Listing 7.6: An example of Java code with VerCors annotations that formally verifies the rewrite rules applied to a target lock object.

added to the locking data structure appropriately: a lock may have disappeared from the structure, and on top of that, it is not verified whether the acquisition operation of a lock has sufficient ownership to access the variables used within its index. Similarly, the latter also requires contextual information, and therefore, it is infeasible to verify the lock object inclusion requirement within the scope of the verification model. As a result, the validity of the latter two requirements will have to be extrapolated through the correctness of the generated locking data structure.

7.2.2.3 Phase 3: Rewrite Rules

Within the third and last phase of the locking mechanism validation process, it is verified whether the rewrite rules are applied correctly to the locking targets within the generated code, i.e., it is validated whether the target index of a lock acquisition has been adjusted appropriately to compensate for the value alterations made by assignments, such that it can be ensured that the ownership is gained over the appropriate element. To start with, each method is to be granted full write permissions over all state machine and class variables that are associated with the target thread, such that all of the variables used in the rewrite rules are guaranteed to be owned by the thread in question. Furthermore, the verification of rewrite rules is performed solely for locks that target array variables. Specifically, observe that variables of a non-array type do not require adjustments, and as such, the verification thereof would be superfluous.

The verification of the rewrite rules is implemented as follows. The verification process focuses solely on the correct application of the rewrite rules, and as such, the statements, transitions and decision structures that are contained within the target model are excluded from the generated code: instead, a verification method is rendered for each lock within the model that targets an array variable. The first statement within the method body creates a temporary variable in which the adjusted index is stored, i.e., the index that should be used during the acquisition of a lock that has been derived through the application of the rewrite rules. Next, all of the assignments associated with the rewrite rules are rendered, such that the used variables are altered appropriately. The last statement within the method body verifies whether the adjusted index is equivalent to the target index. Subsequently, the appropriate variable permissions are added to the contract of

the verification method. Finally, all of the resulting methods are added to the body of the state machine. The verification model will be deemed invalid if the adjusted index and the target index are not equivalent in value, which in turn would imply that the effects of assignments are not accounted for correctly. An example of the rewrite rule validation is presented in Listing 7.6. With all three phases of the verification process completed, the verification of the locking mechanism has reached its conclusion.

7.3 Issues and Challenges

During the implementation of the verification models that have been introduced within Section 7.2, several limitations and inconsistencies pertaining to the VerCors tool set have been encountered that warrant further discussion. The following sections will go into the aforementioned shortcomings, which have been subdivided into two categories respectively. First, the practical limitations of the VerCors tool set will be elaborated upon in Section 7.3.1, with the former including but not being limited to unsupported language constructs and a perceived lack of permission management tools. Subsequently, the issues pertaining to the tool set's documentation and the available tool set versions will be brought forward in Section 7.3.2.

7.3.1 Language Limitations

As observed during the structural verification of the generated code, the VerCors tool set does not support the verification of Java code that contains enumeration types or nested classes. Therefore, major structural changes had to be made to the generated code, which is undesirable, since the verified code is no longer consistent with the code rendered during normal operation. Alternatively, the base code generator could be adjusted to no longer utilize said language constructs. However, the replacement of said language constructs would be detrimental to the quality and readability of the generated code. Furthermore, both language constructs are considered to be common practice in Java, and as such, the base code will remain unaltered.

On top of that, short-circuit evaluations are not handled appropriately by the VerCors tool set. During the implementation process, it has been observed that the evaluation of a conjunction or disjunction expression always results in a state in which all of the terms contained therein are executed, even if said term is excluded due to the application of short-circuit evaluations. Additionally, the effects that one term may have upon the state of the program are not correctly conveyed to the next term in the sequence, i.e., each term seems to be evaluated under the state that the program resides in at the start of evaluating the statement. Both of the aforementioned observations are in conflict with Java's semantics, and hence, a workaround had to be created to compensate for said inconsistencies. The latter is achieved by replacing all conjunctions and disjunctions with nested and sequential if-statements respectively, where each term is rendered within its own if-statement. The given solution is not ideal, since it will lead to inconsistencies between the base code and the verification model, since expressed behavior in the former will need to be overwritten by the latter to attain a valid result. Unfortunately, no alternative workarounds could be found that attain the desired behavior without the need of structural alterations to the base code. Consequently, the presence of inconsistencies will needs to be accepted.

Lastly, the permissions system in the VerCors tool set seemingly does not provide a mechanism through which the ownership over a variable can be acquired or relinquished. Technically, the desired behavior can be achieved through the exploitation of permission leaks, but the latter does not seem to be the intended behavior within the tool set's design. As discussed within the tool set's documentation, permissions can be reasoned with through the application of a magic wand operator. Regrettably, no clear or working examples are given that demonstrate the application thereof. Therefore, it needs to be considered that the ownership mechanism is not intended to be used in this manner. Nevertheless, the introduction of language constructs that perform locking operations would be invaluable for the verification of the locking mechanism.

7.3.2 VerCors Versions and Documentation

The VerCors tool set provides two primary sources of documentation, namely on its official website[1] and the wiki page[2] that can be accessed through its GitHub repository. However, upon closer inspection during the implementation process, several discrepancies were observed between the documentation provided by the two sources. For instance, the given examples through which concepts and operators are demonstrated may at times use a different (outdated) syntax. Furthermore, several pieces of crucial documentation are missing at both locations: for example, no usage instructions are provided for the inhale and exhale operators, which, judging from the examples, may provide the means to properly manipulate permissions during the verification process.

On top of that, multiple versions of the VerCors tool set are available, but they are not consistent in their functionality and supported language constructs. Specifically, the online variant of the tool invalidates models that are otherwise verified to be correct by other versions of the tool set, such as the Windows and Linux versions thereof, and vice versa. Furthermore, axiomatic data times, such as bags and maps, are not supported by all versions of the tool set. Unfortunately, the list of language constructs supported by each of the aforementioned versions remains undocumented, and as such, the syntactical and semantical validity of a model cannot be predicted without performing a verification of the model in all versions of the tool set. The latter proved to be problematic, since the majority of the exploratory experimentation has been performed within the online version of the tool set. However, the latter proved to be too slow for more complex and sizable models, and as such, the Linux version of the tool set had to be used instead. Consequently, major modifications had to be made to the verification models to make the latter compatible with the offline versions of the VerCors tool set, which took valuable time and effort.

The issues described above onto themselves are inconvenient, but the inconsistency between versions is especially problematic when combined with the lack of consistent and complete documentation, since it becomes increasingly difficult to pinpoint the cause of erroneous or unexpected behavior within the written model and find alternative solutions. Consequently, the research into and implementation of the formal verification process ended up being far more arduous and time-consuming than earlier anticipated, which in turn has led to incomplete and arguably lackluster results that do not meet the author's original goals and expectations.

[1] The official website can be found at `https://vercors.ewi.utwente.nl/wiki/#introduction`
[2] The wiki can be found at `https://github.com/utwente-fmt/vercors/wiki`

Chapter 8

Performance Analysis

Throughout the preceding chapters, several crucial observations have been made that are predominantly theoretical in nature, and hence, it is important to verify whether the performance improvements derived from these concepts hold up in practice. Moreover, the revised implementation may still have imperfections, and as such, the performance analysis can be used to discover additional shortcomings that can be improved upon in the future. First of all, it is important to note that the focus in this chapter will be fully on the performance of the generated code, and not that of the code generator itself. In a practical setting, the code generator will only be used sporadically, and hence, searching for potential improvements in the efficiency of the code generator is not as important as optimizing the generated code itself. Furthermore, observe that the performance of the revised code generator will not be compared to that of the original generator, since the latter does not collect the performance measurements required for the analysis.

The chapter is structured as follows. First, the testing methodology will be detailed in Section 8.1. Secondly, the models used during the performance analysis will be introduced in Section 8.2. Next, the implementation and verification of the target performance measurements will be discussed in Section 8.3. Subsequently, the performance analysis of the decision structures and locking mechanism will be conducted within Sections 8.4 and 8.5 respectively. Lastly, the results attained throughout the scope of the chapter will be discussed in Section 8.6.

8.1 Testing Methodology

The tests conducted within the scope of this chapter have been executed on a Windows Server 2016 machine, running an Intel i7-8700k (6c/12t) @3.7GHz (boost @4.7GHz) processor, with a total of 32 GB of DDR4 system memory. For maximum reliability and consistency, the power settings have been set to maximum performance. Furthermore, besides the Windows sub-processes, no other major programs or processes are running on the system during the experiment, since it has been observed that other programs can heavily influence the dynamic between the Java threads of the program, which in turn leads to inconsistent and unreliable results.

Each trial is given a fixed time to run, such that a one-to-one comparison can be made between the results of different test runs. Additionally, each test is configured to have a maximum running time of 30 seconds. Given the unpredictable nature of concurrency, the experiments need to be ran multiple times such that the mean and standard deviation can be reported instead, with the number of trials being set at 20. Lastly, all of the tests are executed in sequence to ensure that the run of one experiment does not inadvertently influence the performance of another.

Two metrics are defined that give an indication of the overall performance of a target program. The first metric is the number of transition executions that have been completed successfully, i.e., the number of times that a transition has been executed with the guard statement holding true. Observe that the first metric increases proportionally to the productivity of a program, i.e., a higher value indicates that more work is done in the same amount of time. The second metric is defined to be the ratio between the number of successful and total transition executions, and as such, said metric favors the efficiency of a program over its productivity. The process of measuring the data required to gather said performance metrics will be detailed in Section 8.3. Afterwards, the measured values will be reported through a series of bar plots and tables. The 95% confidence intervals of each measurement are shown as errors bars within the bar plots to emphasise the overall variance within the results. Additionally, the standard deviation of the measurements will be included within the associated data tables.

To determine the performance and efficiency of the code generator implementation, several test models have been created. The selected models consist of a mixture of synthetic and practical models that can be used to investigate the effectiveness of the revised decision structures and locking mechanism. The Slco models used by the analysis are introduced in Section 8.2. The analysis of the decision structures focuses on investigating the difference between a sequential and random pick approach to resolving non-deterministic decisions. On top of that, the effectiveness of the deterministic decision structures is evaluated by generating code with and without said structures. Note that the effect of the available solver configurations will not be analysed, since the selected models do not produce meaningful differences. The analysis of the locking mechanism aims to measure the effectiveness of each of the available locking modes described in Section 5.4.2.1, with the exception of the no locks mode, which has been excluded from the analysis since the option is deemed to be irrelevant. The testing procedure generates a considerable amount of data, and as such, the effectiveness of the revised decision structures and locking mechanism will be analyzed independently from one another to limit the number of code generator configurations that need to be investigated. As such, the investigation of said components is limited to configurations using the default settings as a base, unless explicitly stated otherwise. The analysis of the decision structures and locking mechanism will be performed in Sections 8.4 and 8.5 respectively.

8.1.1 Locking Fairness Policy

Finally, it needs to be stressed that, oppositely to the approach taken by the original code generator, the generated programs use reentrant locks without fairness constraints. Originally, the intention was to enable the fairness constraints during the testing of the code, but unfortunately, the fairness constraints of the reentrant locks were not enabled due to an oversight made in the code template of the locking mechanism: an erroneous assumption was made that reentrant locks use fair locking by default, while the opposite holds true instead. In response, the necessary adjustments have been made to the code generator to remedy the aforementioned issue by adding a command-line argument that allows for fair locking to be enabled. Moreover, all of the experiments have been executed again with fairness enabled and the results thereof have been analyzed.

The latter led to an interesting observation, namely that the list of transitions visited by programs with and without the fairness policy enabled is not equivalent. Particularly, several of the transitions that are visited a handful of times by programs using an unfair approach to locking are not visited by their fair counterparts. The latter is presumably caused by the fact that the fairness policy has a heavy impact upon the performance of the code, i.e., the number of transitions executed by the program in the 30 second running time is lower, and as such, the chance of finding the aforementioned transitions in an active state reduces proportionally. Nevertheless, both configurations produce compelling results, but due to the sheer amount of data and time limitations, it is infeasible to investigate both of the available configurations. Hence, it has been decided to report solely on the results attained through locking without fairness, since subjectively, the results thereof provide a more complete picture of the program's behavior.

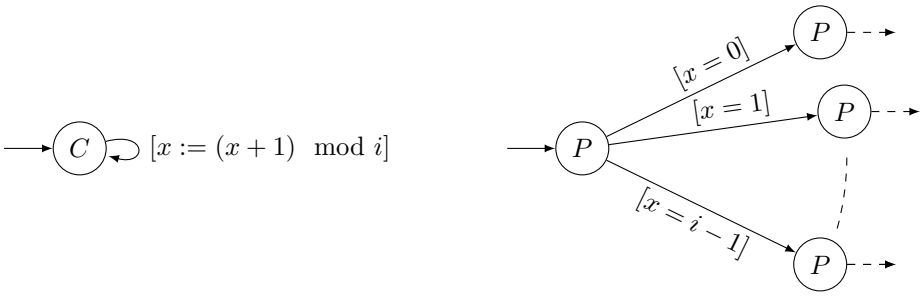
8.2 Target Models

Within the scope of this section, the models used within the performance analysis will be discussed[1]. The models will be categorized into two groups, namely synthetic and practical models. The synthetic category contains models that are designed to demonstrate and validate characteristics of the decision structure and/or locking mechanism. The practical tests consist of several real-world example models that are included within the SLCO repository. The models used in the performance analysis will be introduced in their own respective sections, in which the interesting characteristics of each model will be highlighted. Lastly, the expected effectiveness of the available decision and locking modes will be elaborated upon for each model.

8.2.1 Synthetic Test: CounterDistributor

The first synthetic test will use a variant of the model depicted in Figure 4.1, with the primary difference being that the model will only have two states and that all transitions are self-loops with stricter guard expressions. The `CounterDistributor.i` model used in this experiment has been visualized in Figure 8.1. The model consists of the two state machines `Counter` and `Distributor`, each having only a single state for the sake of simplicity. The `Counter` state machine counts up x from zero to $i-1$, after which the pattern is repeated by resetting the value of x to zero. On top of that, the `Distributor` state machine has a set of i transitions, with guards that ensure that only one transition of the state machine is active at any time.

Hence, the behavior of `Distributor` is completely deterministic, since there are no transitions with guard that have overlapping solution spaces. Consequently, a decision mode using deterministic structures will always be able to find a transition that is active under the current value of x. Oppositely, the chance of success on randomly picking an active transition in `Distributor` is expected to be $\frac{1}{i}$, and as such, the chance of picking the right transition becomes significantly lower at higher values of i. Therefore, a value of $i = 10$ will be used during the experiments for demonstration purposes. Given the earlier observations, there should be a clear performance difference between the available decision modes, especially within the state machine `Distributor`. On the other hand, the model contains only a single class variable `x`, and therefore, no major differences in performance are expected between the available locking modes.



(a) State Machine `Counter` (b) State Machine `Distributor`

Figure 8.1: Model `CounterDistributor.i` consisting of the two state machines `Counter` and `Distributor` with a shared variable x, the latter of which has the initial value $x := 0$. Additionally, note that state machine `Distributor` uses an alternative notation to depict a self-loop, i.e., all nodes depicted within represent the same target state P.

[1]The SLCO models and the generated programs are available at https://github.com/melroy999/2IMC00-SLCO/tree/master/SLCOtoJAVA3.0/test_models

8.2.2 Synthetic Test: Tokens

The second synthetic test has been designed to investigate the measure of concurrency between state machines. The `Tokens` model depicts a system in which the state machines produce tokens that are to be consumed by other state machines within the model. The program consists of the state machines `A`, `B` and `C`, where `A` produces tokens for `B`, `B` produces tokens for `C`, and `C` produces tokens for `A`. A state machine prioritizes the consumption of its target token. Additionally, a state machine is forced to halt its token production until its partner state machine has successfully consumed said token, i.e., no tokens will be produced if the target token is already in an active state. The tokens within the system are represented by a Boolean array class variable `tokens` with a length of three, where each slot corresponds with the index of target state machine respectively. A token is in an active state if the associated Boolean in the array is true, and inactive otherwise. On top of that, the model defines three class variables `a`, `b` and `c` of type integer that act as counter variables for a specific transition within the model: the aim of these variables is to increase the workload of the locking mechanism, such that the overall effect of the available locking modes can be measured by the model. The `Tokens` model has been visualized in Figure 8.2.

The state machines `A`, `B` and `C` are similar in structure, and as such, the description thereof will be generalized. Each state machine consists of the states `act`, `update` and `wait`, with `act` being the initial state. Furthermore, each state machine has an integer typed local variable `x` that is used to pseudo-randomize the activation of a token. The function of each state is as follows:

- The focus of the state `act` is two-fold. First of all, the state `act` prioritizes the consumption of the token associated with the target state machine: the consumption of tokens is depicted as a self-loop transition in which the value of the associated token is set to false if the appropriate conditions are met. Secondly, if no token is to be consumed, the state attempts to produce a token for its partner state machine. A new token is produced when the least significant digit of `x` is equivalent to zero. The state machine transitions to state `update` if the aforementioned condition is not met. Otherwise, a token is produced for the state machine's partner, after which the state machine transitions to state `wait`. Observe that the priority of the latter two transitions is set to one, such that the consumption of tokens takes precedence over the other available actions.

- The sole purpose of state `update` is to update the value of `x`, such that state `act` can make another attempt at producing a token. The aim is to update `x` in such a manner that the next value of `x` is unpredictable, i.e., the value of `x` should be randomized. The latter ensures that the state machines do not end up in a synchronized state, due to the fact that the time between token activations is not a constant period. The SLCO framework does not provide a random statement, and as such, a linear congruential generator of the shape $x_i = (a \cdot x_{i-1} + c) \bmod m$ will be used to implement a basic pseudo-random number generator. To increase variety, each state machine is given a different configuration for the parameters (x_0, a, c, m): the configuration are chosen in such a manner that the generator has a full period, i.e., the cycle length of the generator is m. The configurations used by the state machines `A`, `B` and `C` are $(1, 641, 718, 1009)$, $(42, 193, 953, 1009)$ and $(1, 811, 31, 1009)$ respectively. Given that the sequence of numbers is a full period of length $m = 1009$, it follows that the probability of the least significant digit being zero is approximately $\frac{1}{10}$, which in turn implies that a token activates with the same probability. The state `update` contains a single transition to state `act` which updates the value of `x` through the aforementioned procedure. Additionally, the transition increments the counter associated with the associated state machine. Lastly, note that the state `update` does not contain a token consumption transition due to the fact that its sole transition is unguarded. Consequently, a program-wide deadlock will not occur, since the associated state machine may transition freely from state `update` to state `act`.

- The state `wait` is used to interpret the responsiveness of the program. Similarly to state `act`, the state `wait` prioritizes the consumption of the token associated with the target state machine. Analogously, the consumption of tokens is implemented as a self-loop transition.

The second transition in state `act` allows for the state machine to proceed to the `update` state once the token produced for its partner state machine has been consumed by the latter, i.e., the state machine can only transition from state wait to state `update` if the associated value in the tokens array is false. Intuitively, the ratio between the total and number of successful executions of the aforementioned transition may be used to get an interpretation on the degree of concurrency within the generated program, i.e., a program with a high degree of concurrency should exhibit a higher success ratio. The reasoning behind the latter observation is that a higher degree of concurrency will make the program more reactive due to the state machines running simultaneously, and consequently, it should take less time for the partner state machine to consume the token and allow for said transition to execute successfully. Lastly, the priority of the transition to state `update` is set to one, such that the consumption of its own target token takes precedence.

The performance characteristics of the `Tokens` model are expected to be as follows. In terms of structure, the model contains both deterministic and non-deterministic decisions: states `act` and `update` always have exactly one transition active, while state `wait` may have zero or more transition active at any time. However, priorities have been set on the transitions in these decisions, which in turn dictate the order in which the latter are visited during the selection process. Furthermore, each state has only a small pool of available options to choose from, with states `act`, `update` and `wait` having three, one and two transitions respectively. Given both observations, it is expected that the performance difference between the available decision modes will be marginal.

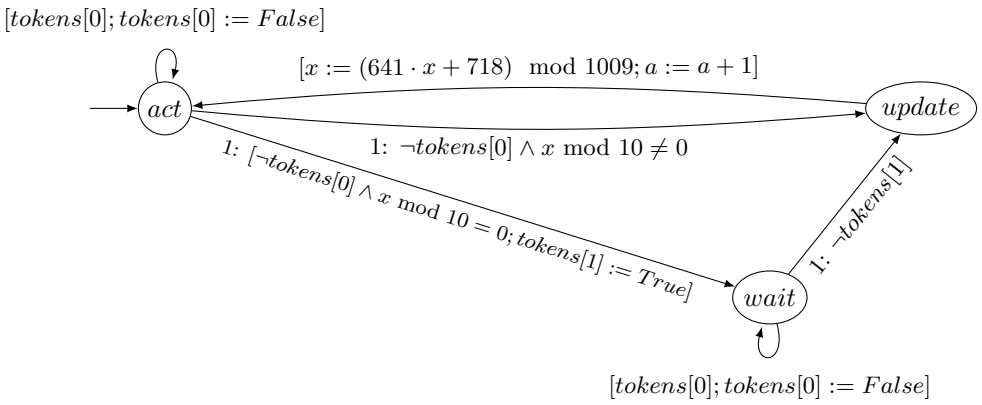
The effect of the available locking modes upon the performance of the program is expected to differ based on which transitions are targeted. First of all, it needs to be noted that the model contains only a few class variables. On top of that, the model does not require any costly unpacking operations, and the pool of variables used in each transition generally does not overlap with that of transitions in other state machines. The exception to the latter is the array variable `tokens`, where each state machine accesses two out of three available slots in state `wait`. Consequently, the majority of transitions are expected to benefit from a fine-grained approach to locking, while different results may be observed in the transitions of state `wait` due to shared variables. Particularly, differences are expected in state machine `C`, due to the lock ordering requirements, i.e., `tokens[0]` needs to be acquired at the same time as `tokens[2]` to respect the strict lock ordering. Nevertheless, it is hoped for that the available locking modes will cause clear performance differences in the state `wait`, given the latter's sensitivity to the program's degree of concurrency: fine-grained approaches to locking should facilitate a higher degree of concurrency, and as such, element-scoped locking should lead to a considerably higher success rate for the transition in question.

8.2.3 Practical Test: Elevator

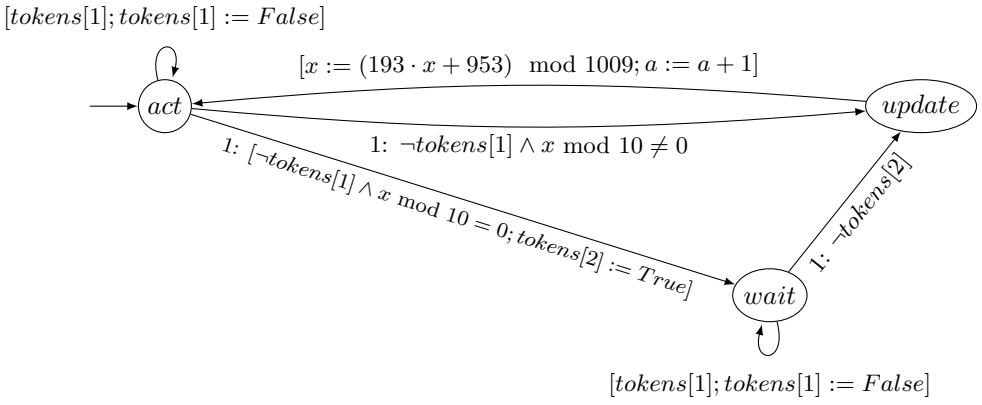
The first practical test will investigate the performance of the `elevator2.1` model as described in Section 2.3.2. An in-depth discussion of the model is included in said section, and as such, this section will focus solely on discussing the expected performance characteristics of the `elevator2.1` model under the available decision structure and locking mechanism configurations.

The types of decisions to be made within the `elevator2.1` model are a mixture between deterministic and non-deterministic structures. The `environment` state machine may have more than one option active at any moment in time, and as such, the decisions made therein are not expected to benefit from a deterministic approach. The decision process of the `cabin` and `controller` state machines is entirely deterministic, i.e., the available options are mutually exclusive, and as such, only a single transition will be active at any time. Furthermore, the deterministic decisions in the model are made over two or three options. Hence, the performance difference between the decision modes is expected to be minimal, with deterministic structures having a minor advantage.

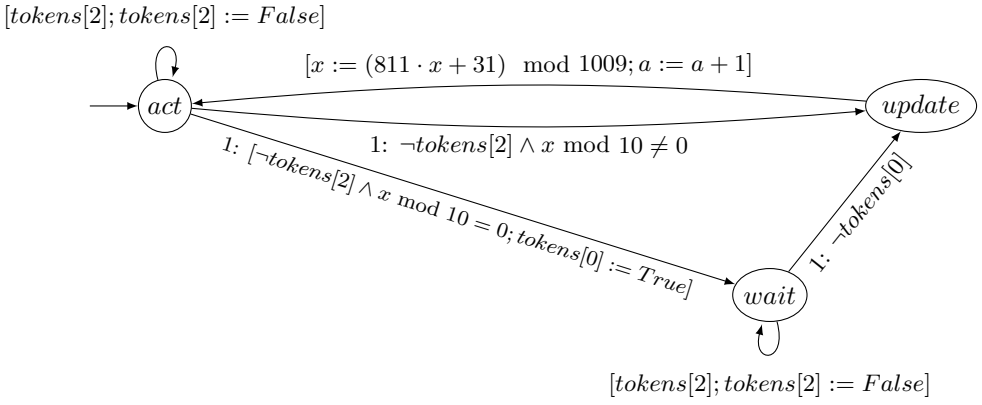
The `elevator2.1` model contains only a few class variables, with the sole array variable having a length of four, and as such, an unpacking operation is expected to be limited in impact. The model contains a single unavoidable lock location sensitivity violation in the `controller` state



(a) State Machine `A` with initial value $x := 1$



(b) State Machine `B` with initial value $x := 42$



(c) State Machine `C` with initial value $x := 308$

Figure 8.2: Model `Tokens` consisting of the state machines `A`, `B` and `C`. The model has defines a shared Boolean array variable *tokens* with a length of three and shared integer variables a, b and c. Additionally, each state machine has a local integer variable x.

machine. The aforementioned violation is caused by the ordering of the transitions within the decision structure. Consequently, it is expected that the `controller` state machine will perform worse due to the unpacking operations that need to be performed. Conversely, the majority of the transitions in the model seem to be using distinct combinations of class variable targets, and as such, it is expected that a fine-grained locking mode will outperform less fine-grained alternatives for the majority of the transitions and decision nodes.

8.2.4 Practical Test: ToadsAndFrogs

The second practical test investigates the performance of a model depicting a single player variant of the *Toads and Frogs* puzzle game. The aforementioned model is equivalent to the third concrete example of the SLCO syntax given in Section 2.3.3, and therefore, this section will focus solely on discussing the expected performance characteristics of the `ToadsAndFrogs` model under the available decision structure and locking mechanism configurations.

In terms of structure, the model is completely deterministic, i.e., the options that are provided for a given state are mutually exclusive, and as such, only a single transition will be active at any time. Consequently, the use of deterministic structures should lead to an advantage in terms of a decision node's success to total ratio. Oppositely, the use of sequential approach to decision making may prove to be detrimental to achieving a successful outcome of the game, due to the fact that the game can only be won if the toads and frogs adhere to a specific ordering of moves. As such, it may be a beneficial property for a state machine to inadvertently pass on making a move through the use of a random pick approach. Furthermore, it has been observed during preliminary testing that concurrent programs in Java have a tendency to perform multiple iterations in sequence, regardless of the fairness constraints set upon the reentrant locks. As per the documentation of reentrant locks, the latter is caused by the fact that the fairness of locks does not guarantee fairness of thread scheduling, since the latter is managed by the operating system instead. The game cannot be won if a toad or frog makes more than one move at the start of the game, and as such, the lack of fair thread scheduling may prove to be problematic. Hence, a random pick approach may achieve better results, due to the $\frac{1}{4}$ chance of a player passing on making a move.

The expected effect of the available locking modes on the generated program's performance is more difficult to ascertain. All of the transitions in the state machines `toad` and `frog` use three class variables, while all but two transitions in the state machine `control` uses at least two. The use of the variables `a` and `y` is a common factor for all state machines: the aforementioned variables are used in fourteen out of eighteen transitions. Furthermore, it has been observed during the conversion to Java code that the `ToadsAndFrogs` model contains transitions that, in conjunction with the decision structure, lead to location sensitivity violations on variable `a`. As such, several unpacking operations need to be performed. Given the size of array variable `a`, such an unpacking operation is predicted to be costly. However, it needs to be noted that the presence of location sensitivity violations depends upon the chosen configuration for the decision structure: no violations are detected when a random pick approach is used in conjunction with the exclusion of deterministic structures. Given the heavy use of class variables, the prediction is that statement-scoped locking should outperform the variable and element-scoped locking modes.

8.2.5 Practical Test: Telephony

The last practical test will investigate the performance of another example model given in the SLCO repository. In this section, the `Telephony.8` model will be introduced, which is a model that depicts a telephony system between four users. The model itself is rather complex, with many states being present within the model itself. Unfortunately, the `Telephony.8` model lacks documentation, and hence, an effort will be made to give an interpretation of the model.

The model consists of four state machines, with each state machine `User_`u depicting a single user, with $u \in [0, 3]$ representing the user's identity. The model has fourteen states, which differentiates

this test suite from the previously conducted experiments, which all have a comparatively limited number of states. Furthermore, the model does not have a central controller: instead, all the users manage the connections within the telephone system by themselves by altering the class variables. Given the overall complexity of and the number of states in the model, it is impractical to create a comprehensible graph or a detailed description for the user's behavior. Hence, in the remainder of this section, the focus will remain primarily on giving a brief description of the included class variables. Additionally, the general structure of the state machine will be investigated, such that a prediction can be made on the effectiveness of the decision structures and locking mechanism within this particular test suite.

The class variables in the model will be discussed first. The model has four class variables, all of which are of an array type with a length that is equal to the number of users depicted in the telephone system. The variable `chan` depicts the identity of the channel that the two users are making the call over. The channel itself is a byte, with the value being 255 for the associated user when the user resides in an idle state. Similarly, the variable `partner` simply refers to the identity of the user that the caller is partnered with during the call, having the same value 255 for the associated user when being in an idle state. The variable `callforwardbusy` is used to map one user to another, should the latter be busy, and has the value 255 set for the user if call forwarding is unsupported. The last variable is `record`. The exact purpose of this variable remains unclear, but it is assumed that it represents the action of recording missed calls, such that a callback can be made to the appropriate user when applicable.

In terms of structure, the `Telephony.8` model is predominantly deterministic. Random decisions are only made in one state of the model, namely the `Dialing` state, with the number of options being equal to six. The first option sends the user to an idle state, whereas the remaining options initiate a call to one of the available partners. However, it is important to observe that the aforementioned transitions do not have a guard statement and are thus always active. Consequently, only the first option will be visited when a sequential approach is taken to decision making, and as such, a considerable proportion of the state machine ends up being unreachable. The latter is undesirable, and as such, all tests conducted with the model should have the random pick option enabled to achieve meaningful results. Furthermore, it needs to be considered that the deterministic decisions in the model are made over a small pool of options, with most states combining an equality and inequality over a certain variable, or a collection thereof. Hence, the performance difference between random picking with and without deterministic structures enabled is expected to be minimal, with the latter being expected to have a minor advantage.

Oppositely, it is predicted that the performance of the available locking modes will differ considerably, due to several unique characteristics of the model in terms of locking behavior. During the application of the code generator to the `Telephony.8` model, it has been observed that the model contains a large number of location sensitive locks, with the majority of the latter being deemed unavoidable violations, i.e., unpacking operations need to be performed to enforce the strict locking order regardless of the chosen locking identities or decision mode due to the internal structure of the model's statements. Consequently, the unpacking operations will affect all but one of the model's class variables, the exception being the variable `partner`, and as such, it is likely that the variable and statement-scoped locking modes will outperform the default element-scoped locking mode. Furthermore, it has been observed that the majority of the transitions make use of one or more of the unpacked variables, and as such, it is expected that statement-scoped locking will outperform the variable-scoped locking mode as well.

8.3 Performance Measurements

The performance of the code generator is measured by two different methods. The first method adheres to a simple and reliable counting-based approach: for each transition, two long-typed variables are added to the class of the associated state machine, one counting the number of

executions of the target transition, and the other counting the number of successful executions thereof. The former is incremented on reaching the entry point of said transition, while the latter is incremented upon reaching the success exit point. The counter variables are state machine local, and as such, the counters will have a minimal effect upon the performance of the program since no locking operations need to be employed. Similar counters are added for all of the root decision nodes contained within the model, such that the effectiveness of the decisions made within each state of a state machine can be measured independently as well.

The counting-based approach cannot be used to measure the degree of concurrency within the generated program, since it does not capture the ordering of transitions within the execution of a program. Therefore, it is non-trivial to measure the degree of concurrency through the first approach, since the measurements lack the required context. As such, a second method of measurement has been introduced that captures the ordering of transitions within the program execution through a logging-based approach: the entry and exit events of a given transition are logged within a log file, such that concurrent behavior can be measured through the mutual ordering of events originating from different state machines. Unfortunately, the results attained with logging-based measurements have proven to be unreliable. Consequently, the measurements made through the latter approach cannot be used by the performance analysis conducted in Sections 8.4 and 8.5. Nevertheless, the implementation and analysis of the logging-based approach has led to interesting insights that could be valuable for future reference, and as such, the continuation of this section will focus on detailing the intricacies and the observed failings of logging-based approach.

The remainder of this section is outlined as follows. First, the concept of logging-based measurements and the advantages and disadvantages thereof will be introduced in Section 8.3.1. Next, the actual implementation of the logging-based measurements within the code generator will be discussed in Section 8.3.2. Subsequently, the ordering and throughput consistency of logging measurements will be analyzed in Section 8.3.3. Additionally, the representativeness of the logging-based measurements will be verified by comparing the attained results to counting-based measurements in Section 8.3.4. Lastly, the observed advantages and shortcomings of the logging-based measurement approach will be discussed in Section 8.3.5.

8.3.1 Logging-Based Measurements

The primary purpose of the logging-based measurements approach is to provide data through which the degree of concurrency can be analyzed, i.e., the goal of the logging-based approach is to measure concurrency by performing a quantitative analysis on the overlapping execution periods of transitions in different state machines as recorded within the log file. The latter can be achieved by logging the start and finish of each transition execution in the run of a test model. Subsequently, the log file can be parsed line by line to find instances within the program execution where two or more transitions depict concurrent behavior, i.e., the latest transition events of two or more state machines denote the start of a transition execution, which in turn implies that two transitions are in an active state simultaneously. Note that the latter depends on the requirement that the logging messages are processed in the order they are issued in, i.e., a strict first-in, first-out (FIFO) ordering needs to be adhered to without fairness constraints. The logging framework and a configuration thereof that adheres to said requirements will be discussed in Section 8.3.2.

A distinctive advantage of using a logging framework to capture the behavior of a program is that it provides a rich amount of data to analyze, i.e., the tests do not have to be re-run if the analysis target changes, since all of the required data is already present within the log file. As such, the logging-based approach to performance measurements is more flexible than its counting-based counterpart, since the measuring of results is decoupled from the analysis thereof. Nevertheless, the use of a logger also has major disadvantages. First of all, the utilization of a logger introduces a measurable performance overhead in high-throughput environments. As a result, a generated program using a logging-based measurement approach will execute significantly fewer transitions than a counting-based alternative in an equivalent span of time, with differences that are over

an order of magnitude not being an uncommon occurrence. Another closely related drawback of logging-based measurements is that the logging process may have an impact upon the dynamics of the generated program, i.e., the logged data and the results deduced from the latter may not be representative if the logging process changes the behavioral characteristics of the program. In other words, the aim is to measure the performance of the generated program, and not the performance of the logging framework. Consequently, it is important to verify whether the utilization of a logging framework has a limited impact upon the distribution of transition executions within and the concurrency characteristics of the generated program.

The verification process of the logging-based measurements is envisioned as follows. To start with, the integrity and consistency of the logging measurements will be investigated. Note that the latter does not consider the contents of log messages, i.e., the analysis focuses entirely on the characteristics of the logging process itself, instead of the transition events logged therein. The aforementioned analysis will be performed in Section 8.3.3, which focuses on validating the message ordering requirement and reporting on the overall throughput and consistency of messages stored within the log file. Furthermore, it needs to be verified whether the data recorded by the logging-based measurements is representative of the behavior depicted by generated code that does not contain performance metrics. The logging-based measurements are deemed representative if the distribution of transition executions over the model is similar to that of the target program without measurements. On top of that, the concurrency characteristics of the two programs need to be comparable. The former of the two requirement is straightforward to validate, since the transition distribution of the logging-based measurements can be compared directly to their counting-based counterparts. Oppositely, the validation of the measured concurrency characteristics is non-trivial, since there does not exist a different source of measurements that has the level of detail required to capture the activity span of transitions. As such, a more creative solution is required through which the concurrency characteristics can be verified. The analysis of the behavior of logging-based measurements will be continued in Section 8.3.4.

8.3.2 Logging Approach and Configuration

The logging-based measurements have been implemented through the use of the highly optimized and feature rich Log4j logging framework. Log4j uses a garbage free implementation, and as such, the framework will not trigger Java's garbage collection. The latter is important, since it ensures that the behavior of the program is not influenced by an increased frequency of resource intensive garbage collection sweeps. Additionally, the Log4j logger can be configured to log asynchronously. The latter is a invaluable property due to the fact that it guarantees that the concurrency characteristics of the generated program are effected as minimally as possible, i.e., it ensures that the state machine threads will not halt when attempting to log a transition event message. Moreover, Log4j guarantees that log messages are written in the order they are received in, which the logging-based measurement approach relies upon as expected behavior.

Furthermore, several additional measures have been taken to ensure that the logging process does not influence the behavioral characteristics of the target program. First of all, the ring buffer used by the asynchronous logger has been configured to have $4096 \cdot 1024$ slots instead of the default $256 \cdot 1024$. The increased size of the buffer ensures that the logger can deal with large bursts of activity at the cost of a higher memory usage. Additionally, the rollover procedure for logging files has been tuned to ensure that the impact of a rollover event is limited: the maximum file size of a log file has been set to 100MB, with the compression level being reduced from the default value of 9 to a compression level of 3. As such, a compressed log file will be flushed to disk in a short burst, but also at a marginally larger in size due to a lower compression level. On top of that, note that the use of compression within the log files is necessary, since each test would otherwise produce a log file that is over 2GB in size: given the number of trials, it is infeasible to store the data without compression. Lastly, log messages are formatted preemptively to be of an equivalent text length. The latter is a precaution and has the goal of ensuring that each log message has equal weight within the logger's ring buffer.

8.3.3 Verifying Logging Consistency

First, the integrity and consistency of the logging measurements will be investigated. As stated previously, the verification process consists of two parts. First, it will be validated whether message ordering requirement is satisfied by the Log4j logging framework, i.e., the logger needs to adhere to a strict FIFO ordering. The latter will be investigated in Section 8.3.3.1. Furthermore, it needs to be verified whether the throughput of log messages is consistent, i.e., it is required that the logger can consistently keep up with the influx of log messages. The analysis of the throughput of the logging process will be discussed in Section 8.3.3.2.

8.3.3.1 Ordering of Messages Analysis

The ordering of messages requirement will be verified as follows. First, it is important to observe that the ordering cannot be verified for an arbitrary model, i.e., the model needs to be constructed in a manner that the expected ordering of messages can be deduced from the semantics of the model. Unfortunately, the test models introduced in Section 8.2 do not meet this requirement: the ordering of transitions within the same state machine can be deducted, but the ordering of transitions between different state machines is not well-defined. For simplicity, the ordering requirement has hence been verified through an auxiliary piece of code[2] that has not been generated through Slco framework. The latter is not an issue, since the sole communication between the two components is performed through the Log4j's API calls, i.e., the Log4j logging framework and the generated code are loosely coupled. Consequently, the results attained through the auxiliary code should be representative as long as the logger adheres to the same configuration.

The program depicted by the auxiliary code spawns m threads. Each thread is assigned an unique identity `thread_id` $\in [0, m)$. All of the threads in the program have shared access to a variable `current_thread` which refers to the identity of the thread that is currently allowed to perform actions. The thread identity is unique, and as such, only a single thread may perform actions at any time. The code performed by the threads are described in Procedure 15, which takes the values `thread_id` and n as an input. The value denoted by `thread_id` corresponds to the identity of the thread that is performing the actions, whereas n is defined to be the number of actions that the thread needs to perform. The value of n is required to be equivalent for all m threads.

The OrderingTestProcedure(thread_id, n) algorithm depicts a system in which m threads are simultaneously attempting to log their thread identity (line 4). Furthermore, a thread is only allowed to perform a logging action if the value of its identity is equivalent to the value of `current_thread` (line 3). After every log action, the value of `current_thread` is updated to be the identity of the next thread in the sequence (line 5). The latter ensures that the threads send log messages in a predefined order, and as such, the message ordering requirements can be verified through the log output of the program. Lastly, the algorithm terminates once n log actions have been performed by the thread (line 2). The value of n is equal for all threads m, and therefore, it is guaranteed that all threads will eventually terminate.

Algorithm 15: OrderingTestProcedure(thread_id, n)

```
1 i ← 0
2 while i < n do
3     if thread_id = current_thread then
4         log(thread_id)
5         current_thread ← (current_thread + 1) mod m
6         i ← i + 1
```

[2]The Java code used for the verification of the message ordering requirement is available at https://github.com/melroy999/2IMC00-SLCO-JAVA/tree/f226344864816f6460a1e6b35d5930f74a6d45aa

During the validation process, the auxiliary code has been executed with the values $m = 8$ and $n = 10^6$ respectively, i.e., the experiment employs eight threads, where each thread makes one million log calls. The resulting log file has been verified by iterating over each message within the log and comparing the value contained therein with the expected sequence of values. The results did not contain any inconsistencies, and as such, it can be concluded with relative safety that the asynchronous logger made available by the Log4j logging framework adheres to a strict FIFO processing ordering, and hence, it follows that the message ordering requirements are met.

8.3.3.2 Log Message Throughput Analysis

Next, the throughput of log messages will be investigated. Structural deviations within the number of log entries per time unit may indicate that the logging framework acts as a bottleneck within the performance of the tested program, and as such, it would be preferable that the data gathered by the logging-based measurement approach lacks the aforementioned deficiencies, since it reduces the likelihood that the logging framework has an influence upon the behavioral characteristics of the executed program. However, it needs to be stressed that the presence of deficiencies within the message throughput of a model does not directly imply that the gathered data is not representative, i.e., the message throughput may be influenced by other factors, such as the operating system or the characteristics of the model itself. Consequently, the data attained through the logging-based approach cannot be discredited based solely on the observations made within the scope of this investigation. Nevertheless, the analysis performed within the scope of this section could provide valuable insights that may help to identify and resolve concurrency related issues.

The log message throughput will be investigated for all models introduced in Section 8.2. The performance analysis of the decision structures and locking mechanism both have the default configuration of the code generator as one of the test cases, and as such, only the default configuration will be considered within the scope of this analysis. The exception to the latter is the `Telephony` model, which has been configured to use a random pick approach to resolve non-deterministic decisions–the latter is required, since said model will not work appropriately otherwise, which is due to the non-deterministic nature of the desired behavior defined therein. By default, the code selects transitions through a deterministic decision structure: non-deterministic decisions are resolved by visiting all options in sequence until an active transition is found. On top of that, the default setting uses the element-scoped locking mode to generate the locking graph.

The results of the logging throughput analysis are given in Appendix A. For each target model, two figures are presented that visualize the throughput of the logging process as a heat map. The first figure focuses on reporting the total throughput of log messages per time unit, whereas the second figure reports the difference between the throughput values of state machines within the model. The values reported by the latter correspond to the sum difference to the row minimum value, which is defined as follows. Let $T(i)$ be a list of the number of log messages made by each state machine at timestamp i, and let $T_s(i) = \sum_{v \in T(i)}(v)$ be the sum total throughput at timestamp i. Under the aforementioned environment, the sum difference to the row minimum value $T_d(i)$ is equivalent to $T_d(i) = \sum_{v \in T(i)}(v - \min(T(i))) = T_s(i) - |T(i)| \cdot \min(T(i))$. Each of the figures can be used to detect different types of throughput deficiencies: the total throughput plot can be used to investigate fluctuations in the throughput volume, while the difference plot may highlight disparities between the throughput of state machines at a given time unit.

The timestamps included within the log files have millisecond accuracy, and as such, each time unit corresponds to a millisecond. Furthermore, the presented measurements are grouped by the log file that the data originates from in an effort to keep the data readable. Consequently, the timestamp values have been adjusted accordingly such that the results of each log file start at time unit zero. As noted in Section 8.3.2, each log file has an uncompressed size of 100MB. Given that each message is of equal length, it hence follows that the total number of messages contained within each log file is equivalent, with the exception of the last log file, which may contain fewer messages. Consequently, a higher number of log files implies that the given run of a target model

has been able to maintain a higher log message throughput. Additionally, observe that the time it takes for a log file to fill is not constant. As a result, each log file will have trailing white space at the right-hand side of the figure of variable length: the trailing white space does not contain measurements, and thus, it should be ignored when interpreting the data within the figure.

Lastly, it needs to be stressed that for each model, the throughput results will be visualized for a single run only, namely the tenth run, since it is infeasible to include figures for all twenty runs. The latter is not a major concern, since all runs display similar throughput characteristics, i.e., the patterns observed within the tenth run of a given model are deemed to be representative for all runs of the same model. On top of that, a table is presented for each model that reports upon the results measured in all twenty runs. The table provides summary statistics on the number of log messages recorded per time unit at the state machine and global level. In addition, the summary statistics of the sum difference to the row minimum metric are reported upon.

Result Analysis To start with, a general observation will be made that applies to the attained results of all target models. A common pattern present within all figures given in Appendix A is that the first few log files display behavior that deviates strongly from subsequent ones. The number of affected log files differs per target model: the behavior stabilizes after the second log file for the `CounterDistributor`, `ToadsAndFrogs` and `Telephony` models, whereas the behavior of the `Tokens` and `Elevator` models stabilizes only after the fifth log file. The aforementioned deviations are likely caused by the fact that the Java Virtual Machine (JVM) needs time to warm-up before reaching optimal performance. Consequently, it is advisable that the first five seconds of gathered measurements will not be considered during the performance analysis process, since it otherwise cannot be guaranteed that the measurements are behaviorally consistent.

Next, the throughput characteristics of each model will be highlighted individually, starting with the `CounterDistributor` model as reported in Section A.1. Within Figure A.1, it can be observed that the message throughput of the model seems to be consistent: the color of the figure is approximately uniform, with only a few gaps and outliers. The observed inconsistencies seem to be periodical, and as such, it is deemed unlikely that said inconsistencies are caused by an external factor, i.e., the fluctuations are likely caused by by the logging framework or the generated code. Additionally, the figure shows several short periods of approximately 40 milliseconds in which the throughput is momentarily lower than average. The latter is not periodical, and as such, it is presumed that the cause thereof is an external factor, i.e., the target program may have had to share system resources with the operating system or other processes running on the machine. The difference measure presented in Figure A.2 yields similar observations: the difference to the row minimum seem to be consistent in value, and all outliers and gaps correspond with those observed in Figure A.1. The latter is to be expected, since $T_s(i)$ is used in the definition of $T_d(i)$.

Table A.1 reports standard deviations of $5.94 \cdot 10^2$ and $4.63 \cdot 10^2$ respectively for the total and difference measurements, which indicates that all measured values are reasonably close to the mean and thus predominantly consistent. Furthermore, the mean and median values reported for all categories are similar in value, and as such, the distribution of throughput values seems to be relatively balanced and/or symmetrical. An unique characteristic of the throughput measurements of the `CounterDistributor` model is that the mean value of the difference measurement is substantial. The latter is explained by the fact that the average number of messages logged by the `Counter` state machine differs by nearly an order of magnitude from the value reported for the `Distributor` state machine. The latter is not an adverse effect of using logging-based measurements: instead, the observed behavior is a characteristic of the model, due to the number of transitions in each state machine, i.e., the `Counter` state machine only has a single transition to open, whereas the `Distributor` state machine has ten.

The `Tokens` model, with the results thereof having been reported in Section A.2, paints a different picture. Within Figure A.3, it can be observed that there are a substantial number of timestamps at which no log messages are registered. The large gaps seem to be periodical, with a period that

is approximately double as long as the period observed for the `CounterDistributor` model. On the other hand, the smaller gaps in the throughput measurements are less predictable. Given the sheer number of gaps, it may be the case that the logger cannot keep up with the influx of log messages. Furthermore, the density of gaps seems to be higher around the 160 millisecond mark. The latter may indicate that the gaps at the given location are related to the rollover procedure of log files, i.e., the pause may be caused by the compression of the previous log file. Nevertheless, the measurements outside of the gaps seem to be relatively consistent in value. The differences reported in Figure A.4 show only small fluctuations in value, and as such, the measurements seem to be mostly consistent. However, it needs to be noted that the plot highlights a substantial number of outliers, which indicates that one or more threads may occasionally be starved of system resources. The gaps are consistent with those observed in Figure A.3.

Furthermore, Table A.2 reports a standard deviation for the total throughput that is significantly higher than that reported for the `CounterDistributor` model. Furthermore, the median value of all categories is consistently larger than its mean value, which implies that the data is skewed towards higher values. Both observations are likely a product of the number of pauses in the data gathering. Oppositely, the mean and standard deviation of the difference measure is comparatively low, which implies that the difference measurements are relatively consistent. Lastly, the table shows that the number of messages is distributed evenly over all state machines, with the throughput of state machine `B` being only slightly higher than that of `A` and `C`.

The results of the `Elevator` model are presented in Section A.3. Figure A.5 shows characteristics that are very similar to those observed for the `Tokens` model: the heat map contains a lot of gaps, in which large gaps are periodical and small gaps are not. Furthermore, the density of gaps seems to be higher around the 160 millisecond mark. However, the throughput of the `Elevator` model differs in the sense that there are a substantial number of timestamps that show an abnormally high number of messages. The aforementioned outliers do not follow a periodical pattern. The difference characteristics displayed in Figure A.6 are analogous to those observed in Figure A.4, and as such, all of the same observations apply. Additionally, it needs to be noted that the outliers observed in Figure A.5 are not present in Figure A.6. As such, the outliers within the model do not seem to have a major impact on the distribution of messages between state machines. The results presented in Table A.3 are structurally similar to those listed in Table A.2, and as such, the throughput summary table of the `Elevator` model will not be discussed in further detail.

The results of the `ToadsAndFrogs` model are reported in Section A.4. The characteristics of the sum throughput that are displayed in Figure A.7 are similar to those observed for the `CounterDistributor` model. The heat map contains only a few gaps and outliers that appear to be periodical in nature, albeit with a shorter period than observed for the `CounterDistributor` model. Additionally, the throughput of messages seems to be largely consistent, with only minor color fluctuations. The `ToadsAndFrogs` model differentiates itself from other models by the relatively low number of log files that have been generated. Given the fixed size of log messages, it is hence implied that the average number of messages made by the model is considerably lower than that of its counterparts. The difference measure presented in Figure A.8 displays a high degree of color fluctuation, which may indicate that the message distribution between state machines lacks consistency. However, it also needs to be considered that the maximum value of the color scale is comparatively low, and as such, minor value differences will be more apparent in the figure.

The data reported in Table A.4 support said assessment. The standard deviation of both the total and difference metrics are lower than those reported for the `CounterDistributor` model, and as such, the throughput is considered to be consistent. Furthermore, the mean and median values reported for all categories are similar in value, and as such, the distribution of throughput values seems to be relatively balanced and/or symmetrical. An exception to the latter is the value reported for the difference metric, which seems to be skewed to lower values. Lastly, the table shows that the number of messages is distributed close to evenly over all state machines, with the throughput of state machine `control` being about 20% higher than that of `frog` and `toad`.

The results for the `Telephony` model are listed in Section A.5. The behavior displayed in Figure A.9 is a curious mixture of the throughput characteristics that have been observed for the previous models. For the first half of the log files, only a few gaps are observed, while the latter half contains a considerable number of gaps that do not seem to be periodical in nature. The outliers within the results seem to be periodical, with the period duration being slightly different before and after the observed tipping point. Otherwise, measured values seem to be generally consistent: small fluctuations are observed throughput the heat map, and the density of fluctuations seems to increase after the tipping point at approximately the 40 and 200 millisecond timestamps. Furthermore, the density of gaps is higher at the 200 millisecond mark, which is presumably caused by the rollover procedure of the log files. The measure of difference reported in Figure A.10 seem to be consistent. Additionally, the difference measurements contain remarkably few outliers, with most outliers occurring after the tipping point.

Finally, Table A.5 reports standard deviations for all measurements that are proportionally low when compared to the mean values. On top of that, the mean and median values of all categories are highly similar, which indicates that the measurements are not skewed and thus symmetrical. As such, the table supports the observation that the data is mostly consistent. Lastly, the table shows that the throughput of messages is spread evenly over all state machines.

Discussion In summary, it has been observed that there are a lot of behavioral differences between the log message throughput of the models. The `Tokens`, `Elevator` and `Telephony` models seem to struggle under the current configuration of the logger, given the number of pauses in the flow of data. As such, there is evidence that the settings of the logger are inadequate and need to be readjusted to compensate. However, it is likely that there does not exist a common configuration that works optimally for all models. Furthermore, the bottleneck may be located elsewhere, i.e., the observed behavior may be related to system limitations, such as the maximum speed at which data can be read from and written to disk. Nevertheless, consistency seems to be a common trend when the gaps in the data are not considered. Therefore, the investigation concludes that there is no immediate cause for concern in terms of message throughput.

8.3.4 Measurement Representativeness Analysis

Lastly, it needs to be verified whether the results attained through the logging-based measurements are representative of a normal run of the target program. The logging-based measurements are deemed representative if the distribution of transition executions over the model is similar to that of its counting-based counterpart. However, it needs to be stressed that it cannot be guaranteed that the results gathered by counting-based approach are fully representative of a program run without measurements. Nevertheless, said measurements are required to reason about the program's performance at an appropriate level of detail, and as such, measurements should be used that have minimal effect upon the performance of a program. Hence, the use of counting-based measurements as a baseline is deemed appropriate, since the latter are considered to be a class of non-intrusive and lightweight measurements. The distribution requirement can be verified through an arbitrary model: the comparison of distributions depends solely on the number of times that each transition is performed, i.e., the exact structure of the state machine is irrelevant. Therefore, the results of all models listed in Section 8.2 will be considered within the analysis.

Furthermore, the concurrency characteristics captured by the logging-based and counting-based measurements need to be comparable in terms of depicted behavior. Unfortunately, performing said comparison directly is a non-trivial task, since the activity span of transitions cannot be deduced from the data gathered by the counting-based measurements. Consequently, an alternative approach is required through which the concurrency characteristics of both approaches can be investigated based solely on the available data. The latter will be achieved by investigating the distribution of transition executions of a model that is designed to be highly sensitive to the degree of concurrency within a program, namely the `Tokens` model described in Section 8.2.2. The design of the `Tokens` model ensures that a run with a high degree of concurrency results in a higher

success ratio on the transitions from state `wait` to state `update` due to the fact that the program is more reactive to state changes, i.e., it should take less time for the partner state machine to consume the token and allow for said transition to be executed successfully. The success ratio of a transition affects the distribution of transition executions, and as such, the degree of concurrency can be observed through the latter. Conveniently, the `Tokens` model is already included within the test suite, and as such, no further adjustments are required to the analysis procedure.

The performance analysis of the decision structures and locking mechanism both have the default configuration of the code generator as one of the test cases, and as such, only the default configuration will be considered within the scope of this analysis. An exception to the latter is the `Telephony` model, which uses a random pick approach to resolve non-deterministic decisions due to the requirements of the model. Both of the performance metrics introduced in Section 8.1 will be reported upon in a series of bar plots and tables. The first metric is the number of transition executions (se) that have been completed successfully. The second metric is defined to be the ratio between the number of successful and total transition executions (sr). The bar plots and tables report the mean value of said metrics that have been measured over 20 trials. The figures will report the 95% confidence interval in the shape of error bars, whereas the table will instead report upon the standard deviation of the sample. On top of that, the table also contains the mean and standard deviation of the total number of transition executions.

The analysis of each model in the test suite will be structured as follows. First, a bar plot will be given that reports upon the aggregate success count (se) and ratio (sr) for each state machine in the target model. The aim of the first figure is to allow for discrepancies within the distribution of transition executions to be observed at a quick glance, i.e., major differences in the aggregate results imply that the distributions of the reported categories are dissimilar. The success count will be reported on the left-hand side of the provided figure, whereas the success to total ratio is reported as a percentage on the right-hand side. Furthermore, note that the scale of the bar plot depicting the counting data may differ depending on the range of values. The measurements are reported in linear scale on default, but occasionally, the measurements will be depicted in a symmetric-logarithmic scale such that a wide range of values can be reported in a compact manner: the symmetric-logarithmic scale adheres to a linear scale within the first column; all subsequent columns are logarithmic. Furthermore, observe that the use of a linear scale within the first column allows for the value zero to be included within the plot. The use of a symmetric-logarithmic scale will be stated explicitly within the axis label and the figure's caption.

Secondly, a bar plot will be presented that reports on the success count (se) and ratio (sr) for each individual transition within the model. Additionally, said metrics are provided for the root decision nodes of the states in each state machine, such that the effectiveness of the decision structure can be measured: the entries associated with a root decision node have a subject label that is rendered in a bold font. The detailed bar plot allows for the exact behavior of the program to be analyzed, i.e., it can be observed which transitions behave differently. The structure of the bar plot is equivalent to that of the first figure: the success count (se) and ratio (sr) metrics are reported on the left- and right-hand sides of the figure respectively, and a symmetric-logarithmic scale is used when applicable. Occasionally, the observations made at the hand of the bar plot will be substantiated by consulting the values reported within the associated table. The detailed bar plots and the associated data tables do not fit within the flow of the report due to their page-filling dimensions, and as such, said figures will be provided in Appendices B and C respectively.

8.3.4.1 Synthetic Test: CounterDistributor

The results for the `CounterDistributor` model will be discussed first. The aggregate success count and ratio values for the state machines `Counter` and `Distributor` are presented in Figure 8.3. Observe that the success count is reported on the left-hand side in symmetric-logarithmic scale. The success count shows that the number of successful transition executions captured by the counting-based measurements are at least one order of magnitude larger than the values reported

by the logging-based measurements. The latter shows that the use of a logging framework incurs a considerable performance overhead. Furthermore, the bars are not equally spaced, which shows that the state machines are not equally affected by the logging overhead. The success ratio of transitions in the state machine `Counter` are equivalent: the latter contains only a single transition that is always active, and as such, said behavior is to be expected. The ratio reported for the `Distributor` state machine differs slightly, with the counting-based approach outperforming the logging-based approach by a small margin. The confidence intervals shown in the figure are narrow, which implies that the gathered measurements remain consistent over all 20 trials.

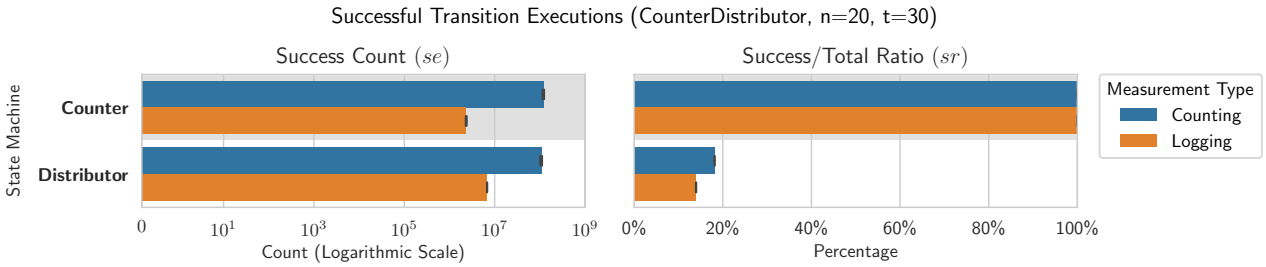


Figure 8.3: A bar plot that reports the number of successful executions (se) and the percentage of successful executions (sr) for each state machine in the target model `CounterDistributor`, where the results are grouped by the measurement type. Observe that the success count is depicted in a logarithmic scale, due to the wide range of measured values: the first column adheres to a linear scale; all subsequent columns are logarithmic. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

The success count and ratio measurements of all transitions and root decision nodes within the model are reported in Figure B.1. For the counting-based measurements, it is observed that the success count is equivalent for all transitions in state machine `Distributor`, i.e., the reported aggregate success count is distributed evenly over all transitions. The latter is not the case for logging-based measurements, which instead reports that the success count of said transitions follows a gradually increasing trend. Nevertheless, the difference between the counting-based and logging-based measurements remains substantial. For the success ratio, it is observed that the logging-based measurements for transitions in state machine `Distributor` are consistently lower than that of its counting-based counterpart. The sole exception is the last transition, which has a success ratio of 100% for both measurement types. Additionally, the success ratio follows an increasing trend for both types of measurements, which is the expected behavior, since the default configuration resolves non-deterministic decisions by sequentially visiting all options until an active transition is found. Furthermore, the decision is treated as an atomic operation, and as such, all variables used within the decision making process are locked until completion.

However, in an ideal situation, the reported success ratios of the two types of measurements should not differ, i.e., by the design of state machine `Counter`, each value of $x \in [0, 9]$ should be equally likely to occur, regardless of the chosen measurement type. The counting-based measurements are reported in Table C.1. The table shows that all transitions in state machine `Distributor` have an mean success count in the range $[1.06 \cdot 10^7, 1.09 \cdot 10^7]$, which implies that the successes are evenly distributed over the pool of transitions. As such, the logging-based measurements meet the expected behavior, since the successes will be distributed evenly if and only if each value of x is equally likely to occur. The logging-based measurements are listed in Table C.2, which reports that the success count of said transitions is in the range $[2.39 \cdot 10^5, 1.35 \cdot 10^6]$. The latter indicates that the successful executions of transitions are not evenly distributed. Therefore, the logging-based measurements deviate from the expected behavior: given the success counts reported for state machine `Distributor`, it would seem that higher values of x are more likely to occur. Regardless, the distribution of transition executions differs between the two types of measurements, and as such, the representativeness of the logging-based measurements is questioned.

8.3.4.2 Synthetic Test: Tokens

Next, the measurement representativeness will be investigated for the `Tokens` model. As stated previously, the `Tokens` model allows for the degree of concurrency to be quantified. As such, the model can be used to compare the concurrency characteristics of the counting-based and logging-based measurement types. The aggregate success count and ratio values of the state machines `A`, `B` and `C` are presented in Figure 8.4. Once again, the success count reported by the counting-based measurements are larger than those given by its logging-based counterpart, with the difference being over an order of magnitude. The success count reports an interesting balance between the state machines–for both types of measurements, state machines `A` and `C` report similar counts, whereas state machine `B` seems to outperform the former two state machines by a respectable margin. The success ratio of the logging-based measurements is slightly higher than that of its counting-based counterpart, and hence, it is likely that the distribution of transition executions differs for both measurement types. The confidence intervals shown in the figure are relatively narrow, which implies that the recorded measurements are consistent over all 20 trials.

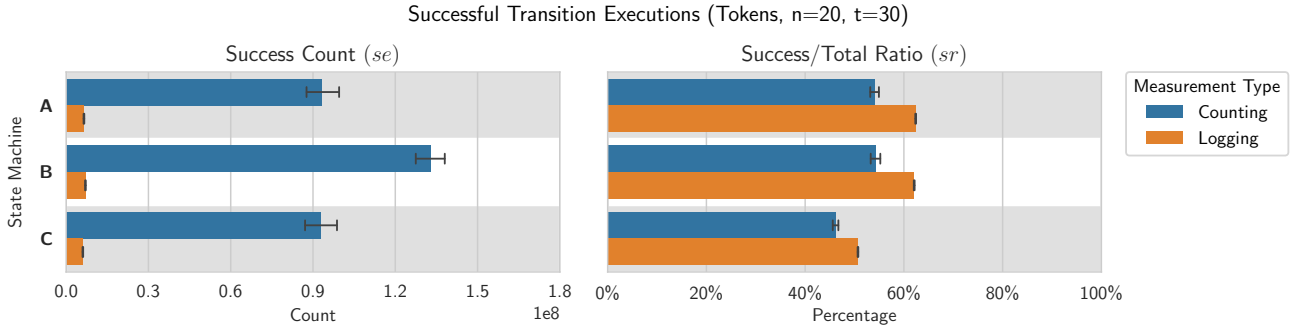


Figure 8.4: A bar plot that reports the number of successful executions (se) and the percentage of successful executions (sr) for each state machine in the target model `Tokens`, where the results are grouped by the measurement type. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

The success count and ratio measurements of all transitions and root decision nodes within the model are reported in Figure B.2. Observe that the success count is reported in a symmetric-logarithmic scale. On top of that, the associated counting-based and logging-based measurements are listed in Tables C.3 and C.4 respectively. Through visual inspection, it is observed that the success counts reported by the counting-based and logging-based approaches do not seem to follow a similar distribution: the bars are not equally spaced, and as such, the recorded differences are not of a constant order of magnitude. Furthermore, major differences in success ratio can be observed for transitions between states `wait` and `update`, where the success ratio of the logging-based approach is more than double that of its counting-based alternative. As discussed in Section 8.2.2, the success ratios of the aforementioned transitions depends strongly on the degree of concurrency within the program: a high-valued success ratio implies a high degree of concurrency and vice versa. Therefore, it is implied that the logging-based measurements display a higher degree of concurrency than the counting-based measurements. Consequently, there are severe doubts about the representativeness of the logging-based measurements, given that both the distribution of transition executions and the concurrency characteristics are dissimilar.

8.3.4.3 Practical Test: Elevator

Within the scope of this section, the representativeness of the logging-based measurements will be analyzed for the `Elevator` model. The aggregate success count and ratio values for the state machines `cabin`, `controller` and `environment` are visualized in Figure 8.5. The success count bar plot indicates that the counting-based approach produces approximately five times as many

successful transition executions than its logging-based counterpart, which is the smallest difference encountered so far. The success ratios reported by the logging-based measurements are consistency higher than that of its counting-based counterpart. The differences in success ratio are minor for the state machine `controller`, but more substantial for the state machines `cabin` and `environment`. Furthermore, note that the confidence intervals bars within the figure are relatively narrow, which indicates that all 20 runs of the program produced similar results.

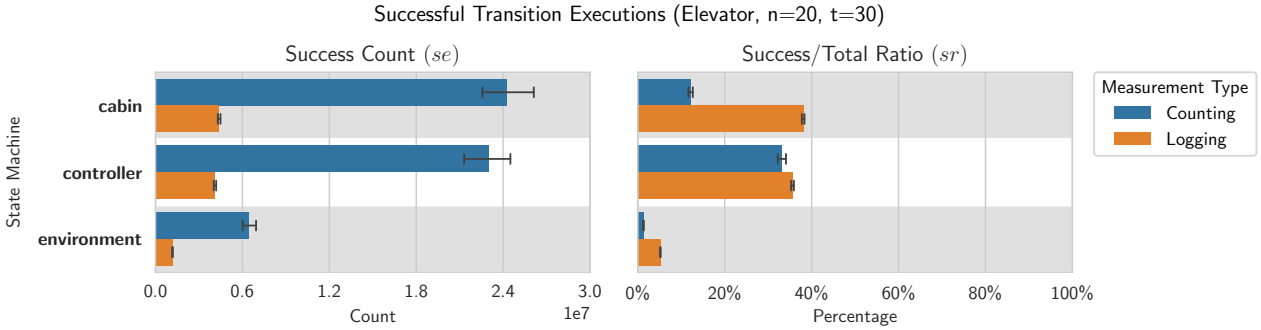


Figure 8.5: A bar plot that reports the number of successful executions (se) and the percentage of successful executions (sr) for each state machine in the target model `Elevator`, where the results are grouped by the measurement type. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

The success count and ratio of each transition and root decision node within the model is illustrated in Figure B.3, which presents the success count bar plot in symmetric-logarithmic scale. Additionally, the counting-based and logging-based measurements associated with the figure are listed in Tables C.5 and C.6 respectively. For the success count, it is observed that visually, the differences between the counting-based and logging-based measurements are of equal size, and as such, both types of measurement adhere to a similar distribution of successful transition executions. The sole exception to the latter is the self-loop transition in state `work` of the state machine `controller`, for which the difference between the two measurements is considerably larger. Oppositely, the values presented for the success ratio show clear differences: the logging-based measurements present values that are consistently larger than those of its counting-based counterpart. The success ratio differences within the state machine `controller` are minor, whereas the differences in the `cabin` and `environment` state machines are more sizeable. The reported success ratios indicate that the distribution of transition executions is different between the two measurement types, and therefore, the representativeness of logging-based measurements is questioned.

8.3.4.4 Practical Test: ToadsAndFrogs

The measurement representativeness for `ToadsAndFrogs` model will be investigated next. The aggregate success count and ratio values of the state machines `control`, `frog` and `toad` are presented in Figure 8.6. The success count differs significantly between the counting-based and logging-based measurements: the former shows considerable differences between the success counts of each individual state machine, whereas the latter reports that the successes are distributed evenly instead. Consequently, the magnitude of the difference varies per state machine: the minimum difference is a factor of four, whereas the maximum is approximately a factor of eight. Similarly, the success ratio shows mixed results. The success ratio reported by the logging-based measurements is marginally lower for state machine `control`, about double as high for state machine `frog`, and moderately higher for state machine `toad`. The confidence intervals included within the figure are relatively narrow, and as such, all 20 runs report matching results.

The success count and ratio measurements of all transitions and root decision nodes within the `ToadsAndFrogs` model are reported in Figure B.4. On top of that, observe that the success count

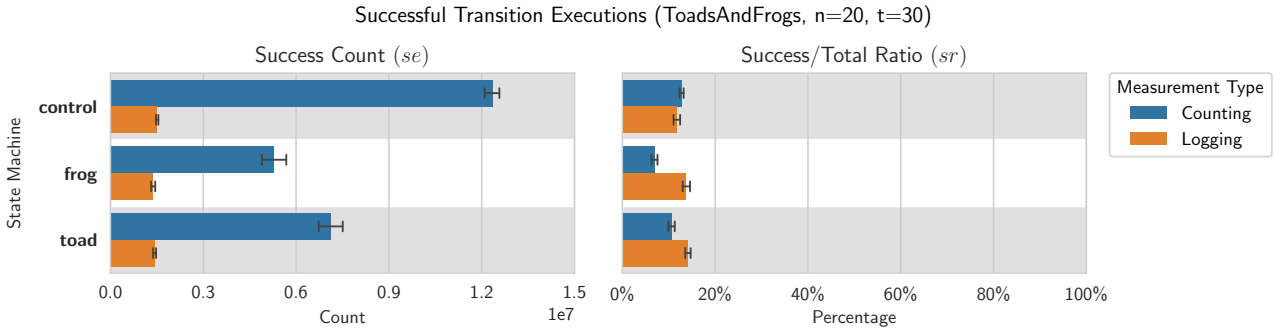


Figure 8.6: A bar plot that reports the number of successful executions (se) and the percentage of successful executions (sr) for each state machine in the target model `ToadsAndFrogs`, where the results are grouped by the measurement type. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

is reported in symmetric-logarithmic scale. The counting-based and logging-based measurements associated with the figure are listed in Tables C.7 and C.8 respectively. The success counts given by the counting-based and logging-based measurements differ considerably. The differences are most notable in state `running` of state machine `control`: the logging-based measurements report that the successful executions within said state are distributed relatively evenly over all transitions therein, whereas the counting-based counterpart heavily favours two of the five transitions. Additionally, sizeable differences are observed for the state `success` as well. Based on the measurements reported in the associated tables, it is clear that the logging-based approach has a higher chance at winning the game: the counting-based approach indicates that the game is won on average 0.80 times per run, whereas its logging-based counterpart wins 36.6 times per run. Given the observed differences, it is clear that the distribution of transition executions differs considerably between both types of measurements. Similar differences can be observed within the given success ratios: a vast amount of the ratios reported by the counting-based approach are notably close to zero, whereas the logging-based approach reports ratios that are consistently higher than 2%. An exception to the latter is the first transition in state `done` of state machine `control`, which reports a success ratio of $2.58 \cdot 10^{-7}$ for the counter-based measurements and $1.01 \cdot 10^{-4}$ for its logging-based counterpart. Therefore, it is safe to conclude that the logging-based measurements are not representative of a normal run of the `ToadsAndFrogs` model.

8.3.4.5 Practical Test: Telephony

Lastly, the results for the `Telephony` model will be discussed. Once again, it needs to be stressed that a different default configuration is used for the `Telephony` model: the latter does not function correctly when non-deterministic decisions are resolved through a sequential decision process, and as such, the code generator has been configured to use a random pick approach instead. The aggregate success count and ratio values of the state machines `User_0`, `User_1`, `User_2` and `User_3` are reported in Figure 8.7. The success count bar plot indicates that the counting-based approach produces approximately ten times as many successful transition executions than its logging-based counterpart, and as such, the productivity of both measurement types differs by an order of magnitude. Furthermore, the reported success ratios are similar for all state machines, regardless of the used measurement type. The confidence intervals reported for the success count are narrow for the logging-based measurements, but relatively wide for its counting-based counterpart. Likewise, the confidence intervals of the success ratio are remarkably broad when compared to results presented by previous models, which is a sign of concern in terms of measurement consistency. Nevertheless, all state machines seem to be affected by an equal degree, and as such, it is presumed that the higher degree of variation will not subvert the significance of the earlier observations.

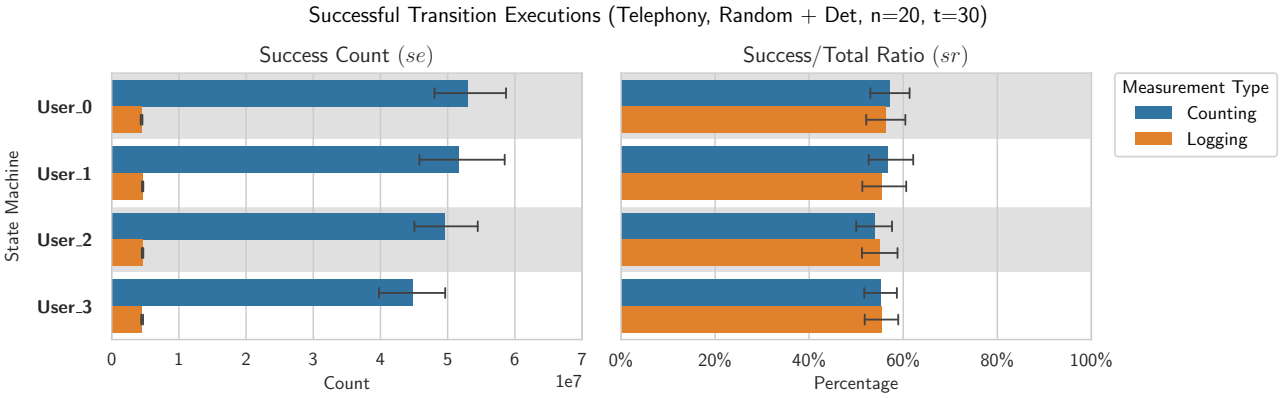


Figure 8.7: A bar plot that reports the number of successful executions (se) and the percentage of successful executions (sr) for each state machine in the target model `Telephony`, where the results are grouped by the measurement type. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

The success count and ratio measurements of all transitions and root decision nodes within the `Telephony` model are reported in Figures B.5-B.8, where each figure reports the data associated with state machine `User_0`, `User_1`, `User_2` and `User_3` in the given order. Additionally, observe that the success count is given in symmetric-logarithmic scale. The associated counting-based and logging-based measurements are presented in Tables C.9-C.12 and Tables C.13-C.16 respectively. For regularly occurring transitions, it is observed that the difference between the mean success count values reported by both types of measurement are fairly constant in terms of percentages, i.e., the size difference between the bars of both categories seems to be of constant width. However, the bar plots also show that the remaining transitions are visited solely by the counting-based measurements, albeit sporadically. Said observation is supported by the values listed within the associated tables, which report that said transitions have not been recorded by the logging-based measurements. On the other hand, it is observed for state machine `User_2` that the confidence intervals of the counting-based measurements are exceptionally broad for the majority of the reported values, especially on the lower end of the spectrum. A similar observation can be made for state `tconnected` in state machine `User_3`. As such, the values are shown to be highly valuable, and thus, the reliability of the gathered message count data is questioned.

The success ratio shows that the majority of commonly executed transitions always succeed. The remainder of the transitions show mixed results, but generally, it is observed that the reported means are within 5% of each other. The differences in success ratio are substantial for sporadically visited transitions. The latter is logical, since the logging-based measurements never visit said transitions: consequently, the success ratio is listed as an undefined number in the tables. Lastly, it needs to be observed that the confidence intervals of the counting-based measurements show a considerable amount of variation for several measurements, whereas the measurements of the logging-based counterpart seem mostly consistent. An exception to the latter are the transitions between states `idle` and `dialing`, for which a high variance in success ratio is reported for both types of measurements. Based on the observations made above, the measurement representativeness of the data is difficult to interpret: the set of transitions visited by both measurement types is not consistent, and the attained results show a high degree of variance. Hence, the measurement representativeness for the `Telephony` model remains unclear. Furthermore, the suitability of the `Telephony` model as a test program is questioned due to the observed inconsistencies.

8.3.5 Viability of the Logging-Based Measurements

Within Section 8.3, concerns have been raised about the possibility that the logging-based measurements may inadvertently alter the performance characteristics of the target program. In response, said concerns have been investigated, which has led the following findings. First, it has been shown in Section 8.3.3.1 that asynchronous logger provided by the Log4j logging framework adheres to the strict FIFO ordering that is required to extract meaningful information from the gathered data. Secondly, the investigation conducted in Section 8.3.3.2 concludes that the message throughput of the logger and the distribution of messages over the state machines are generally consistent, and as such, there is no immediate cause for concern in terms of message throughput. On the other hand, the analysis of the representativeness of the logging-based measurement in Section 8.3.4 has yielded conflicting results: for all target models, it has been observed that the distribution of transition executions depicted by the counting-based and logging-based measurements are dissimilar. Therefore, the behavior captured by the logging-based measurements does not represent the performance characteristics that a normal run of the affected program would display, i.e., the use of logging-based measurements alters the behavior of the program.

Due to the latter observation, it has been decided that it is inappropriate to use the logging-based measurements within the performance analysis–after all, the aim of the performance analysis is to interpret the behavior of the target program, and not that of the Log4j logging framework. Nevertheless, it needs to be stressed that the aforementioned decision does not imply that the results measured by the logging-based approach are inferior to those reported by its counting-based counterpart. On the contrary: it has been observed at multiple occasions that the logging-based approach increases the efficiency of the target program, i.e., the success ratio of transitions is comparatively higher than those reported by its counting-based counterpart. Consequently, the difference in performance is an aspect that may warrant further investigation.

Moreover, further investigation into the generated code has revealed that the logging-based measurements cannot fulfill their intended purpose in their current shape, due to the fact the implementation thereof is inherently flawed. As discussed in Section 8.3.1, the execution period of a transition is defined to be the span of time between its commencement and conclusion. Therefore, it is crucial that the logging statements that process said events are placed at an appropriate location within the generated code, such that the desired behavior can be captured. Within the current implementation of the logging-based measurements, the aforementioned logging operations are executed prior to and directly after the call to the method that attempts to fire the associated transition. Consequently, it is possible for a transition to halt during its execution period, i.e., the transition may perform several lock acquisition operations within the scope of its execution, and as such, the associated thread may be blocked until ownership over said lock objects is attained. The latter is problematic, since the described behavior is not captured by the logging procedure, i.e., the sequence of events in the log files may indicate that two transitions run concurrently, while in reality, said transitions are forced to run in sequence by the locking mechanism.

Finding a resolution to said shortcoming is complicated. Theoretically, the log messages that signify the start of a transition could be moved, such that the logging operations are performed within the active region of the acquired locks, i.e., the commencement of a transition should only be registered once all of its locks have been acquired. However, due to the fine-grained nature of the locking mechanism, said moment is not clearly defined, e.g., it is possible that the active regions of two locks used by the transition do not intersect. Furthermore, the suggested solution deviates from the given definition of a transition's execution period. Specifically, the execution period that is reported by said approach is not guaranteed to cover the complete execution period of the transition, which in turn may result in actual concurrent behavior not being captured by the logging-based measurements. Consequently, a more refined approach will be required to effectively measure and report upon the concurrency within a program's execution. Nevertheless, the position of the logging statements does not influence the measurements that the decision of not using the logging-based measurements is based on, and as such, said resolution remains unchanged.

8.4 Decision Structure Analysis

The discussion of the intricacies of the performance analysis has been concluded, and therefore, the chapter may proceed to the investigation of the gathered measurements. To start with, the performance characteristics of the revised decision structures will be investigated. The aim of the analysis is to determine the effectiveness of the improvements made to the decision making process in Chapter 4, i.e., it needs to be verified whether the inclusion of deterministic structures leads to measurable improvements in terms of efficiency and/or performance. Moreover, it needs to be determined which approach to resolving non-deterministic decisions attains the most optimal results. The available solver configurations are not considered within the scope of this analysis, due to the fact that none of the selected target models take full advantage of said solver configurations, i.e., all solver configurations generate the same decision structure, and as such, the inclusion thereof does not produce any meaningful differences in efficiency or performance.

The collection of command-line arguments through which the construction of decision structures can be configured are provided in Section 6.3.1. A combination of configuration options will be referred to as a decision mode. The decision modes that will be evaluated within the scope of this analysis are presented in Table 8.1, which lists the identifier of the decision mode and the associated command-line arguments. The analysis of the decision structures will adhere to an analysis process that is analogous to the one used by the measurement representativeness analysis in Section 8.3.4, with the sole difference being the categories that are targeted during the investigation, i.e., the decision modes will be assessed instead of the measurement types. As discussed in Section 8.3.5, the logging-based measurements are not representative of a normal run of the program, and as such, the performance analysis will be based solely on the counting-based measurements.

Decision mode	Command-line arguments
Sequential + Det	
Random + Det	`-use_random_pick`
Sequential	`-no_deterministic_structures`
Random	`-use_random_pick`, `-no_deterministic_structures`

Table 8.1: The decision modes that are explored during the performance analysis of the decision structures. For each decision mode, the associated command-line arguments are listed.

8.4.1 Synthetic Test: CounterDistributor

The results of the `CounterDistributor` model will be discussed first. The aggregate success count and ratio values for state machines `Counter` and `Distributor` are given in Figure 8.8. For state machine `Counter`, it is observed that all decision modes perform roughly the same number of successful transition executions. Oppositely, it is shown for state machine `Distributor` that the random approach without deterministic structures performs severely worse than its alternatives in terms of success count. In contrast, the success count of the remaining decision modes are observed to be alike. Similar results are reported for the success ratio, which shows that the random decision mode has a lower chance to succeed in state machine `Distributor` than its counterparts, and once again, the results given for the remainder of the decision modes are practically identical. The fact that the first three decision modes report similar results is to be expected, given the fact that said decision modes generate the same code for this particular model. The latter shows why it is crucial to take the variance into consideration: minor differences can be observed within the success count bar plot, but due to the variance within the measurements, said differences fall within the margin of error and are therefore insignificant.

The success count and ratio measurements of all transitions and root decision nodes contained within the model are illustrated in Figure B.9. In addition, observe that the success count is reported in symmetric-logarithmic scale. For the sake of completeness, the exact values of the

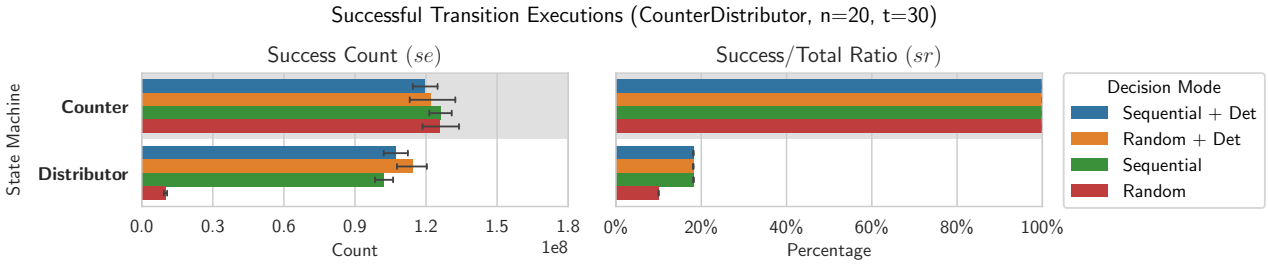


Figure 8.8: A bar plot that reports the number of successful executions (se) and the percentage of successful executions (sr) for each state machine in the target model `CounterDistributor`, where the results are grouped by the decision mode. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

measurements presented by the figure are provided in Tables C.1, C.17, C.18 and C.19 respectively. Through visual inspection, it is observed that the success counts reported by each decision modes are virtually identical to each other, with the exception of the results reported by the random approach for the `Distributor` state machine, which shows values that are lower by an order of magnitude. Furthermore, the whiskers of the confidence interval are narrow, which indicates that the measurements taken during all of the 20 trials show consistent results. The reported success ratios show predictable results. First of all, the state machine `Counter` has only a single transition without a guard, and as such, the execution thereof is always successful regardless of the decision mode used. On top of that, a 10% success ratio for picking an active transition is reported for the root decision node of state machine `Distributor`, which contrasts the 100% ratio that is reported by all of its counterparts. The latter is expected behavior, given to the fact that both the deterministic structures and the sequential approach to resolving non-deterministic decisions continue to visit options until no transitions remain or an active transition is found. On top of that, the choice in the decision node is mutually exclusive, and as such, the transition selection will always proceed. Said observation is supported by the success ratios reported for the individual transitions: the success ratio of the random decision mode remains constant, whereas the ratio reported by its alternatives gradually increases, i.e., the success ratios reported for the latter category follow a trend that corresponds to the principles of sampling without replacement.

Based on the observations made throughout this section, it is safe to conclude that deterministic structures have a positive effect upon both the performance and efficiency of the code generated for the `CounterDistributor` model. Similarly, the use of a sequential approach to the resolution of non-deterministic decisions is encouraged, but not strictly necessary: the deterministic structures are capable of constructing more complex and fine-grained decision structures, and as such, the use thereof should be prioritized over the application of sequential decision making.

8.4.2 Synthetic Test: Tokens

Next, the performance characteristics of the decision modes will be investigated for the `Tokens` model. The aggregate success count and ratio for the state machines `A`, `B` and `C` are presented in Figure 8.9. The success count shows that the decision modes that use a sequential approach to the resolution of non-deterministic decisions outperform their random picking counterparts by a respectable margin. Furthermore, it can be observed that the use of deterministic structures is beneficial to the performance of state machines `A` and `C` if random picking is used, but yet, not benefit is observed for state machine `B`. Furthermore, it is observed that state machine `B` achieves success counts that are consistently higher than that of state machines `A` and `C` by a significant degree. The latter two observations are unexpected, given that the implementations of all state machines are structurally equivalent. The success ratio shows that the random decision mode consistently performs worse than its alternatives. The remaining decision modes report success

ratios that are practically identical, i.e., the minor differences that can be observed fall within the margin of error. The confidence intervals shown for the success count and ratio are relatively narrow, which implies that the recorded measurements are consistent over all 20 trials.

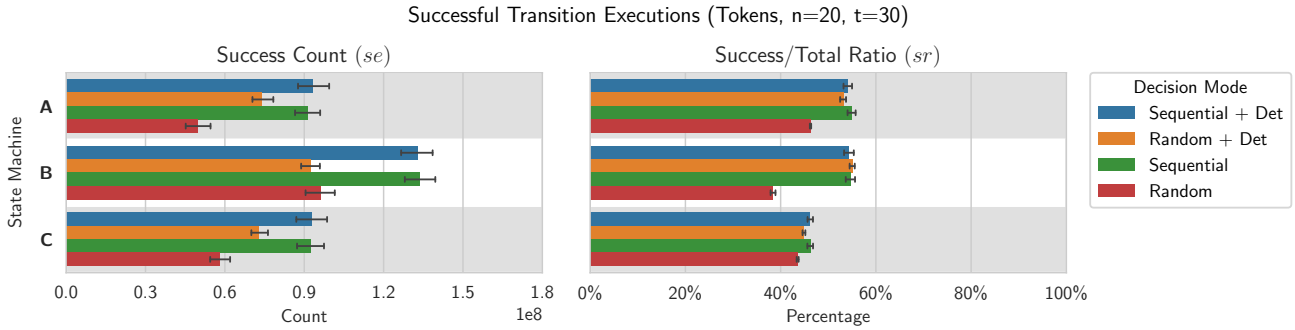


Figure 8.9: A bar plot that reports the number of successful executions (se) and the percentage of successful executions (sr) for each state machine in the target model `Tokens`, where the results are grouped by the decision mode. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

The success count and ratio for each transition and root decision node contained in the `Tokens` model are given in Figure B.10, which presents the success count bar plot in symmetric-logarithmic scale. For the sake of completeness, the exact values of the measurements presented by the figure are provided in Tables C.3, C.20, C.21 and C.22 respectively. For the success count, it is observed that the decision modes that use a sequential approach consistently outperform their random pick counterparts. Oppositely, the success ratio reports mixed results. For state `act`, it is observed that the use of a sequential approach increases the success ratio to 100%. Additionally, it is observed that the use of deterministic structures improves the overall success ratio of state `act`, but only by a limited degree. Curiously, no decisive ranking can be observed for the success ratios in state `wait`, given that each state machine reports different behavior.

Nevertheless, it can be concluded that a sequential approach to decision making with or without deterministic structures outperforms its random picking counterpart in the majority of cases. On top of that, the inclusion of deterministic structures within the decision structure seems to lead to a marginal increase in terms of efficiency and performance.

8.4.3 Practical Test: Elevator

The `Elevator` model will be considered next. The aggregate success count and ratio for state machines `cabin`, `controller` and `environment` are given in Figure 8.10. Unfortunately, no conclusive ranking can be reported based on the success count alone. At first glance, it seems safe to conclude that the random decision mode is outperformed by its counterparts. However, the whiskers of the confidence intervals are comparatively wide, and as such, it is difficult to determine which remaining option is the most optimal given the margin of error. Oppositely, the behavior shown by the success ratio seems more decisive. For state machine `controller`, it is shown that all available modes outperform the random decision mode by a considerable margin. The success ratios reported for the other state machines show that all of the decision modes produce similar results. Peculiarly, slightly higher success ratios are reported for the random decision mode in state machine `cabin` and the random with deterministic structures decision mode in state machine `environment`. The cause of the latter remains unclear. The confidence intervals of the success ratio are relatively narrow, and as such, it can be assumed that said results are accurate.

The success count and ratio of the transitions and root decision nodes contained within the `Elevator` model are listed in Figure B.11. Note that the success ratio is reported in symmetric-logarithmic scale. For the sake of completeness, the exact values of the measurements presented

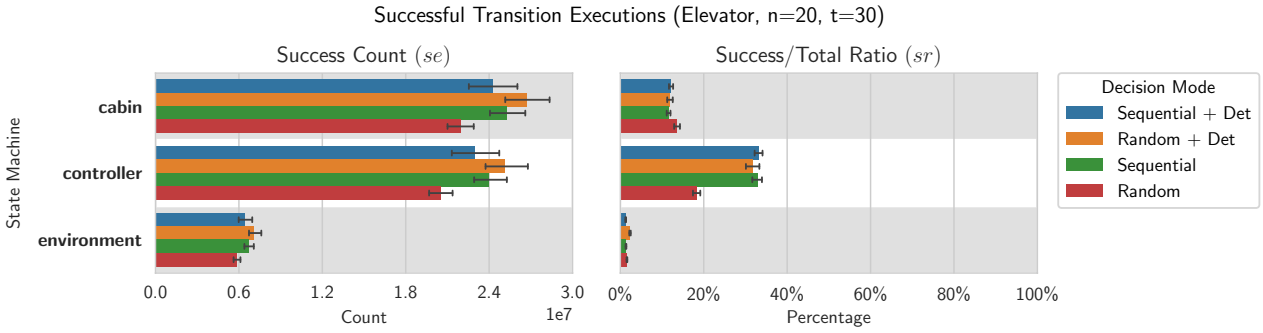


Figure 8.10: A bar plot that reports the number of successful executions (se) and the percentage of successful executions (sr) for each state machine in the target model `Elevator`, where the results are grouped by the decision mode. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

by the figure are given in Tables C.5, C.23, C.24 and C.25 respectively. The success count does not report any significant differences, except for the last transition going from state `work` to `work` in the state machine `controller`, which indicates that the random mode reports a success count that is over an order of magnitude less than its alternatives. Similarly, the success count of the random mode with deterministic structures is lower, but only by a comparatively small amount. Furthermore, the confidence intervals shown for the latter transition are comparatively wide when compared to that of other transitions, which may indicate that the activation of said transition depends heavily upon concurrent behavior, i.e., the transition is reactionary in nature.

On the other hand, mixed results are observed for the success ratio. For state `idle` of state machine `cabin`, it is observed that the random decision mode reports a success ratio that is about a percentage higher than that of its counterparts. Oppositely, all of the other transitions in the state machines `cabin` and `controller` report success ratios for the random decision mode that are consistently lower than that of the other decision modes. In addition, the latter category reports success ratios that are within margin of error of each other. In contrast, different behavior is observed for state `read` of state machine `environment`. For the root decision node, it is reported that a sequential approach to resolving non-deterministic decisions more than doubles the chances of success, whereas the transitions themselves show that a random approach with deterministic decisions yields the best results. The difference between the success ratios of the root decision node and the transitions contained therein is logical, given that the sequential approach is guaranteed to visit all options until an active transition is found, whereas the random approach only visits one option per selection routine. Moreover, note that the decision between the transitions starting in state `read` is non-deterministic, due to the fact that multiple transitions may be active at the same time. Consequently, the use of deterministic structures does not affect the success ratio of the root decision node, since the sequential nature of a deterministic structure cannot be exploited if no deterministic behavior is present within the selection procedure.

Lastly, it is curious that several of the root decision nodes report success ratios that differ per decision mode, even though said decision nodes contains only a single transition, i.e., the same code is generated for all decision modes. The latter indicates that the success ratio of said transitions is determined by an outside source, which in turn leads to two observations: the success ratio is influenced by the concurrency characteristics of the generated program; the success ratio demonstrates that the decision modes have an impact on said concurrency characteristics.

Nevertheless, the analysis of the `Elevator` model has led to the following conclusions. First of all, the values reported for the random decision mode are consistently worse than that of its counterparts in terms of both efficiency and performance, and as such, the random decision mode should be avoided. Furthermore, similar performance characteristics are reported for the remaining

decision modes, and as a result, said decision modes are considered to be equally viable.

8.4.4 Practical Test: ToadsAndFrogs

Next, the `ToadsAndFrogs` model will be investigated. The aggregate success count and ratio of state machines `control`, `frog` and `toad` are presented in Figure 8.11. The success count indicates that the random decision mode is outperformed by its counterparts in state machines `control` and `toad`, whereas the same performance is achieved in state machine `frog`. The latter is unexpected, given that fact that the implementations of state machines `frog` and `toad` are virtually identical, i.e., the associated player merely move in opposite directions and the types of moves are in matching order. The differences observed for the remaining decision modes fall within the margin of error and are thus insignificant. The success ratio shows a similar trend, i.e., the reported values seem to be practically identical for all decision modes. The random decision mode is an exception to the latter: a minor increase in success ratio is observed in state machine `control`, whereas a decrease in success ratio is observed in state machine `toad`. The confidence intervals included within the figure are relatively narrow, and as such, all 20 runs seem to report consistent results.

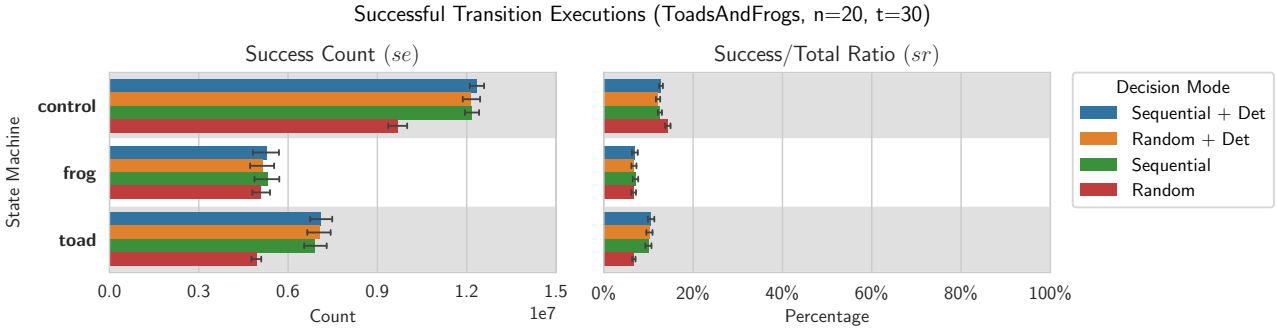


Figure 8.11: A bar plot that reports the number of successful executions (se) and the percentage of successful executions (sr) for each state machine in the target model `ToadsAndFrogs`, where the results are grouped by the decision mode. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

The success count and ratio measurements of all transitions and root decision nodes contained within the model are given in Figure B.12. Furthermore, observe that the success count is reported in symmetric-logarithmic scale. For the sake of completeness, the exact values of the measurements presented by the figure are provided in Tables C.7, C.26, C.27 and C.28 respectively. The success count shows interesting results in terms of the dynamics of the game itself: the random decision mode seems to lead to a lot more win states than its counterparts, with said difference being over two orders of magnitude. The latter can be explained by the fact that the random decision mode seems to create move sequences that are more arbitrary in nature, which is substantiated by the observation that a considerable increase in success count can be observed in several transitions starting in state `running` of state machine `control`. Similar observations can be made for the last two transitions of state machines `frog` and `toad`, which also show a considerable increase in success count when the random decision mode is used. At the same time, no significant differences are observed within the success counts of the remaining decision nodes.

The success ratio shows that the chance of winning the game is still incredibly low: the success ratios of the transition between states `done` and `success` of state machine `control` show that the chances of winning the game are $2.58 \cdot 10^{-7}$, $2.98 \cdot 10^{-7}$, $2.95 \cdot 10^{-7}$ and $3.28 \cdot 10^{-4}$ respectively. Likewise, low success ratios can be observed for several other transitions in the model, namely those in state `running` of state machine `control` and the last two transitions of state machines `frog` and `toad`. On the other hand, all other modes of decision making seem to be behaving similarly in terms of success ratio, with the observed differences falling within margin of error.

Furthermore, it can be observed that deterministic structures are generated for several decision nodes, which is evidenced by the fact that the success ratios of the random and random with deterministic structures decision modes differ in the majority of the root decision nodes.

Lastly, the cause of the disparity between the performance characteristics of the state machines `frog` and `toad` remains unclear. The transitions and root decision node of state machine `toad` seems to have higher success ratios than that of state machine `frog`, which could explain the fact that said transitions have a higher success count as well. Hypothetically, the difference in success ratio may be caused by a non-critical race condition between the two state machines, i.e., the success ratio is highly dependent upon the mutual timing of transition activations. By the construction of the model, it can be observed that two consecutive moves by the same state machine could result in a game state in which the other state machine is no longer capable of making moves until the end of the game. Consequently, the impact of said race condition may be exacerbated by the occurrence of the aforementioned snowball effect. As such, the observed behavior may be explained by the possibility that the state machine `toad` is, for one reason or another, more likely to make two consecutive turns. Within the generated code, it can be observed that the `toad` thread is instantiated before the `frog` thread, and as such, it is presumed that the starting order of threads may have an influence on the concurrency characteristics of the generated program. By construction, both threads are assigned the same (default) priority, and therefore, one may expect that all threads are weighted equally. However, in reality, it may be the case that a FIFO prioritization is adhered to if threads are assigned equal priorities[3]. Regrettably, the data required to verify said claims cannot not be gathered due to time limitations.

Based on the observations made throughout this section, it is safe to conclude that the random decision mode performs best in terms of the dynamics of the game depicted by the model. In contrast, it has been observed that the random decision mode should be avoided if the aim is to maximize the performance and efficiency of the generated program. Similar performance characteristics are reported for the remaining decision modes, and as such, they are equally viable.

8.4.5 Practical Test: Telephony

Finally, the results for the `Telephony` model will be discussed. First, it needs to be stressed that a different set of decision modes will be used for the `Telephony` model: the latter is known for not functioning correctly when non-deterministic decisions are resolved through a sequential decision process, and therefore, only the decision modes using random picking will be considered. On top of that, concerns have been raised in Section 8.3.4.5 about the viability of the model as a test case due to the inconsistent and highly variable nature of the gathered measurements. Nevertheless, an effort will be made to discuss the results, albeit at a more superficial level if necessary.

The aggregate success count and ratio of state machines `User_0`, `User_1`, `User_2` and `User_3` are visualized in Figure 8.12. For all state machines, it is observed that the success count of the random decision mode is lower than of its deterministic structures counterpart by a significant margin. Nevertheless, the reported confidence intervals are relatively wide, and as such, it cannot be guaranteed that the aforementioned ranking holds for all combinations of runs. Similarly, it is observed for all state machines that the success ratio of the random decision mode is approximately 10% less than its alternative. The confidence intervals of the success ratios are comparably wide as well, but no overlaps are observed between the confidence intervals of different categories, and as such, the aforementioned ranking remains applicable regardless of the reported variance.

The success count and ratio measurements of all transitions and root decision nodes within the `Telephony` model are reported in Figures B.13-B.16, where each figure reports the data associated with state machine `User_0`, `User_1`, `User_2` and `User_3` in the given order. Furthermore, observe that the success count is reported in symmetric-logarithmic scale. For the sake of completeness, the exact values of the presented measurements are provided in Tables C.9-C.12 and C.29-C.32

[3] `https://www.baeldung.com/java-thread-priority#overview-of-thread-execution` (visited on 29-09-2022)

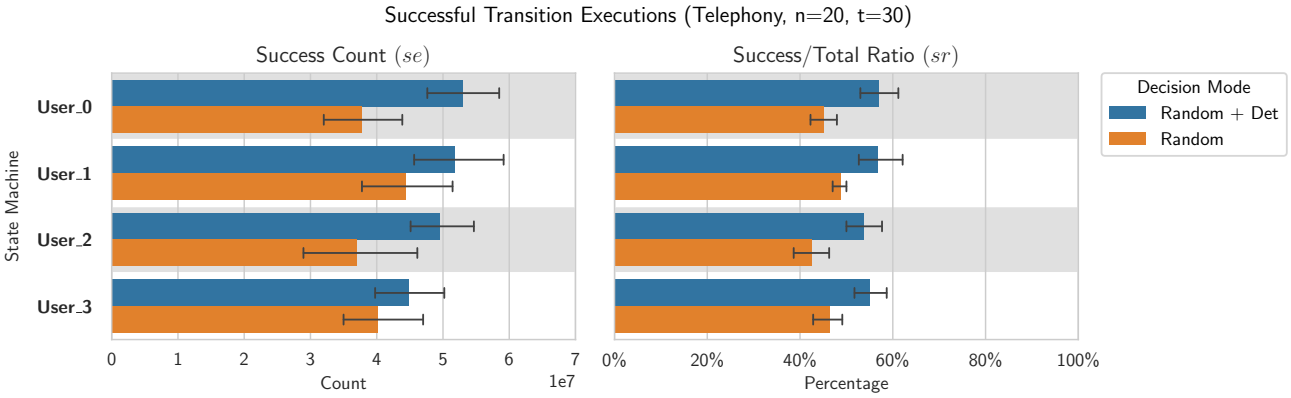


Figure 8.12: A bar plot that reports the number of successful executions (se) and the percentage of successful executions (sr) for each state machine in the target model `Telephony`, where the results are grouped by the decision mode. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

respectively. Firstly, it needs to be observed that the majority of transitions in state machines `User_1` and `User_2` once again report an abnormally wide confidence interval for the success count, with said variance being located predominantly in the lower end of the spectrum. The variance is not consistent in terms of decision mode: the aforementioned variance is observed for the random decision mode in state machine `User_1`, whereas state machine `User_2` reports said variance for its counterpart instead. Regrettably, the exact cause of the described behavior remains unclear.

For the success count, it is observed that the random decision mode is consistently outperformed by its deterministic structure counterpart within the majority of the transitions. A clear exception to the latter is the success ratio reported for state `tconnected` in state machine `User_2`, which shows a comparatively high success count for the random decision mode. The exact opposite is observed for state `tconnected` in state machine `User_3`, which instead reports a high success count for its counterpart. However, at the same time, it is also noted that certain transitions are visited solely by the random decision mode, albeit sporadically.

The success ratio shows that the use of deterministic structures generally leads to more optimal behavior: the latter is especially noticeable for the transitions that start in state `calling` of all state machines. In contrast, it can be observed for the sporadically visited transitions that the random decision node reports a success ratio, whereas it counterpart does not. The latter is logical, given that said transitions are not visited by its counterpart. Nevertheless, the results do report several instances in which the random decision mode outperforms its counterpart, and as such, its is not definitive that the use of deterministic structures is the best choice for all transitions and root decision nodes. Lastly, it needs to be noted that both decision nodes occasionally report confidence intervals for the success ratio that are comparatively wide, which indicates that the behavioral characteristics of the model varies considerably from run to run.

Based upon the results of the investigation, it can be concluded that both decision modes have their own strengths and weaknesses. For the majority of transitions, the analysis indicates that the inclusion of deterministic structures has a positive effect upon its overall performance and efficiency. However, the use of decision structures may lead to situations in which certain transitions are not visited, which is undesirable. The random decision mode does visit said transitions, and as such, the decision between the two decision modes depends fully upon the desired behavior.

8.5 Locking Mechanism Analysis

Secondly, the performance characteristics of the revised locking mechanism will be investigated. The aim of the analysis is to determine the effectiveness of each locking mode provided in Section 5.4.2.1, i.e., it needs to be verified whether a program using element-scoped locking targets is guaranteed to outperform its broader-scoped counterparts in terms of efficiency and/or performance. Furthermore, observe that the no locks locking mode will be excluded from the investigation, since the use thereof does not guarantee the atomicity of statements requirement that the test models rely upon. Consequently, the generated code may not adhere to the behavior described by its parent model, and as such, the inclusion thereof may violate the integrity of the analysis.

The collection of command-line arguments through which the locking mode can be controlled are listed in Section 6.3.2. The locking modes that will be considered during the performance analysis are given in Table 8.2, which lists the identifier of the locking mode and the associated command-line arguments. The analysis of the locking mechanism will adhere to an analysis process that is analogous to the one used by the measurement representativeness analysis in Section 8.3.4, with the sole difference being the categories that are targeted during the investigation, i.e., the locking modes will be assessed instead of the measurement types. As discussed in Section 8.3.5, the logging-based measurements are not representative of a normal run of the program, and as such, the performance analysis will be based solely on the counting-based measurements.

Locking mode	**Command-line arguments**
Element	
Variable	`-lock_array`
Statement	`-statement_level_locking`

Table 8.2: The locking modes that are explored during the performance analysis of the locking mechanism. For each locking mode, the associated command-line arguments are listed.

8.5.1 Synthetic Test: CounterDistributor

First, the results of the `CounterDistributor` model will be discussed. The aggregate success count and ratio values for state machines `Counter` and `Distributor` are given in Figure 8.13. For state machine `Counter`, it is observed that the success counts given for the locking modes are practically identical to each other. For state machine `Distributor`, a minor difference is observed between the success count of the element locking mode and its counterparts, but given the observed variance, said difference falls within the margin of error. Furthermore, the success ratios of the available locking modes are equivalent for all state machines. The confidence intervals shown within the figure are relatively narrow for the success count and spot on for the success ratio, and as such, it can be concluded that all 20 trials reported similar results.

The success count and ratio measurements of all transitions and root decision nodes contained within the model are presented in Figure B.17. For the sake of completeness, the exact values of the measurements shown by the figure are provided in Tables C.1, C.33 and C.34 respectively. Through visual inspection, it is observed that all of the observations made for the aggregate measurements remain applicable: for all state machines, it can be observed that all transitions show similar behavior regardless of the chosen locking mode. For the transitions in state machine `Distributor`, it can be observed that the confidence interval of the variable locking mode is slightly wider than that of its counterparts. Said difference is by all likelihood a coincidental occurrence: the `CounterDistributor` model contains only a single class variable that is of a non-array type, and as such, all locking modes generate the same code. Consequently, it is of no surprise that the choice of locking mode does not have a measurable impact upon the performance and efficiency of the `CounterDistributor` model, given its simplistic locking requirements.

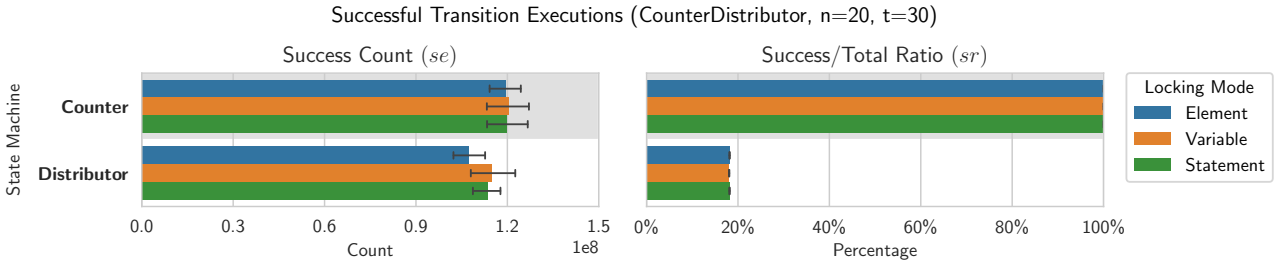


Figure 8.13: A bar plot that reports the number of successful executions (se) and the percentage of successful executions (sr) for each state machine in the target model `CounterDistributor`, where the results are grouped by the locking mode. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

8.5.2 Synthetic Test: Tokens

Secondly, the results of the `Tokens` model will be considered. The aggregate success count and ratio values for state machines `A`, `B` and `C` are given in Figure 8.14. In terms of the success count, it is observed that the values reported for the element locking mode are consistently higher than that of its alternatives. Furthermore, the variable locking mode seems to outperform the statement locking mode by a very small margin. The same trend can be observed for the success ratio, which shows that the element locking mode succeeds approximately twice as often when compared to its counterparts. On top of that, the success ratio of the variable locking mode is 2% higher than that of the statement locking mode. Additionally, the reported confidence intervals are narrow, which indicates that all of the performed trials report similar and consistent behavior.

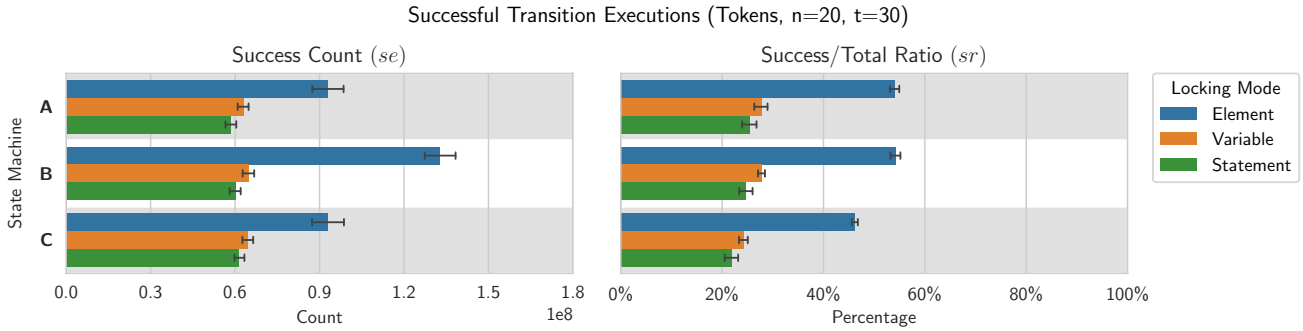


Figure 8.14: A bar plot that reports the number of successful executions (se) and the percentage of successful executions (sr) for each state machine in the target model `Tokens`, where the results are grouped by the locking mode. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

The success count and ratio measurements of all transitions and root decision nodes contained within the model are given in Figure B.18. Observe that the success count is reported in symmetric-logarithmic scale. For the sake of completeness, the exact values of the measurements presented by the success count and ratio bar plots are listed in Tables C.3, C.35 and C.36 respectively. For the majority of the transitions, it is observed that the use of the element locking mode yields the highest success count. The sole exception to the latter is the self-loop transition in state `wait` of all state machines, for which the exact opposite is observed. The behavior shown by the aforementioned transition is unexpected, given the fact that the identical self-loop in state `act` does not display similar behavior. Therefore, it is theorized that the reduction in success count is caused by the fact that state machines resides in state `wait` for a shorter period of time when the element locking mode is used. To clarify, recall that the transition between states `wait` and

`update` has been designed to be highly sensitive to the degree of concurrency within the program, i.e., a program with a high degree of concurrency exhibits a higher success ratio on said transition. The element locking mode has a fine-grained approach to locking, and as such, it is expected that the degree of concurrency is highest when said locking mode is used. Consequently, the overall number of opportunities at which the self-loop in state `wait` can be activated will be lower, which in turn may translate to a lower success count for the latter transition.

For the success ratio, it is observed that the element locking mode outperforms its alternatives by a considerable margin. An exception to the latter is the self-loop in state `wait` of state machine `B`, where the performance of all locking modes is about equal. Similar behavior is observed for the transition between states `act` and `wait` in state machine `C`. The comparatively high success ratio for the transition between states `wait` and `update` of all state machines supports the claim that the `Tokens` model is highly sensitive to the degree of concurrency of the model, i.e., the element locking mode allows for more concurrency to take place than its counterparts due to the former's fine-grained approach to locking. On top of that, the increase in success ratio for the transition between states `wait` and `update` substantiates the earlier theory that the decrease in success count of the self-loop in state `wait` is a direct result of the fact that the affected state machine resides within state `wait` for a shorter amount of time when the element locking mode is used. In contrast, only minor differences are observed for the variable and statement locking modes for both the success count and ratio. Lastly, the reported confidence intervals are narrow, which indicates that the variance between the measurements of all 20 trials is low.

Based upon the observations made within the scope of this section, it is clear that the use of the element-scoped locking mode yields the optimal behavior in terms of both the performance and efficiency of the generated program. The performance of the variable and statement locking modes is observed to be similar, with the former having a minor advantage over the latter.

8.5.3 Practical Test: Elevator

Next, the `Elevator` model will be investigated. The aggregate success count and ratio for state machines `cabin`, `controller` and `environment` are presented in Figure 8.15. The success count shows that the performance of the element and variable locking modes is equivalent. In contrast, the performance of the statement locking mode is considerably worse, with the success counts of the latter being approximately half that of its alternatives. For the success ratio, it is shown that the element locking mode yields the highest success ratio on all state machines. The variable locking mode performs second best in state machines `cabin` and `controller` by a small margin. On top of that, a significant difference is observed between the success ratio of the variable and statement locking modes in state machine `controller`. Oppositely, it is observed for state machine `environment` that the success ratios of the variable and statement locking modes are practically identical. The confidence intervals of the measurements are relatively narrow, and as such, it can be concluded that all 20 trials report similar results.

The success count and ratio of the transitions and root decision nodes contained within the model are listed in Figure B.19. Note that the success ratio is reported in symmetric-logarithmic scale. For the sake of completeness, the exact values of the gathered measurements are given in Tables C.5, C.37 and C.38 respectively. The success counts reported for the individual transitions follow the same trend as observed within the aggregate data: the performance of the element and variable locking modes is equivalent; the latter two locking modes outperform the statement locking mode by a respectable margin. The sole exception to the aforementioned ranking is the self-loop transition in state `work` of state machine `controller`, for which it is observed that the statement locking mode outperforms both of its counterparts. The difference between the success count of the statement and element locking modes is minor, but more significant differences are observed between the statement and variable locking modes. The exact cause of said behavior is unknown, but it is worthy of note that an unpacking operation had to be performed in the decision node that said transition is part of. However, the latter observation is only applicable to the

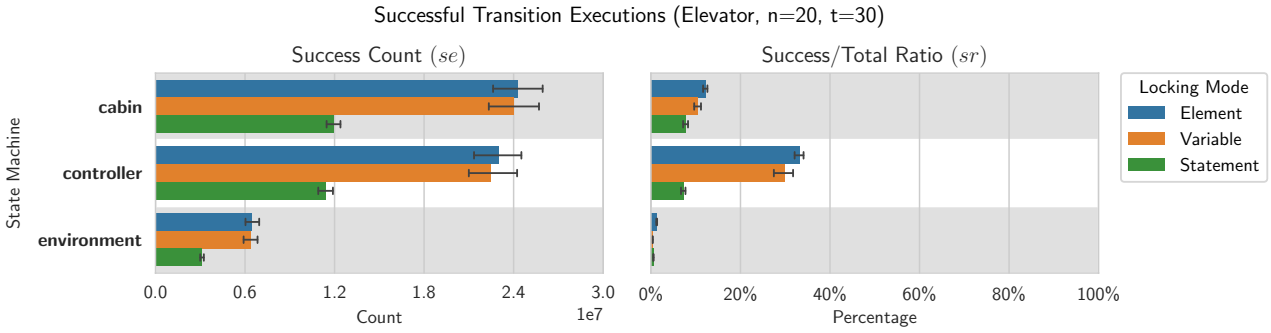


Figure 8.15: A bar plot that reports the number of successful executions (se) and the percentage of successful executions (sr) for each state machine in the target model `Elevator`, where the results are grouped by the locking mode. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

element-scoped locking mode, and as such, it would not explain the comparatively bad performance of the variable locking mode. Furthermore, the performance of the element locking mode does not seem to be hindered by the unpacking operation within the previous two transitions.

Mixed results are observed for the success ratio. Generally, the element locking mode seems to be the optimal decision, with it only being outperformed by the variable locking mode in the transition from state `work` to `done` in state machine `controller`. For state machines `cabin` and `controller`, it is observed that the variable locking mode outperforms the statement locking mode. The observed difference is minor in state machines `cabin`, but more significant in state machine `controller`. For state machine `environment`, it is observed that the difference between the success ratios of the variable and statement locking modes is marginal, with the latter having a small advantage over the former. Lastly, it has been established within Section 8.4.3 that the success ratio of the root decision nodes in state `idle` of state machine `cabin` and state `wait` of state machine `controller` provide an indication of the degree of concurrency within the program. Consequently, it can be concluded that the use of the statement locking mode has a negative effect upon the degree of concurrency within the generated program. The latter is not surprising, given the fact that only a single thread may access class variables at any moment time, i.e., state machines are practically forced to execute transitions in sequence instead of parallel.

Given the observations above, it may be concluded that the element and variable locking modes are equally viable options for the `Elevator` model. Nevertheless, the element locking mode pulls slightly ahead in terms of performance and efficiency. On the other hand, it has been observed that the statement locking mode has a detrimental effect upon the concurrency characteristics of the generated code, and as such, the use thereof is discouraged.

8.5.4 Practical Test: ToadsAndFrogs

Within the scope of this section, the performance characteristics of the `ToadsAndFrogs` model will be analyzed. The aggregate success count and ratio for state machines `control`, `frog` and `toad` are provided in Figure 8.16. Mixed results are reported for the success count. For state machine `control`, it can be observed that the statement locking mode performs slightly better than the element locking mode. Furthermore, the variable locking mode is outperformed by both of the latter locking modes. The success counts for state machine `frog` show that the statement locking mode outperforms both of its alternatives. Additionally, the element and variable locking modes perform about the same number of successful executions, with the latter having a minor advantage. For state machine `toad`, it is reported that the element locking mode achieves a slightly higher success count than the statement locking mode, and once again, the variable locking mode is outperformed by both alternatives. In contrast, the results shown for the success ratio are more

decisive, with all state machines reporting the same ranking. The element locking mode is ahead of its counterparts by a significant margin. Moreover, it can be observed that the success ratio of the variable locking mode is approximately twice as high as that of the statement locking mode. The confidence intervals are narrow, and as such, all 20 trials report similar behavior.

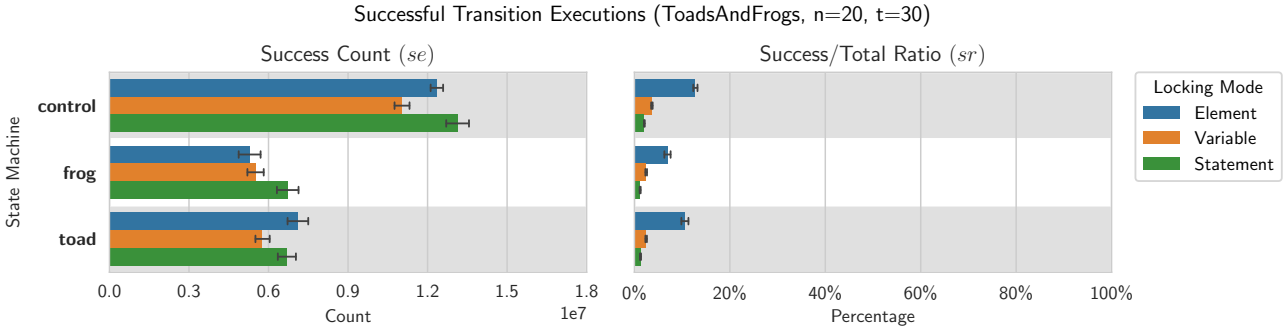


Figure 8.16: A bar plot that reports the number of successful executions (se) and the percentage of successful executions (sr) for each state machine in the target model `ToadsAndFrogs`, where the results are grouped by the locking mode. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

The success count and ratio of the transitions and root decision nodes contained within the model are illustrated in Figure B.20, which presents the success count bar plot in symmetric-logarithmic scale. For the sake of completeness, the exact values of the gathered measurements are listed in Tables C.7, C.39 and C.40 respectively. For the majority of the regularly occurring transitions, it can be observed the statement locking mode is slightly ahead of the element and variable locking modes in terms of success count. Different behavior is observed in the transitions of state machine `toad`, i.e., the element and statement locking modes report success counts that are practically identical. For less regularly occurring transitions, it is noted that the statement locking mode outperforms its counterparts by a considerable margin. In terms of the dynamics of the game, it can be concluded that the statement locking mode is most likely to lead to a win condition, with the difference in success count being over an order of magnitude when compared to the alternative locking modes. The latter indicates that the statement locking mode has a beneficial effect upon the randomness of the move sequences made by the two players, which is substantiated by the fact that, in state `running` of state machine `control`, the use of the statement locking mode vastly increases the success count for the second, third and fifth transition. Nevertheless, the total number of win conditions remains exceptionally low when compared to the number of failures.

The success ratio paints an opposite picture. For the regularly occurring transitions, it can be observed that element locking mode reports significantly higher ratios than its counterparts for the majority of transitions. In addition, the variable locking mode is about quite as successful as the statement locking mode for the aforementioned set of transitions. For the remainder of transitions, it can be observed within the associated measurement tables that the use of the statement locking mode generally leads to a higher success ratio than its counterparts in state machine `control`. Furthermore, the difference between the success ratios of the element and variable locking modes is significant, with the former outperforming the latter by at least a factor of two. Conversely, the success ratios reported for the state machines `frog` and `toad` are irregular, and as such, no decisive ranking can be produced for the locking modes in the aforementioned state machines.

Given all of the observations, it remains unclear which locking mode yields the best results for the `ToadsAndFrogs` model. As such, the optimal choice of locking mode depends completely upon the desired behavior. In terms of the performance and the dynamics of the game, the statement locking mode outperforms its counterparts by a respectable margin. If efficiency is the primary concern, then the element locking mode should be used instead.

8.5.5 Practical Test: Telephony

Finally, the results of the `Telephony` model will be discussed. First, it needs to be stressed that a different default configuration is used for the `Telephony` model: the latter does not function correctly when non-deterministic decisions are resolved through a sequential decision process, and as such, the code generator has been configured to use a random pick approach instead. On top of that, concerns have been raised in Section 8.3.4.5 about the viability of the model as a test case due to the inconsistent and highly variable nature of the gathered measurements. Nevertheless, an effort will be made to discuss the results, albeit at a more superficial level if necessary.

The aggregate success count and ratio of state machines `User_0`, `User_1`, `User_2` and `User_3` are presented in Figure 8.17. For all state machines, it can be observed that the highest success count is achieved by the statement locking mode. A smaller difference is observed between the variable and element locking modes, where the former consistently outperforms the latter. The highest success counts of the element and statement locking modes are observed in state machines `User_0` and `User_3` respectively, whereas the successes of the variable locking mode seems to be spread more or less evenly over all state machines. The results shown by the success ratio are less conclusive. For state machines `User_0` and `User_2`, the statement locking mode seems to be optimal. Conversely, state machines `User_1` and `User_3` report the highest success count when the variable locking mode is used. Nevertheless, high variability is observed within the reported success ratios, and therefore, it cannot be concluded that every trial adheres to said ranking.

Figure 8.17: A bar plot that reports the number of successful executions (se) and the percentage of successful executions (sr) for each state machine in the target model `Telephony`, where the results are grouped by the locking mode. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

The success count and ratio measurements of all transitions and root decision nodes within the `Telephony` model are reported in Figures B.21-B.24, where each figure reports the data associated with state machine `User_0`, `User_1`, `User_2` and `User_3` in the given order. Furthermore, observe that the success count is reported in symmetric-logarithmic scale. For the sake of completeness, the exact values of the presented measurements are provided in Tables C.9-C.12, C.41-C.44 and C.45-C.48 respectively. For state machines `User_0` and `User_2`, it is observed that the success count is highest for the majority of the transitions when the statement locking mode is used. In contrast, the variable locking mode performs best for the majority of transitions in state machine `User_1`. The latter is counter-intuitive, given the fact that the aggregate results report otherwise. However, it needs to be considered that the data is provided in logarithmic scale: within the most commonly visited transitions, namely the transitions between states `idle` and `qi`, the statement locking mode has a clear advantage in terms of success count. The reported values are two orders of magnitude larger than that of an average transition, and as such, the additional successful executions thereof are represented more heavily within the aggregate data, which in turn explains the different

conclusions. On top of that, a slight advantage is observed for the statement locking mode in state machine `User_3` on approximately half of the transitions. Lastly, it is important to observe that several of the transitions in the model are not visited by programs using the element locking mode. The same issue is observed for the variable locking mode, but to a more limited extent.

Regrettably, no consistent behavior can be observed within the reported success ratios, i.e., the reported values are highly variable, with the same transitions of different state machines reporting wildly different results. Consequently, the recommendation on which locking mode to use will be based solely on the aggregate results and the success count. In conclusion, it would seem that the statement locking mode is the most optimal configuration.

8.6 Discussion

Given the results that have been presented within the preceding sections, it is clear that there does not exist a configuration that performs optimally for all models, i.e., the choice of configuration depends heavily on the structure of the model and the desired behavioral characteristics of the generated program. As such, having the options to configure the code generator is beneficial, since it allows for said optimizations to be made to the program with a relatively low amount of effort. A tentative ranking of the decision and locking modes is provided by Table 8.3, which assigns a rank $p/e/b$ to each model and configuration combination. The ranks p and e order configurations based upon the latter's performance and efficiency respectively. The optional rank b orders configurations based upon the observed behavior: a configuration that is closest to the desired behavior of a model is assigned the highest ranking. The given ranks adhere to an ordinal scale, and therefore, the difference between the value of two ranks merely indicates that one configuration is more optimal than the other, i.e., the magnitude of the difference cannot be deduced from the assigned ranks. Furthermore, an inverse scale is used for the given ranking: the lowest rank value is assigned to the configuration that is most optimal. Lastly, two configurations with equivalent ranks are considered to be more or less equally optimal, but not necessarily identical.

	Decision mode				**Locking mode**		
Model	*Sequential + Det*	*Random + Det*	*Sequential*	*Random*	*Element*	*Variable*	*Statement*
`CounterDistributor`	0/0	0/0	0/0	3/3	0/0	0/0	0/0
`Tokens`	0/0	2/2	0/0	3/3	0/0	1/1	2/2
`Elevator`	0/0	0/0	0/0	3/3	0/0	0/1	2/2
`ToadsAndFrogs`	0/0/1	0/0/1	0/0/1	3/3/0	1/0/2	2/1/1	0/2/0
`Telephony`	-	0/0/1	-	1/1/0	2/2/2	1/0/1	0/0/0

Table 8.3: A table that contains the ranking between the given sets of configurations. Each cell provides up to three values: the first factor is the performance rank, the second factor is the efficiency rank, and the (optional) third factor is the desired behavior rank. A lower rank value indicates that the given configuration is more optimal than its counterparts with higher rank values. The given rankings are in an ordinal scale, and as such, the numerical difference between two ranks holds no significance in terms of magnitude.

8.6.1 Expectations Vs. Reality

To start with, the attained results will be compared to the expectations made within the scope of Section 8.2. For the `CounterDistributor` model, it has been predicted that a clear difference will be observed between the available decision modes, and oppositely, the locking modes will not make a meaningful impact on the results. Based upon the rankings listed in Table 8.3, it can be deduced that the use of deterministic structures and sequential decision making produce similar results. As such, the observed differences are not as predominant as originally expected, but nevertheless, a clear difference is observed for one of the configurations. Oppositely, the expectations on the effectiveness of the locking modes meet expectations, i.e., the locking modes do not have a meaningful impact upon the `CounterDistributor` model.

Similarly, the `Tokens` model shows unexpected results for the decision modes as well. Originally, it has been predicted that the choice of decision mode will have a marginal impact. However, in reality, a clear difference between the random and sequential decision modes can be observed. The latter is likely caused by the fact that sequential decision making increases the reactiveness of the state machines to state changes within the program. In contrast, the impact of the locking modes meets expectations: a fine-grained approach leads to a more optimal program.

For the `Elevator` model, it has been predicted that the choice of decision mode has a minor impact upon the program, with deterministic structures having a minor advantage. The ranking concurs partially with said assessment: the random approach is less optimal, but the sequential approach to decision making is on par with the inclusion of deterministic structures. The latter can be explained by the fact that the use of sequential decisions results in a decision structure that is equivalent to that of decision modes using deterministic structures. Similarly to the previous models, the listed rankings concur with the expectations set for the locking modes: a fine-grained approach to locking results in more optimal behavior within the program.

The predictions made for the `ToadsAndFrogs` model have proven to be mostly correct. In terms of the effectiveness of the decision modes, it has been projected that the use of deterministic structures will be beneficial to the program's performance and efficiency, but detrimental to the dynamics of the game. The rankings listed in Table 8.3 concur with both predictions. On the other hand, the expectations set upon the effectiveness of the locking modes is only partially correct. Given the heavy use of class variables, it has been predicted that the statement locking mode outperforms its counterparts. The latter holds true for the rankings assigned to the performance and dynamics of the game metrics. However, in terms of efficiency, the opposite ranking is observed, i.e., fine-grained approaches are more optimal. The latter indicates that the reactiveness of a program has a greater impact upon the efficiency of a program than originally anticipated. Lastly, it is noteworthy that the overhead of locking operations is clearly observed in the results of the `ToadsAndFrogs` model: a higher degree of efficiency is achieved with a fine-grained approach due to an increase in concurrent behavior, but in spite of that, the total number of successes ends up being lower due to the increased workload of the locking mechanism.

Finally, the predictions of the `Telephony` model will be discussed. Originally, the expectation was that the use of deterministic structures will have a minor detrimental impact upon the behavior of the program. However, the listed rankings instead show the opposite: the use of deterministic structures yields better results. The cause of said discrepancy remains unclear, but it is theorized that the cause thereof is the increased reactivity of the state machines to state changes, i.e., the use of deterministic structures has a beneficial impact upon the concurrency characteristics of the program. On the other hand, it can be observed that the predictions made for the effectiveness of the locking modes are correct: fine-grained approaches are expected to be less optimal, due to the fact that unpacking operations need to be performed on a regular basis. Furthermore, all transitions require access to multiple class variables, and as such, the locking mechanism has been predicted to be the limiting factor. The rankings indicate that fine-grained approaches are less optimal, and as such, the observed behavior is in accordance with the expected behavior.

8.6.2 Potential Improvements

Several improvements have been identified that may potentially improve the quality and effectiveness of the generated code. Firstly, it would be beneficial to create a selection procedure that automatically picks a decision and locking mode combination that is optimal for the target model. The latter requires additional research into the behavioral characteristics of models, i.e., the decision and locking patterns that are present within the models need to be identified, such that said patterns can be connected to an optimal configuration. Furthermore, the performance analysis has considered merely a subset of all possible configurations of the code generator: the effectiveness of the decision structure and locking mechanism has been analyzed independently, i.e., the effect that the decision modes have on the locking modes and vice versa has not been investigated. Consequently, a combination of two non-default decision and locking modes may lead to unpredictable performance characteristics that contradict the observations made throughout this chapter, and as such, it is advisable to investigate said combinations in the future.

On top of that, several structural improvements to the generated code have been suggested within the scope of Chapters 4, 5 and 6 that may have a profound impact upon the behavioral characteristics of the program. Regrettably, said improvements could not be incorporated into the code generator due to time restrictions, and as such, the effectiveness thereof could not be taken into account during the performance analysis. However, the garnered results do substantiate several of the concerns that said improvements are based upon, and hence, the implementation and analysis of said enhancements is considered to be a worthwhile endeavor.

Additionally, several additional areas of improvements have been identified based upon the results of the performance analysis. For one, it has been observed that the effectiveness of a decision mode may differ between state machines, i.e., a decision mode that is optimal for one state machine may not be optimal for another and vice versa. As such, it may be advantageous to allow for a given decision mode to be applied to only a part of the model for added flexibility. Similarly, the classes within a model may benefit from a more flexible class-based application of the available locking modes. Moreover, the variable locking mode could be applied to a subset of the class variables, i.e., one may use the variable-scoped locking mode for variables that are subjected to unpacking operations, while using the element-scoped locking mode for all remaining variables instead. In extension to that, it may be possible to create a grouping of lock targets instead of using individual locks: if all transitions use the same collection of locks, then those locks can be combined into a singular lock object instead. The benefit of the latter is that only a single lock operation needs to be performed, and as such, additional expensive lock operations can be avoided.

Lastly, it has been noted that the performance characteristics of programs with logging-based measurements are at times better than programs that use counting-based measurements. Consequently, it may theoretically be beneficial to add pauses to the primary loops of the state machines to achieve a similar effect. Likewise, the use of wait-notify constructions to avoid busy-waiting may accomplish the same goal. The exact cause of the observed behavioral differences remains unclear, and therefore, a deeper investigation into said phenomenon is required.

8.6.3 Behavioral Analysis and Process Mining

Within the previous section, several improvements have been discussed that may have a beneficial impact upon the performance of the program. Nevertheless, all of said improvements have a common factor: for the measures to be effective, a greater measure of understanding of the behavioral characteristics of the program run is required. The primary intention of the logging-based measurement was to provide said information, but unfortunately, it has been concluded that said measurements have a major impact upon the dynamics of the program run, i.e., the behavioral characteristics exhibited by the execution of programs with and without logging-based measurements are dissimilar, and as such, the logging-based measurements cannot be used, since the results thereof are not representative of the dynamics displayed by a normal run of the program.

Consequently, further research is required into finding a methodology through which said measurements can be gathered without affecting the behavioral characteristics of the program.

Likewise, an approach through which the logging-based measurements can be analyzed is required. Originally, the primary purpose of said measurements was to provide data through which the degree of concurrency within the program can be analyzed, i.e., overlapping active periods of transitions within the log files imply concurrent behavior. However, the aforementioned target does not give a complete interpretation of the behavior of the program, i.e., the concurrency characteristics are only a single aspect of a program's behavior, and as such, a substantial amount of information within the log files remains uninvestigated. Furthermore, Section 8.3.5 indicates that the current implementation of the logging-based measurement approach is inherently flawed, and consequently, substantial changes to the logging process are required to capture the desired data. Given the aforementioned challenges, it could be beneficial to redesign the logging-based measurements such that process mining techniques can be applied to the gathered data instead: existing process mining toolkits are robust and reliable, and hence, they are expected to provide a more stable and suitable foundation for the logging-based performance analysis.

8.6.4 Data and Methodology Transparency

Lastly, it needs to be noted for the sake of transparency that all of the code associated with the performance analysis is publicly available on GitHub. The experiments have been conducted within the scope of the `2IMC00-SLCO-JAVA`[4] repository. The latter includes the generated Java programs associated with all of the investigated models and configuration combinations. On top of that, all of the run configurations used during the performance testing are provided as xml files in the `.runs` directory. Similarly, the Log4j configuration is included within the `resources` folder. Lastly, the post-processed performance measurements of all runs are provided within the `results` directory. The log files have been excluded from the repository due to their size.

The analysis of the results is carried out within the `2IMC00-SLCO-ANALYSIS`[5] repository. The latter provides the Python scripts through which all of the presented results have been compiled and visualized. Additionally, the source code of all figures and tables generated during the analysis process are included within the `figures` and `tables` directories respectively.

[4]The `2IMC00-SLCO-JAVA` repository is available at `https://github.com/melroy999/2IMC00-SLCO-JAVA`
[5]The `2IMC00-SLCO-ANALYSIS` repository is available at `https://github.com/melroy999/2IMC00-SLCO-ANALYSIS`

Chapter 9

Conclusions

In this report, several enhancements have been introduced that aim to improve the performance, reliability and maintainability of the Java code generator component of the Slco framework and the code it generates. In hindsight, it is clear that the focus of this project has been too broad to fit within the scope of a thesis, and as such, a wide range of improvements can undoubtedly be made to the majority of the components. Nevertheless, an earnest attempt has been made at providing a comprehensive investigation into all of the included components.

The conclusion will be structured as follows. To start with, the contributions made within the scope of this thesis will be summarized in Section 9.1. Subsequently, several promising improvements will be discussed in Section 9.2 that warrant further investigation.

9.1 Summary

The contributions made throughout this report can be summarized as follows. In Chapter 3, several changes have been made that aim to resolve various issues within the parser component of the Slco framework. To start with, the definition of the Slco 2.0 language has been modified to make it more consistent with the semantics of the target Java language. Furthermore, a preprocessing procedure has been created that makes several structural improvements to the resulting syntax tree, such that the latter is more straightforward to work with.

Secondly, the use of (nested) deterministic structures has been investigated in Chapter 4 to improve the overall performance and predictability of the generated code. The inclusion of deterministic structures ensures that the generated code makes proper use of the mutual exclusivity of guard statement combinations. The decision structures are generated by describing the issue at hand as a Satisfiability Modulo Theories (SMT) optimization problem. Two decision structure construction algorithms have been provided: the first solution applies a greedy problem-solving heuristic, whilst the other is guaranteed to give an optimal solution. In addition, several overlap constraints have been defined through which the desired structural characteristics of the resulting decision structure can be specified. Lastly, a procedure has been introduced that simplifies a given decision structure without affecting its functionality or correctness.

Afterwards, a new locking data structure has been introduced in Chapter 5 that ensures the atomicity of statements and lock-deadlock freedom requirements. The atomicity of statements is achieved through a lock propagation procedure that moves locks to the appropriate location within the locking data structure, whereas the lock-deadlock freedom is accomplished by adhering to a strict lock ordering based upon a lock's assigned identity. The inter-lock-dependencies between locks within the model have been resolved through the application of phased locking. Furthermore, the concept of weak unpacking has been introduced to tackle circular references in said

dependencies. On top of that, the locking data structure has been designed to strictly follows the control flow of the generated Java program, which allows for short-circuit evaluations to be dealt with appropriately. The location sensitivity issues caused by short-circuit evaluations are resolved through strict unpacking. Moreover, the assignment of lock identities has been refined through the addition of selection heuristics, and several locking modes have been introduced that allow for an optimal locking target scope to be picked for a given model. Finally, a proper approach to exception handling has been proposed to ensure that the program will not enter a lock-deadlock upon the occurrence of an exception in one or more of the state machines.

In Chapter 6, structural improvements have been made to the code generator and its code templates. The code generator has been rewritten from scratch in an effort to increase the quality and maintainability of the code by adhering to a more modular approach to code rendering. The revised code has migrated to object-scoped code templates, accompanied by similarly scoped view models to decouple the logic from the rendering. Likewise, the code generator project has been subdivided into several modules, where each module focuses upon a singular aspect of the rendering process. Additionally, the code templates of decision structures have been revised to support sequential decisions and encompassing guard statements. On top of that, the locking mechanism has been overhauled such that it makes proper use of the new locking data structure and its more robust atomicity preserving features. Lastly, several discrepancies between the semantics of the Slco language and the code generator have been resolved, and an effort has been made to ensure that the generated programs partially enforce the property of memory consistency.

Additionally, a procedure has been introduced in Chapter 7 through which the functional correctness of a generated program can be formally verified. The verification process is implemented using the VerCors verification tool set and has been subdivided into two distinct parts. First, the structural characteristics of the generated code are (partially) validated, such that it can be ensured that the rendered code executes the behavior expressed by the input model appropriately. Next, the correctness of the locking mechanism is verified through a three-step approach that validates (1) the structural integrity of the locking mechanism, (2) the adequacy of the lock coverage in terms of atomicity, and (3) the correctness of the applied rewrite rules respectively.

Lastly, the performance of the generated programs has been analyzed in Chapter 8 to measure the overall effectiveness of the improvements made throughout this work. The performance analysis is performed on two synthetic models and three practical models. To start with, the use of logging-based performance measurements has been investigated in an attempt to get a more detailed look into the concurrency characteristics of the generated programs. Regrettably, it has been observed that said measurements have a profound impact upon the behavioral characteristics of the target program, and as such, the results thereof are excluded from the analysis. Subsequently, the analysis investigates the effectiveness of the revamped decision structures by comparing the performance and efficiency characteristics of the available decision modes. Analogously, the effectiveness of the revised locking mechanism and its locking modes has been evaluated. Based upon the results of the performance analysis, it can be concluded that the effectiveness of the introduced improvements/options depends heavily upon the desired behavior of a model. Moreover, several areas of improvement have been identified that require further attention.

9.2 Future Work

The work conducted within the scope of this thesis is not complete, i.e., several improvements have been identified that remain to be researched and/or implemented. To start with, several measures have been proposed through which the general functionality of the generated code can be completed and/or enhanced. In Section 6.2.1, it has been observed that several components of the Slco language lack an implementation for their model to code conversion. Specifically, delay statements remain to be implemented in Java. In addition, the verified implementation of the communication channels and ports used within a previous iteration of the Slco framework

has to be integrated into the revised code generator. Furthermore, the current implementation of the code generator does not guarantee the memory consistency of array-typed variables. On top of that, a more efficient alternative needs to be found for the busy-waiting loops used within the transition selection procedure. Note that potential solutions to the latter two issues are provided in Sections 6.2.4 and 6.2.5 respectively. Lastly, Section 5.8.3 suggests that the property of dependency safety in exception handling can be attained through the introduction of failboxes.

Additionally, several improvements have been put forward that are applicable to the construction and rendering of the decision structures. First of all, it has been observed in Section 8.6.2 that the creation of a selection procedure that automatically picks a decision mode that is optimal for the given target model would be beneficial. Secondly, it may be advantageous to allow for a decision mode to be applied to a specific part of the model for added flexibility in the optimization process. Furthermore, Section 6.2.3 stresses that further optimizations are required with respect to rendering of decision structures, i.e., it may occur that a conditional is needlessly rendered more than once within the same branch of the decision structure due to the presence of encompassing guard statements. The latter is deemed problematic, since it may cause unavoidable location sensitivity violations to occur with the locking data structure. Lastly, alternative code templates for deterministic decisions are suggested in Section 6.2.3.1 that warrant further investigation.

Similarly, various elements have been identified within the locking mechanism that can be improved upon. For one, it has been observed in Section 5.5.1 that the need of two passes within the lock propagation procedure actively limits the flexibility of the propagation heuristics used therein. Furthermore, the need of a more optimal and robust selection heuristic for the resolution of circular references is emphasized in Section 5.6.3. On top of that, several measures are suggested that aim to limit the impact of strict unpacking operations. Firstly, two alternative approaches to strict unpacking are proposed in Section 5.7.2.1. Secondly, the possibility of reordering decisions within the decision nodes to reduce the number of unavoidable location sensitivity violations is considered in Section 5.8.1. Lastly, an enhancement to the process of marking location sensitivity violations is proposed in Section 5.8.2 to decrease the number of false positives within the location sensitivity markings. Next, the use of alternative synchronization constructs is considered in Section 6.2.2.2, and a less strict scope of atomicity within decision structures is proposed in Section 6.2.2.3. Finally, various improvements are suggested within the scope of Section 8.6.2. To start with, it has been observed that it may be beneficial to make the locking modes variable-scoped. Additionally, the option of merging multiple variables into a single lock object warrants further investigation. Lastly, the creation of a selection procedure that automatically picks a locking mode that is optimal for the given target model may prove to be advantageous.

Next, the required future work on and research into the formal verification process of the generated code will be highlighted. First of all, Section 7.2.1 points out that the structural verification is incomplete: the validity of the decision structure and the state changes upon transitioning remain unverified within the current iteration of the verification process. Secondly, it is observed in Section 7.2.2.2 that the verification process of the locking mechanism is incompatible with the alternative locking modes. Lastly, concerns are raised in Section 7.3.1 about the structural limitations of the VerCors tool set and its potential incompatibility with the task at hand.

Finally, several areas of interest have been identified for the performance analysis that require additional investigation. As noted in Section 8.1.1, the effect of the locking fairness policy upon the behavior of the generated programs requires further examination. In addition, the effectiveness of the available solver configurations used within the construction of decision structures has not been considered in Section 8.4 due to a lack of compatible test models. Similarly, concerns are raised in Section 8.6.2 that the effect of combining two non-default decision and locking modes remains uninvestigated. Lastly, it is stressed in Section 8.6.3 that a proper understanding of a program's behavior is a crucial factor of the optimization process. Therefore, the creation of more accurate and effective performance measurements and a versatile analysis process to interpret said results warrants further attention.

Bibliography

[1] Jean-Raymond Abrial and A Hoare. *The B-book: assigning programs to meanings*, volume 1. Cambridge university press Cambridge, 1996. 3

[2] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT press, 2008. 3, 87

[3] Bernhard Beckert, Reiner Hähnle, and Peter H Schmitt. *Verification of Object-Oriented Software. The KeY Approach: Foreword by K. Rustan M. Leino*, volume 4334. Springer, 2007. 88

[4] Jan A Bergstra, Alban Ponse, and Daan JC Staudt. Short-circuit logic. *arXiv preprint arXiv:1010.3674*, 2010. 49

[5] Stefan Blom, Saeed Darabi, Marieke Huisman, and Wytse Oortwijn. The vercors tool set: Verification of parallel and concurrent software. In Nadia Polikarpova and Steve Schneider, editors, *Integrated Formal Methods*, pages 102–110, Cham, 2017. Springer International Publishing. 87

[6] Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. A formally verified compiler for lustre. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 586–601, 2017. 3

[7] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. Vcc: A practical system for verifying concurrent c. In *International Conference on Theorem Proving in Higher Order Logics*, pages 23–42. Springer, 2009. 88

[8] David R Cok. Openjml: Jml for java 7 by extending openjdk. In *NASA formal methods symposium*, pages 472–479. Springer, 2011. 88

[9] Sjoerd Cranen, Jan Friso Groote, Jeroen J. A. Keiren, Frank P. M. Stappers, Erik P. de Vink, Wieger Wesselink, and Tim A. C. Willemse. An overview of the mcrl2 toolset and its recent advances. In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 199–213, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. 3, 87

[10] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. 29

[11] Leonardo De Moura and Nikolaj Bjørner. Satisfiability modulo theories: introduction and applications. *Communications of the ACM*, 54(9):69–77, 2011. 28

[12] Sander de Putter and Anton Wijs. A formal verification technique for behavioural model-to-model transformations. *Formal Aspects of Computing*, 30(1):3–43, 2018. 87

[13] Sander de Putter, Anton Wijs, and Dan Zhang. The SLCO framework for verified, model-driven construction of component software. In Kyungmin Bae and Peter Csaba Ölveczky, editors, *Formal Aspects of Component Software - 15th International Conference, FACS 2018, Pohang, South Korea, October 10-12, 2018, Proceedings*, volume 11222 of *Lecture Notes in Computer Science*, pages 288–296. Springer, 2018. 3, 7, 12, 15

[14] Igor Dejanović, Renata Vaderna, Gordana Milosavljević, and Željko Vuković. TextX: A python tool for domain-specific languages implementation. *Knowledge-Based Systems*, 115:1 – 4, 2017. 4

[15] Simulink Documentation. Simulation and model-based design, 2020. 3

[16] François Dormoy. Scade 6 a model based solution for safety critical software development. In *Embedded Real Time Software and Systems (ERTS2008)*, 2008. 3

[17] Andreas Fürst, Thai Son Hoang, David Basin, Krishnaji Desai, Naoto Sato, and Kunihiko Miyazaki. Code generation for event-b. In *International Conference on Integrated Formal Methods*, pages 323–338. Springer, 2014. 3

[18] James W. Havender. Avoiding deadlock in multitasking systems. *IBM Syst. J.*, 7(2):74–84, 1968. 50

[19] Bart Jacobs and Frank Piessens. Failboxes: Provably safe exception handling. In Sophia Drossopoulou, editor, *ECOOP 2009 – Object-Oriented Programming*, pages 470–494, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. 73

[20] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for c and java. In *NASA formal methods symposium*, pages 41–55. Springer, 2011. 88

[21] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 348–370. Springer, 2010. 88

[22] Anany Levitin and Maria Levitin. *Algorithmic puzzles*, page 53. Oxford University Press, 2011. 18

[23] Robert Nieuwenhuis and Albert Oliveras. On sat modulo theories and optimization problems. In *International conference on theory and applications of satisfiability testing*, pages 156–169. Springer, 2006. 29

[24] Oracle Java Documentation: Memory Consistency Errors. `https://docs.oracle.com/javase/tutorial/essential/concurrency/memconsist.html/`, accessed 22-06-2022. 82

[25] Colin O'Halloran. Automated verification of code automatically generated from simulink®. *Automated Software Engineering*, 20(2):237–264, 2013. 3

[26] Anton Wijs and Luc Engelen. Efficient property preservation checking of model refinements. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 565–579. Springer, 2013. 87

[27] Anton Wijs and Luc Engelen. Refiner: towards formal verification of model transformations. In *NASA Formal Methods Symposium*, pages 258–263. Springer, 2014. 87

[28] Dan Zhang, Dragan Bosnacki, Mark van den Brand, Cornelis Huizing, Bart Jacobs, Ruurd Kuiper, and Anton Wijs. Dependency safety for java - implementing and testing failboxes. *Sci. Comput. Program.*, 184, 2019. 73

Appendix A

Log Throughput Analysis

A.1 Synthetic Test: CounterDistributor

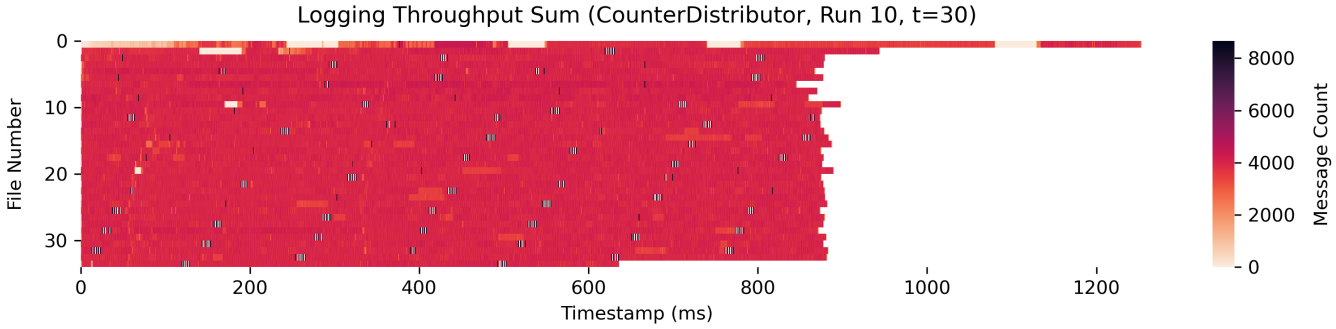


Figure A.1: A heat map plot that reports on the total number of log messages per time unit (milliseconds) for each log file generated during run 10 of target model `CounterDistributor`. Each file has a maximum size of 100MB. The results have been measured over a time span of 30 seconds.

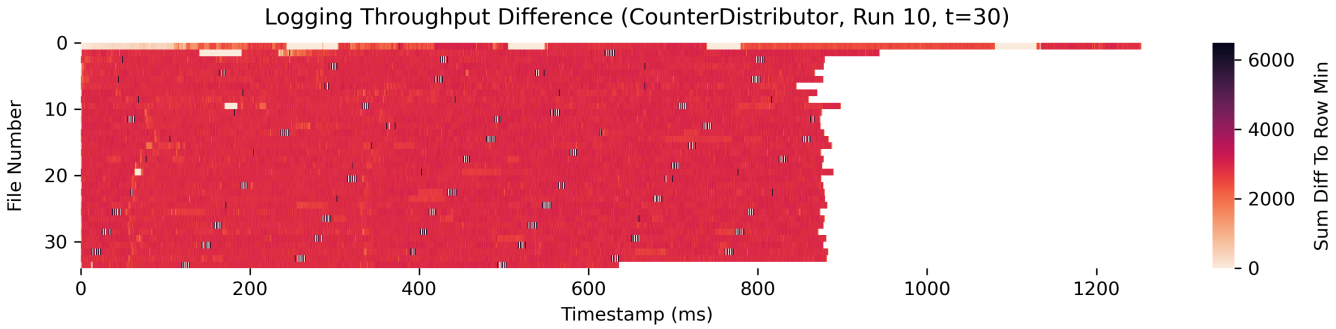


Figure A.2: A heat map plot that reports on the sum difference to the row minimum of the state machines' log message counts per time unit (milliseconds) for each log file generated during run 10 of target model `CounterDistributor`. Each file has a maximum size of 100MB. The results have been measured over a time span of 30 seconds.

Log message throughput statistics for target model '`CounterDistributor`' ($n = 20, t = 30$)

Target	N	$min(e)$	$max(e)$	$median(e)$	$\mu(e)$	$\sigma(e)$
`Counter`	600020	0.00	$2.87 \cdot 10^3$	$4.64 \cdot 10^2$	$4.66 \cdot 10^2$	$9.62 \cdot 10^1$
`Distributor`	600020	0.00	$7.47 \cdot 10^3$	$3.47 \cdot 10^3$	$3.40 \cdot 10^3$	$5.26 \cdot 10^2$
total	600020	0.00	$8.65 \cdot 10^3$	$3.94 \cdot 10^3$	$3.87 \cdot 10^3$	$5.94 \cdot 10^2$
difference	600020	0.00	$6.48 \cdot 10^3$	$2.99 \cdot 10^3$	$2.94 \cdot 10^3$	$4.63 \cdot 10^2$

Table A.1: A table containing statistics on the log messages per milliseconds (e) measured during the execution of the target model `CounterDistributor` grouped by state machine. The total and difference entries at the end of the table depict the row sum and the sum difference to the row minimum respectively. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

A.2 Synthetic Test: Tokens

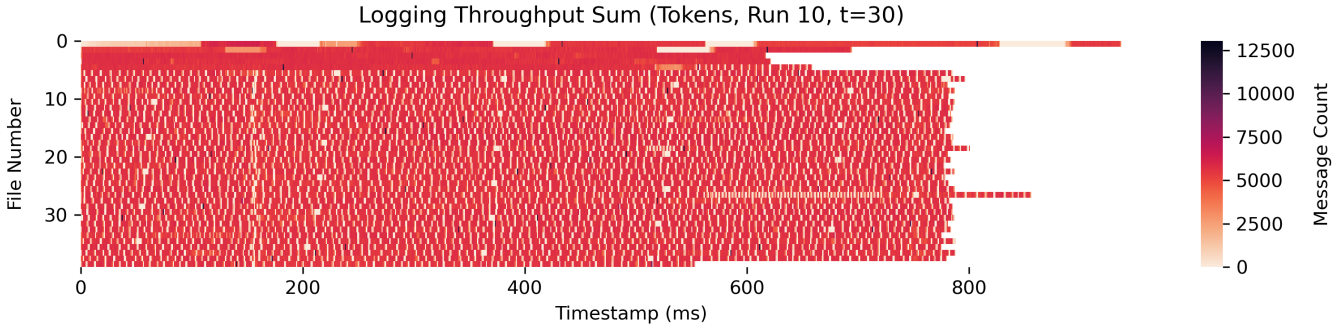


Figure A.3: A heat map plot that reports on the total number of log messages per time unit (milliseconds) for each log file generated during run 10 of target model `Tokens`. Each file has a maximum size of 100MB. The results have been measured over a time span of 30 seconds.

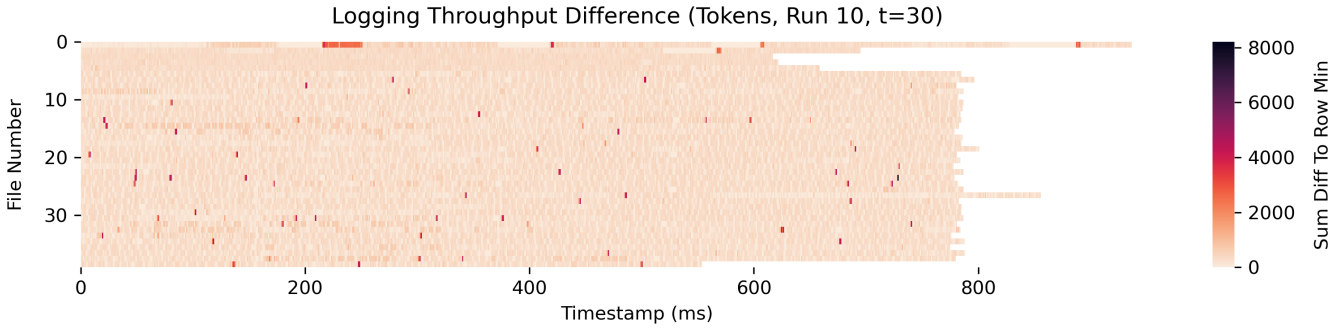


Figure A.4: A heat map plot that reports on the sum difference to the row minimum of the state machines' log message counts per time unit (milliseconds) for each log file generated during run 10 of target model `Tokens`. Each file has a maximum size of 100MB. The results have been measured over a time span of 30 seconds.

Log message throughput statistics for target model '`Tokens`' ($n = 20, t = 30$)

Target	N	$min(e)$	$max(e)$	$median(e)$	$\mu(e)$	$\sigma(e)$
`A`	600020	0.00	$8.36 \cdot 10^3$	$1.80 \cdot 10^3$	$1.51 \cdot 10^3$	$6.72 \cdot 10^2$
`B`	600020	0.00	$4.78 \cdot 10^3$	$1.96 \cdot 10^3$	$1.65 \cdot 10^3$	$7.28 \cdot 10^2$
`C`	600020	0.00	$4.88 \cdot 10^3$	$1.82 \cdot 10^3$	$1.54 \cdot 10^3$	$6.83 \cdot 10^2$
total	600020	0.00	$1.30 \cdot 10^4$	$5.57 \cdot 10^3$	$4.70 \cdot 10^3$	$2.05 \cdot 10^3$
difference	600020	0.00	$8.20 \cdot 10^3$	$2.08 \cdot 10^2$	$2.29 \cdot 10^2$	$2.78 \cdot 10^2$

Table A.2: A table containing statistics on the log messages per milliseconds (e) measured during the execution of the target model `Tokens` grouped by state machine. The total and difference entries at the end of the table depict the row sum and the sum difference to the row minimum respectively. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

A.3 Practical Test: Elevator

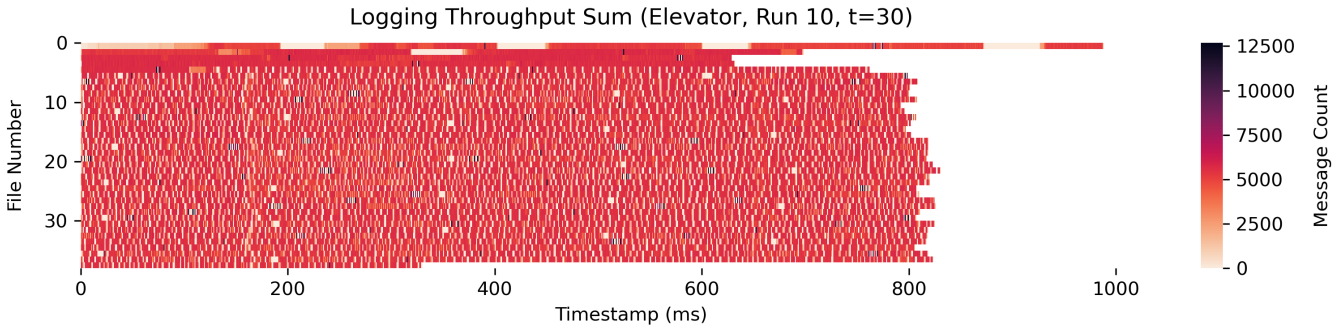


Figure A.5: A heat map plot that reports on the total number of log messages per time unit (milliseconds) for each log file generated during run 10 of target model `Elevator`. Each file has a maximum size of 100MB. The results have been measured over a time span of 30 seconds.

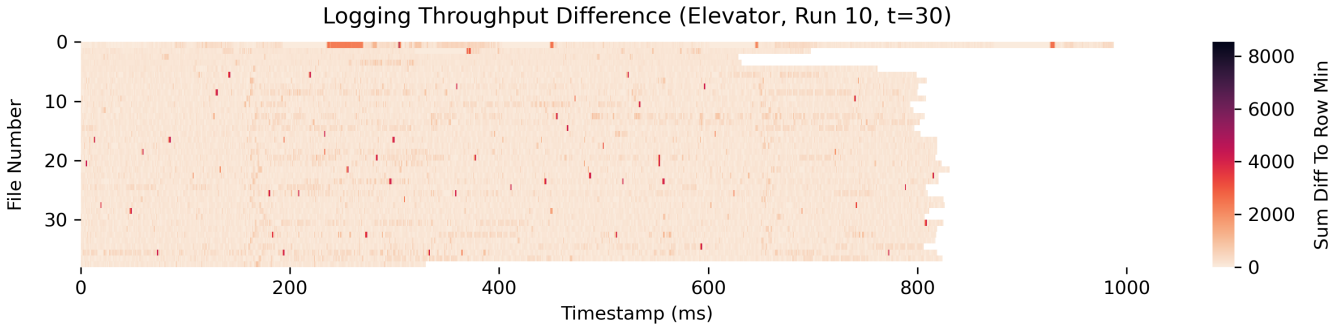


Figure A.6: A heat map plot that reports on the sum difference to the row minimum of the state machines' log message counts per time unit (milliseconds) for each log file generated during run 10 of target model `Elevator`. Each file has a maximum size of 100MB. The results have been measured over a time span of 30 seconds.

Log message throughput statistics for target model '`Elevator`' ($n = 20, t = 30$)

Target	N	$min(e)$	$max(e)$	$median(e)$	$\mu(e)$	$\sigma(e)$
`cabin`	600040	0.00	$6.87 \cdot 10^3$	$1.80 \cdot 10^3$	$1.54 \cdot 10^3$	$6.46 \cdot 10^2$
`controller`	600040	0.00	$8.07 \cdot 10^3$	$1.77 \cdot 10^3$	$1.52 \cdot 10^3$	$6.37 \cdot 10^2$
`environment`	600040	0.00	$8.54 \cdot 10^3$	$1.82 \cdot 10^3$	$1.57 \cdot 10^3$	$6.55 \cdot 10^2$
total	600040	0.00	$1.27 \cdot 10^4$	$5.40 \cdot 10^3$	$4.63 \cdot 10^3$	$1.91 \cdot 10^3$
difference	600040	0.00	$8.53 \cdot 10^3$	$1.25 \cdot 10^2$	$1.64 \cdot 10^2$	$2.51 \cdot 10^2$

Table A.3: A table containing statistics on the log messages per milliseconds (e) measured during the execution of the target model `Elevator` grouped by state machine. The total and difference entries at the end of the table depict the row sum and the sum difference to the row minimum respectively. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

A.4 Practical Test: ToadsAndFrogs

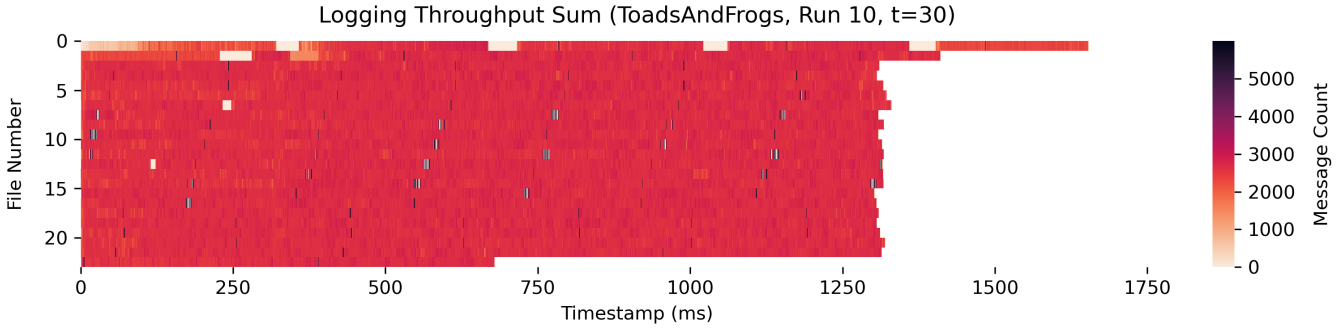


Figure A.7: A heat map plot that reports on the total number of log messages per time unit (milliseconds) for each log file generated during run 10 of target model `ToadsAndFrogs`. Each file has a maximum size of 100MB. The results have been measured over a time span of 30 seconds.

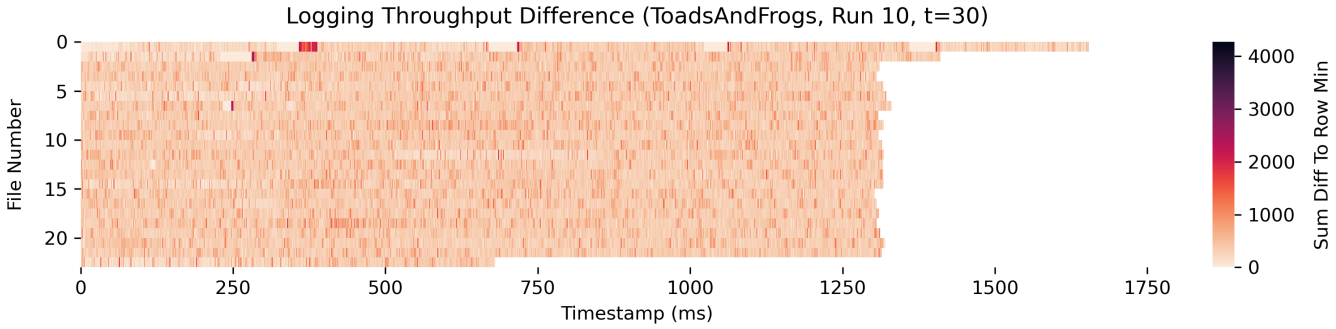


Figure A.8: A heat map plot that reports on the sum difference to the row minimum of the state machines' log message counts per time unit (milliseconds) for each log file generated during run 10 of target model `ToadsAndFrogs`. Each file has a maximum size of 100MB. The results have been measured over a time span of 30 seconds.

Log message throughput statistics for target model '`ToadsAndFrogs`' ($n = 20, t = 30$)

Target	N	$min(e)$	$max(e)$	$median(e)$	$\mu(e)$	$\sigma(e)$
`control`	600020	0.00	$3.27 \cdot 10^3$	$9.64 \cdot 10^2$	$9.68 \cdot 10^2$	$2.13 \cdot 10^2$
`frog`	600020	0.00	$2.63 \cdot 10^3$	$7.93 \cdot 10^2$	$7.92 \cdot 10^2$	$1.54 \cdot 10^2$
`toad`	600020	0.00	$4.25 \cdot 10^3$	$7.96 \cdot 10^2$	$7.99 \cdot 10^2$	$1.60 \cdot 10^2$
total	600020	0.00	$6.00 \cdot 10^3$	$2.58 \cdot 10^3$	$2.56 \cdot 10^3$	$3.80 \cdot 10^2$
difference	600020	0.00	$4.27 \cdot 10^3$	$2.82 \cdot 10^2$	$3.38 \cdot 10^2$	$2.54 \cdot 10^2$

Table A.4: A table containing statistics on the log messages per milliseconds (e) measured during the execution of the target model `ToadsAndFrogs` grouped by state machine. The total and difference entries at the end of the table depict the row sum and the sum difference to the row minimum respectively. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

A.5 Practical Test: Telephony

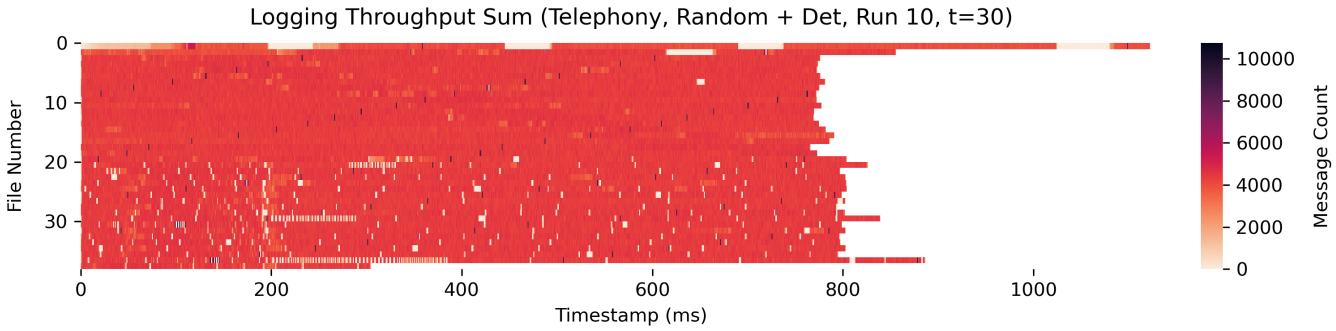


Figure A.9: A heat map plot that reports on the total number of log messages per time unit (milliseconds) for each log file generated during run 10 of target model `Telephony`. Each file has a maximum size of 100MB. The results have been measured over a time span of 30 seconds.

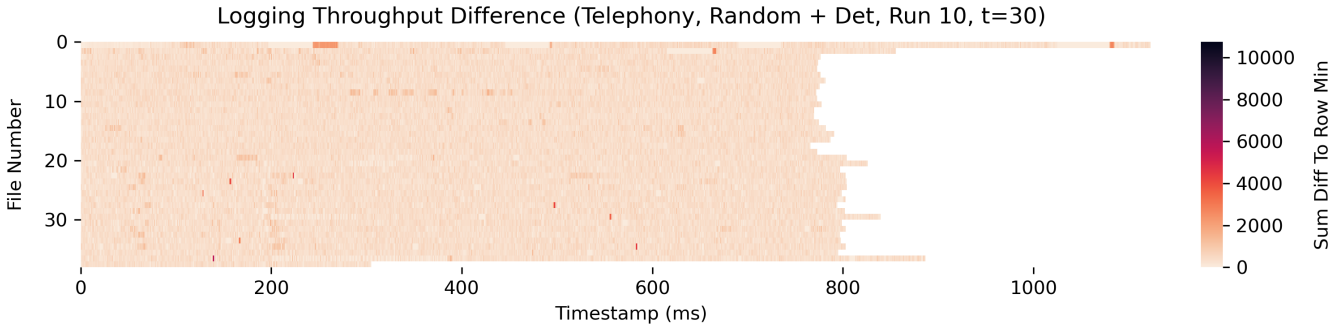


Figure A.10: A heat map plot that reports on the sum difference to the row minimum of the state machines' log message counts per time unit (milliseconds) for each log file generated during run 10 of target model `Telephony`. Each file has a maximum size of 100MB. The results have been measured over a time span of 30 seconds.

Log message throughput statistics for target model '`Telephony`' ($Random + Det, n = 20, t = 30$)

Target	N	$min(e)$	$max(e)$	$median(e)$	$\mu(e)$	$\sigma(e)$
`User_0`	600020	0.00	$4.89 \cdot 10^3$	$1.08 \cdot 10^3$	$1.06 \cdot 10^3$	$1.87 \cdot 10^2$
`User_1`	600020	0.00	$4.35 \cdot 10^3$	$1.12 \cdot 10^3$	$1.11 \cdot 10^3$	$1.84 \cdot 10^2$
`User_2`	600020	0.00	$6.30 \cdot 10^3$	$1.12 \cdot 10^3$	$1.10 \cdot 10^3$	$1.94 \cdot 10^2$
`User_3`	600020	0.00	$5.37 \cdot 10^3$	$1.10 \cdot 10^3$	$1.09 \cdot 10^3$	$1.91 \cdot 10^2$
total	600020	0.00	$1.07 \cdot 10^4$	$4.46 \cdot 10^3$	$4.36 \cdot 10^3$	$6.50 \cdot 10^2$
difference	600020	0.00	$1.07 \cdot 10^4$	$4.15 \cdot 10^2$	$4.39 \cdot 10^2$	$2.34 \cdot 10^2$

Table A.5: A table containing statistics on the log messages per milliseconds (e) measured during the execution of the target model `Telephony` grouped by state machine. The total and difference entries at the end of the table depict the row sum and the sum difference to the row minimum respectively. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

Appendix B

Performance Analysis: Figures

B.1 Counting-Logging Frequency Bar Charts

B.1.1 Synthetic Test: CounterDistributor

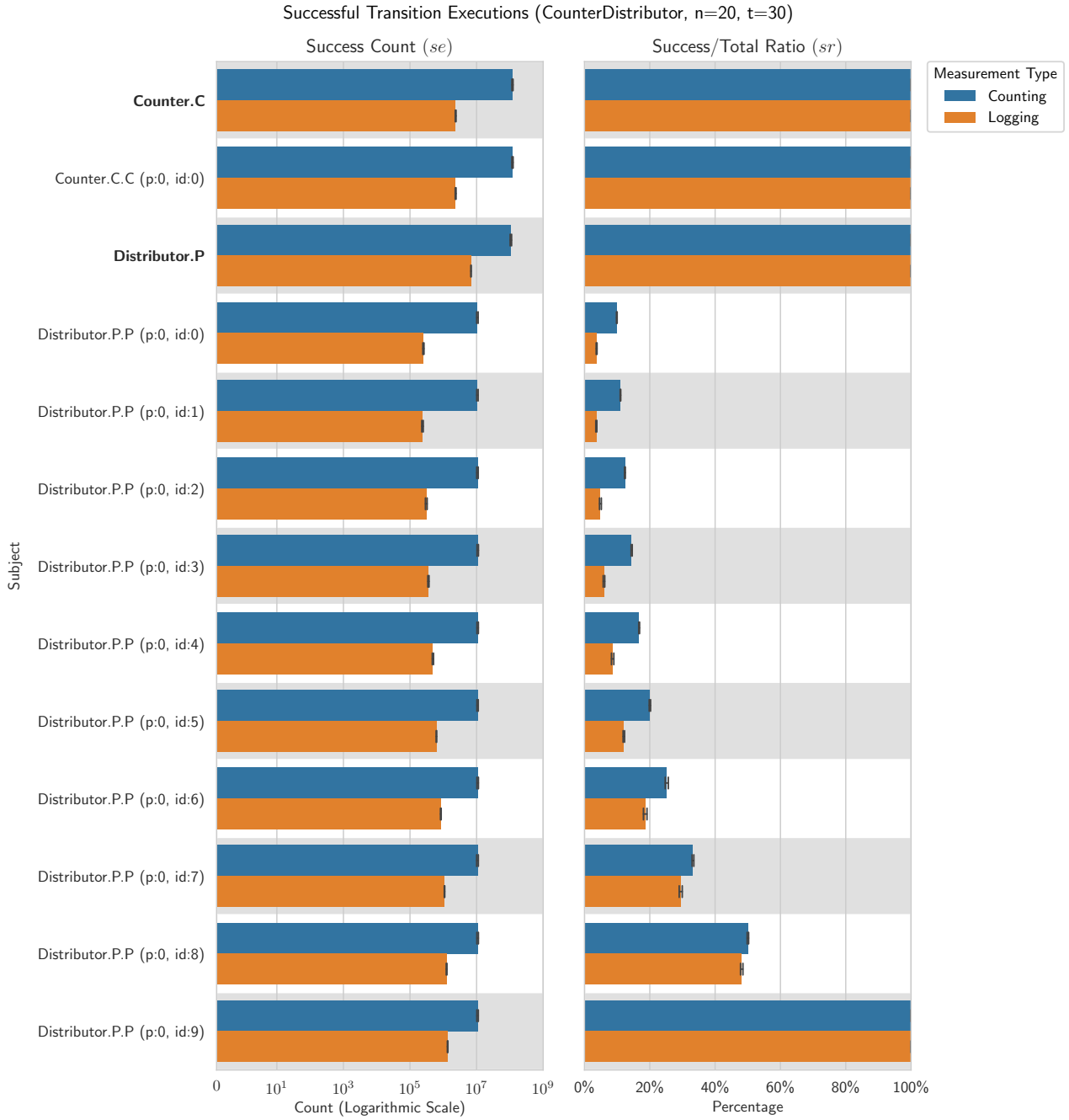


Figure B.1: A bar plot that reports the number of successful executions (se) and the percentage of successful executions (sr) for each decision node and transition in the target model `CounterDistributor`, where the results are grouped by the measurement type. Observe that the success count is depicted in a logarithmic scale, due to the wide range of measured values: the first column adheres to a linear scale; all subsequent columns are logarithmic. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

B.1.2 Synthetic Test: Tokens

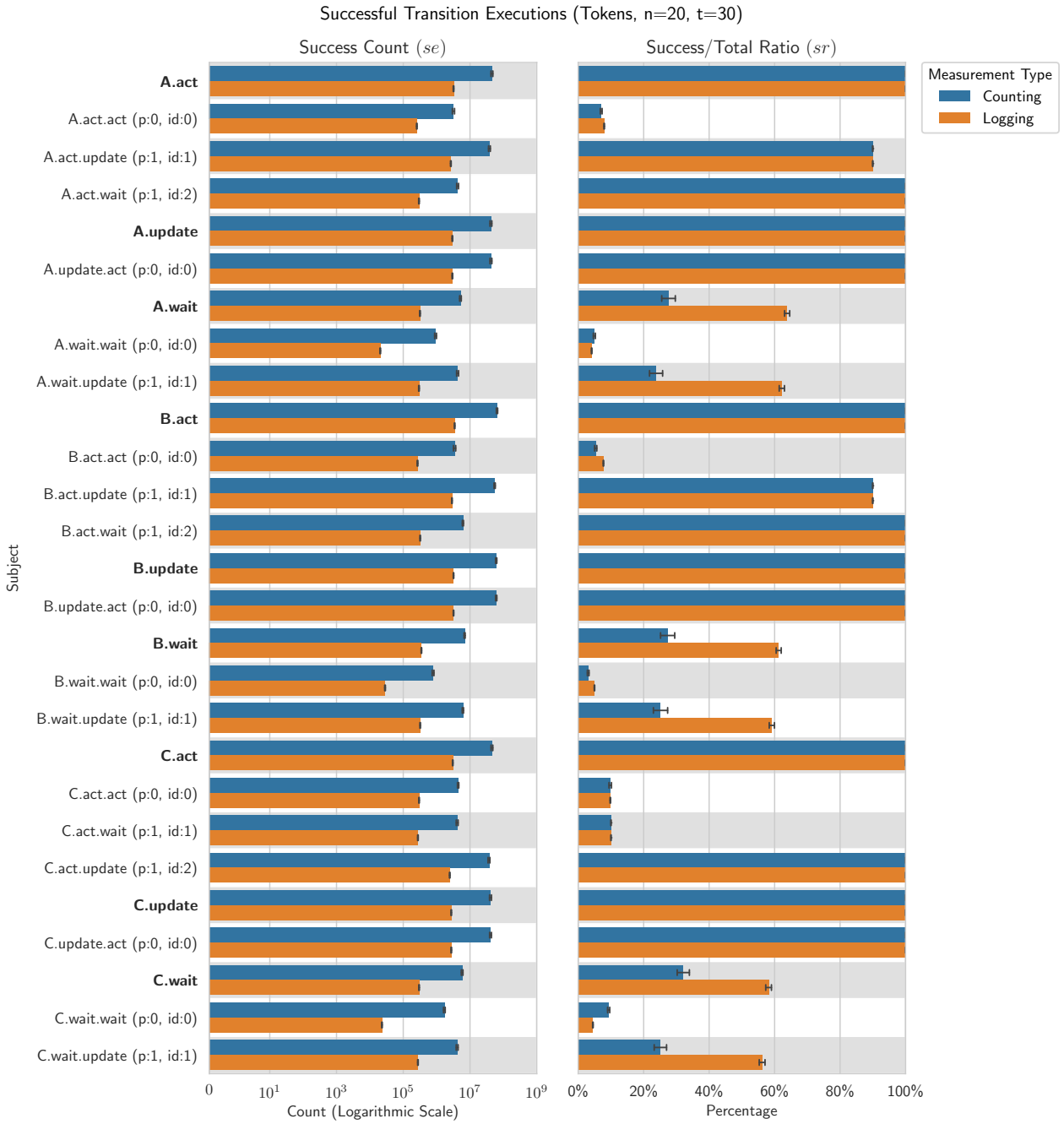


Figure B.2: A bar plot that reports the number of successful executions (se) and the percentage of successful executions (sr) for each decision node and transition in the target model `Tokens`, where the results are grouped by the measurement type. Observe that the success count is depicted in a logarithmic scale, due to the wide range of measured values: the first column adheres to a linear scale; all subsequent columns are logarithmic. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

B.1.3 Practical Test: Elevator

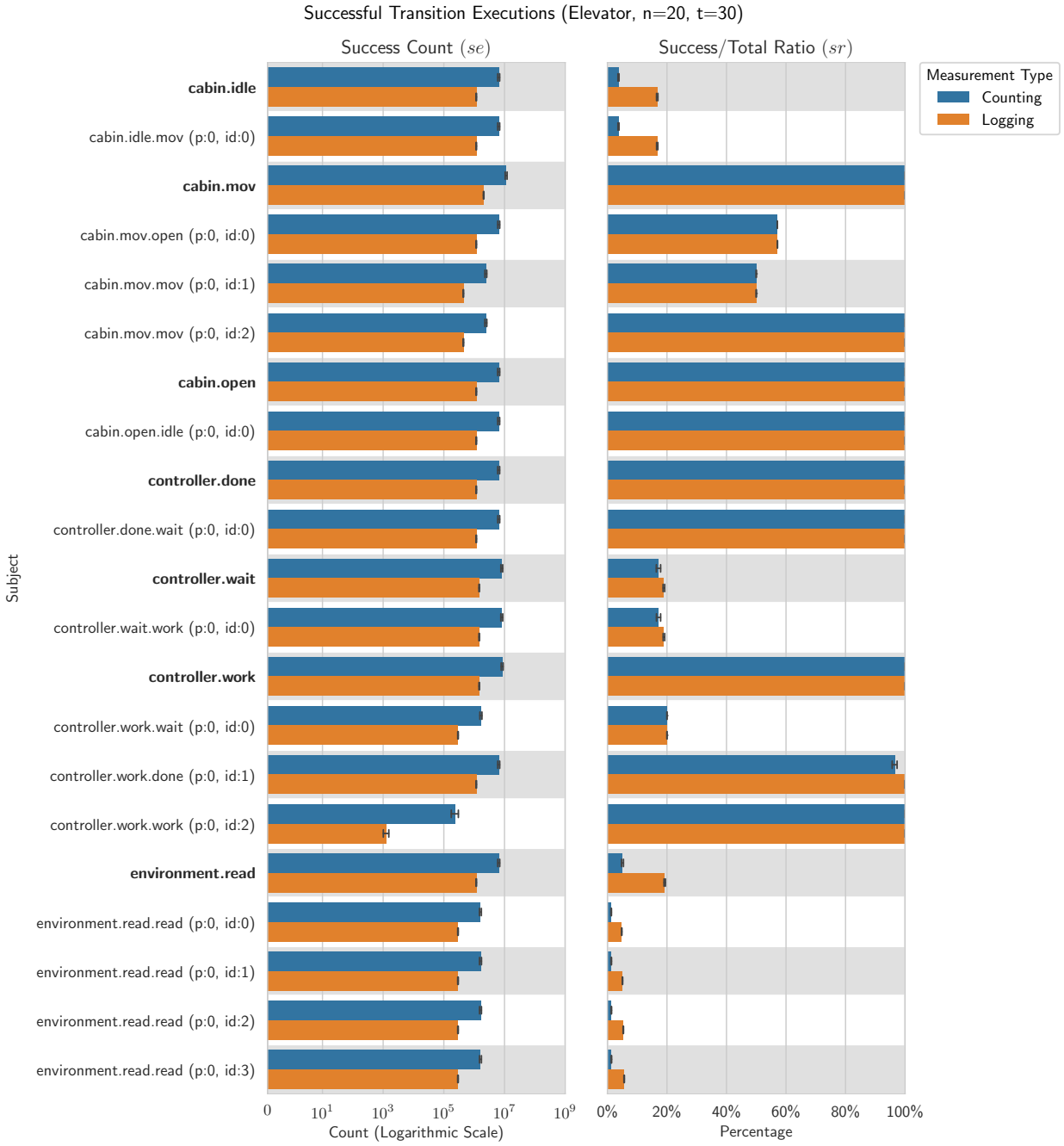


Figure B.3: A bar plot that reports the number of successful executions (se) and the percentage of successful executions (sr) for each decision node and transition in the target model `Elevator`, where the results are grouped by the measurement type. Observe that the success count is depicted in a logarithmic scale, due to the wide range of measured values: the first column adheres to a linear scale; all subsequent columns are logarithmic. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

B.1.4 Practical Test: ToadsAndFrogs

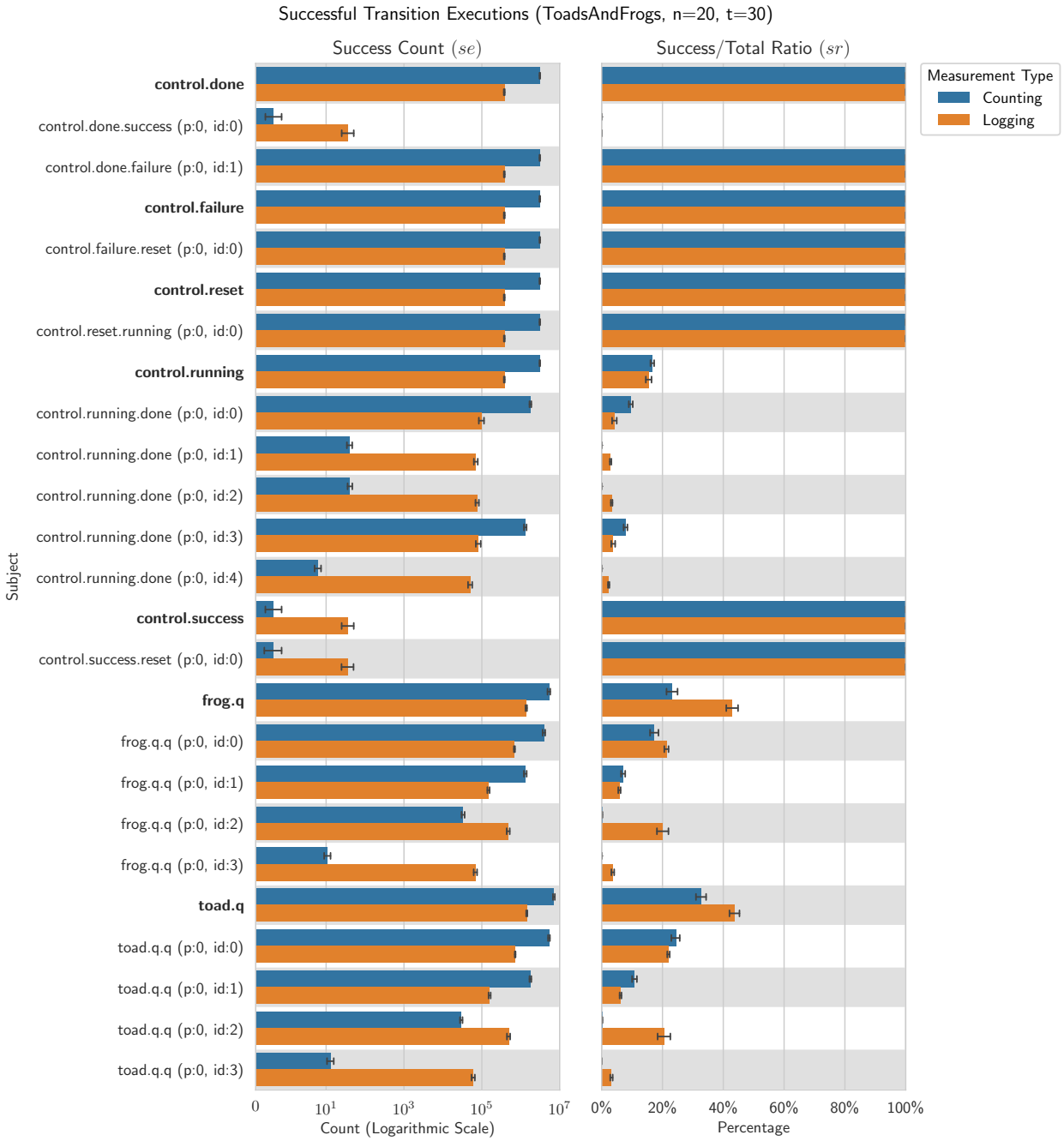


Figure B.4: A bar plot that reports the number of successful executions (se) and the percentage of successful executions (sr) for each decision node and transition in the target model `ToadsAndFrogs`, where the results are grouped by the measurement type. Observe that the success count is depicted in a logarithmic scale, due to the wide range of measured values: the first column adheres to a linear scale; all subsequent columns are logarithmic. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

B.1.5 Practical Test: Telephony

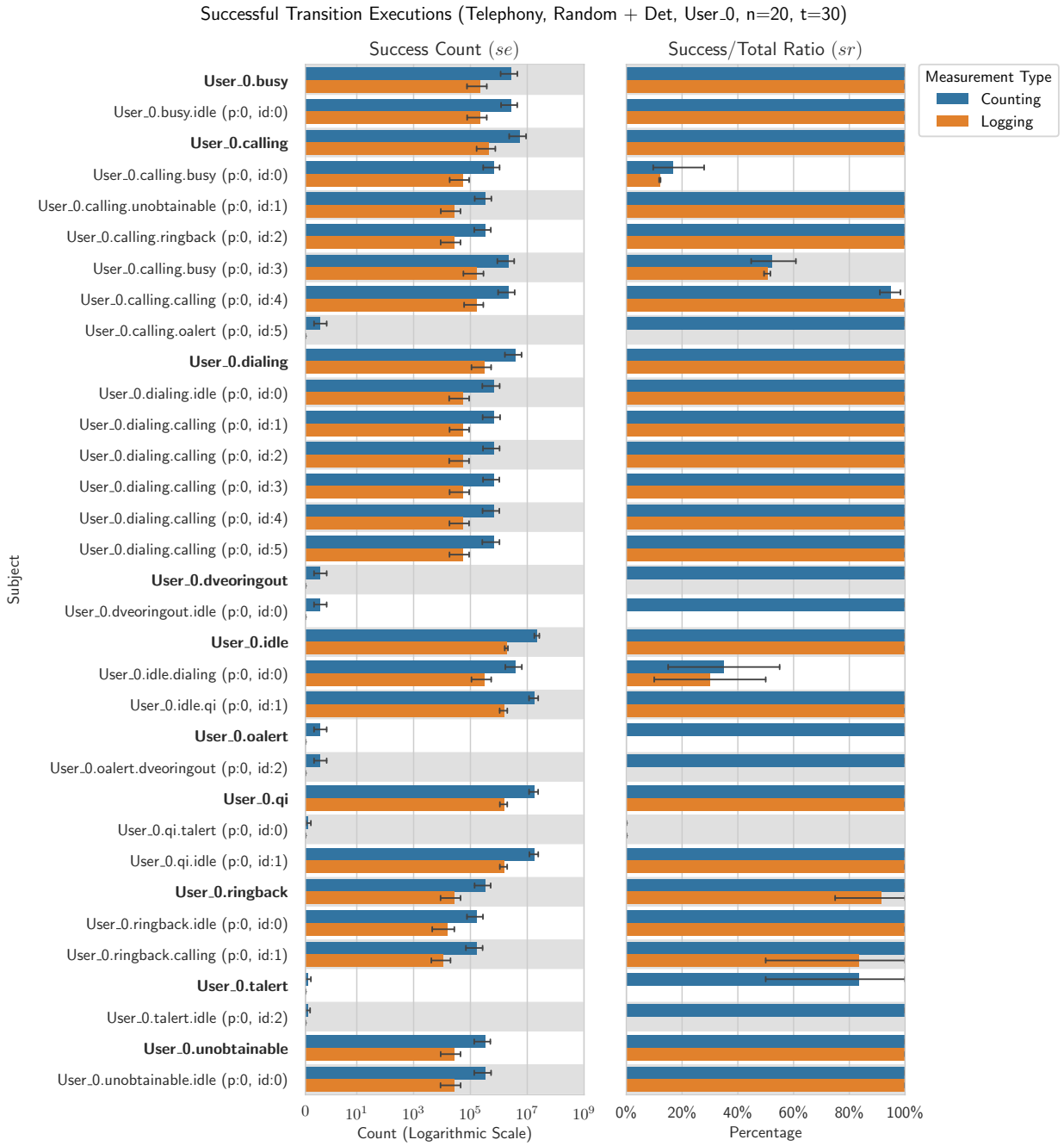


Figure B.5: A bar plot that reports the number of successful executions (se) and the percentage of successful executions (sr) for each decision node and transition in the target model `Telephony`, where the results are grouped by the measurement type. Observe that the success count is depicted in a logarithmic scale, due to the wide range of measured values: the first column adheres to a linear scale; all subsequent columns are logarithmic. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

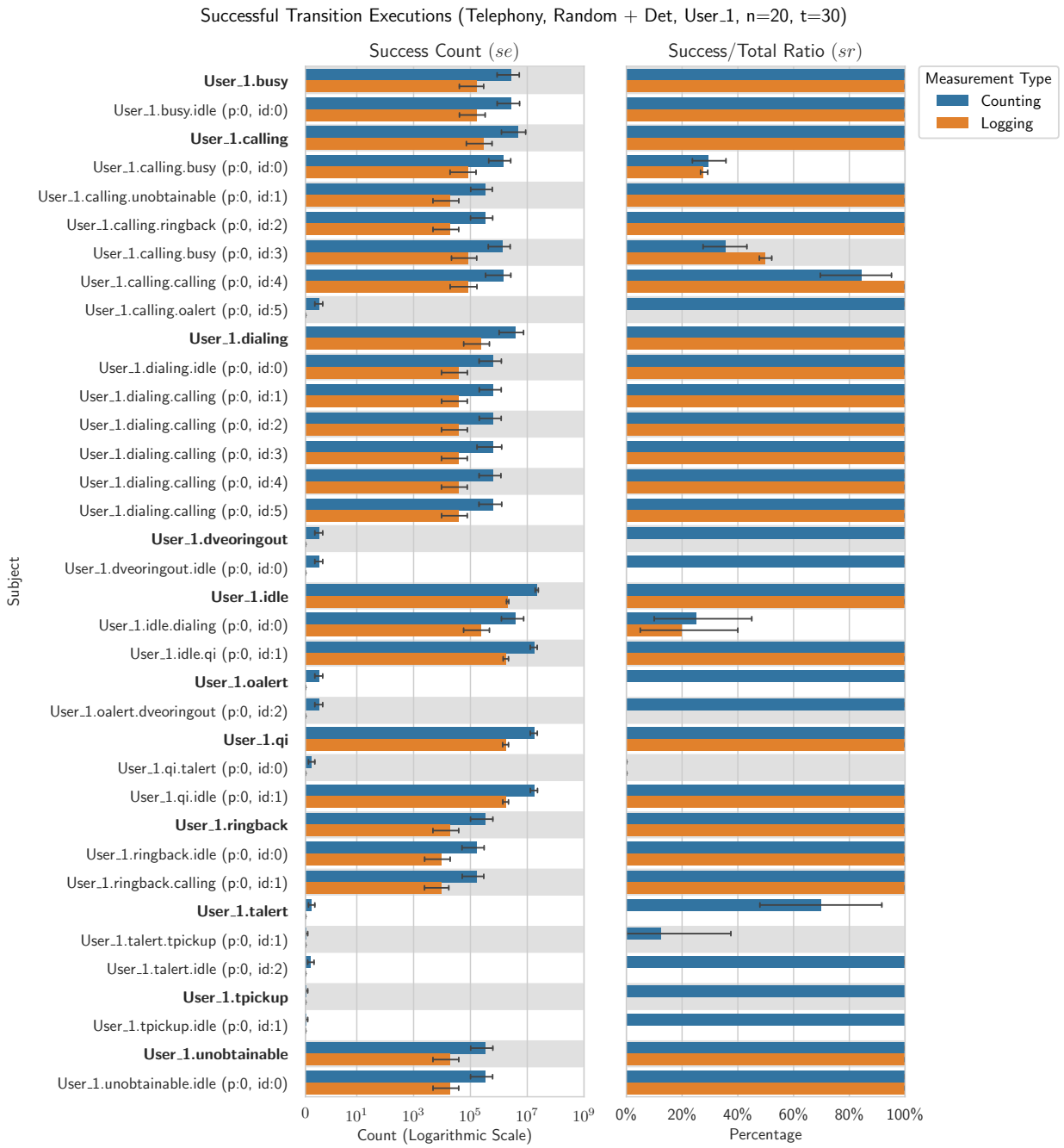


Figure B.6: A bar plot that reports the number of successful executions (se) and the percentage of successful executions (sr) for each decision node and transition in the target model `Telephony`, where the results are grouped by the measurement type. Observe that the success count is depicted in a logarithmic scale, due to the wide range of measured values: the first column adheres to a linear scale; all subsequent columns are logarithmic. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

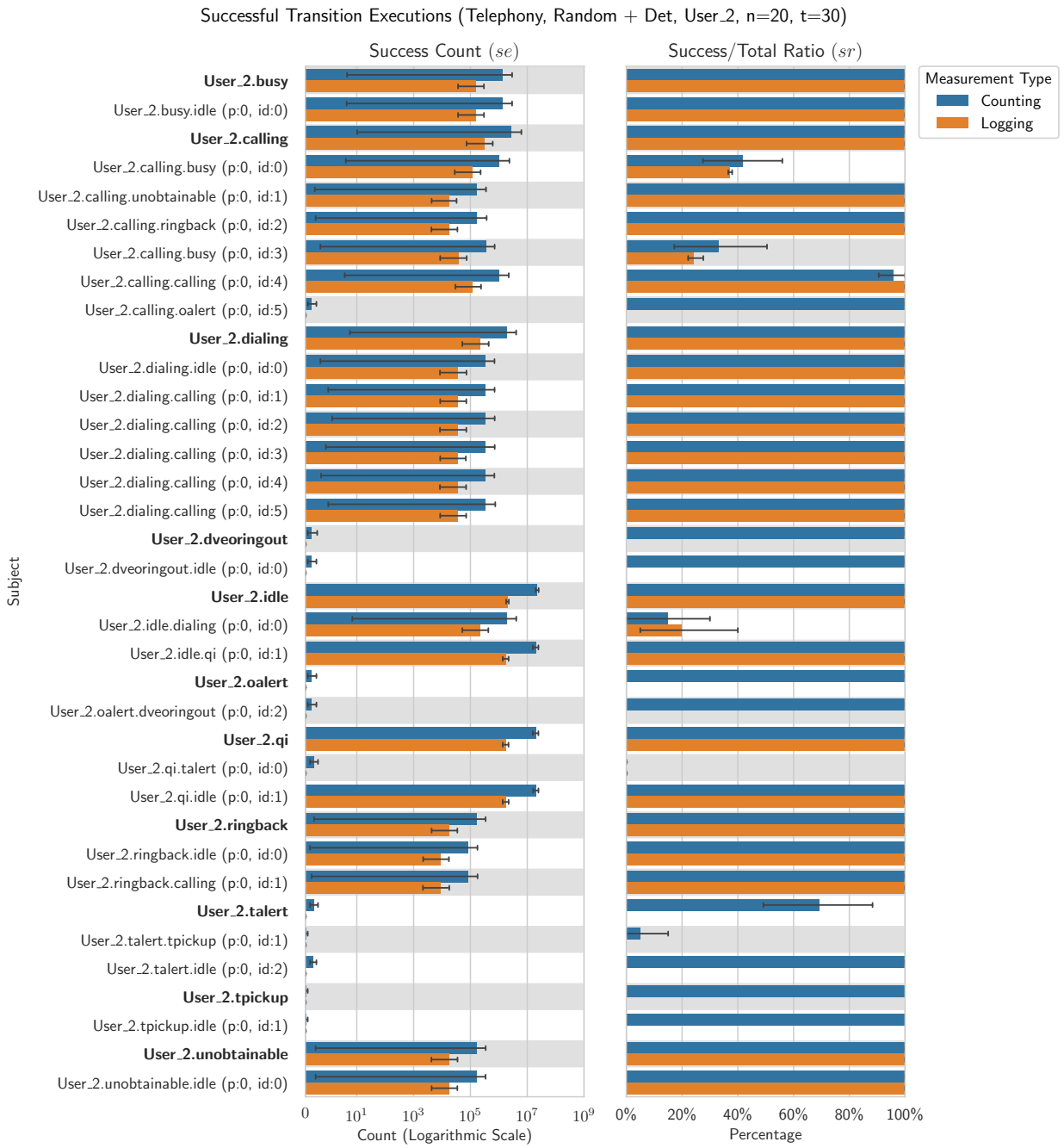


Figure B.7: A bar plot that reports the number of successful executions (se) and the percentage of successful executions (sr) for each decision node and transition in the target model `Telephony`, where the results are grouped by the measurement type. Observe that the success count is depicted in a logarithmic scale, due to the wide range of measured values: the first column adheres to a linear scale; all subsequent columns are logarithmic. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

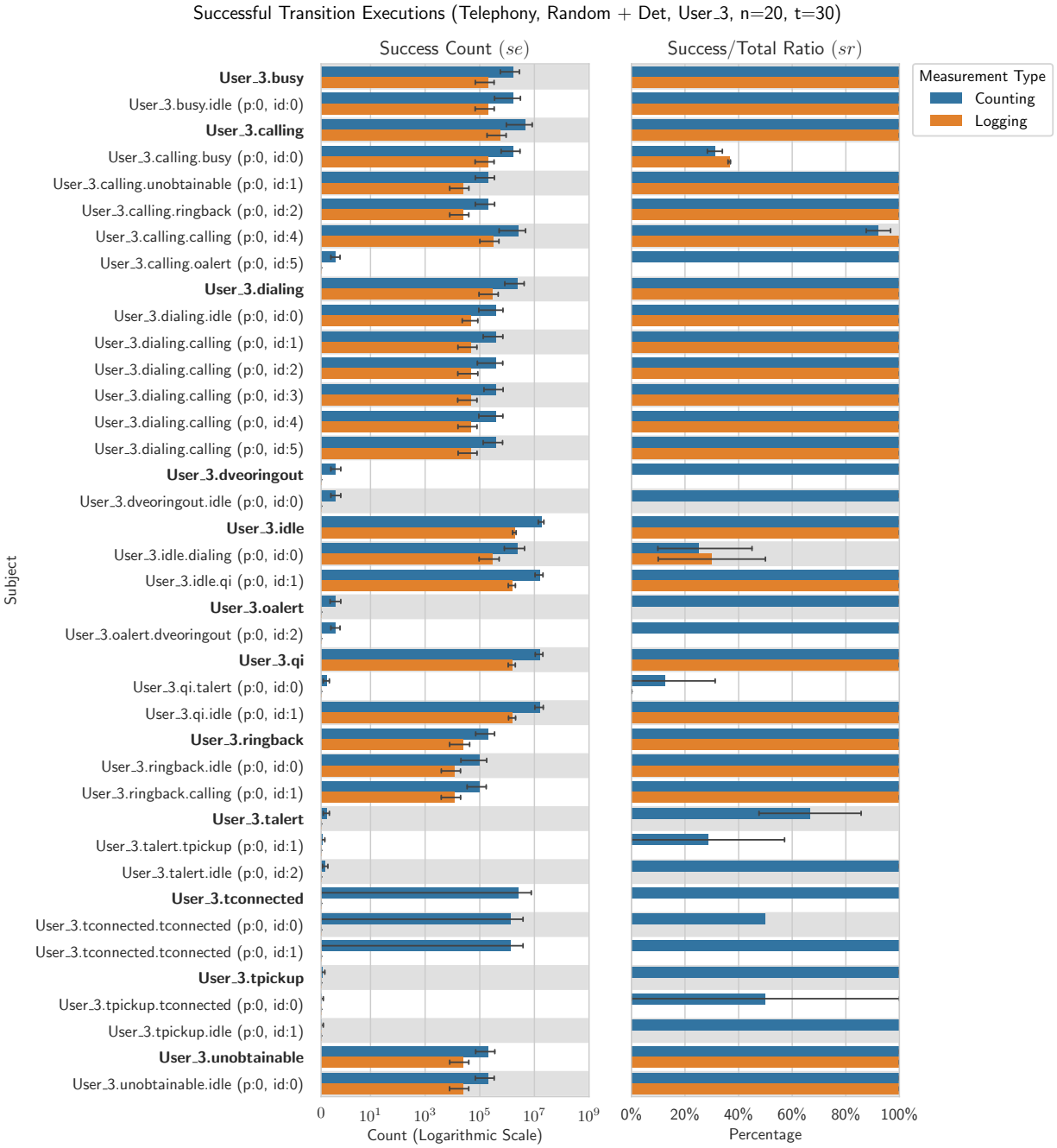


Figure B.8: A bar plot that reports the number of successful executions (se) and the percentage of successful executions (sr) for each decision node and transition in the target model `Telephony`, where the results are grouped by the measurement type. Observe that the success count is depicted in a logarithmic scale, due to the wide range of measured values: the first column adheres to a linear scale; all subsequent columns are logarithmic. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

B.2 Decision Mode Frequency Bar Charts

B.2.1 Synthetic Test: CounterDistributor

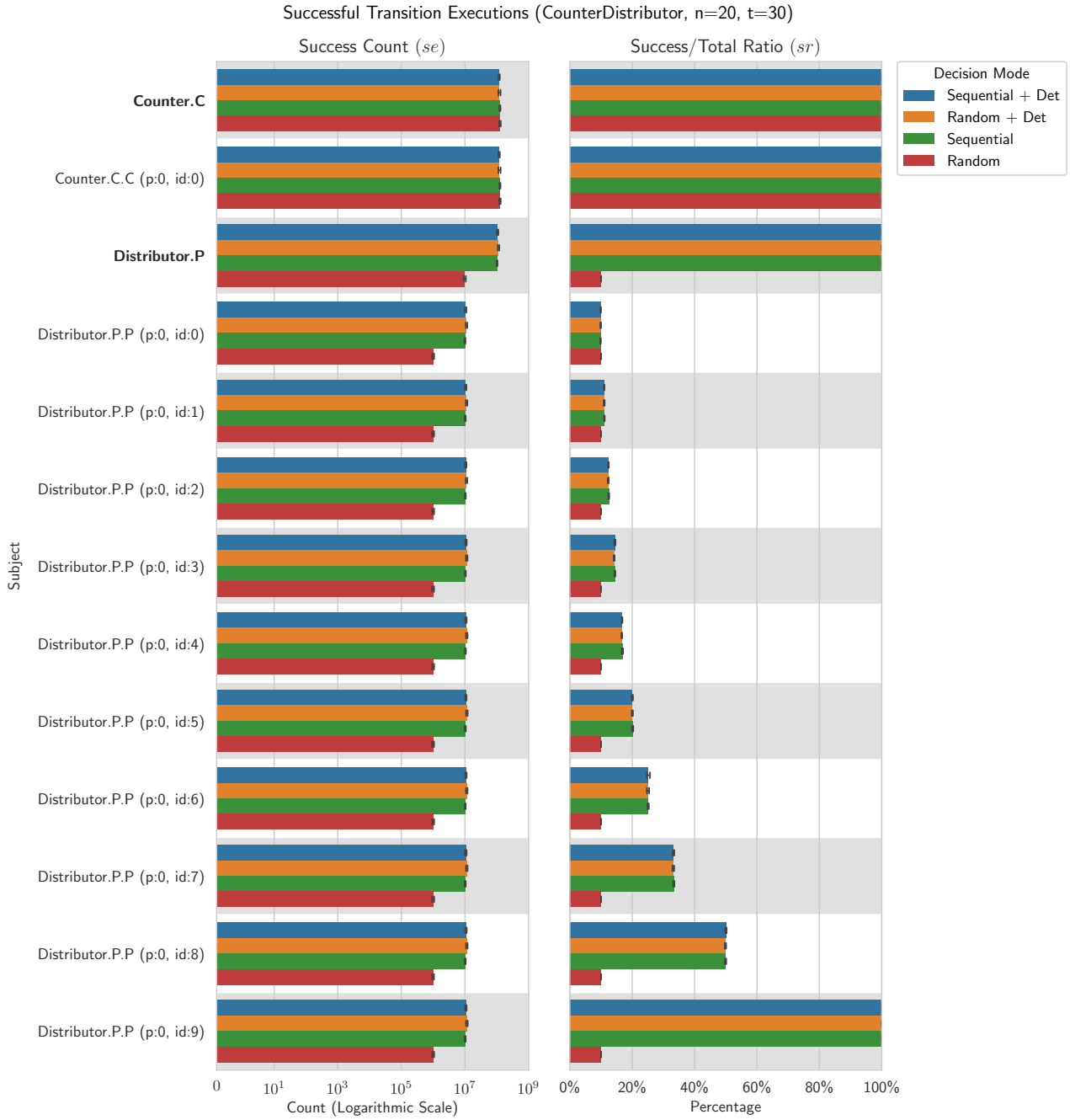


Figure B.9: A bar plot that reports the number of successful executions (se) and the percentage of successful executions (sr) for each decision node and transition in the target model `CounterDistributor`, where the results are grouped by the decision mode. Observe that the success count is depicted in a logarithmic scale, due to the wide range of measured values: the first column adheres to a linear scale; all subsequent columns are logarithmic. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

B.2.2 Synthetic Test: Tokens

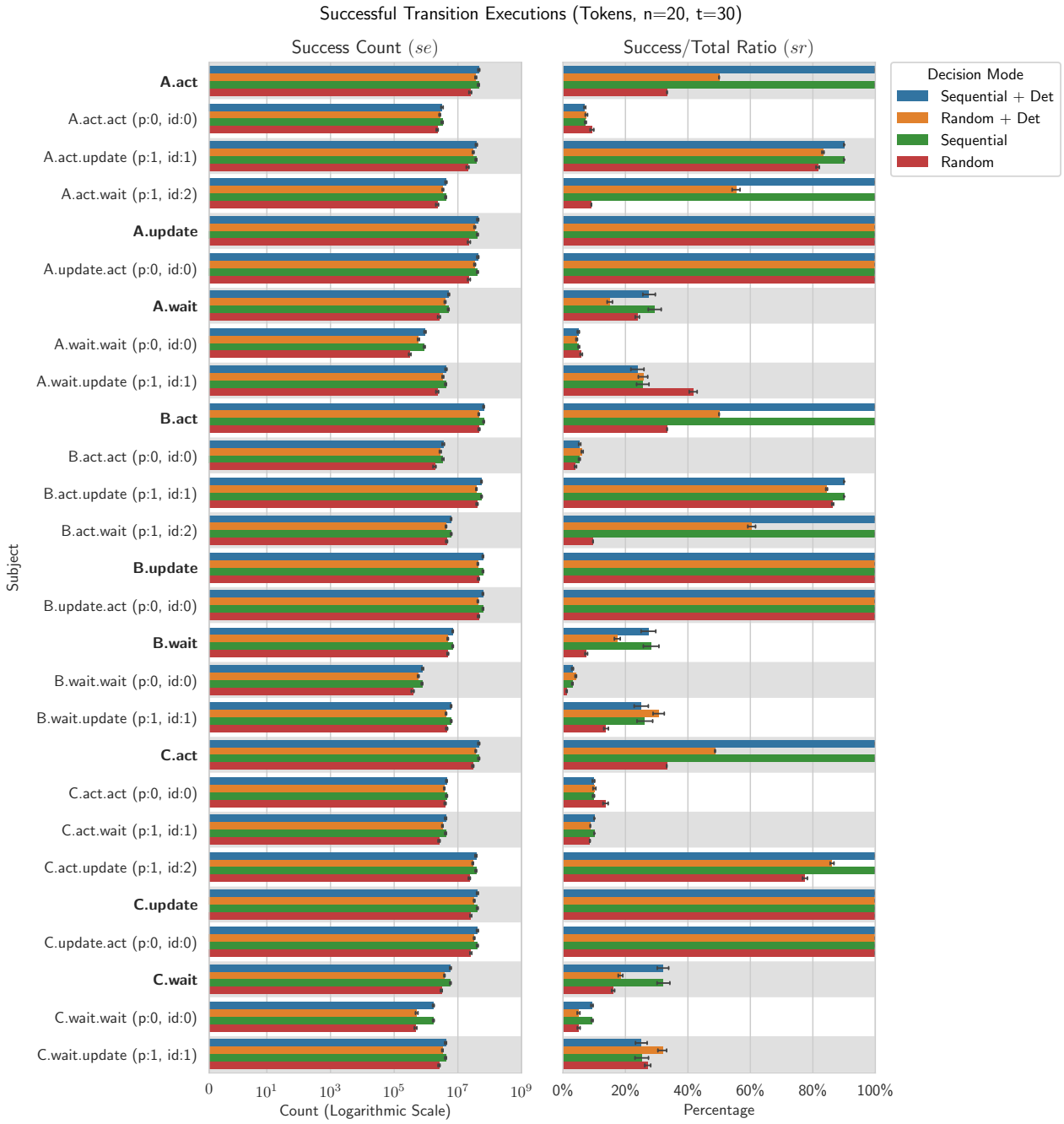


Figure B.10: A bar plot that reports the number of successful executions (se) and the percentage of successful executions (sr) for each decision node and transition in the target model `Tokens`, where the results are grouped by the decision mode. Observe that the success count is depicted in a logarithmic scale, due to the wide range of measured values: the first column adheres to a linear scale; all subsequent columns are logarithmic. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

B.2.3 Practical Test: Elevator

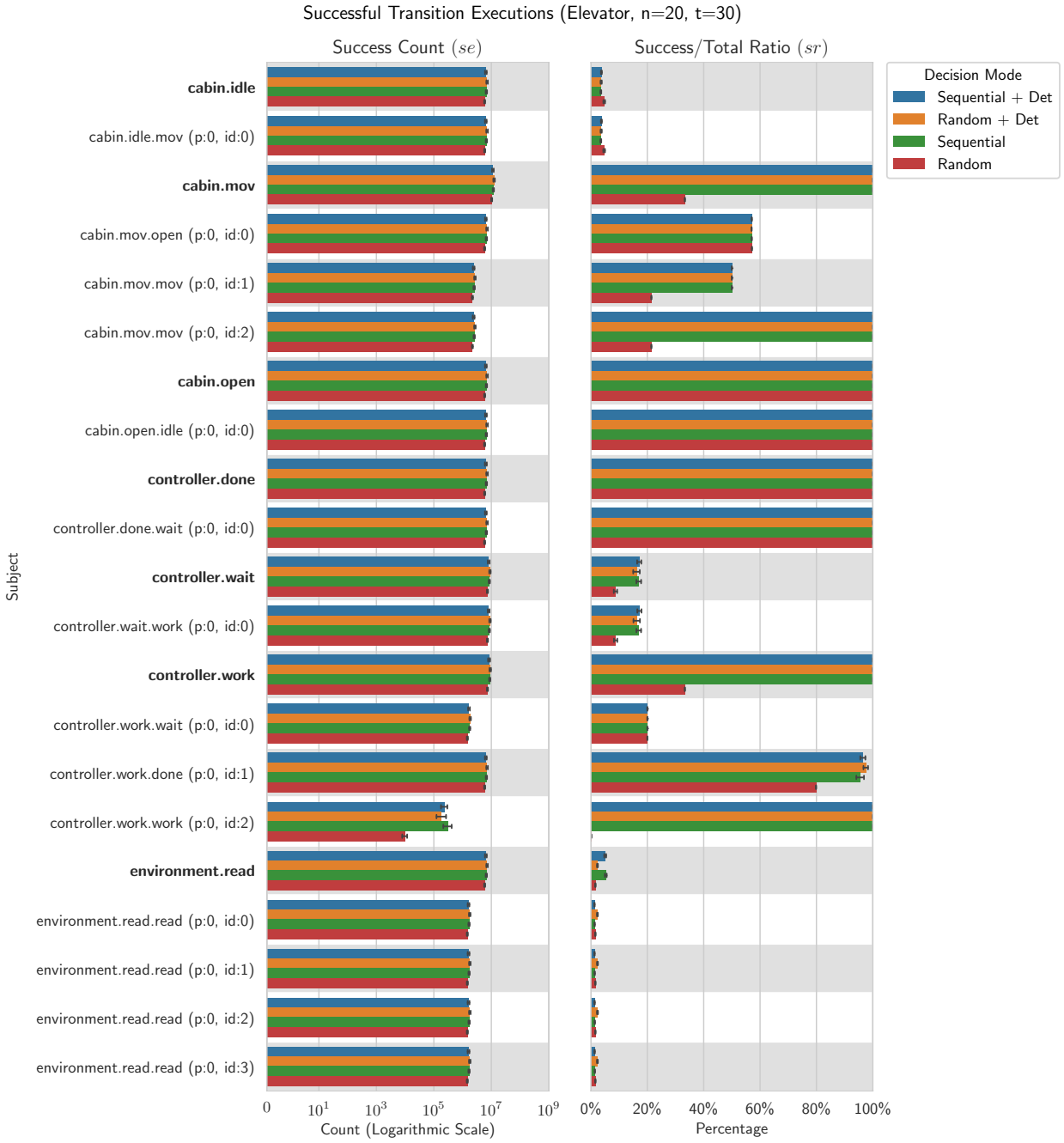


Figure B.11: A bar plot that reports the number of successful executions (se) and the percentage of successful executions (sr) for each decision node and transition in the target model `Elevator`, where the results are grouped by the decision mode. Observe that the success count is depicted in a logarithmic scale, due to the wide range of measured values: the first column adheres to a linear scale; all subsequent columns are logarithmic. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

B.2.4 Practical Test: ToadsAndFrogs

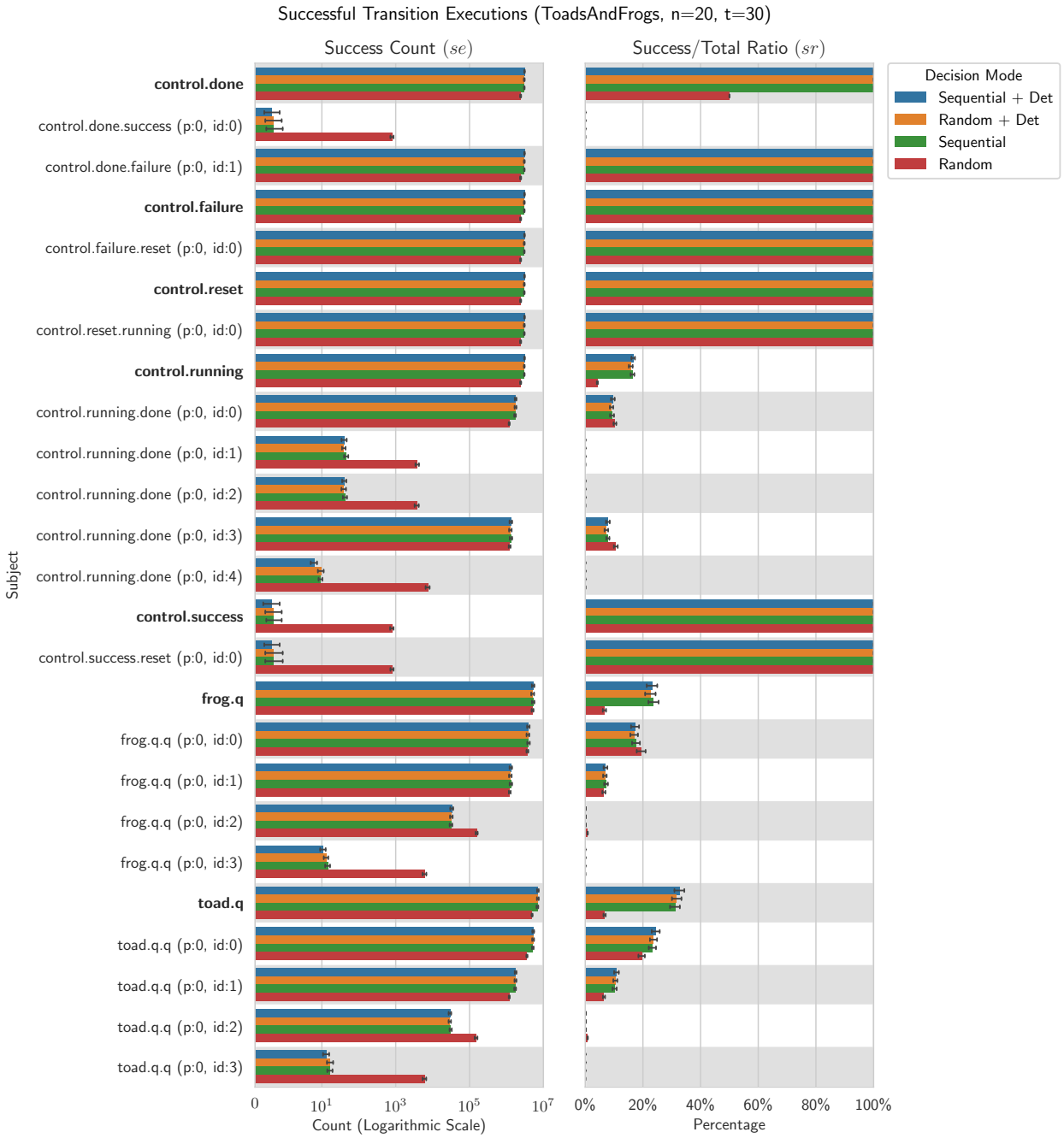


Figure B.12: A bar plot that reports the number of successful executions (se) and the percentage of successful executions (sr) for each decision node and transition in the target model `ToadsAndFrogs`, where the results are grouped by the decision mode. Observe that the success count is depicted in a logarithmic scale, due to the wide range of measured values: the first column adheres to a linear scale; all subsequent columns are logarithmic. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

B.2.5 Practical Test: Telephony

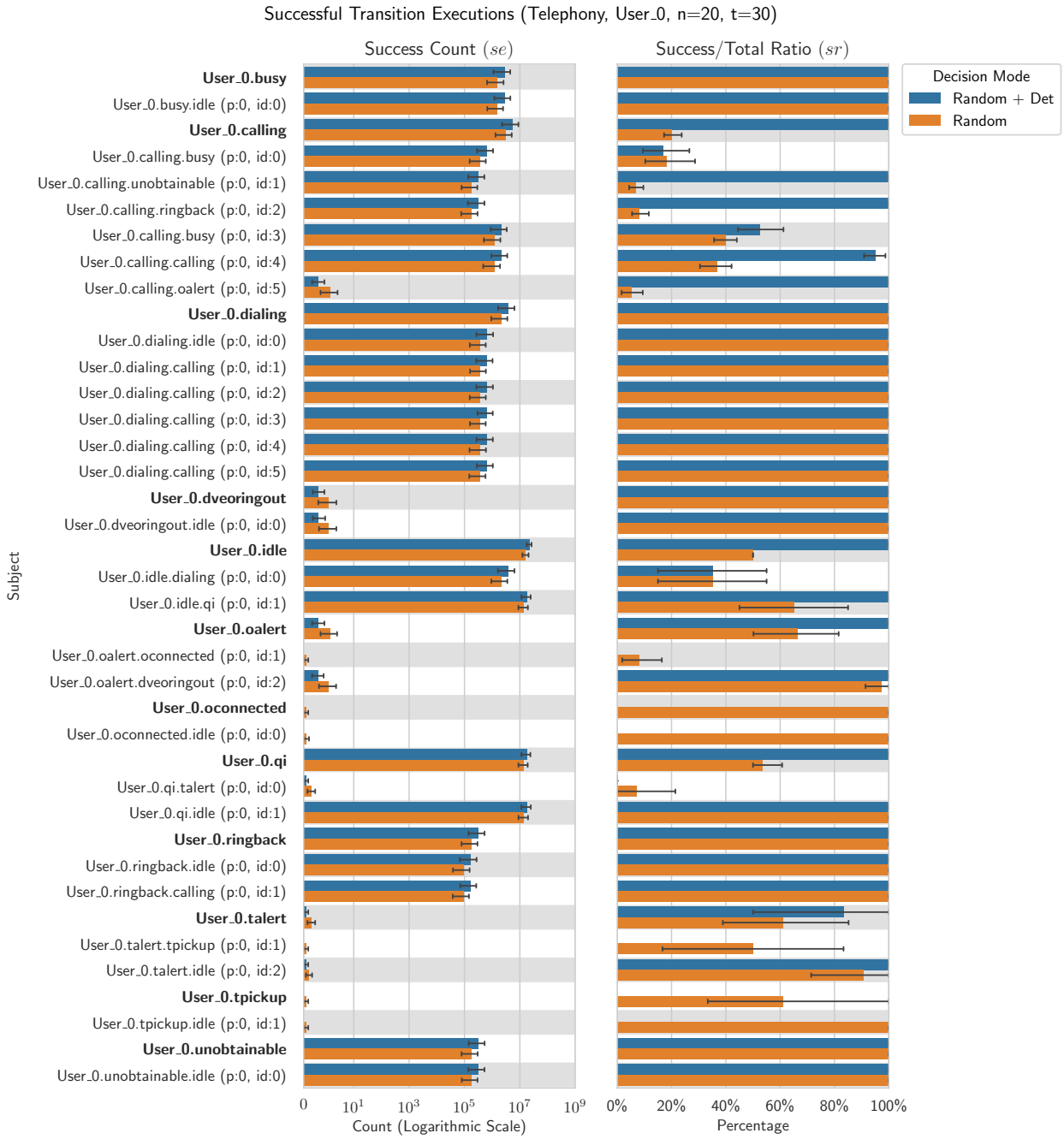


Figure B.13: A bar plot that reports the number of successful executions (se) and the percentage of successful executions (sr) for each decision node and transition in the target model `Telephony`, where the results are grouped by the decision mode. Observe that the success count is depicted in a logarithmic scale, due to the wide range of measured values: the first column adheres to a linear scale; all subsequent columns are logarithmic. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

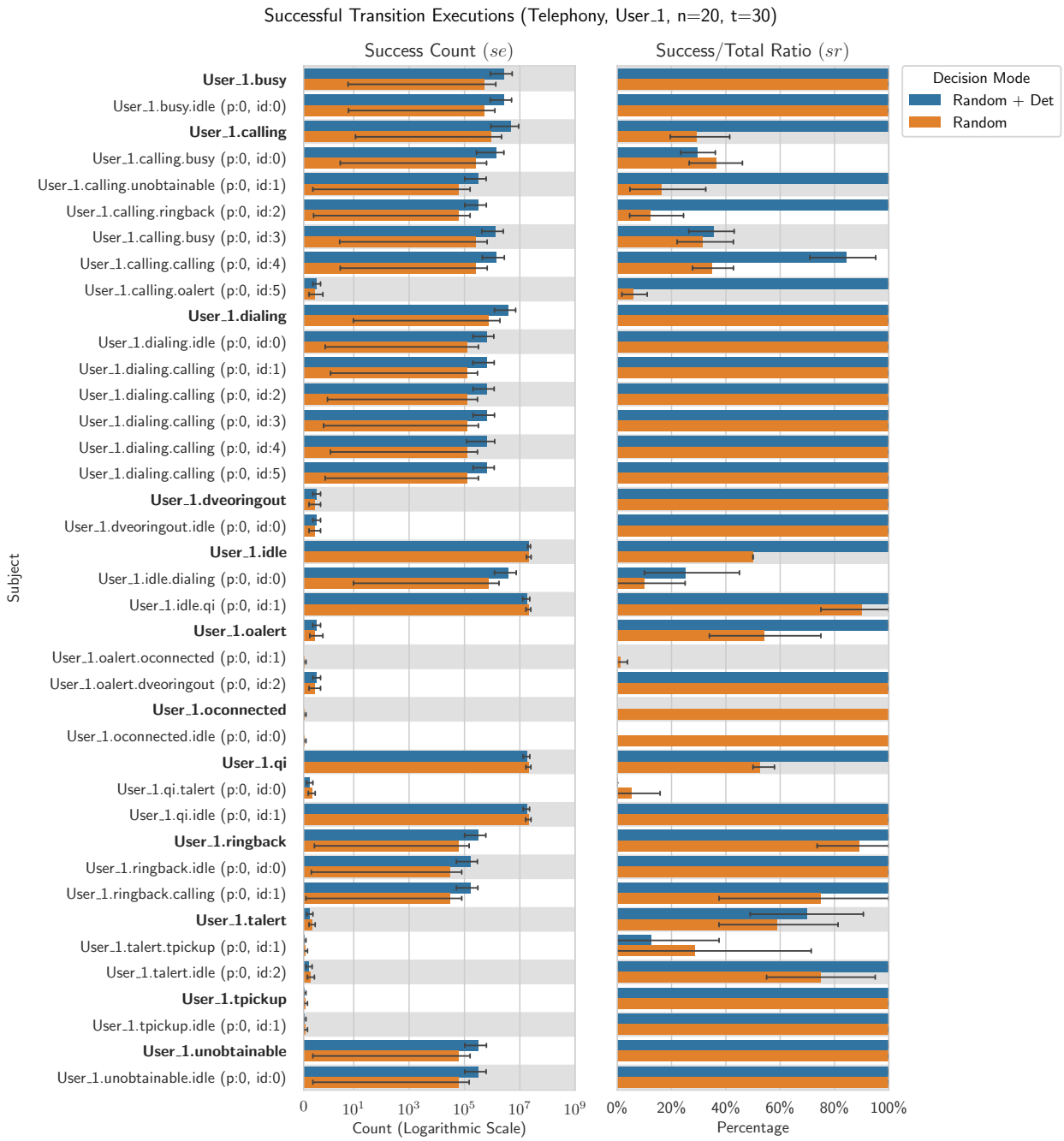


Figure B.14: A bar plot that reports the number of successful executions (se) and the percentage of successful executions (sr) for each decision node and transition in the target model `Telephony`, where the results are grouped by the decision mode. Observe that the success count is depicted in a logarithmic scale, due to the wide range of measured values: the first column adheres to a linear scale; all subsequent columns are logarithmic. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

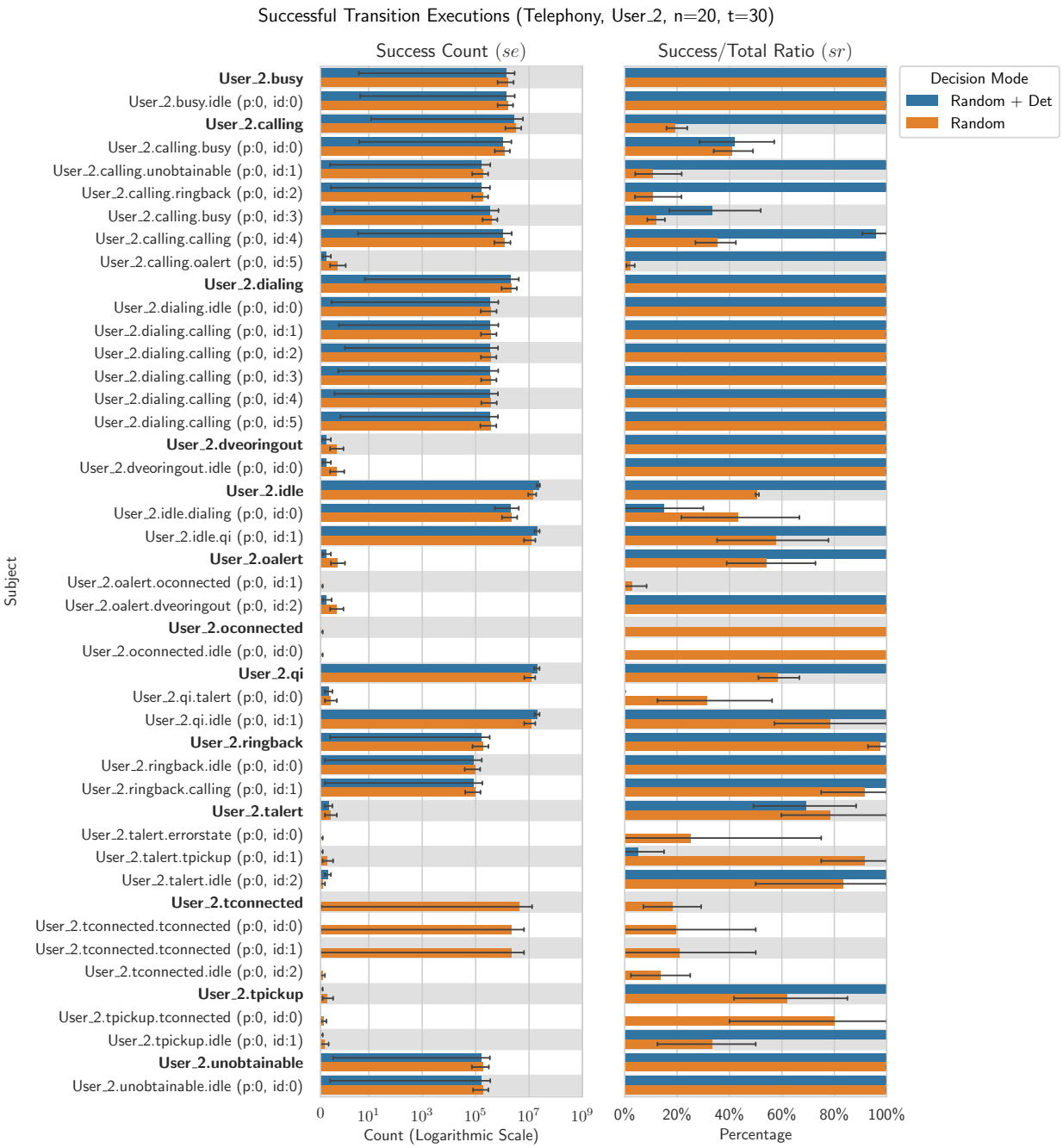


Figure B.15: A bar plot that reports the number of successful executions (se) and the percentage of successful executions (sr) for each decision node and transition in the target model `Telephony`, where the results are grouped by the decision mode. Observe that the success count is depicted in a logarithmic scale, due to the wide range of measured values: the first column adheres to a linear scale; all subsequent columns are logarithmic. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

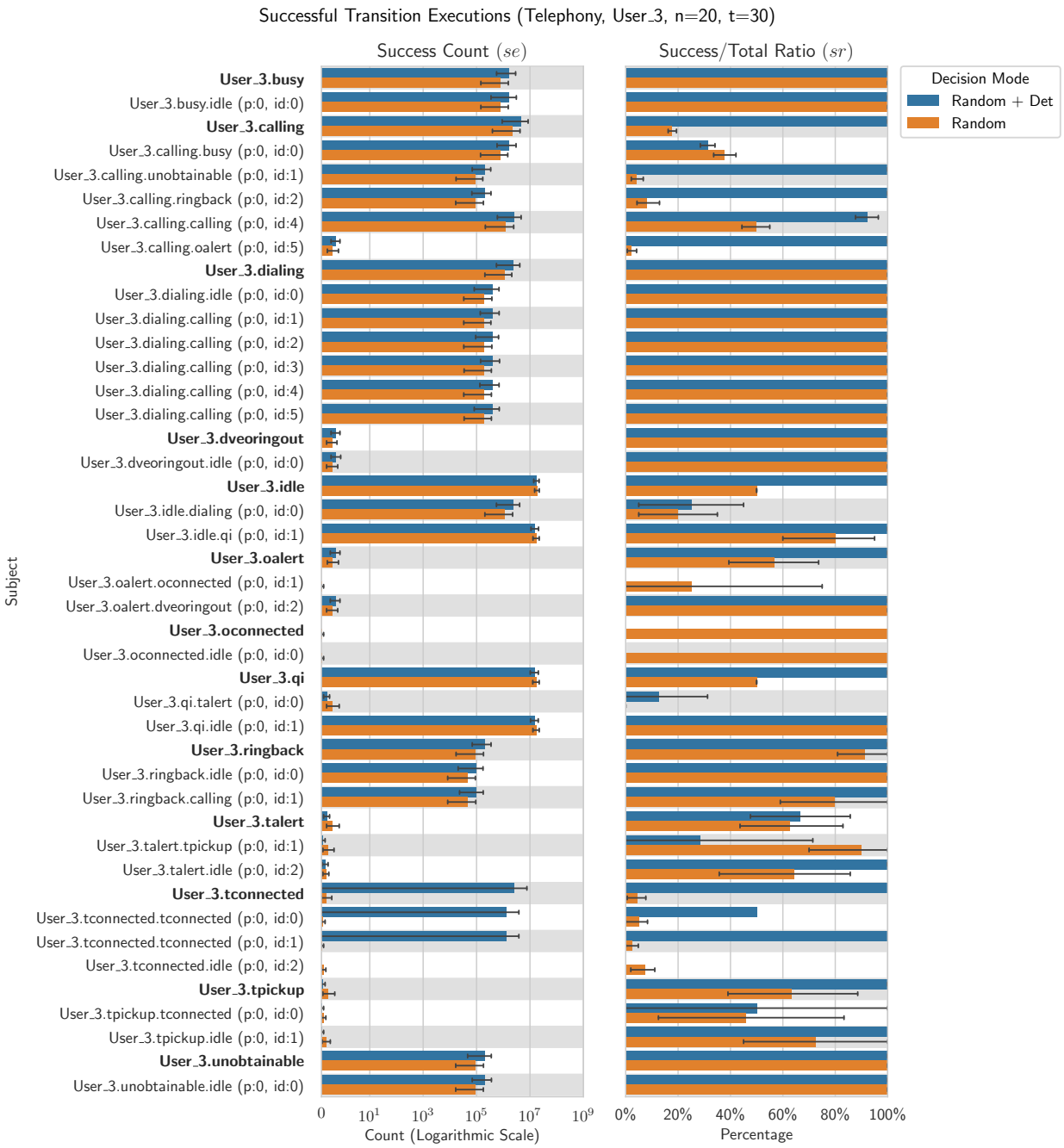


Figure B.16: A bar plot that reports the number of successful executions (se) and the percentage of successful executions (sr) for each decision node and transition in the target model `Telephony`, where the results are grouped by the decision mode. Observe that the success count is depicted in a logarithmic scale, due to the wide range of measured values: the first column adheres to a linear scale; all subsequent columns are logarithmic. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

B.3 Locking Mode Frequency Bar Charts

B.3.1 Synthetic Test: CounterDistributor

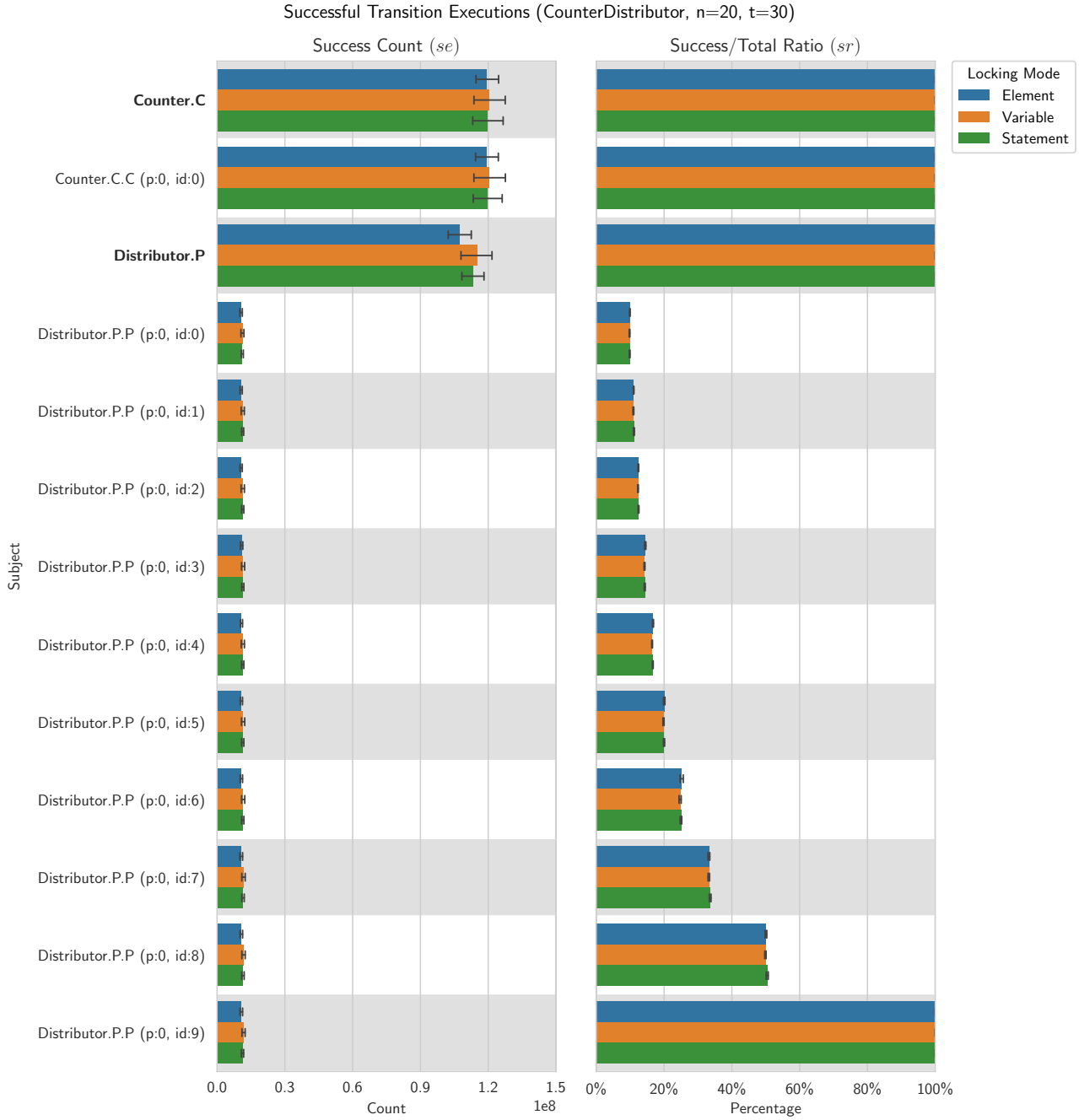


Figure B.17: A bar plot that reports the number of successful executions (se) and the percentage of successful executions (sr) for each decision node and transition in the target model `CounterDistributor`, where the results are grouped by the locking mode. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

B.3.2 Synthetic Test: Tokens

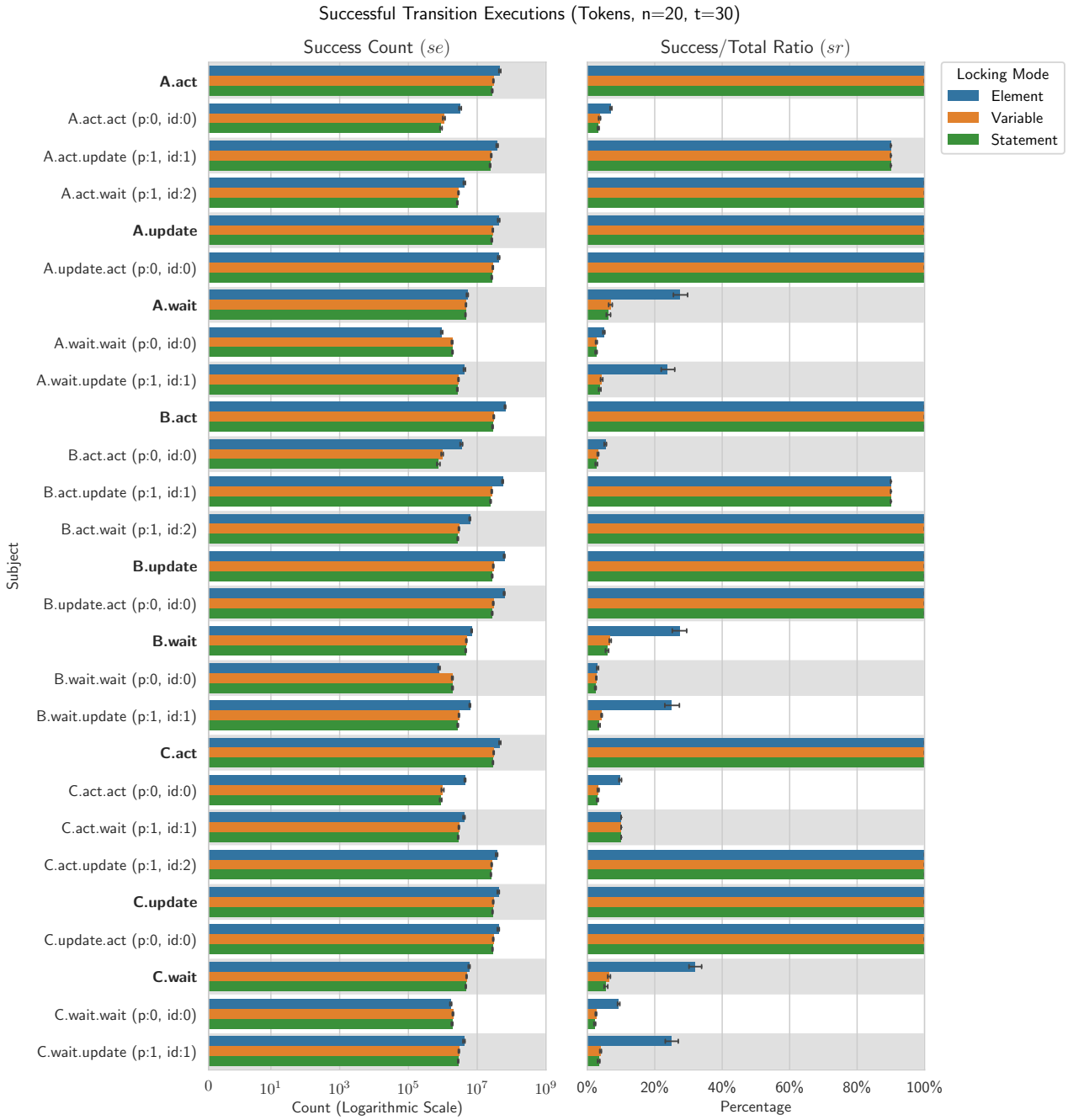


Figure B.18: A bar plot that reports the number of successful executions (se) and the percentage of successful executions (sr) for each decision node and transition in the target model `Tokens`, where the results are grouped by the locking mode. Observe that the success count is depicted in a logarithmic scale, due to the wide range of measured values: the first column adheres to a linear scale; all subsequent columns are logarithmic. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

B.3.3 Practical Test: Elevator

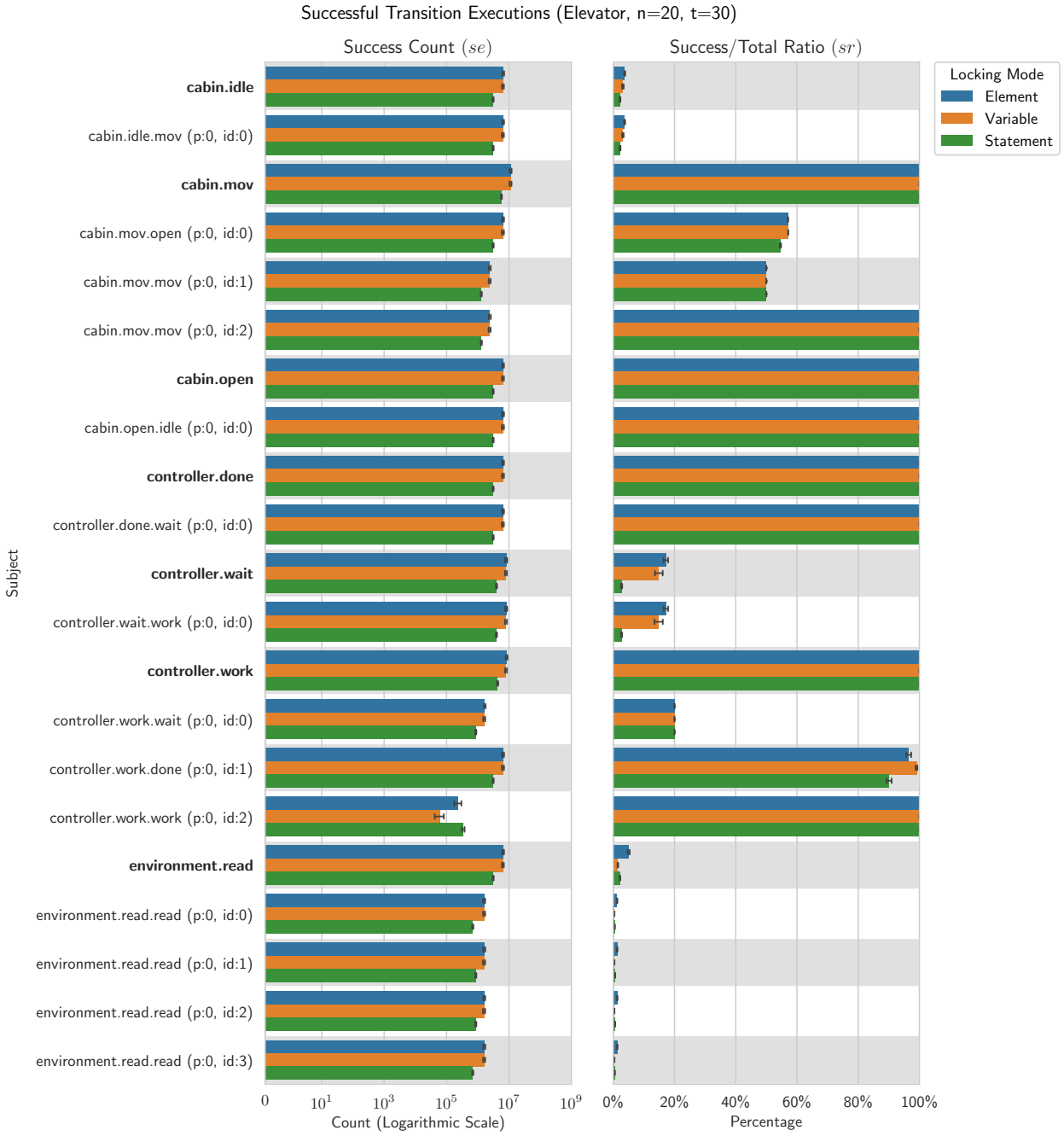


Figure B.19: A bar plot that reports the number of successful executions (se) and the percentage of successful executions (sr) for each decision node and transition in the target model `Elevator`, where the results are grouped by the locking mode. Observe that the success count is depicted in a logarithmic scale, due to the wide range of measured values: the first column adheres to a linear scale; all subsequent columns are logarithmic. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

B.3.4 Practical Test: ToadsAndFrogs

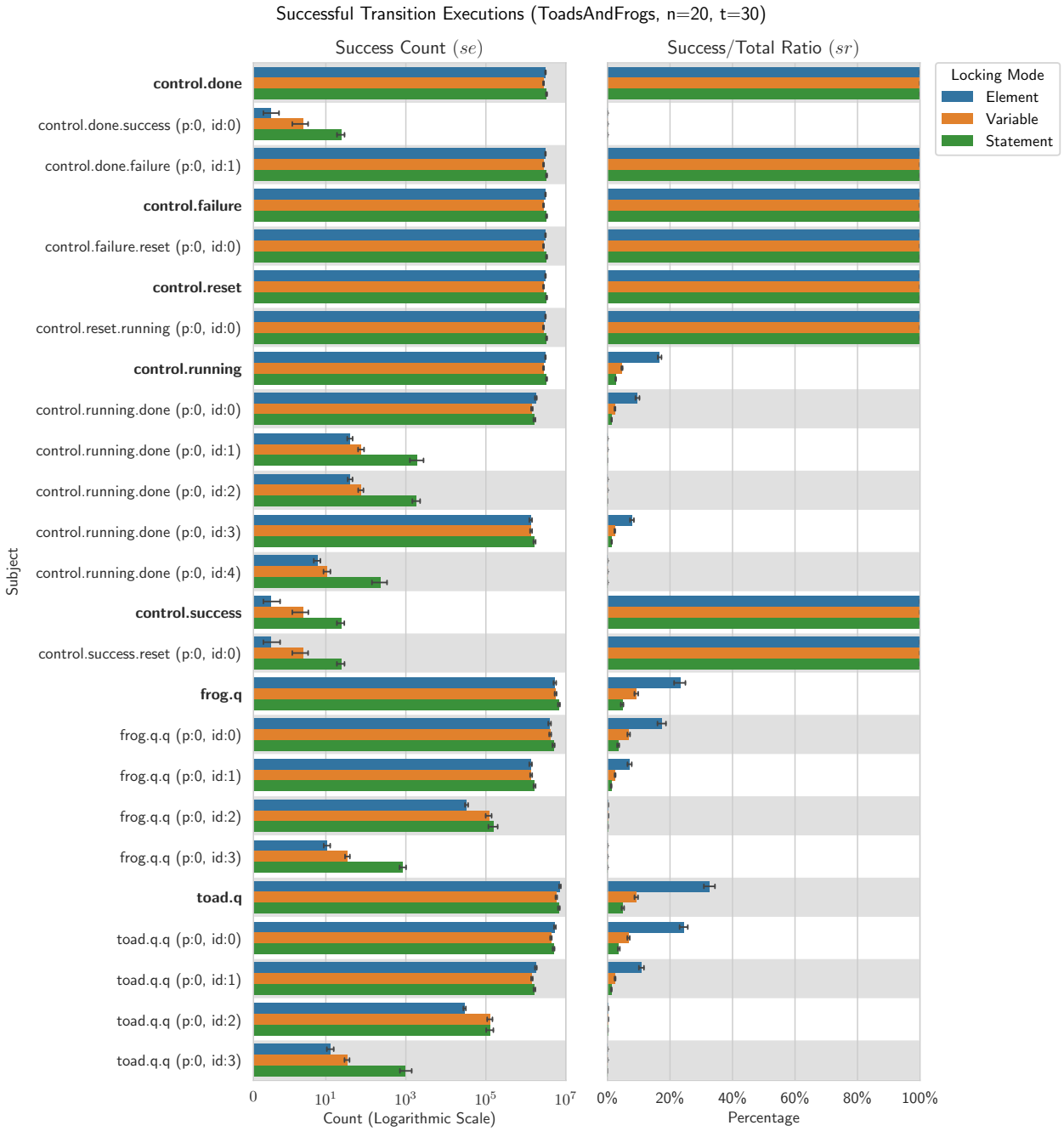


Figure B.20: A bar plot that reports the number of successful executions (se) and the percentage of successful executions (sr) for each decision node and transition in the target model `ToadsAndFrogs`, where the results are grouped by the locking mode. Observe that the success count is depicted in a logarithmic scale, due to the wide range of measured values: the first column adheres to a linear scale; all subsequent columns are logarithmic. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

B.3.5 Practical Test: Telephony

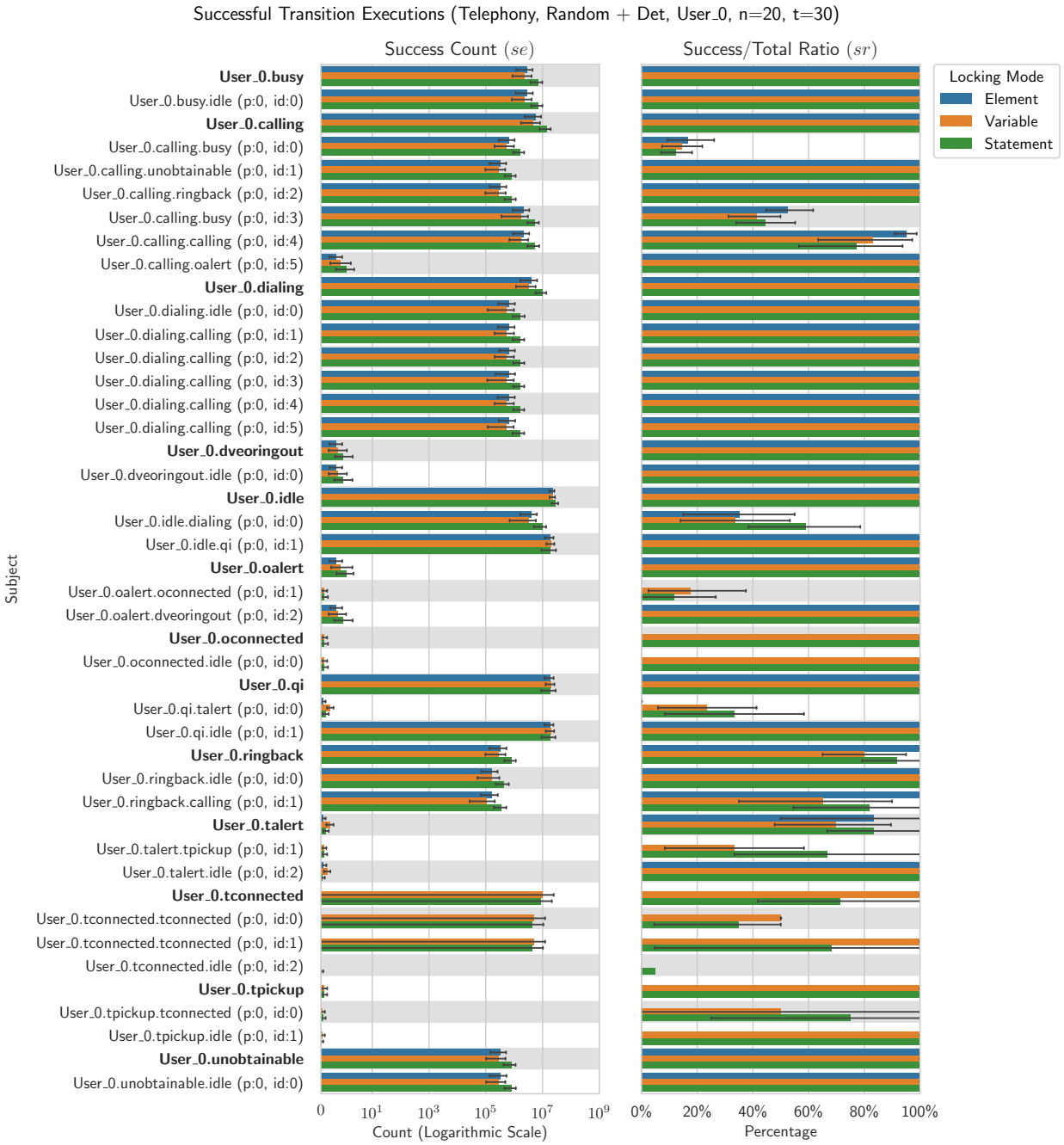


Figure B.21: A bar plot that reports the number of successful executions (se) and the percentage of successful executions (sr) for each decision node and transition in the target model `Telephony`, where the results are grouped by the locking mode. Observe that the success count is depicted in a logarithmic scale, due to the wide range of measured values: the first column adheres to a linear scale; all subsequent columns are logarithmic. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

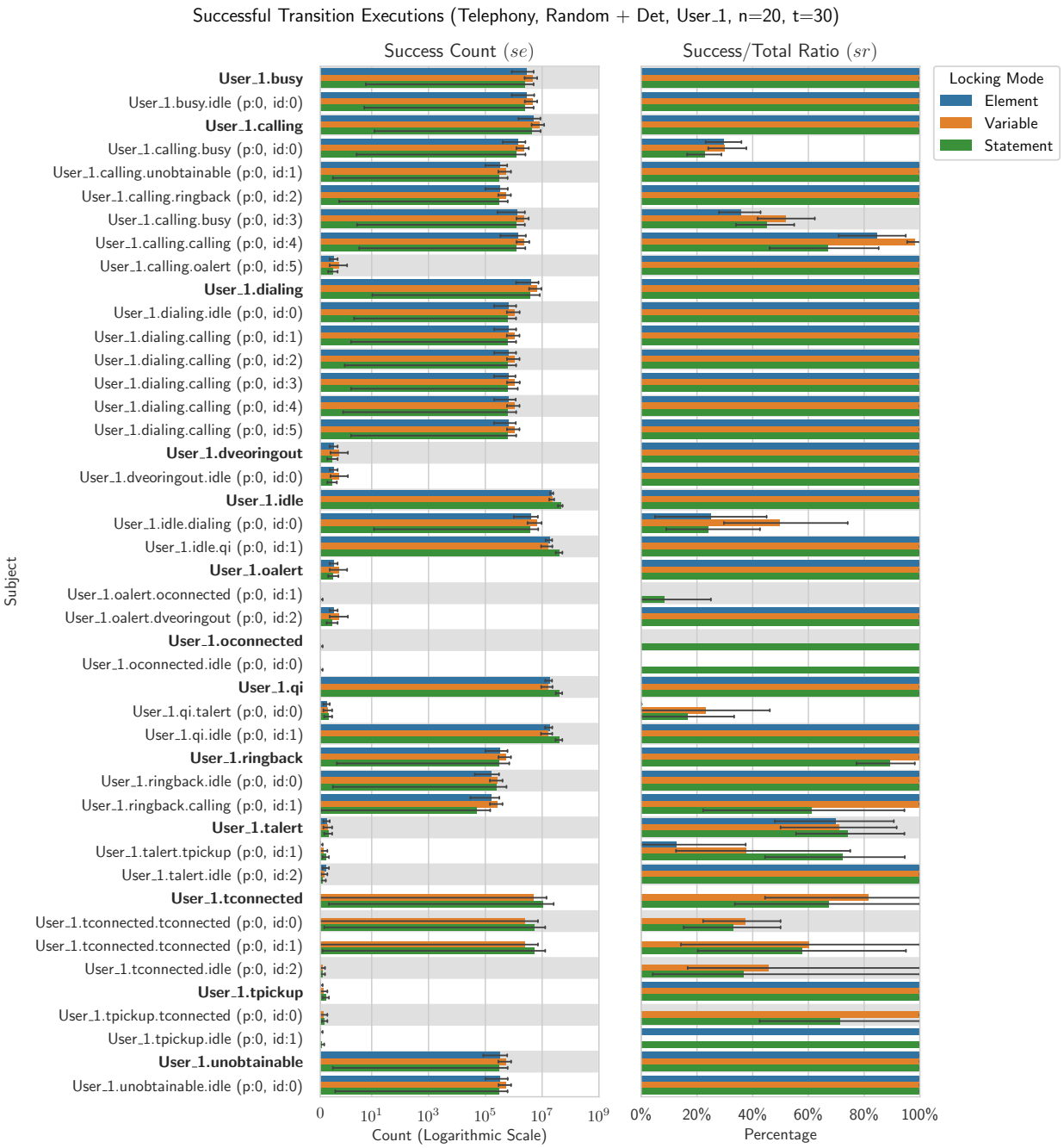


Figure B.22: A bar plot that reports the number of successful executions (se) and the percentage of successful executions (sr) for each decision node and transition in the target model `Telephony`, where the results are grouped by the locking mode. Observe that the success count is depicted in a logarithmic scale, due to the wide range of measured values: the first column adheres to a linear scale; all subsequent columns are logarithmic. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

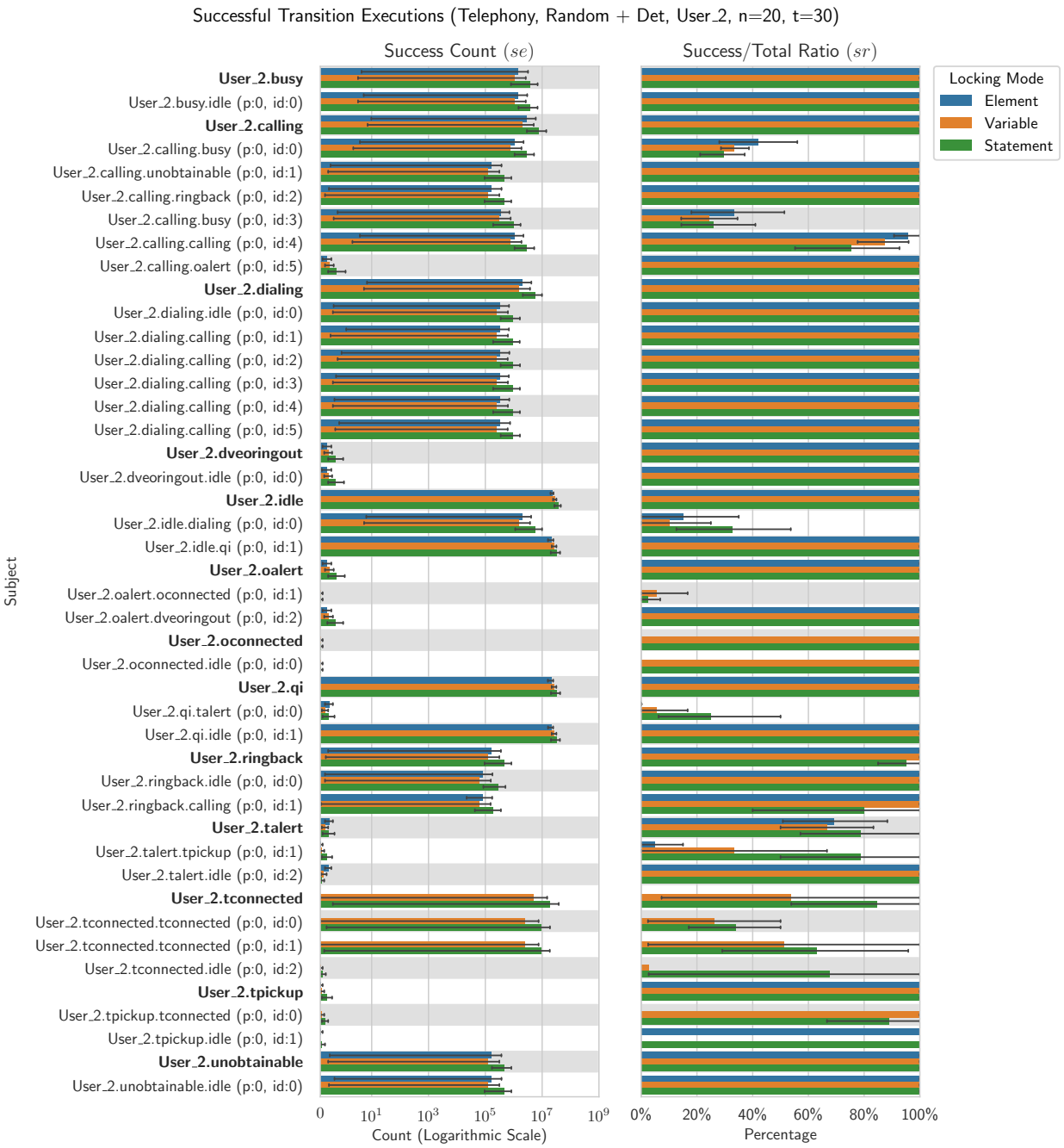


Figure B.23: A bar plot that reports the number of successful executions (se) and the percentage of successful executions (sr) for each decision node and transition in the target model `Telephony`, where the results are grouped by the locking mode. Observe that the success count is depicted in a logarithmic scale, due to the wide range of measured values: the first column adheres to a linear scale; all subsequent columns are logarithmic. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

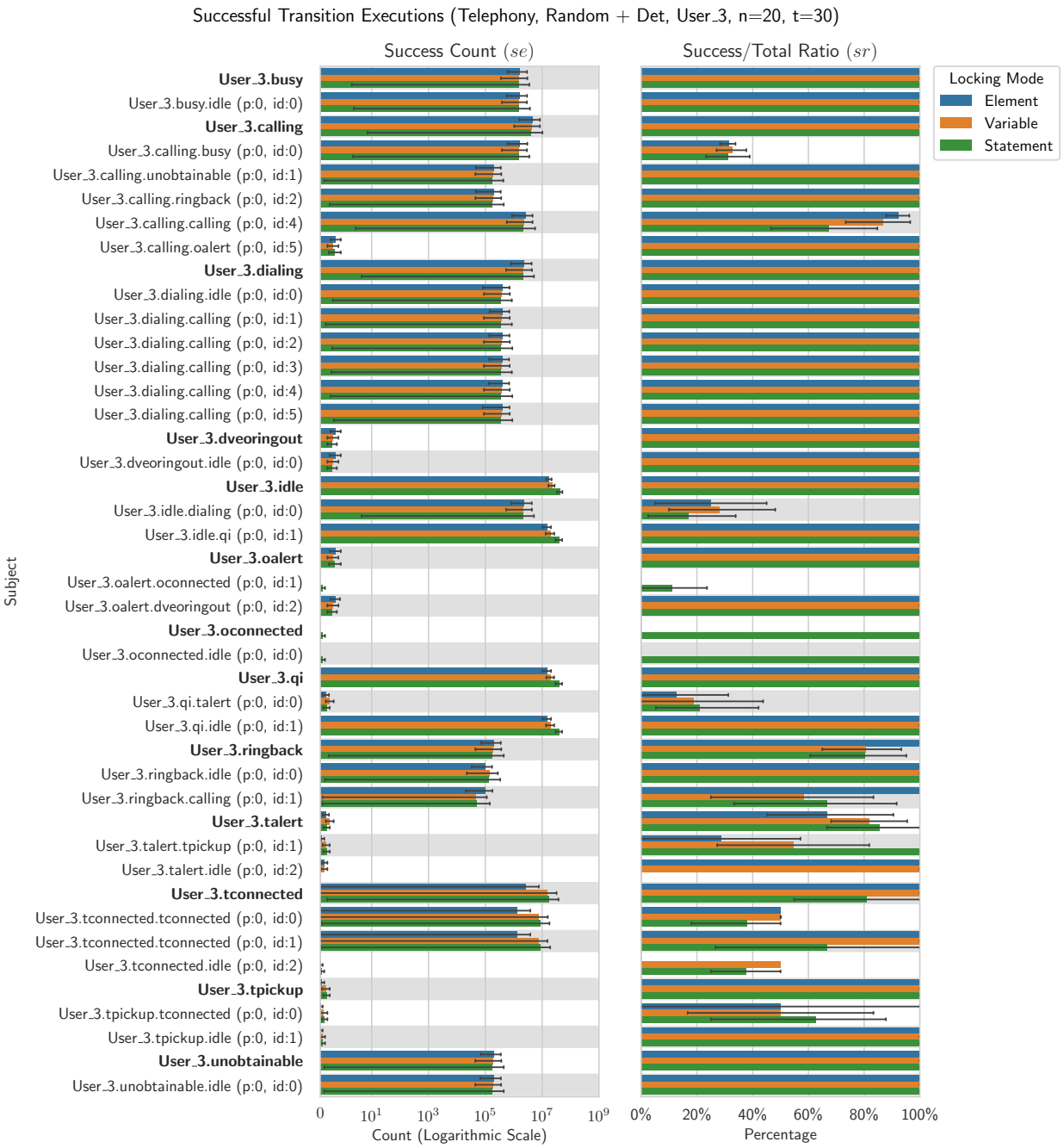


Figure B.24: A bar plot that reports the number of successful executions (se) and the percentage of successful executions (sr) for each decision node and transition in the target model `Telephony`, where the results are grouped by the locking mode. Observe that the success count is depicted in a logarithmic scale, due to the wide range of measured values: the first column adheres to a linear scale; all subsequent columns are logarithmic. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

Appendix C

Performance Analysis: Tables

C.1 Counting-Logging Frequency Tables

C.1.1 Synthetic Test: CounterDistributor

Performance results for target model '`CounterDistributor`' ($n = 20, t = 30$, Default)

Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
`Counter.C`	$1.19 \cdot 10^8$	$1.19 \cdot 10^7$	$1.19 \cdot 10^8$	$1.19 \cdot 10^7$	1.00	0.00
Counter.C.C (p:0, id:0)	$1.19 \cdot 10^8$	$1.19 \cdot 10^7$	$1.19 \cdot 10^8$	$1.19 \cdot 10^7$	1.00	0.00
`Distributor.P`	$1.07 \cdot 10^8$	$1.24 \cdot 10^7$	$1.07 \cdot 10^8$	$1.24 \cdot 10^7$	1.00	0.00
Distributor.P.P (p:0, id:0)	$1.07 \cdot 10^8$	$1.24 \cdot 10^7$	$1.06 \cdot 10^7$	$1.26 \cdot 10^6$	$9.90 \cdot 10^{-2}$	$2.23 \cdot 10^{-3}$
Distributor.P.P (p:0, id:1)	$9.68 \cdot 10^7$	$1.12 \cdot 10^7$	$1.07 \cdot 10^7$	$1.19 \cdot 10^6$	$1.10 \cdot 10^{-1}$	$1.87 \cdot 10^{-3}$
Distributor.P.P (p:0, id:2)	$8.61 \cdot 10^7$	$1.00 \cdot 10^7$	$1.07 \cdot 10^7$	$1.24 \cdot 10^6$	$1.24 \cdot 10^{-1}$	$2.73 \cdot 10^{-3}$
Distributor.P.P (p:0, id:3)	$7.55 \cdot 10^7$	$8.81 \cdot 10^6$	$1.09 \cdot 10^7$	$1.26 \cdot 10^6$	$1.44 \cdot 10^{-1}$	$4.35 \cdot 10^{-3}$
Distributor.P.P (p:0, id:4)	$6.46 \cdot 10^7$	$7.59 \cdot 10^6$	$1.08 \cdot 10^7$	$1.22 \cdot 10^6$	$1.67 \cdot 10^{-1}$	$3.03 \cdot 10^{-3}$
Distributor.P.P (p:0, id:5)	$5.38 \cdot 10^7$	$6.39 \cdot 10^6$	$1.08 \cdot 10^7$	$1.22 \cdot 10^6$	$2.00 \cdot 10^{-1}$	$4.76 \cdot 10^{-3}$
Distributor.P.P (p:0, id:6)	$4.30 \cdot 10^7$	$5.20 \cdot 10^6$	$1.08 \cdot 10^7$	$1.26 \cdot 10^6$	$2.51 \cdot 10^{-1}$	$1.10 \cdot 10^{-2}$
Distributor.P.P (p:0, id:7)	$3.22 \cdot 10^7$	$4.04 \cdot 10^6$	$1.07 \cdot 10^7$	$1.41 \cdot 10^6$	$3.32 \cdot 10^{-1}$	$6.04 \cdot 10^{-3}$
Distributor.P.P (p:0, id:8)	$2.15 \cdot 10^7$	$2.65 \cdot 10^6$	$1.08 \cdot 10^7$	$1.35 \cdot 10^6$	$5.00 \cdot 10^{-1}$	$5.31 \cdot 10^{-3}$
Distributor.P.P (p:0, id:9)	$1.07 \cdot 10^7$	$1.30 \cdot 10^6$	$1.07 \cdot 10^7$	$1.30 \cdot 10^6$	1.00	0.00

Table C.1: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model `CounterDistributor`. The Java code has been generated using the default settings. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

Performance results for target model '`CounterDistributor`' ($n = 20, t = 30$, Logging)

Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
`Counter.C`	$2.35 \cdot 10^6$	$1.49 \cdot 10^5$	$2.35 \cdot 10^6$	$1.49 \cdot 10^5$	1.00	$9.86 \cdot 10^{-8}$
Counter.C.C (p:0, id:0)	$2.35 \cdot 10^6$	$1.49 \cdot 10^5$	$2.35 \cdot 10^6$	$1.49 \cdot 10^5$	1.00	$1.78 \cdot 10^{-7}$
`Distributor.P`	$6.80 \cdot 10^6$	$1.88 \cdot 10^5$	$6.80 \cdot 10^6$	$1.88 \cdot 10^5$	1.00	$4.53 \cdot 10^{-8}$
Distributor.P.P (p:0, id:0)	$6.80 \cdot 10^6$	$1.88 \cdot 10^5$	$2.53 \cdot 10^5$	$2.34 \cdot 10^4$	$3.72 \cdot 10^{-2}$	$2.99 \cdot 10^{-3}$
Distributor.P.P (p:0, id:1)	$6.55 \cdot 10^6$	$1.75 \cdot 10^5$	$2.39 \cdot 10^5$	$2.75 \cdot 10^4$	$3.64 \cdot 10^{-2}$	$3.65 \cdot 10^{-3}$
Distributor.P.P (p:0, id:2)	$6.31 \cdot 10^6$	$1.55 \cdot 10^5$	$3.07 \cdot 10^5$	$4.37 \cdot 10^4$	$4.86 \cdot 10^{-2}$	$6.47 \cdot 10^{-3}$
Distributor.P.P (p:0, id:3)	$6.00 \cdot 10^6$	$1.40 \cdot 10^5$	$3.57 \cdot 10^5$	$3.51 \cdot 10^4$	$5.95 \cdot 10^{-2}$	$5.62 \cdot 10^{-3}$
Distributor.P.P (p:0, id:4)	$5.65 \cdot 10^6$	$1.33 \cdot 10^5$	$4.86 \cdot 10^5$	$4.71 \cdot 10^4$	$8.60 \cdot 10^{-2}$	$7.85 \cdot 10^{-3}$
Distributor.P.P (p:0, id:5)	$5.16 \cdot 10^6$	$1.26 \cdot 10^5$	$6.22 \cdot 10^5$	$2.51 \cdot 10^4$	$1.20 \cdot 10^{-1}$	$5.25 \cdot 10^{-3}$
Distributor.P.P (p:0, id:6)	$4.54 \cdot 10^6$	$1.23 \cdot 10^5$	$8.42 \cdot 10^5$	$4.73 \cdot 10^4$	$1.86 \cdot 10^{-1}$	$1.26 \cdot 10^{-2}$
Distributor.P.P (p:0, id:7)	$3.70 \cdot 10^6$	$1.40 \cdot 10^5$	$1.09 \cdot 10^6$	$3.94 \cdot 10^4$	$2.95 \cdot 10^{-1}$	$1.11 \cdot 10^{-2}$
Distributor.P.P (p:0, id:8)	$2.61 \cdot 10^6$	$1.24 \cdot 10^5$	$1.26 \cdot 10^6$	$7.25 \cdot 10^4$	$4.82 \cdot 10^{-1}$	$8.51 \cdot 10^{-3}$
Distributor.P.P (p:0, id:9)	$1.35 \cdot 10^6$	$5.77 \cdot 10^4$	$1.35 \cdot 10^6$	$5.77 \cdot 10^4$	1.00	0.00

Table C.2: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model `CounterDistributor`. The Java code has been generated using the default settings and the results have been attained through logging-based measurements. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

C.1.2 Synthetic Test: Tokens

Performance results for target model '`Tokens`' ($n = 20, t = 30$, Default)						
Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
`A.act`	$4.55 \cdot 10^7$	$6.79 \cdot 10^6$	$4.55 \cdot 10^7$	$6.79 \cdot 10^6$	1.00	0.00
A.act.act (p:0, id:0)	$4.55 \cdot 10^7$	$6.79 \cdot 10^6$	$3.20 \cdot 10^6$	$5.81 \cdot 10^5$	$7.02 \cdot 10^{-2}$	$6.53 \cdot 10^{-3}$
A.act.update (p:1, id:1)	$4.23 \cdot 10^7$	$6.30 \cdot 10^6$	$3.81 \cdot 10^7$	$5.66 \cdot 10^6$	$9.00 \cdot 10^{-1}$	$3.42 \cdot 10^{-8}$
A.act.wait (p:1, id:2)	$4.24 \cdot 10^6$	$6.31 \cdot 10^5$	$4.24 \cdot 10^6$	$6.31 \cdot 10^5$	1.00	0.00
`A.update`	$4.23 \cdot 10^7$	$6.30 \cdot 10^6$	$4.23 \cdot 10^7$	$6.30 \cdot 10^6$	1.00	0.00
A.update.act (p:0, id:0)	$4.23 \cdot 10^7$	$6.30 \cdot 10^6$	$4.23 \cdot 10^7$	$6.30 \cdot 10^6$	1.00	0.00
`A.wait`	$1.91 \cdot 10^7$	$2.58 \cdot 10^6$	$5.17 \cdot 10^6$	$6.59 \cdot 10^5$	$2.75 \cdot 10^{-1}$	$4.87 \cdot 10^{-2}$
A.wait.wait (p:0, id:0)	$1.91 \cdot 10^7$	$2.58 \cdot 10^6$	$9.32 \cdot 10^5$	$1.43 \cdot 10^5$	$4.92 \cdot 10^{-2}$	$7.25 \cdot 10^{-3}$
A.wait.update (p:1, id:1)	$1.82 \cdot 10^7$	$2.51 \cdot 10^6$	$4.24 \cdot 10^6$	$6.31 \cdot 10^5$	$2.38 \cdot 10^{-1}$	$4.87 \cdot 10^{-2}$
`B.act`	$6.46 \cdot 10^7$	$6.43 \cdot 10^6$	$6.46 \cdot 10^7$	$6.43 \cdot 10^6$	1.00	0.00
B.act.act (p:0, id:0)	$6.46 \cdot 10^7$	$6.43 \cdot 10^6$	$3.46 \cdot 10^6$	$6.70 \cdot 10^5$	$5.35 \cdot 10^{-2}$	$7.87 \cdot 10^{-3}$
B.act.update (p:1, id:1)	$6.12 \cdot 10^7$	$6.04 \cdot 10^6$	$5.51 \cdot 10^7$	$5.43 \cdot 10^6$	$9.00 \cdot 10^{-1}$	$4.24 \cdot 10^{-8}$
B.act.wait (p:1, id:2)	$6.13 \cdot 10^6$	$6.05 \cdot 10^5$	$6.13 \cdot 10^6$	$6.05 \cdot 10^5$	1.00	0.00
`B.update`	$6.12 \cdot 10^7$	$6.04 \cdot 10^6$	$6.12 \cdot 10^7$	$6.04 \cdot 10^6$	1.00	0.00
B.update.act (p:0, id:0)	$6.12 \cdot 10^7$	$6.04 \cdot 10^6$	$6.12 \cdot 10^7$	$6.04 \cdot 10^6$	1.00	0.00
`B.wait`	$2.61 \cdot 10^7$	$5.17 \cdot 10^6$	$6.91 \cdot 10^6$	$6.00 \cdot 10^5$	$2.73 \cdot 10^{-1}$	$5.28 \cdot 10^{-2}$
B.wait.wait (p:0, id:0)	$2.61 \cdot 10^7$	$5.17 \cdot 10^6$	$7.80 \cdot 10^5$	$1.12 \cdot 10^5$	$3.07 \cdot 10^{-2}$	$6.03 \cdot 10^{-3}$
B.wait.update (p:1, id:1)	$2.53 \cdot 10^7$	$5.15 \cdot 10^6$	$6.13 \cdot 10^6$	$6.05 \cdot 10^5$	$2.50 \cdot 10^{-1}$	$5.16 \cdot 10^{-2}$
`C.act`	$4.57 \cdot 10^7$	$6.55 \cdot 10^6$	$4.57 \cdot 10^7$	$6.55 \cdot 10^6$	1.00	0.00
C.act.act (p:0, id:0)	$4.57 \cdot 10^7$	$6.55 \cdot 10^6$	$4.42 \cdot 10^6$	$4.73 \cdot 10^5$	$9.75 \cdot 10^{-2}$	$8.61 \cdot 10^{-3}$
C.act.wait (p:1, id:1)	$4.12 \cdot 10^7$	$6.18 \cdot 10^6$	$4.13 \cdot 10^6$	$6.19 \cdot 10^5$	$1.00 \cdot 10^{-1}$	$9.38 \cdot 10^{-8}$
C.act.update (p:1, id:2)	$3.71 \cdot 10^7$	$5.56 \cdot 10^6$	$3.71 \cdot 10^7$	$5.56 \cdot 10^6$	1.00	0.00
`C.update`	$4.12 \cdot 10^7$	$6.18 \cdot 10^6$	$4.12 \cdot 10^7$	$6.18 \cdot 10^6$	1.00	0.00
C.update.act (p:0, id:0)	$4.12 \cdot 10^7$	$6.18 \cdot 10^6$	$4.12 \cdot 10^7$	$6.18 \cdot 10^6$	1.00	0.00
`C.wait`	$1.85 \cdot 10^7$	$2.64 \cdot 10^6$	$5.84 \cdot 10^6$	$7.54 \cdot 10^5$	$3.20 \cdot 10^{-1}$	$4.43 \cdot 10^{-2}$
C.wait.wait (p:0, id:0)	$1.85 \cdot 10^7$	$2.64 \cdot 10^6$	$1.71 \cdot 10^6$	$2.32 \cdot 10^5$	$9.30 \cdot 10^{-2}$	$7.54 \cdot 10^{-3}$
C.wait.update (p:1, id:1)	$1.68 \cdot 10^7$	$2.45 \cdot 10^6$	$4.13 \cdot 10^6$	$6.19 \cdot 10^5$	$2.50 \cdot 10^{-1}$	$4.49 \cdot 10^{-2}$

Table C.3: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model `Tokens`. The Java code has been generated using the default settings. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

Performance results for target model '`Tokens`' ($n = 20, t = 30$, Logging)						
Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
`A.act`	$3.21 \cdot 10^6$	$1.56 \cdot 10^5$	$3.21 \cdot 10^6$	$1.56 \cdot 10^5$	1.00	$1.52 \cdot 10^{-7}$
A.act.act (p:0, id:0)	$3.21 \cdot 10^6$	$1.56 \cdot 10^5$	$2.55 \cdot 10^5$	$1.10 \cdot 10^4$	$7.94 \cdot 10^{-2}$	$1.80 \cdot 10^{-3}$
A.act.update (p:1, id:1)	$2.96 \cdot 10^6$	$1.47 \cdot 10^5$	$2.66 \cdot 10^6$	$1.32 \cdot 10^5$	$9.00 \cdot 10^{-1}$	$9.88 \cdot 10^{-7}$
A.act.wait (p:1, id:2)	$2.96 \cdot 10^5$	$1.47 \cdot 10^4$	$2.96 \cdot 10^5$	$1.47 \cdot 10^4$	1.00	0.00
`A.update`	$2.96 \cdot 10^6$	$1.47 \cdot 10^5$	$2.96 \cdot 10^6$	$1.47 \cdot 10^5$	1.00	$1.39 \cdot 10^{-7}$
A.update.act (p:0, id:0)	$2.96 \cdot 10^6$	$1.47 \cdot 10^5$	$2.96 \cdot 10^6$	$1.47 \cdot 10^5$	1.00	$7.28 \cdot 10^{-8}$
`A.wait`	$4.97 \cdot 10^5$	$2.76 \cdot 10^4$	$3.17 \cdot 10^5$	$1.56 \cdot 10^4$	$6.38 \cdot 10^{-1}$	$1.89 \cdot 10^{-2}$
A.wait.wait (p:0, id:0)	$4.97 \cdot 10^5$	$2.76 \cdot 10^4$	$2.04 \cdot 10^4$	$1.05 \cdot 10^3$	$4.11 \cdot 10^{-2}$	$1.82 \cdot 10^{-3}$
A.wait.update (p:1, id:1)	$4.76 \cdot 10^5$	$2.69 \cdot 10^4$	$2.96 \cdot 10^5$	$1.47 \cdot 10^4$	$6.22 \cdot 10^{-1}$	$1.91 \cdot 10^{-2}$
`B.act`	$3.48 \cdot 10^6$	$1.39 \cdot 10^5$	$3.48 \cdot 10^6$	$1.39 \cdot 10^5$	1.00	$1.34 \cdot 10^{-7}$
B.act.act (p:0, id:0)	$3.48 \cdot 10^6$	$1.39 \cdot 10^5$	$2.68 \cdot 10^5$	$1.33 \cdot 10^4$	$7.71 \cdot 10^{-2}$	$1.57 \cdot 10^{-3}$
B.act.update (p:1, id:1)	$3.21 \cdot 10^6$	$1.27 \cdot 10^5$	$2.89 \cdot 10^6$	$1.14 \cdot 10^5$	$9.00 \cdot 10^{-1}$	$7.15 \cdot 10^{-7}$
B.act.wait (p:1, id:2)	$3.21 \cdot 10^5$	$1.27 \cdot 10^4$	$3.21 \cdot 10^5$	$1.27 \cdot 10^4$	1.00	0.00
`B.update`	$3.21 \cdot 10^6$	$1.27 \cdot 10^5$	$3.21 \cdot 10^6$	$1.27 \cdot 10^5$	1.00	$1.16 \cdot 10^{-7}$
B.update.act (p:0, id:0)	$3.21 \cdot 10^6$	$1.27 \cdot 10^5$	$3.21 \cdot 10^6$	$1.27 \cdot 10^5$	1.00	$7.38 \cdot 10^{-8}$
`B.wait`	$5.72 \cdot 10^5$	$2.76 \cdot 10^4$	$3.50 \cdot 10^5$	$1.41 \cdot 10^4$	$6.12 \cdot 10^{-1}$	$1.90 \cdot 10^{-2}$
B.wait.wait (p:0, id:0)	$5.72 \cdot 10^5$	$2.76 \cdot 10^4$	$2.84 \cdot 10^4$	$1.49 \cdot 10^3$	$4.97 \cdot 10^{-2}$	$1.53 \cdot 10^{-3}$
B.wait.update (p:1, id:1)	$5.43 \cdot 10^5$	$2.64 \cdot 10^4$	$3.21 \cdot 10^5$	$1.27 \cdot 10^4$	$5.92 \cdot 10^{-1}$	$1.95 \cdot 10^{-2}$
`C.act`	$3.05 \cdot 10^6$	$1.30 \cdot 10^5$	$3.05 \cdot 10^6$	$1.30 \cdot 10^5$	1.00	$1.67 \cdot 10^{-7}$
C.act.act (p:0, id:0)	$3.05 \cdot 10^6$	$1.30 \cdot 10^5$	$2.98 \cdot 10^5$	$1.18 \cdot 10^4$	$9.80 \cdot 10^{-2}$	$1.98 \cdot 10^{-3}$
C.act.wait (p:1, id:1)	$2.75 \cdot 10^6$	$1.20 \cdot 10^5$	$2.75 \cdot 10^5$	$1.20 \cdot 10^4$	$1.00 \cdot 10^{-1}$	$1.05 \cdot 10^{-6}$
C.act.update (p:1, id:2)	$2.47 \cdot 10^6$	$1.08 \cdot 10^5$	$2.47 \cdot 10^6$	$1.08 \cdot 10^5$	1.00	$1.26 \cdot 10^{-7}$
`C.update`	$2.75 \cdot 10^6$	$1.20 \cdot 10^5$	$2.75 \cdot 10^6$	$1.20 \cdot 10^5$	1.00	$1.52 \cdot 10^{-7}$
C.update.act (p:0, id:0)	$2.75 \cdot 10^6$	$1.20 \cdot 10^5$	$2.75 \cdot 10^6$	$1.20 \cdot 10^5$	1.00	$1.17 \cdot 10^{-7}$
`C.wait`	$5.13 \cdot 10^5$	$2.59 \cdot 10^4$	$2.98 \cdot 10^5$	$1.29 \cdot 10^4$	$5.82 \cdot 10^{-1}$	$2.03 \cdot 10^{-2}$
C.wait.wait (p:0, id:0)	$5.13 \cdot 10^5$	$2.59 \cdot 10^4$	$2.30 \cdot 10^4$	$1.06 \cdot 10^3$	$4.49 \cdot 10^{-2}$	$1.79 \cdot 10^{-3}$
C.wait.update (p:1, id:1)	$4.90 \cdot 10^5$	$2.52 \cdot 10^4$	$2.75 \cdot 10^5$	$1.20 \cdot 10^4$	$5.62 \cdot 10^{-1}$	$2.06 \cdot 10^{-2}$

Table C.4: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model `Tokens`. The Java code has been generated using the default settings and the results have been attained through logging-based measurements. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

C.1.3 Practical Test: Elevator

Performance results for target model 'Elevator' ($n = 20, t = 30$, Default)						
Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
`cabin.idle`	$1.77 \cdot 10^8$	$3.69 \cdot 10^7$	$6.47 \cdot 10^6$	$1.09 \cdot 10^6$	$3.70 \cdot 10^{-2}$	$4.12 \cdot 10^{-3}$
cabin.idle.mov (p:0, id:0)	$1.77 \cdot 10^8$	$3.69 \cdot 10^7$	$6.47 \cdot 10^6$	$1.09 \cdot 10^6$	$3.70 \cdot 10^{-2}$	$4.12 \cdot 10^{-3}$
`cabin.mov`	$1.13 \cdot 10^7$	$1.92 \cdot 10^6$	$1.13 \cdot 10^7$	$1.92 \cdot 10^6$	1.00	0.00
cabin.mov.open (p:0, id:0)	$1.13 \cdot 10^7$	$1.92 \cdot 10^6$	$6.47 \cdot 10^6$	$1.09 \cdot 10^6$	$5.71 \cdot 10^{-1}$	$4.01 \cdot 10^{-4}$
cabin.mov.mov (p:0, id:1)	$4.87 \cdot 10^6$	$8.21 \cdot 10^5$	$2.43 \cdot 10^6$	$4.11 \cdot 10^5$	$5.00 \cdot 10^{-1}$	$1.06 \cdot 10^{-7}$
cabin.mov.mov (p:0, id:2)	$2.43 \cdot 10^6$	$4.11 \cdot 10^5$	$2.43 \cdot 10^6$	$4.11 \cdot 10^5$	1.00	0.00
`cabin.open`	$6.47 \cdot 10^6$	$1.09 \cdot 10^6$	$6.47 \cdot 10^6$	$1.09 \cdot 10^6$	1.00	0.00
cabin.open.idle (p:0, id:0)	$6.47 \cdot 10^6$	$1.09 \cdot 10^6$	$6.47 \cdot 10^6$	$1.09 \cdot 10^6$	1.00	0.00
`controller.done`	$6.47 \cdot 10^6$	$1.09 \cdot 10^6$	$6.47 \cdot 10^6$	$1.09 \cdot 10^6$	1.00	0.00
controller.done.wait (p:0, id:0)	$6.47 \cdot 10^6$	$1.09 \cdot 10^6$	$6.47 \cdot 10^6$	$1.09 \cdot 10^6$	1.00	0.00
`controller.wait`	$4.78 \cdot 10^7$	$8.59 \cdot 10^6$	$8.14 \cdot 10^6$	$1.38 \cdot 10^6$	$1.72 \cdot 10^{-1}$	$1.69 \cdot 10^{-2}$
controller.wait.work (p:0, id:0)	$4.78 \cdot 10^7$	$8.59 \cdot 10^6$	$8.14 \cdot 10^6$	$1.38 \cdot 10^6$	$1.72 \cdot 10^{-1}$	$1.69 \cdot 10^{-2}$
`controller.work`	$8.38 \cdot 10^6$	$1.42 \cdot 10^6$	$8.38 \cdot 10^6$	$1.42 \cdot 10^6$	1.00	0.00
controller.work.wait (p:0, id:0)	$8.38 \cdot 10^6$	$1.42 \cdot 10^6$	$1.68 \cdot 10^6$	$2.84 \cdot 10^5$	$2.00 \cdot 10^{-1}$	$2.63 \cdot 10^{-8}$
controller.work.done (p:0, id:1)	$6.70 \cdot 10^6$	$1.14 \cdot 10^6$	$6.47 \cdot 10^6$	$1.09 \cdot 10^6$	$9.65 \cdot 10^{-1}$	$1.99 \cdot 10^{-2}$
controller.work.work (p:0, id:2)	$2.34 \cdot 10^5$	$1.45 \cdot 10^5$	$2.34 \cdot 10^5$	$1.45 \cdot 10^5$	1.00	0.00
`environment.read`	$1.30 \cdot 10^8$	$2.49 \cdot 10^7$	$6.47 \cdot 10^6$	$1.09 \cdot 10^6$	$5.03 \cdot 10^{-2}$	$7.92 \cdot 10^{-3}$
environment.read.read (p:0, id:0)	$1.30 \cdot 10^8$	$2.49 \cdot 10^7$	$1.62 \cdot 10^6$	$2.74 \cdot 10^5$	$1.26 \cdot 10^{-2}$	$1.96 \cdot 10^{-3}$
environment.read.read (p:0, id:1)	$1.29 \cdot 10^8$	$2.48 \cdot 10^7$	$1.62 \cdot 10^6$	$2.74 \cdot 10^5$	$1.27 \cdot 10^{-2}$	$2.05 \cdot 10^{-3}$
environment.read.read (p:0, id:2)	$1.27 \cdot 10^8$	$2.46 \cdot 10^7$	$1.62 \cdot 10^6$	$2.74 \cdot 10^5$	$1.29 \cdot 10^{-2}$	$2.11 \cdot 10^{-3}$
environment.read.read (p:0, id:3)	$1.26 \cdot 10^8$	$2.44 \cdot 10^7$	$1.62 \cdot 10^6$	$2.73 \cdot 10^5$	$1.31 \cdot 10^{-2}$	$2.16 \cdot 10^{-3}$

Table C.5: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model `Elevator`. The Java code has been generated using the default settings. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

Performance results for target model 'Elevator' ($n = 20, t = 30$, Logging)						
Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
`cabin.idle`	$7.01 \cdot 10^6$	$3.54 \cdot 10^5$	$1.17 \cdot 10^6$	$5.48 \cdot 10^4$	$1.67 \cdot 10^{-1}$	$5.15 \cdot 10^{-3}$
cabin.idle.mov (p:0, id:0)	$7.01 \cdot 10^6$	$3.54 \cdot 10^5$	$1.17 \cdot 10^6$	$5.48 \cdot 10^4$	$1.67 \cdot 10^{-1}$	$5.15 \cdot 10^{-3}$
`cabin.mov`	$2.05 \cdot 10^6$	$9.60 \cdot 10^4$	$2.05 \cdot 10^6$	$9.60 \cdot 10^4$	1.00	$2.20 \cdot 10^{-7}$
cabin.mov.open (p:0, id:0)	$2.05 \cdot 10^6$	$9.60 \cdot 10^4$	$1.17 \cdot 10^6$	$5.48 \cdot 10^4$	$5.71 \cdot 10^{-1}$	$5.23 \cdot 10^{-5}$
cabin.mov.mov (p:0, id:1)	$8.80 \cdot 10^5$	$4.11 \cdot 10^4$	$4.40 \cdot 10^5$	$2.06 \cdot 10^4$	$5.00 \cdot 10^{-1}$	$8.41 \cdot 10^{-7}$
cabin.mov.mov (p:0, id:2)	$4.40 \cdot 10^5$	$2.06 \cdot 10^4$	$4.40 \cdot 10^5$	$2.06 \cdot 10^4$	1.00	$5.24 \cdot 10^{-7}$
`cabin.open`	$1.17 \cdot 10^6$	$5.48 \cdot 10^4$	$1.17 \cdot 10^6$	$5.48 \cdot 10^4$	1.00	$2.59 \cdot 10^{-7}$
cabin.open.idle (p:0, id:0)	$1.17 \cdot 10^6$	$5.48 \cdot 10^4$	$1.17 \cdot 10^6$	$5.48 \cdot 10^4$	1.00	0.00
`controller.done`	$1.17 \cdot 10^6$	$5.48 \cdot 10^4$	$1.17 \cdot 10^6$	$5.48 \cdot 10^4$	1.00	$1.95 \cdot 10^{-7}$
controller.done.wait (p:0, id:0)	$1.17 \cdot 10^6$	$5.48 \cdot 10^4$	$1.17 \cdot 10^6$	$5.48 \cdot 10^4$	1.00	$1.95 \cdot 10^{-7}$
`controller.wait`	$7.73 \cdot 10^6$	$4.77 \cdot 10^5$	$1.47 \cdot 10^6$	$6.86 \cdot 10^4$	$1.90 \cdot 10^{-1}$	$6.08 \cdot 10^{-3}$
controller.wait.work (p:0, id:0)	$7.73 \cdot 10^6$	$4.77 \cdot 10^5$	$1.47 \cdot 10^6$	$6.86 \cdot 10^4$	$1.90 \cdot 10^{-1}$	$6.08 \cdot 10^{-3}$
`controller.work`	$1.47 \cdot 10^6$	$6.89 \cdot 10^4$	$1.47 \cdot 10^6$	$6.89 \cdot 10^4$	1.00	$3.06 \cdot 10^{-7}$
controller.work.wait (p:0, id:0)	$1.47 \cdot 10^6$	$6.89 \cdot 10^4$	$2.93 \cdot 10^5$	$1.38 \cdot 10^4$	$2.00 \cdot 10^{-1}$	$2.41 \cdot 10^{-7}$
controller.work.done (p:0, id:1)	$1.17 \cdot 10^6$	$5.52 \cdot 10^4$	$1.17 \cdot 10^6$	$5.48 \cdot 10^4$	$9.99 \cdot 10^{-1}$	$4.90 \cdot 10^{-4}$
controller.work.work (p:0, id:2)	$1.27 \cdot 10^3$	$6.11 \cdot 10^2$	$1.27 \cdot 10^3$	$6.11 \cdot 10^2$	1.00	0.00
`environment.read`	$6.10 \cdot 10^6$	$2.87 \cdot 10^5$	$1.17 \cdot 10^6$	$5.48 \cdot 10^4$	$1.92 \cdot 10^{-1}$	$5.34 \cdot 10^{-3}$
environment.read.read (p:0, id:0)	$6.10 \cdot 10^6$	$2.87 \cdot 10^5$	$2.93 \cdot 10^5$	$1.37 \cdot 10^4$	$4.81 \cdot 10^{-2}$	$1.34 \cdot 10^{-3}$
environment.read.read (p:0, id:1)	$5.80 \cdot 10^6$	$2.76 \cdot 10^5$	$2.93 \cdot 10^5$	$1.37 \cdot 10^4$	$5.05 \cdot 10^{-2}$	$1.48 \cdot 10^{-3}$
environment.read.read (p:0, id:2)	$5.51 \cdot 10^6$	$2.65 \cdot 10^5$	$2.93 \cdot 10^5$	$1.37 \cdot 10^4$	$5.32 \cdot 10^{-2}$	$1.64 \cdot 10^{-3}$
environment.read.read (p:0, id:3)	$5.22 \cdot 10^6$	$2.54 \cdot 10^5$	$2.93 \cdot 10^5$	$1.37 \cdot 10^4$	$5.62 \cdot 10^{-2}$	$1.83 \cdot 10^{-3}$

Table C.6: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model `Elevator`. The Java code has been generated using the default settings and the results have been attained through logging-based measurements. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

C.1.4 Practical Test: ToadsAndFrogs

Performance results for target model '`ToadsAndFrogs`' ($n = 20, t = 30$, Default)						
Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
`control.done`	$3.09 \cdot 10^{6}$	$1.43 \cdot 10^{5}$	$3.09 \cdot 10^{6}$	$1.43 \cdot 10^{5}$	1.00	0.00
control.done.success (p:0, id:0)	$3.09 \cdot 10^{6}$	$1.43 \cdot 10^{5}$	$8.00 \cdot 10^{-1}$	$8.94 \cdot 10^{-1}$	$2.58 \cdot 10^{-7}$	$2.83 \cdot 10^{-7}$
control.done.failure (p:0, id:1)	$3.09 \cdot 10^{6}$	$1.43 \cdot 10^{5}$	$3.09 \cdot 10^{6}$	$1.43 \cdot 10^{5}$	1.00	0.00
`control.failure`	$3.09 \cdot 10^{6}$	$1.43 \cdot 10^{5}$	$3.09 \cdot 10^{6}$	$1.43 \cdot 10^{5}$	1.00	0.00
control.failure.reset (p:0, id:0)	$3.09 \cdot 10^{6}$	$1.43 \cdot 10^{5}$	$3.09 \cdot 10^{6}$	$1.43 \cdot 10^{5}$	1.00	0.00
`control.reset`	$3.09 \cdot 10^{6}$	$1.43 \cdot 10^{5}$	$3.09 \cdot 10^{6}$	$1.43 \cdot 10^{5}$	1.00	0.00
control.reset.running (p:0, id:0)	$3.09 \cdot 10^{6}$	$1.43 \cdot 10^{5}$	$3.09 \cdot 10^{6}$	$1.43 \cdot 10^{5}$	1.00	0.00
`control.running`	$1.86 \cdot 10^{7}$	$1.14 \cdot 10^{6}$	$3.09 \cdot 10^{6}$	$1.43 \cdot 10^{5}$	$1.67 \cdot 10^{-1}$	$1.40 \cdot 10^{-2}$
control.running.done (p:0, id:0)	$1.86 \cdot 10^{7}$	$1.14 \cdot 10^{6}$	$1.77 \cdot 10^{6}$	$2.24 \cdot 10^{5}$	$9.56 \cdot 10^{-2}$	$1.49 \cdot 10^{-2}$
control.running.done (p:0, id:1)	$1.68 \cdot 10^{7}$	$1.20 \cdot 10^{6}$	$4.03 \cdot 10^{1}$	$1.53 \cdot 10^{1}$	$2.39 \cdot 10^{-6}$	$8.86 \cdot 10^{-7}$
control.running.done (p:0, id:2)	$1.68 \cdot 10^{7}$	$1.20 \cdot 10^{6}$	$4.06 \cdot 10^{1}$	$1.31 \cdot 10^{1}$	$2.41 \cdot 10^{-6}$	$7.77 \cdot 10^{-7}$
control.running.done (p:0, id:3)	$1.68 \cdot 10^{7}$	$1.20 \cdot 10^{6}$	$1.32 \cdot 10^{6}$	$2.42 \cdot 10^{5}$	$7.84 \cdot 10^{-2}$	$1.48 \cdot 10^{-2}$
control.running.done (p:0, id:4)	$1.55 \cdot 10^{7}$	$1.19 \cdot 10^{6}$	6.25	2.71	$4.02 \cdot 10^{-7}$	$1.78 \cdot 10^{-7}$
`control.success`	$8.00 \cdot 10^{-1}$	$8.94 \cdot 10^{-1}$	$8.00 \cdot 10^{-1}$	$8.94 \cdot 10^{-1}$	1.00	0.00
control.success.reset (p:0, id:0)	$8.00 \cdot 10^{-1}$	$8.94 \cdot 10^{-1}$	$8.00 \cdot 10^{-1}$	$8.94 \cdot 10^{-1}$	1.00	0.00
`frog.q`	$2.29 \cdot 10^{7}$	$9.71 \cdot 10^{5}$	$5.30 \cdot 10^{6}$	$9.69 \cdot 10^{5}$	$2.32 \cdot 10^{-1}$	$4.36 \cdot 10^{-2}$
frog.q.q (p:0, id:0)	$2.29 \cdot 10^{7}$	$9.71 \cdot 10^{5}$	$3.95 \cdot 10^{6}$	$7.30 \cdot 10^{5}$	$1.73 \cdot 10^{-1}$	$3.28 \cdot 10^{-2}$
frog.q.q (p:0, id:1)	$1.89 \cdot 10^{7}$	$1.26 \cdot 10^{6}$	$1.32 \cdot 10^{6}$	$2.42 \cdot 10^{5}$	$7.04 \cdot 10^{-2}$	$1.56 \cdot 10^{-2}$
frog.q.q (p:0, id:2)	$1.76 \cdot 10^{7}$	$1.42 \cdot 10^{6}$	$3.30 \cdot 10^{4}$	$7.06 \cdot 10^{3}$	$1.88 \cdot 10^{-3}$	$4.06 \cdot 10^{-4}$
frog.q.q (p:0, id:3)	$1.76 \cdot 10^{7}$	$1.42 \cdot 10^{6}$	$1.08 \cdot 10^{1}$	4.71	$6.14 \cdot 10^{-7}$	$2.60 \cdot 10^{-7}$
`toad.q`	$2.18 \cdot 10^{7}$	$1.20 \cdot 10^{6}$	$7.11 \cdot 10^{6}$	$8.99 \cdot 10^{5}$	$3.27 \cdot 10^{-1}$	$4.21 \cdot 10^{-2}$
toad.q.q (p:0, id:0)	$2.18 \cdot 10^{7}$	$1.20 \cdot 10^{6}$	$5.31 \cdot 10^{6}$	$6.77 \cdot 10^{5}$	$2.44 \cdot 10^{-1}$	$3.17 \cdot 10^{-2}$
toad.q.q (p:0, id:1)	$1.65 \cdot 10^{7}$	$1.28 \cdot 10^{6}$	$1.77 \cdot 10^{6}$	$2.24 \cdot 10^{5}$	$1.08 \cdot 10^{-1}$	$1.85 \cdot 10^{-2}$
toad.q.q (p:0, id:2)	$1.47 \cdot 10^{7}$	$1.38 \cdot 10^{6}$	$2.97 \cdot 10^{4}$	$5.26 \cdot 10^{3}$	$2.02 \cdot 10^{-3}$	$3.27 \cdot 10^{-4}$
toad.q.q (p:0, id:3)	$1.47 \cdot 10^{7}$	$1.37 \cdot 10^{6}$	$1.31 \cdot 10^{1}$	5.97	$9.10 \cdot 10^{-7}$	$4.62 \cdot 10^{-7}$

Table C.7: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model `ToadsAndFrogs`. The Java code has been generated using the default settings. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

Performance results for target model '`ToadsAndFrogs`' ($n = 20, t = 30$, Logging)						
Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
`control.done`	$3.78 \cdot 10^5$	$2.34 \cdot 10^4$	$3.78 \cdot 10^5$	$2.34 \cdot 10^4$	1.00	$6.28 \cdot 10^{-7}$
control.done.success (p:0, id:0)	$3.78 \cdot 10^5$	$2.34 \cdot 10^4$	$3.66 \cdot 10^1$	$3.06 \cdot 10^1$	$1.01 \cdot 10^{-4}$	$9.19 \cdot 10^{-5}$
control.done.failure (p:0, id:1)	$3.78 \cdot 10^5$	$2.35 \cdot 10^4$	$3.78 \cdot 10^5$	$2.35 \cdot 10^4$	1.00	$6.28 \cdot 10^{-7}$
`control.failure`	$3.78 \cdot 10^5$	$2.35 \cdot 10^4$	$3.78 \cdot 10^5$	$2.35 \cdot 10^4$	1.00	$6.15 \cdot 10^{-7}$
control.failure.reset (p:0, id:0)	$3.78 \cdot 10^5$	$2.35 \cdot 10^4$	$3.78 \cdot 10^5$	$2.35 \cdot 10^4$	1.00	$6.15 \cdot 10^{-7}$
`control.reset`	$3.78 \cdot 10^5$	$2.34 \cdot 10^4$	$3.78 \cdot 10^5$	$2.34 \cdot 10^4$	1.00	$1.19 \cdot 10^{-6}$
control.reset.running (p:0, id:0)	$3.78 \cdot 10^5$	$2.34 \cdot 10^4$	$3.78 \cdot 10^5$	$2.34 \cdot 10^4$	1.00	$1.19 \cdot 10^{-6}$
`control.running`	$2.47 \cdot 10^6$	$2.09 \cdot 10^5$	$3.78 \cdot 10^5$	$2.34 \cdot 10^4$	$1.54 \cdot 10^{-1}$	$2.13 \cdot 10^{-2}$
control.running.done (p:0, id:0)	$2.47 \cdot 10^6$	$2.09 \cdot 10^5$	$9.86 \cdot 10^4$	$3.44 \cdot 10^4$	$4.12 \cdot 10^{-2}$	$1.81 \cdot 10^{-2}$
control.running.done (p:0, id:1)	$2.38 \cdot 10^6$	$2.40 \cdot 10^5$	$7.07 \cdot 10^4$	$1.89 \cdot 10^4$	$2.94 \cdot 10^{-2}$	$5.92 \cdot 10^{-3}$
control.running.done (p:0, id:2)	$2.31 \cdot 10^6$	$2.26 \cdot 10^5$	$7.62 \cdot 10^4$	$1.85 \cdot 10^4$	$3.28 \cdot 10^{-2}$	$6.44 \cdot 10^{-3}$
control.running.done (p:0, id:3)	$2.23 \cdot 10^6$	$2.14 \cdot 10^5$	$8.16 \cdot 10^4$	$2.79 \cdot 10^4$	$3.79 \cdot 10^{-2}$	$1.60 \cdot 10^{-2}$
control.running.done (p:0, id:4)	$2.15 \cdot 10^6$	$2.36 \cdot 10^5$	$5.07 \cdot 10^4$	$1.48 \cdot 10^4$	$2.33 \cdot 10^{-2}$	$5.37 \cdot 10^{-3}$
`control.success`	$3.66 \cdot 10^1$	$3.06 \cdot 10^1$	$3.66 \cdot 10^1$	$3.06 \cdot 10^1$	1.00	0.00
control.success.reset (p:0, id:0)	$3.66 \cdot 10^1$	$3.06 \cdot 10^1$	$3.66 \cdot 10^1$	$3.06 \cdot 10^1$	1.00	0.00
`frog.q`	$3.23 \cdot 10^6$	$9.63 \cdot 10^4$	$1.38 \cdot 10^6$	$1.48 \cdot 10^5$	$4.29 \cdot 10^{-1}$	$4.77 \cdot 10^{-2}$
frog.q.q (p:0, id:0)	$3.23 \cdot 10^6$	$9.63 \cdot 10^4$	$6.88 \cdot 10^5$	$5.26 \cdot 10^4$	$2.13 \cdot 10^{-1}$	$1.62 \cdot 10^{-2}$
frog.q.q (p:0, id:1)	$2.54 \cdot 10^6$	$1.00 \cdot 10^5$	$1.48 \cdot 10^5$	$2.40 \cdot 10^4$	$5.83 \cdot 10^{-2}$	$9.21 \cdot 10^{-3}$
frog.q.q (p:0, id:2)	$2.39 \cdot 10^6$	$9.73 \cdot 10^4$	$4.78 \cdot 10^5$	$1.04 \cdot 10^5$	$2.01 \cdot 10^{-1}$	$4.66 \cdot 10^{-2}$
frog.q.q (p:0, id:3)	$1.91 \cdot 10^6$	$1.69 \cdot 10^5$	$6.84 \cdot 10^4$	$1.63 \cdot 10^4$	$3.65 \cdot 10^{-2}$	$1.09 \cdot 10^{-2}$
`toad.q`	$3.27 \cdot 10^6$	$1.18 \cdot 10^5$	$1.43 \cdot 10^6$	$1.12 \cdot 10^5$	$4.37 \cdot 10^{-1}$	$3.87 \cdot 10^{-2}$
toad.q.q (p:0, id:0)	$3.27 \cdot 10^6$	$1.18 \cdot 10^5$	$7.18 \cdot 10^5$	$3.68 \cdot 10^4$	$2.20 \cdot 10^{-1}$	$9.71 \cdot 10^{-3}$
toad.q.q (p:0, id:1)	$2.55 \cdot 10^6$	$1.03 \cdot 10^5$	$1.59 \cdot 10^5$	$2.27 \cdot 10^4$	$6.22 \cdot 10^{-2}$	$7.23 \cdot 10^{-3}$
toad.q.q (p:0, id:2)	$2.39 \cdot 10^6$	$8.69 \cdot 10^4$	$4.90 \cdot 10^5$	$1.11 \cdot 10^5$	$2.06 \cdot 10^{-1}$	$4.86 \cdot 10^{-2}$
toad.q.q (p:0, id:3)	$1.90 \cdot 10^6$	$1.59 \cdot 10^5$	$6.01 \cdot 10^4$	$1.42 \cdot 10^4$	$3.22 \cdot 10^{-2}$	$9.41 \cdot 10^{-3}$

Table C.8: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model `ToadsAndFrogs`. The Java code has been generated using the default settings and the results have been attained through logging-based measurements. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

C.1.5 Practical Test: Telephony

Performance results for target model '`Telephony`' ($n = 20, t = 30$, Random + Det, User_0)

Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
`User_0.busy`	$2.77 \cdot 10^{6}$	$3.92 \cdot 10^{6}$	$2.77 \cdot 10^{6}$	$3.92 \cdot 10^{6}$	1.00	0.00
User_0.busy.idle (p:0, id:0)	$2.77 \cdot 10^{6}$	$3.92 \cdot 10^{6}$	$2.77 \cdot 10^{6}$	$3.92 \cdot 10^{6}$	1.00	0.00
`User_0.calling`	$5.58 \cdot 10^{6}$	$7.86 \cdot 10^{6}$	$5.58 \cdot 10^{6}$	$7.86 \cdot 10^{6}$	1.00	0.00
User_0.calling.busy (p:0, id:0)	$5.58 \cdot 10^{6}$	$7.86 \cdot 10^{6}$	$6.53 \cdot 10^{5}$	$9.22 \cdot 10^{5}$	$1.67 \cdot 10^{-1}$	$2.14 \cdot 10^{-1}$
User_0.calling.unobtainable (p:0, id:1)	$3.26 \cdot 10^{5}$	$4.61 \cdot 10^{5}$	$3.26 \cdot 10^{5}$	$4.61 \cdot 10^{5}$	1.00	0.00
User_0.calling.ringback (p:0, id:2)	$3.26 \cdot 10^{5}$	$4.61 \cdot 10^{5}$	$3.26 \cdot 10^{5}$	$4.61 \cdot 10^{5}$	1.00	0.00
User_0.calling.busy (p:0, id:3)	$4.28 \cdot 10^{6}$	$6.02 \cdot 10^{6}$	$2.12 \cdot 10^{6}$	$3.00 \cdot 10^{6}$	$5.24 \cdot 10^{-1}$	$1.92 \cdot 10^{-1}$
User_0.calling.calling (p:0, id:4)	$2.15 \cdot 10^{6}$	$3.02 \cdot 10^{6}$	$2.15 \cdot 10^{6}$	$3.02 \cdot 10^{6}$	$9.50 \cdot 10^{-1}$	$8.52 \cdot 10^{-2}$
User_0.calling.oalert (p:0, id:5)	$9.50 \cdot 10^{-1}$	$9.45 \cdot 10^{-1}$	$9.50 \cdot 10^{-1}$	$9.45 \cdot 10^{-1}$	1.00	0.00
`User_0.dialing`	$3.92 \cdot 10^{6}$	$5.53 \cdot 10^{6}$	$3.92 \cdot 10^{6}$	$5.53 \cdot 10^{6}$	1.00	0.00
User_0.dialing.idle (p:0, id:0)	$6.53 \cdot 10^{5}$	$9.22 \cdot 10^{5}$	$6.53 \cdot 10^{5}$	$9.22 \cdot 10^{5}$	1.00	0.00
User_0.dialing.calling (p:0, id:1)	$6.53 \cdot 10^{5}$	$9.22 \cdot 10^{5}$	$6.53 \cdot 10^{5}$	$9.22 \cdot 10^{5}$	1.00	0.00
User_0.dialing.calling (p:0, id:2)	$6.53 \cdot 10^{5}$	$9.22 \cdot 10^{5}$	$6.53 \cdot 10^{5}$	$9.22 \cdot 10^{5}$	1.00	0.00
User_0.dialing.calling (p:0, id:3)	$6.53 \cdot 10^{5}$	$9.22 \cdot 10^{5}$	$6.53 \cdot 10^{5}$	$9.22 \cdot 10^{5}$	1.00	0.00
User_0.dialing.calling (p:0, id:4)	$6.53 \cdot 10^{5}$	$9.22 \cdot 10^{5}$	$6.53 \cdot 10^{5}$	$9.22 \cdot 10^{5}$	1.00	0.00
User_0.dialing.calling (p:0, id:5)	$6.53 \cdot 10^{5}$	$9.22 \cdot 10^{5}$	$6.53 \cdot 10^{5}$	$9.22 \cdot 10^{5}$	1.00	0.00
`User_0.dveoringout`	$9.50 \cdot 10^{-1}$	$9.45 \cdot 10^{-1}$	$9.50 \cdot 10^{-1}$	$9.45 \cdot 10^{-1}$	1.00	0.00
User_0.dveoringout.idle (p:0, id:0)	$9.50 \cdot 10^{-1}$	$9.45 \cdot 10^{-1}$	$9.50 \cdot 10^{-1}$	$9.45 \cdot 10^{-1}$	1.00	0.00
`User_0.idle`	$2.20 \cdot 10^{7}$	$1.02 \cdot 10^{7}$	$2.20 \cdot 10^{7}$	$1.02 \cdot 10^{7}$	1.00	0.00
User_0.idle.dialing (p:0, id:0)	$2.20 \cdot 10^{7}$	$1.02 \cdot 10^{7}$	$3.92 \cdot 10^{6}$	$5.53 \cdot 10^{6}$	$3.50 \cdot 10^{-1}$	$4.89 \cdot 10^{-1}$
User_0.idle.qi (p:0, id:1)	$1.81 \cdot 10^{7}$	$1.49 \cdot 10^{7}$	$1.81 \cdot 10^{7}$	$1.49 \cdot 10^{7}$	1.00	0.00
`User_0.oalert`	$9.50 \cdot 10^{-1}$	$9.45 \cdot 10^{-1}$	$9.50 \cdot 10^{-1}$	$9.45 \cdot 10^{-1}$	1.00	0.00
User_0.oalert.dveoringout (p:0, id:2)	$9.50 \cdot 10^{-1}$	$9.45 \cdot 10^{-1}$	$9.50 \cdot 10^{-1}$	$9.45 \cdot 10^{-1}$	1.00	0.00
`User_0.qi`	$1.81 \cdot 10^{7}$	$1.49 \cdot 10^{7}$	$1.81 \cdot 10^{7}$	$1.49 \cdot 10^{7}$	1.00	0.00
User_0.qi.talert (p:0, id:0)	$1.81 \cdot 10^{7}$	$1.49 \cdot 10^{7}$	$1.50 \cdot 10^{-1}$	$3.66 \cdot 10^{-1}$	$8.36 \cdot 10^{-9}$	$1.61 \cdot 10^{-8}$
User_0.qi.idle (p:0, id:1)	$1.81 \cdot 10^{7}$	$1.49 \cdot 10^{7}$	$1.81 \cdot 10^{7}$	$1.49 \cdot 10^{7}$	1.00	0.00
`User_0.ringback`	$3.26 \cdot 10^{5}$	$4.61 \cdot 10^{5}$	$3.26 \cdot 10^{5}$	$4.61 \cdot 10^{5}$	1.00	$5.00 \cdot 10^{-7}$
User_0.ringback.idle (p:0, id:0)	$1.63 \cdot 10^{5}$	$2.30 \cdot 10^{5}$	$1.63 \cdot 10^{5}$	$2.30 \cdot 10^{5}$	1.00	0.00
User_0.ringback.calling (p:0, id:1)	$1.63 \cdot 10^{5}$	$2.31 \cdot 10^{5}$	$1.63 \cdot 10^{5}$	$2.31 \cdot 10^{5}$	1.00	$1.14 \cdot 10^{-6}$
`User_0.talert`	$2.00 \cdot 10^{-1}$	$5.23 \cdot 10^{-1}$	$1.50 \cdot 10^{-1}$	$3.66 \cdot 10^{-1}$	$8.33 \cdot 10^{-1}$	$2.89 \cdot 10^{-1}$
User_0.talert.idle (p:0, id:2)	$1.50 \cdot 10^{-1}$	$3.66 \cdot 10^{-1}$	$1.50 \cdot 10^{-1}$	$3.66 \cdot 10^{-1}$	1.00	0.00
`User_0.unobtainable`	$3.26 \cdot 10^{5}$	$4.61 \cdot 10^{5}$	$3.26 \cdot 10^{5}$	$4.61 \cdot 10^{5}$	1.00	0.00
User_0.unobtainable.idle (p:0, id:0)	$3.26 \cdot 10^{5}$	$4.61 \cdot 10^{5}$	$3.26 \cdot 10^{5}$	$4.61 \cdot 10^{5}$	1.00	0.00

Table C.9: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model `Telephony` (state machine `User_0` only). The Java code has been generated with the 'Random + Det' decision mode enabled. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

Performance results for target model '`Telephony`' ($n = 20, t = 30$, Random + Det, User_1)

Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
`User_1.busy`	$2.75 \cdot 10^{6}$	$5.19 \cdot 10^{6}$	$2.75 \cdot 10^{6}$	$5.19 \cdot 10^{6}$	1.00	0.00
User_1.busy.idle (p:0, id:0)	$2.75 \cdot 10^{6}$	$5.19 \cdot 10^{6}$	$2.75 \cdot 10^{6}$	$5.19 \cdot 10^{6}$	1.00	0.00
`User_1.calling`	$4.80 \cdot 10^{6}$	$9.10 \cdot 10^{6}$	$4.80 \cdot 10^{6}$	$9.10 \cdot 10^{6}$	1.00	0.00
User_1.calling.busy (p:0, id:0)	$4.80 \cdot 10^{6}$	$9.10 \cdot 10^{6}$	$1.40 \cdot 10^{6}$	$2.69 \cdot 10^{6}$	$2.96 \cdot 10^{-1}$	$1.46 \cdot 10^{-1}$
User_1.calling.unobtainable (p:0, id:1)	$3.23 \cdot 10^{5}$	$6.11 \cdot 10^{5}$	$3.23 \cdot 10^{5}$	$6.11 \cdot 10^{5}$	1.00	0.00
User_1.calling.ringback (p:0, id:2)	$3.23 \cdot 10^{5}$	$6.11 \cdot 10^{5}$	$3.23 \cdot 10^{5}$	$6.11 \cdot 10^{5}$	1.00	0.00
User_1.calling.busy (p:0, id:3)	$2.75 \cdot 10^{6}$	$5.19 \cdot 10^{6}$	$1.34 \cdot 10^{6}$	$2.51 \cdot 10^{6}$	$3.56 \cdot 10^{-1}$	$1.81 \cdot 10^{-1}$
User_1.calling.calling (p:0, id:4)	$1.40 \cdot 10^{6}$	$2.69 \cdot 10^{6}$	$1.40 \cdot 10^{6}$	$2.69 \cdot 10^{6}$	$8.44 \cdot 10^{-1}$	$2.99 \cdot 10^{-1}$
User_1.calling.oalert (p:0, id:5)	$8.50 \cdot 10^{-1}$	$5.87 \cdot 10^{-1}$	$8.50 \cdot 10^{-1}$	$5.87 \cdot 10^{-1}$	1.00	0.00
`User_1.dialing`	$3.88 \cdot 10^{6}$	$7.33 \cdot 10^{6}$	$3.88 \cdot 10^{6}$	$7.33 \cdot 10^{6}$	1.00	0.00
User_1.dialing.idle (p:0, id:0)	$6.47 \cdot 10^{5}$	$1.22 \cdot 10^{6}$	$6.47 \cdot 10^{5}$	$1.22 \cdot 10^{6}$	1.00	0.00
User_1.dialing.calling (p:0, id:1)	$6.47 \cdot 10^{5}$	$1.22 \cdot 10^{6}$	$6.47 \cdot 10^{5}$	$1.22 \cdot 10^{6}$	1.00	0.00
User_1.dialing.calling (p:0, id:2)	$6.47 \cdot 10^{5}$	$1.22 \cdot 10^{6}$	$6.47 \cdot 10^{5}$	$1.22 \cdot 10^{6}$	1.00	0.00
User_1.dialing.calling (p:0, id:3)	$6.47 \cdot 10^{5}$	$1.22 \cdot 10^{6}$	$6.47 \cdot 10^{5}$	$1.22 \cdot 10^{6}$	1.00	0.00
User_1.dialing.calling (p:0, id:4)	$6.47 \cdot 10^{5}$	$1.22 \cdot 10^{6}$	$6.47 \cdot 10^{5}$	$1.22 \cdot 10^{6}$	1.00	0.00
User_1.dialing.calling (p:0, id:5)	$6.47 \cdot 10^{5}$	$1.22 \cdot 10^{6}$	$6.47 \cdot 10^{5}$	$1.22 \cdot 10^{6}$	1.00	0.00
`User_1.dveoringout`	$8.50 \cdot 10^{-1}$	$5.87 \cdot 10^{-1}$	$8.50 \cdot 10^{-1}$	$5.87 \cdot 10^{-1}$	1.00	0.00
User_1.dveoringout.idle (p:0, id:0)	$8.50 \cdot 10^{-1}$	$5.87 \cdot 10^{-1}$	$8.50 \cdot 10^{-1}$	$5.87 \cdot 10^{-1}$	1.00	0.00
`User_1.idle`	$2.18 \cdot 10^{7}$	$6.39 \cdot 10^{6}$	$2.18 \cdot 10^{7}$	$6.39 \cdot 10^{6}$	1.00	0.00
User_1.idle.dialing (p:0, id:0)	$2.18 \cdot 10^{7}$	$6.39 \cdot 10^{6}$	$3.88 \cdot 10^{6}$	$7.33 \cdot 10^{6}$	$2.50 \cdot 10^{-1}$	$4.44 \cdot 10^{-1}$
User_1.idle.qi (p:0, id:1)	$1.79 \cdot 10^{7}$	$1.15 \cdot 10^{7}$	$1.79 \cdot 10^{7}$	$1.15 \cdot 10^{7}$	1.00	0.00
`User_1.oalert`	$8.50 \cdot 10^{-1}$	$5.87 \cdot 10^{-1}$	$8.50 \cdot 10^{-1}$	$5.87 \cdot 10^{-1}$	1.00	0.00
User_1.oalert.dveoringout (p:0, id:2)	$8.50 \cdot 10^{-1}$	$5.87 \cdot 10^{-1}$	$8.50 \cdot 10^{-1}$	$5.87 \cdot 10^{-1}$	1.00	0.00
`User_1.qi`	$1.79 \cdot 10^{7}$	$1.15 \cdot 10^{7}$	$1.79 \cdot 10^{7}$	$1.15 \cdot 10^{7}$	1.00	0.00
User_1.qi.talert (p:0, id:0)	$1.79 \cdot 10^{7}$	$1.15 \cdot 10^{7}$	$4.00 \cdot 10^{-1}$	$5.03 \cdot 10^{-1}$	$2.35 \cdot 10^{-8}$	$2.35 \cdot 10^{-8}$
User_1.qi.idle (p:0, id:1)	$1.79 \cdot 10^{7}$	$1.15 \cdot 10^{7}$	$1.79 \cdot 10^{7}$	$1.15 \cdot 10^{7}$	1.00	0.00
`User_1.ringback`	$3.23 \cdot 10^{5}$	$6.11 \cdot 10^{5}$	$3.23 \cdot 10^{5}$	$6.11 \cdot 10^{5}$	1.00	0.00
User_1.ringback.idle (p:0, id:0)	$1.62 \cdot 10^{5}$	$3.06 \cdot 10^{5}$	$1.62 \cdot 10^{5}$	$3.06 \cdot 10^{5}$	1.00	0.00
User_1.ringback.calling (p:0, id:1)	$1.62 \cdot 10^{5}$	$3.05 \cdot 10^{5}$	$1.62 \cdot 10^{5}$	$3.05 \cdot 10^{5}$	1.00	0.00
`User_1.talert`	$7.50 \cdot 10^{-1}$	1.16	$4.00 \cdot 10^{-1}$	$5.03 \cdot 10^{-1}$	$6.98 \cdot 10^{-1}$	$3.33 \cdot 10^{-1}$
User_1.talert.tpickup (p:0, id:1)	$4.00 \cdot 10^{-1}$	$5.03 \cdot 10^{-1}$	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	$1.25 \cdot 10^{-1}$	$3.54 \cdot 10^{-1}$
User_1.talert.idle (p:0, id:2)	$3.50 \cdot 10^{-1}$	$4.89 \cdot 10^{-1}$	$3.50 \cdot 10^{-1}$	$4.89 \cdot 10^{-1}$	1.00	0.00
`User_1.tpickup`	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	1.00	NaN
User_1.tpickup.idle (p:0, id:1)	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	1.00	NaN
`User_1.unobtainable`	$3.23 \cdot 10^{5}$	$6.11 \cdot 10^{5}$	$3.23 \cdot 10^{5}$	$6.11 \cdot 10^{5}$	1.00	0.00
User_1.unobtainable.idle (p:0, id:0)	$3.23 \cdot 10^{5}$	$6.11 \cdot 10^{5}$	$3.23 \cdot 10^{5}$	$6.11 \cdot 10^{5}$	1.00	0.00

Table C.10: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model `Telephony` (state machine `User_1` only). The Java code has been generated with the 'Random + Det' decision mode enabled. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

Performance results for target model 'Telephony' ($n = 20, t = 30$, Random + Det, User_2)

Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
`User_2.busy`	$1.39 \cdot 10^{6}$	$3.41 \cdot 10^{6}$	$1.39 \cdot 10^{6}$	$3.41 \cdot 10^{6}$	1.00	0.00
User_2.busy.idle (p:0, id:0)	$1.39 \cdot 10^{6}$	$3.41 \cdot 10^{6}$	$1.39 \cdot 10^{6}$	$3.41 \cdot 10^{6}$	1.00	0.00
`User_2.calling`	$2.75 \cdot 10^{6}$	$6.78 \cdot 10^{6}$	$2.75 \cdot 10^{6}$	$6.78 \cdot 10^{6}$	1.00	0.00
User_2.calling.busy (p:0, id:0)	$2.75 \cdot 10^{6}$	$6.78 \cdot 10^{6}$	$1.04 \cdot 10^{6}$	$2.57 \cdot 10^{6}$	$4.19 \cdot 10^{-1}$	$3.18 \cdot 10^{-1}$
User_2.calling.unobtainable (p:0, id:1)	$1.63 \cdot 10^{5}$	$4.01 \cdot 10^{5}$	$1.63 \cdot 10^{5}$	$4.01 \cdot 10^{5}$	1.00	0.00
User_2.calling.ringback (p:0, id:2)	$1.63 \cdot 10^{5}$	$4.01 \cdot 10^{5}$	$1.63 \cdot 10^{5}$	$4.01 \cdot 10^{5}$	1.00	0.00
User_2.calling.busy (p:0, id:3)	$1.39 \cdot 10^{6}$	$3.41 \cdot 10^{6}$	$3.49 \cdot 10^{5}$	$8.52 \cdot 10^{5}$	$3.33 \cdot 10^{-1}$	$3.59 \cdot 10^{-1}$
User_2.calling.calling (p:0, id:4)	$1.04 \cdot 10^{6}$	$2.57 \cdot 10^{6}$	$1.04 \cdot 10^{6}$	$2.57 \cdot 10^{6}$	$9.57 \cdot 10^{-1}$	$8.30 \cdot 10^{-2}$
User_2.calling.oalert (p:0, id:5)	$4.00 \cdot 10^{-1}$	$6.81 \cdot 10^{-1}$	$4.00 \cdot 10^{-1}$	$6.81 \cdot 10^{-1}$	1.00	0.00
`User_2.dialing`	$1.96 \cdot 10^{6}$	$4.82 \cdot 10^{6}$	$1.96 \cdot 10^{6}$	$4.82 \cdot 10^{6}$	1.00	0.00
User_2.dialing.idle (p:0, id:0)	$3.26 \cdot 10^{5}$	$8.03 \cdot 10^{5}$	$3.26 \cdot 10^{5}$	$8.03 \cdot 10^{5}$	1.00	0.00
User_2.dialing.calling (p:0, id:1)	$3.26 \cdot 10^{5}$	$8.03 \cdot 10^{5}$	$3.26 \cdot 10^{5}$	$8.03 \cdot 10^{5}$	1.00	0.00
User_2.dialing.calling (p:0, id:2)	$3.26 \cdot 10^{5}$	$8.03 \cdot 10^{5}$	$3.26 \cdot 10^{5}$	$8.03 \cdot 10^{5}$	1.00	0.00
User_2.dialing.calling (p:0, id:3)	$3.26 \cdot 10^{5}$	$8.03 \cdot 10^{5}$	$3.26 \cdot 10^{5}$	$8.03 \cdot 10^{5}$	1.00	0.00
User_2.dialing.calling (p:0, id:4)	$3.26 \cdot 10^{5}$	$8.03 \cdot 10^{5}$	$3.26 \cdot 10^{5}$	$8.03 \cdot 10^{5}$	1.00	0.00
User_2.dialing.calling (p:0, id:5)	$3.26 \cdot 10^{5}$	$8.03 \cdot 10^{5}$	$3.26 \cdot 10^{5}$	$8.03 \cdot 10^{5}$	1.00	0.00
`User_2.dveoringout`	$4.00 \cdot 10^{-1}$	$6.81 \cdot 10^{-1}$	$4.00 \cdot 10^{-1}$	$6.81 \cdot 10^{-1}$	1.00	0.00
User_2.dveoringout.idle (p:0, id:0)	$4.00 \cdot 10^{-1}$	$6.81 \cdot 10^{-1}$	$4.00 \cdot 10^{-1}$	$6.81 \cdot 10^{-1}$	1.00	0.00
`User_2.idle`	$2.26 \cdot 10^{7}$	$6.54 \cdot 10^{6}$	$2.26 \cdot 10^{7}$	$6.54 \cdot 10^{6}$	1.00	0.00
User_2.idle.dialing (p:0, id:0)	$2.26 \cdot 10^{7}$	$6.54 \cdot 10^{6}$	$1.96 \cdot 10^{6}$	$4.82 \cdot 10^{6}$	$1.50 \cdot 10^{-1}$	$3.66 \cdot 10^{-1}$
User_2.idle.qi (p:0, id:1)	$2.06 \cdot 10^{7}$	$1.02 \cdot 10^{7}$	$2.06 \cdot 10^{7}$	$1.02 \cdot 10^{7}$	1.00	0.00
`User_2.oalert`	$4.00 \cdot 10^{-1}$	$6.81 \cdot 10^{-1}$	$4.00 \cdot 10^{-1}$	$6.81 \cdot 10^{-1}$	1.00	0.00
User_2.oalert.dveoringout (p:0, id:2)	$4.00 \cdot 10^{-1}$	$6.81 \cdot 10^{-1}$	$4.00 \cdot 10^{-1}$	$6.81 \cdot 10^{-1}$	1.00	0.00
`User_2.qi`	$2.06 \cdot 10^{7}$	$1.02 \cdot 10^{7}$	$2.06 \cdot 10^{7}$	$1.02 \cdot 10^{7}$	1.00	0.00
User_2.qi.talert (p:0, id:0)	$2.06 \cdot 10^{7}$	$1.02 \cdot 10^{7}$	$5.50 \cdot 10^{-1}$	$6.05 \cdot 10^{-1}$	$3.04 \cdot 10^{-8}$	$2.84 \cdot 10^{-8}$
User_2.qi.idle (p:0, id:1)	$2.06 \cdot 10^{7}$	$1.02 \cdot 10^{7}$	$2.06 \cdot 10^{7}$	$1.02 \cdot 10^{7}$	1.00	0.00
`User_2.ringback`	$1.63 \cdot 10^{5}$	$4.01 \cdot 10^{5}$	$1.63 \cdot 10^{5}$	$4.01 \cdot 10^{5}$	1.00	$2.90 \cdot 10^{-7}$
User_2.ringback.idle (p:0, id:0)	$8.15 \cdot 10^{4}$	$2.01 \cdot 10^{5}$	$8.15 \cdot 10^{4}$	$2.01 \cdot 10^{5}$	1.00	0.00
User_2.ringback.calling (p:0, id:1)	$8.15 \cdot 10^{4}$	$2.01 \cdot 10^{5}$	$8.15 \cdot 10^{4}$	$2.01 \cdot 10^{5}$	1.00	$7.78 \cdot 10^{-7}$
`User_2.talert`	1.00	1.26	$5.50 \cdot 10^{-1}$	$6.05 \cdot 10^{-1}$	$6.92 \cdot 10^{-1}$	$3.33 \cdot 10^{-1}$
User_2.talert.tpickup (p:0, id:1)	$5.50 \cdot 10^{-1}$	$6.05 \cdot 10^{-1}$	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	$5.00 \cdot 10^{-2}$	$1.58 \cdot 10^{-1}$
User_2.talert.idle (p:0, id:2)	$5.00 \cdot 10^{-1}$	$5.13 \cdot 10^{-1}$	$5.00 \cdot 10^{-1}$	$5.13 \cdot 10^{-1}$	1.00	0.00
`User_2.tpickup`	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	1.00	NaN
User_2.tpickup.idle (p:0, id:1)	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	1.00	NaN
`User_2.unobtainable`	$1.63 \cdot 10^{5}$	$4.01 \cdot 10^{5}$	$1.63 \cdot 10^{5}$	$4.01 \cdot 10^{5}$	1.00	0.00
User_2.unobtainable.idle (p:0, id:0)	$1.63 \cdot 10^{5}$	$4.01 \cdot 10^{5}$	$1.63 \cdot 10^{5}$	$4.01 \cdot 10^{5}$	1.00	0.00

Table C.11: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model `Telephony` (state machine `User_2` only). The Java code has been generated with the 'Random + Det' decision mode enabled. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

Performance results for target model '`Telephony`' ($n = 20, t = 30$, Random + Det, User_3)

Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
`User_3.busy`	$1.65 \cdot 10^{6}$	$2.95 \cdot 10^{6}$	$1.65 \cdot 10^{6}$	$2.95 \cdot 10^{6}$	1.00	0.00
User_3.busy.idle (p:0, id:0)	$1.65 \cdot 10^{6}$	$2.95 \cdot 10^{6}$	$1.65 \cdot 10^{6}$	$2.95 \cdot 10^{6}$	1.00	0.00
`User_3.calling`	$4.57 \cdot 10^{6}$	$8.19 \cdot 10^{6}$	$4.57 \cdot 10^{6}$	$8.19 \cdot 10^{6}$	1.00	0.00
User_3.calling.busy (p:0, id:0)	$4.57 \cdot 10^{6}$	$8.19 \cdot 10^{6}$	$1.65 \cdot 10^{6}$	$2.95 \cdot 10^{6}$	$3.14 \cdot 10^{-1}$	$6.34 \cdot 10^{-2}$
User_3.calling.unobtainable (p:0, id:1)	$1.94 \cdot 10^{5}$	$3.48 \cdot 10^{5}$	$1.94 \cdot 10^{5}$	$3.48 \cdot 10^{5}$	1.00	0.00
User_3.calling.ringback (p:0, id:2)	$1.94 \cdot 10^{5}$	$3.47 \cdot 10^{5}$	$1.94 \cdot 10^{5}$	$3.47 \cdot 10^{5}$	1.00	0.00
User_3.calling.calling (p:0, id:4)	$2.54 \cdot 10^{6}$	$4.54 \cdot 10^{6}$	$2.54 \cdot 10^{6}$	$4.54 \cdot 10^{6}$	$9.22 \cdot 10^{-1}$	$1.02 \cdot 10^{-1}$
User_3.calling.oalert (p:0, id:5)	$9.50 \cdot 10^{-1}$	$7.59 \cdot 10^{-1}$	$9.50 \cdot 10^{-1}$	$7.59 \cdot 10^{-1}$	1.00	0.00
`User_3.dialing`	$2.33 \cdot 10^{6}$	$4.17 \cdot 10^{6}$	$2.33 \cdot 10^{6}$	$4.17 \cdot 10^{6}$	1.00	0.00
User_3.dialing.idle (p:0, id:0)	$3.88 \cdot 10^{5}$	$6.95 \cdot 10^{5}$	$3.88 \cdot 10^{5}$	$6.95 \cdot 10^{5}$	1.00	0.00
User_3.dialing.calling (p:0, id:1)	$3.88 \cdot 10^{5}$	$6.96 \cdot 10^{5}$	$3.88 \cdot 10^{5}$	$6.96 \cdot 10^{5}$	1.00	0.00
User_3.dialing.calling (p:0, id:2)	$3.88 \cdot 10^{5}$	$6.95 \cdot 10^{5}$	$3.88 \cdot 10^{5}$	$6.95 \cdot 10^{5}$	1.00	0.00
User_3.dialing.calling (p:0, id:3)	$3.88 \cdot 10^{5}$	$6.95 \cdot 10^{5}$	$3.88 \cdot 10^{5}$	$6.95 \cdot 10^{5}$	1.00	0.00
User_3.dialing.calling (p:0, id:4)	$3.87 \cdot 10^{5}$	$6.95 \cdot 10^{5}$	$3.87 \cdot 10^{5}$	$6.95 \cdot 10^{5}$	1.00	0.00
User_3.dialing.calling (p:0, id:5)	$3.88 \cdot 10^{5}$	$6.95 \cdot 10^{5}$	$3.88 \cdot 10^{5}$	$6.95 \cdot 10^{5}$	1.00	0.00
`User_3.dveoringout`	$9.50 \cdot 10^{-1}$	$7.59 \cdot 10^{-1}$	$9.50 \cdot 10^{-1}$	$7.59 \cdot 10^{-1}$	1.00	0.00
User_3.dveoringout.idle (p:0, id:0)	$9.50 \cdot 10^{-1}$	$7.59 \cdot 10^{-1}$	$9.50 \cdot 10^{-1}$	$7.59 \cdot 10^{-1}$	1.00	0.00
`User_3.idle`	$1.78 \cdot 10^{7}$	$9.30 \cdot 10^{6}$	$1.78 \cdot 10^{7}$	$9.30 \cdot 10^{6}$	1.00	0.00
User_3.idle.dialing (p:0, id:0)	$1.78 \cdot 10^{7}$	$9.30 \cdot 10^{6}$	$2.33 \cdot 10^{6}$	$4.17 \cdot 10^{6}$	$2.50 \cdot 10^{-1}$	$4.44 \cdot 10^{-1}$
User_3.idle.qi (p:0, id:1)	$1.55 \cdot 10^{7}$	$1.20 \cdot 10^{7}$	$1.55 \cdot 10^{7}$	$1.20 \cdot 10^{7}$	1.00	0.00
`User_3.oalert`	$9.50 \cdot 10^{-1}$	$7.59 \cdot 10^{-1}$	$9.50 \cdot 10^{-1}$	$7.59 \cdot 10^{-1}$	1.00	0.00
User_3.oalert.dveoringout (p:0, id:2)	$9.50 \cdot 10^{-1}$	$7.59 \cdot 10^{-1}$	$9.50 \cdot 10^{-1}$	$7.59 \cdot 10^{-1}$	1.00	0.00
`User_3.qi`	$1.55 \cdot 10^{7}$	$1.20 \cdot 10^{7}$	$1.55 \cdot 10^{7}$	$1.20 \cdot 10^{7}$	1.00	0.00
User_3.qi.talert (p:0, id:0)	$1.55 \cdot 10^{7}$	$1.20 \cdot 10^{7}$	$3.50 \cdot 10^{-1}$	$4.89 \cdot 10^{-1}$	$1.25 \cdot 10^{-1}$	$3.42 \cdot 10^{-1}$
User_3.qi.idle (p:0, id:1)	$1.55 \cdot 10^{7}$	$1.20 \cdot 10^{7}$	$1.55 \cdot 10^{7}$	$1.20 \cdot 10^{7}$	1.00	0.00
`User_3.ringback`	$1.94 \cdot 10^{5}$	$3.47 \cdot 10^{5}$	$1.94 \cdot 10^{5}$	$3.47 \cdot 10^{5}$	1.00	0.00
User_3.ringback.idle (p:0, id:0)	$9.68 \cdot 10^{4}$	$1.74 \cdot 10^{5}$	$9.68 \cdot 10^{4}$	$1.74 \cdot 10^{5}$	1.00	0.00
User_3.ringback.calling (p:0, id:1)	$9.70 \cdot 10^{4}$	$1.74 \cdot 10^{5}$	$9.70 \cdot 10^{4}$	$1.74 \cdot 10^{5}$	1.00	0.00
`User_3.talert`	$6.50 \cdot 10^{-1}$	1.04	$3.50 \cdot 10^{-1}$	$4.89 \cdot 10^{-1}$	$6.67 \cdot 10^{-1}$	$3.19 \cdot 10^{-1}$
User_3.talert.tpickup (p:0, id:1)	$3.50 \cdot 10^{-1}$	$4.89 \cdot 10^{-1}$	$1.00 \cdot 10^{-1}$	$3.08 \cdot 10^{-1}$	$2.86 \cdot 10^{-1}$	$4.88 \cdot 10^{-1}$
User_3.talert.idle (p:0, id:2)	$2.50 \cdot 10^{-1}$	$4.44 \cdot 10^{-1}$	$2.50 \cdot 10^{-1}$	$4.44 \cdot 10^{-1}$	1.00	0.00
`User_3.tconnected`	$2.56 \cdot 10^{6}$	$1.14 \cdot 10^{7}$	$2.56 \cdot 10^{6}$	$1.14 \cdot 10^{7}$	1.00	NaN
User_3.tconnected.tconnected (p:0, id:0)	$2.56 \cdot 10^{6}$	$1.14 \cdot 10^{7}$	$1.28 \cdot 10^{6}$	$5.72 \cdot 10^{6}$	$5.00 \cdot 10^{-1}$	NaN
User_3.tconnected.tconnected (p:0, id:1)	$1.28 \cdot 10^{6}$	$5.72 \cdot 10^{6}$	$1.28 \cdot 10^{6}$	$5.72 \cdot 10^{6}$	1.00	NaN
`User_3.tpickup`	$1.00 \cdot 10^{-1}$	$3.08 \cdot 10^{-1}$	$1.00 \cdot 10^{-1}$	$3.08 \cdot 10^{-1}$	1.00	0.00
User_3.tpickup.tconnected (p:0, id:0)	$1.00 \cdot 10^{-1}$	$3.08 \cdot 10^{-1}$	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	$5.00 \cdot 10^{-1}$	$7.07 \cdot 10^{-1}$
User_3.tpickup.idle (p:0, id:1)	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	1.00	NaN
`User_3.unobtainable`	$1.94 \cdot 10^{5}$	$3.48 \cdot 10^{5}$	$1.94 \cdot 10^{5}$	$3.48 \cdot 10^{5}$	1.00	0.00
User_3.unobtainable.idle (p:0, id:0)	$1.94 \cdot 10^{5}$	$3.48 \cdot 10^{5}$	$1.94 \cdot 10^{5}$	$3.48 \cdot 10^{5}$	1.00	0.00

Table C.12: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model `Telephony` (state machine `User_3` only). The Java code has been generated with the 'Random + Det' decision mode enabled. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

Performance results for target model 'Telephony' ($n = 20, t = 30$, Logging, User_0)						
Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
`User_0.busy`	$2.24 \cdot 10^5$	$3.52 \cdot 10^5$	$2.24 \cdot 10^5$	$3.52 \cdot 10^5$	1.00	0.00
User_0.busy.idle (p:0, id:0)	$2.24 \cdot 10^5$	$3.52 \cdot 10^5$	$2.24 \cdot 10^5$	$3.52 \cdot 10^5$	1.00	0.00
`User_0.calling`	$4.44 \cdot 10^5$	$6.97 \cdot 10^5$	$4.44 \cdot 10^5$	$6.97 \cdot 10^5$	1.00	$3.62 \cdot 10^{-7}$
User_0.calling.busy (p:0, id:0)	$4.44 \cdot 10^5$	$6.97 \cdot 10^5$	$5.33 \cdot 10^4$	$8.37 \cdot 10^4$	$1.20 \cdot 10^{-1}$	$3.77 \cdot 10^{-3}$
User_0.calling.unobtainable (p:0, id:1)	$2.66 \cdot 10^4$	$4.18 \cdot 10^4$	$2.66 \cdot 10^4$	$4.18 \cdot 10^4$	1.00	0.00
User_0.calling.ringback (p:0, id:2)	$2.66 \cdot 10^4$	$4.18 \cdot 10^4$	$2.66 \cdot 10^4$	$4.18 \cdot 10^4$	1.00	0.00
User_0.calling.busy (p:0, id:3)	$3.37 \cdot 10^5$	$5.30 \cdot 10^5$	$1.71 \cdot 10^5$	$2.68 \cdot 10^5$	$5.07 \cdot 10^{-1}$	$1.59 \cdot 10^{-2}$
User_0.calling.calling (p:0, id:4)	$1.67 \cdot 10^5$	$2.62 \cdot 10^5$	$1.67 \cdot 10^5$	$2.62 \cdot 10^5$	1.00	0.00
User_0.calling.oalert (p:0, id:5)	0.00	0.00	0.00	0.00	NaN	NaN
`User_0.dialing`	$3.20 \cdot 10^5$	$5.02 \cdot 10^5$	$3.20 \cdot 10^5$	$5.02 \cdot 10^5$	1.00	$3.91 \cdot 10^{-7}$
User_0.dialing.idle (p:0, id:0)	$5.33 \cdot 10^4$	$8.37 \cdot 10^4$	$5.33 \cdot 10^4$	$8.37 \cdot 10^4$	1.00	0.00
User_0.dialing.calling (p:0, id:1)	$5.33 \cdot 10^4$	$8.37 \cdot 10^4$	$5.33 \cdot 10^4$	$8.37 \cdot 10^4$	1.00	0.00
User_0.dialing.calling (p:0, id:2)	$5.33 \cdot 10^4$	$8.36 \cdot 10^4$	$5.33 \cdot 10^4$	$8.36 \cdot 10^4$	1.00	0.00
User_0.dialing.calling (p:0, id:3)	$5.32 \cdot 10^4$	$8.35 \cdot 10^4$	$5.32 \cdot 10^4$	$8.35 \cdot 10^4$	1.00	0.00
User_0.dialing.calling (p:0, id:4)	$5.33 \cdot 10^4$	$8.36 \cdot 10^4$	$5.33 \cdot 10^4$	$8.36 \cdot 10^4$	1.00	0.00
User_0.dialing.calling (p:0, id:5)	$5.33 \cdot 10^4$	$8.36 \cdot 10^4$	$5.33 \cdot 10^4$	$8.36 \cdot 10^4$	1.00	0.00
`User_0.dveoringout`	0.00	0.00	0.00	0.00	NaN	NaN
User_0.dveoringout.idle (p:0, id:0)	0.00	0.00	0.00	0.00	NaN	NaN
`User_0.idle`	$1.86 \cdot 10^6$	$5.52 \cdot 10^5$	$1.86 \cdot 10^6$	$5.52 \cdot 10^5$	1.00	$2.03 \cdot 10^{-7}$
User_0.idle.dialing (p:0, id:0)	$1.86 \cdot 10^6$	$5.52 \cdot 10^5$	$3.20 \cdot 10^5$	$5.02 \cdot 10^5$	$3.00 \cdot 10^{-1}$	$4.70 \cdot 10^{-1}$
User_0.idle.qi (p:0, id:1)	$1.55 \cdot 10^6$	$1.05 \cdot 10^6$	$1.55 \cdot 10^6$	$1.05 \cdot 10^6$	1.00	0.00
`User_0.oalert`	0.00	0.00	0.00	0.00	NaN	NaN
User_0.oalert.dveoringout (p:0, id:2)	0.00	0.00	0.00	0.00	NaN	NaN
`User_0.qi`	$1.55 \cdot 10^6$	$1.05 \cdot 10^6$	$1.55 \cdot 10^6$	$1.05 \cdot 10^6$	1.00	$2.38 \cdot 10^{-7}$
User_0.qi.talert (p:0, id:0)	$1.55 \cdot 10^6$	$1.05 \cdot 10^6$	0.00	0.00	0.00	0.00
User_0.qi.idle (p:0, id:1)	$1.55 \cdot 10^6$	$1.05 \cdot 10^6$	$1.55 \cdot 10^6$	$1.05 \cdot 10^6$	1.00	$1.70 \cdot 10^{-7}$
`User_0.ringback`	$3.12 \cdot 10^4$	$5.28 \cdot 10^4$	$2.66 \cdot 10^4$	$4.18 \cdot 10^4$	$9.16 \cdot 10^{-1}$	$2.05 \cdot 10^{-1}$
User_0.ringback.idle (p:0, id:0)	$1.56 \cdot 10^4$	$2.63 \cdot 10^4$	$1.56 \cdot 10^4$	$2.63 \cdot 10^4$	1.00	0.00
User_0.ringback.calling (p:0, id:1)	$1.56 \cdot 10^4$	$2.65 \cdot 10^4$	$1.11 \cdot 10^4$	$1.97 \cdot 10^4$	$8.33 \cdot 10^{-1}$	$4.08 \cdot 10^{-1}$
`User_0.talert`	0.00	0.00	0.00	0.00	NaN	NaN
User_0.talert.idle (p:0, id:2)	0.00	0.00	0.00	0.00	NaN	NaN
`User_0.unobtainable`	$2.66 \cdot 10^4$	$4.18 \cdot 10^4$	$2.66 \cdot 10^4$	$4.18 \cdot 10^4$	1.00	$4.92 \cdot 10^{-6}$
User_0.unobtainable.idle (p:0, id:0)	$2.66 \cdot 10^4$	$4.18 \cdot 10^4$	$2.66 \cdot 10^4$	$4.18 \cdot 10^4$	1.00	$4.92 \cdot 10^{-6}$

Table C.13: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model `Telephony` (state machine `User_0` only). The Java code has been generated with the 'Random + Det' decision enabled and the results have been attained through logging-based measurements. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

Performance results for target model 'Telephony' ($n = 20, t = 30$, Logging, User_1)

Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
User_1.busy	$1.65 \cdot 10^5$	$3.39 \cdot 10^5$	$1.65 \cdot 10^5$	$3.39 \cdot 10^5$	1.00	$6.14 \cdot 10^{-7}$
User_1.busy.idle (p:0, id:0)	$1.65 \cdot 10^5$	$3.39 \cdot 10^5$	$1.65 \cdot 10^5$	$3.39 \cdot 10^5$	1.00	0.00
User_1.calling	$2.89 \cdot 10^5$	$5.93 \cdot 10^5$	$2.89 \cdot 10^5$	$5.93 \cdot 10^5$	1.00	$4.03 \cdot 10^{-7}$
User_1.calling.busy (p:0, id:0)	$2.89 \cdot 10^5$	$5.93 \cdot 10^5$	$8.01 \cdot 10^4$	$1.65 \cdot 10^5$	$2.77 \cdot 10^{-1}$	$1.62 \cdot 10^{-2}$
User_1.calling.unobtainable (p:0, id:1)	$1.94 \cdot 10^4$	$3.99 \cdot 10^4$	$1.94 \cdot 10^4$	$3.99 \cdot 10^4$	1.00	0.00
User_1.calling.ringback (p:0, id:2)	$1.94 \cdot 10^4$	$3.99 \cdot 10^4$	$1.94 \cdot 10^4$	$3.99 \cdot 10^4$	1.00	0.00
User_1.calling.busy (p:0, id:3)	$1.70 \cdot 10^5$	$3.49 \cdot 10^5$	$8.49 \cdot 10^4$	$1.74 \cdot 10^5$	$5.00 \cdot 10^{-1}$	$2.43 \cdot 10^{-2}$
User_1.calling.calling (p:0, id:4)	$8.50 \cdot 10^4$	$1.75 \cdot 10^5$	$8.50 \cdot 10^4$	$1.75 \cdot 10^5$	1.00	0.00
User_1.calling.oalert (p:0, id:5)	0.00	0.00	0.00	0.00	NaN	NaN
User_1.dialing	$2.33 \cdot 10^5$	$4.78 \cdot 10^5$	$2.33 \cdot 10^5$	$4.78 \cdot 10^5$	1.00	0.00
User_1.dialing.idle (p:0, id:0)	$3.88 \cdot 10^4$	$7.97 \cdot 10^4$	$3.88 \cdot 10^4$	$7.97 \cdot 10^4$	1.00	0.00
User_1.dialing.calling (p:0, id:1)	$3.88 \cdot 10^4$	$7.97 \cdot 10^4$	$3.88 \cdot 10^4$	$7.97 \cdot 10^4$	1.00	0.00
User_1.dialing.calling (p:0, id:2)	$3.88 \cdot 10^4$	$7.97 \cdot 10^4$	$3.88 \cdot 10^4$	$7.97 \cdot 10^4$	1.00	0.00
User_1.dialing.calling (p:0, id:3)	$3.88 \cdot 10^4$	$7.97 \cdot 10^4$	$3.88 \cdot 10^4$	$7.97 \cdot 10^4$	1.00	0.00
User_1.dialing.calling (p:0, id:4)	$3.87 \cdot 10^4$	$7.95 \cdot 10^4$	$3.87 \cdot 10^4$	$7.95 \cdot 10^4$	1.00	0.00
User_1.dialing.calling (p:0, id:5)	$3.89 \cdot 10^4$	$7.98 \cdot 10^4$	$3.89 \cdot 10^4$	$7.98 \cdot 10^4$	1.00	0.00
User_1.dveoringout	0.00	0.00	0.00	0.00	NaN	NaN
User_1.dveoringout.idle (p:0, id:0)	0.00	0.00	0.00	0.00	NaN	NaN
User_1.idle	$2.06 \cdot 10^6$	$4.68 \cdot 10^5$	$2.06 \cdot 10^6$	$4.68 \cdot 10^5$	1.00	$2.17 \cdot 10^{-7}$
User_1.idle.dialing (p:0, id:0)	$2.06 \cdot 10^6$	$4.68 \cdot 10^5$	$2.33 \cdot 10^5$	$4.78 \cdot 10^5$	$2.00 \cdot 10^{-1}$	$4.10 \cdot 10^{-1}$
User_1.idle.qi (p:0, id:1)	$1.83 \cdot 10^6$	$9.41 \cdot 10^5$	$1.83 \cdot 10^6$	$9.41 \cdot 10^5$	1.00	$2.00 \cdot 10^{-7}$
User_1.oalert	0.00	0.00	0.00	0.00	NaN	NaN
User_1.oalert.dveoringout (p:0, id:2)	0.00	0.00	0.00	0.00	NaN	NaN
User_1.qi	$1.83 \cdot 10^6$	$9.41 \cdot 10^5$	$1.83 \cdot 10^6$	$9.41 \cdot 10^5$	1.00	$1.97 \cdot 10^{-7}$
User_1.qi.talert (p:0, id:0)	$1.83 \cdot 10^6$	$9.41 \cdot 10^5$	0.00	0.00	0.00	0.00
User_1.qi.idle (p:0, id:1)	$1.83 \cdot 10^6$	$9.41 \cdot 10^5$	$1.83 \cdot 10^6$	$9.41 \cdot 10^5$	1.00	$1.52 \cdot 10^{-7}$
User_1.ringback	$1.94 \cdot 10^4$	$3.99 \cdot 10^4$	$1.94 \cdot 10^4$	$3.99 \cdot 10^4$	1.00	0.00
User_1.ringback.idle (p:0, id:0)	$9.68 \cdot 10^3$	$1.99 \cdot 10^4$	$9.68 \cdot 10^3$	$1.99 \cdot 10^4$	1.00	0.00
User_1.ringback.calling (p:0, id:1)	$9.74 \cdot 10^3$	$2.00 \cdot 10^4$	$9.74 \cdot 10^3$	$2.00 \cdot 10^4$	1.00	0.00
User_1.talert	0.00	0.00	0.00	0.00	NaN	NaN
User_1.talert.tpickup (p:0, id:1)	0.00	0.00	0.00	0.00	NaN	NaN
User_1.talert.idle (p:0, id:2)	0.00	0.00	0.00	0.00	NaN	NaN
User_1.tpickup	0.00	0.00	0.00	0.00	NaN	NaN
User_1.tpickup.idle (p:0, id:1)	0.00	0.00	0.00	0.00	NaN	NaN
User_1.unobtainable	$1.94 \cdot 10^4$	$3.99 \cdot 10^4$	$1.94 \cdot 10^4$	$3.99 \cdot 10^4$	1.00	0.00
User_1.unobtainable.idle (p:0, id:0)	$1.94 \cdot 10^4$	$3.99 \cdot 10^4$	$1.94 \cdot 10^4$	$3.99 \cdot 10^4$	1.00	0.00

Table C.14: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model Telephony (state machine User_1 only). The Java code has been generated with the 'Random + Det' decision enabled and the results have been attained through logging-based measurements. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

Performance results for target model 'Telephony' ($n = 20, t = 30$, Logging, User_2)						
Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
`User_2.busy`	$1.53 \cdot 10^5$	$3.15 \cdot 10^5$	$1.53 \cdot 10^5$	$3.15 \cdot 10^5$	1.00	0.00
User_2.busy.idle (p:0, id:0)	$1.53 \cdot 10^5$	$3.15 \cdot 10^5$	$1.53 \cdot 10^5$	$3.15 \cdot 10^5$	1.00	0.00
`User_2.calling`	$3.08 \cdot 10^5$	$6.33 \cdot 10^5$	$3.08 \cdot 10^5$	$6.33 \cdot 10^5$	1.00	$3.38 \cdot 10^{-7}$
User_2.calling.busy (p:0, id:0)	$3.08 \cdot 10^5$	$6.33 \cdot 10^5$	$1.15 \cdot 10^5$	$2.35 \cdot 10^5$	$3.72 \cdot 10^{-1}$	$7.66 \cdot 10^{-3}$
User_2.calling.unobtainable (p:0, id:1)	$1.80 \cdot 10^4$	$3.70 \cdot 10^4$	$1.80 \cdot 10^4$	$3.70 \cdot 10^4$	1.00	0.00
User_2.calling.ringback (p:0, id:2)	$1.80 \cdot 10^4$	$3.70 \cdot 10^4$	$1.80 \cdot 10^4$	$3.70 \cdot 10^4$	1.00	0.00
User_2.calling.busy (p:0, id:3)	$1.57 \cdot 10^5$	$3.24 \cdot 10^5$	$3.85 \cdot 10^4$	$8.05 \cdot 10^4$	$2.43 \cdot 10^{-1}$	$3.42 \cdot 10^{-2}$
User_2.calling.calling (p:0, id:4)	$1.19 \cdot 10^5$	$2.44 \cdot 10^5$	$1.19 \cdot 10^5$	$2.44 \cdot 10^5$	1.00	0.00
User_2.calling.oalert (p:0, id:5)	0.00	0.00	0.00	0.00	NaN	NaN
`User_2.dialing`	$2.16 \cdot 10^5$	$4.45 \cdot 10^5$	$2.16 \cdot 10^5$	$4.45 \cdot 10^5$	1.00	$4.68 \cdot 10^{-7}$
User_2.dialing.idle (p:0, id:0)	$3.61 \cdot 10^4$	$7.42 \cdot 10^4$	$3.61 \cdot 10^4$	$7.42 \cdot 10^4$	1.00	0.00
User_2.dialing.calling (p:0, id:1)	$3.60 \cdot 10^4$	$7.41 \cdot 10^4$	$3.60 \cdot 10^4$	$7.41 \cdot 10^4$	1.00	0.00
User_2.dialing.calling (p:0, id:2)	$3.60 \cdot 10^4$	$7.41 \cdot 10^4$	$3.60 \cdot 10^4$	$7.41 \cdot 10^4$	1.00	0.00
User_2.dialing.calling (p:0, id:3)	$3.60 \cdot 10^4$	$7.41 \cdot 10^4$	$3.60 \cdot 10^4$	$7.41 \cdot 10^4$	1.00	0.00
User_2.dialing.calling (p:0, id:4)	$3.60 \cdot 10^4$	$7.40 \cdot 10^4$	$3.60 \cdot 10^4$	$7.40 \cdot 10^4$	1.00	0.00
User_2.dialing.calling (p:0, id:5)	$3.60 \cdot 10^4$	$7.41 \cdot 10^4$	$3.60 \cdot 10^4$	$7.41 \cdot 10^4$	1.00	0.00
`User_2.dveoringout`	0.00	0.00	0.00	0.00	NaN	NaN
User_2.dveoringout.idle (p:0, id:0)	0.00	0.00	0.00	0.00	NaN	NaN
`User_2.idle`	$2.05 \cdot 10^6$	$5.10 \cdot 10^5$	$2.05 \cdot 10^6$	$5.10 \cdot 10^5$	1.00	$2.16 \cdot 10^{-7}$
User_2.idle.dialing (p:0, id:0)	$2.05 \cdot 10^6$	$5.10 \cdot 10^5$	$2.16 \cdot 10^5$	$4.45 \cdot 10^5$	$2.00 \cdot 10^{-1}$	$4.10 \cdot 10^{-1}$
User_2.idle.qi (p:0, id:1)	$1.83 \cdot 10^6$	$9.46 \cdot 10^5$	$1.83 \cdot 10^6$	$9.46 \cdot 10^5$	1.00	$1.13 \cdot 10^{-7}$
`User_2.oalert`	0.00	0.00	0.00	0.00	NaN	NaN
User_2.oalert.dveoringout (p:0, id:2)	0.00	0.00	0.00	0.00	NaN	NaN
`User_2.qi`	$1.83 \cdot 10^6$	$9.46 \cdot 10^5$	$1.83 \cdot 10^6$	$9.46 \cdot 10^5$	1.00	$2.22 \cdot 10^{-7}$
User_2.qi.talert (p:0, id:0)	$1.83 \cdot 10^6$	$9.46 \cdot 10^5$	0.00	0.00	0.00	0.00
User_2.qi.idle (p:0, id:1)	$1.83 \cdot 10^6$	$9.46 \cdot 10^5$	$1.83 \cdot 10^6$	$9.46 \cdot 10^5$	1.00	0.00
`User_2.ringback`	$1.80 \cdot 10^4$	$3.70 \cdot 10^4$	$1.80 \cdot 10^4$	$3.70 \cdot 10^4$	1.00	0.00
User_2.ringback.idle (p:0, id:0)	$9.01 \cdot 10^3$	$1.85 \cdot 10^4$	$9.01 \cdot 10^3$	$1.85 \cdot 10^4$	1.00	0.00
User_2.ringback.calling (p:0, id:1)	$8.99 \cdot 10^3$	$1.85 \cdot 10^4$	$8.99 \cdot 10^3$	$1.85 \cdot 10^4$	1.00	0.00
`User_2.talert`	0.00	0.00	0.00	0.00	NaN	NaN
User_2.talert.tpickup (p:0, id:1)	0.00	0.00	0.00	0.00	NaN	NaN
User_2.talert.idle (p:0, id:2)	0.00	0.00	0.00	0.00	NaN	NaN
`User_2.tpickup`	0.00	0.00	0.00	0.00	NaN	NaN
User_2.tpickup.idle (p:0, id:1)	0.00	0.00	0.00	0.00	NaN	NaN
`User_2.unobtainable`	$1.80 \cdot 10^4$	$3.70 \cdot 10^4$	$1.80 \cdot 10^4$	$3.70 \cdot 10^4$	1.00	0.00
User_2.unobtainable.idle (p:0, id:0)	$1.80 \cdot 10^4$	$3.70 \cdot 10^4$	$1.80 \cdot 10^4$	$3.70 \cdot 10^4$	1.00	0.00

Table C.15: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model `Telephony` (state machine `User_2` only). The Java code has been generated with the 'Random + Det' decision enabled and the results have been attained through logging-based measurements. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

Performance results for target model 'Telephony' ($n = 20, t = 30$, Logging, User_3)						
Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
User_3.busy	$2.00 \cdot 10^5$	$3.14 \cdot 10^5$	$2.00 \cdot 10^5$	$3.14 \cdot 10^5$	1.00	$5.98 \cdot 10^{-7}$
User_3.busy.idle (p:0, id:0)	$2.00 \cdot 10^5$	$3.14 \cdot 10^5$	$2.00 \cdot 10^5$	$3.14 \cdot 10^5$	1.00	$5.98 \cdot 10^{-7}$
User_3.calling	$5.47 \cdot 10^5$	$8.59 \cdot 10^5$	$5.47 \cdot 10^5$	$8.59 \cdot 10^5$	1.00	$2.90 \cdot 10^{-7}$
User_3.calling.busy (p:0, id:0)	$5.47 \cdot 10^5$	$8.59 \cdot 10^5$	$2.00 \cdot 10^5$	$3.14 \cdot 10^5$	$3.66 \cdot 10^{-1}$	$6.47 \cdot 10^{-3}$
User_3.calling.unobtainable (p:0, id:1)	$2.36 \cdot 10^4$	$3.69 \cdot 10^4$	$2.36 \cdot 10^4$	$3.69 \cdot 10^4$	1.00	0.00
User_3.calling.ringback (p:0, id:2)	$2.36 \cdot 10^4$	$3.69 \cdot 10^4$	$2.36 \cdot 10^4$	$3.69 \cdot 10^4$	1.00	0.00
User_3.calling.calling (p:0, id:4)	$3.00 \cdot 10^5$	$4.71 \cdot 10^5$	$3.00 \cdot 10^5$	$4.71 \cdot 10^5$	1.00	$4.31 \cdot 10^{-7}$
User_3.calling.oalert (p:0, id:5)	0.00	0.00	0.00	0.00	NaN	NaN
User_3.dialing	$2.83 \cdot 10^5$	$4.43 \cdot 10^5$	$2.83 \cdot 10^5$	$4.43 \cdot 10^5$	1.00	$4.46 \cdot 10^{-7}$
User_3.dialing.idle (p:0, id:0)	$4.72 \cdot 10^4$	$7.40 \cdot 10^4$	$4.72 \cdot 10^4$	$7.40 \cdot 10^4$	1.00	0.00
User_3.dialing.calling (p:0, id:1)	$4.71 \cdot 10^4$	$7.39 \cdot 10^4$	$4.71 \cdot 10^4$	$7.39 \cdot 10^4$	1.00	0.00
User_3.dialing.calling (p:0, id:2)	$4.70 \cdot 10^4$	$7.37 \cdot 10^4$	$4.70 \cdot 10^4$	$7.37 \cdot 10^4$	1.00	0.00
User_3.dialing.calling (p:0, id:3)	$4.71 \cdot 10^4$	$7.39 \cdot 10^4$	$4.71 \cdot 10^4$	$7.39 \cdot 10^4$	1.00	0.00
User_3.dialing.calling (p:0, id:4)	$4.71 \cdot 10^4$	$7.39 \cdot 10^4$	$4.71 \cdot 10^4$	$7.39 \cdot 10^4$	1.00	0.00
User_3.dialing.calling (p:0, id:5)	$4.71 \cdot 10^4$	$7.39 \cdot 10^4$	$4.71 \cdot 10^4$	$7.39 \cdot 10^4$	1.00	0.00
User_3.dveoringout	0.00	0.00	0.00	0.00	NaN	NaN
User_3.dveoringout.idle (p:0, id:0)	0.00	0.00	0.00	0.00	NaN	NaN
User_3.idle	$1.87 \cdot 10^6$	$6.40 \cdot 10^5$	$1.87 \cdot 10^6$	$6.40 \cdot 10^5$	1.00	$2.95 \cdot 10^{-7}$
User_3.idle.dialing (p:0, id:0)	$1.87 \cdot 10^6$	$6.40 \cdot 10^5$	$2.83 \cdot 10^5$	$4.43 \cdot 10^5$	$3.00 \cdot 10^{-1}$	$4.70 \cdot 10^{-1}$
User_3.idle.qi (p:0, id:1)	$1.59 \cdot 10^6$	$1.08 \cdot 10^6$	$1.59 \cdot 10^6$	$1.08 \cdot 10^6$	1.00	$1.58 \cdot 10^{-7}$
User_3.oalert	0.00	0.00	0.00	0.00	NaN	NaN
User_3.oalert.dveoringout (p:0, id:2)	0.00	0.00	0.00	0.00	NaN	NaN
User_3.qi	$1.59 \cdot 10^6$	$1.08 \cdot 10^6$	$1.59 \cdot 10^6$	$1.08 \cdot 10^6$	1.00	$2.34 \cdot 10^{-7}$
User_3.qi.talert (p:0, id:0)	$1.59 \cdot 10^6$	$1.08 \cdot 10^6$	0.00	0.00	0.00	0.00
User_3.qi.idle (p:0, id:1)	$1.59 \cdot 10^6$	$1.08 \cdot 10^6$	$1.59 \cdot 10^6$	$1.08 \cdot 10^6$	1.00	$1.62 \cdot 10^{-7}$
User_3.ringback	$2.36 \cdot 10^4$	$3.69 \cdot 10^4$	$2.36 \cdot 10^4$	$3.69 \cdot 10^4$	1.00	0.00
User_3.ringback.idle (p:0, id:0)	$1.18 \cdot 10^4$	$1.85 \cdot 10^4$	$1.18 \cdot 10^4$	$1.85 \cdot 10^4$	1.00	0.00
User_3.ringback.calling (p:0, id:1)	$1.18 \cdot 10^4$	$1.84 \cdot 10^4$	$1.18 \cdot 10^4$	$1.84 \cdot 10^4$	1.00	0.00
User_3.talert	0.00	0.00	0.00	0.00	NaN	NaN
User_3.talert.tpickup (p:0, id:1)	0.00	0.00	0.00	0.00	NaN	NaN
User_3.talert.idle (p:0, id:2)	0.00	0.00	0.00	0.00	NaN	NaN
User_3.tconnected	0.00	0.00	0.00	0.00	NaN	NaN
User_3.tconnected.tconnected (p:0, id:0)	0.00	0.00	0.00	0.00	NaN	NaN
User_3.tconnected.tconnected (p:0, id:1)	0.00	0.00	0.00	0.00	NaN	NaN
User_3.tpickup	0.00	0.00	0.00	0.00	NaN	NaN
User_3.tpickup.tconnected (p:0, id:0)	0.00	0.00	0.00	0.00	NaN	NaN
User_3.tpickup.idle (p:0, id:1)	0.00	0.00	0.00	0.00	NaN	NaN
User_3.unobtainable	$2.36 \cdot 10^4$	$3.69 \cdot 10^4$	$2.36 \cdot 10^4$	$3.69 \cdot 10^4$	1.00	0.00
User_3.unobtainable.idle (p:0, id:0)	$2.36 \cdot 10^4$	$3.69 \cdot 10^4$	$2.36 \cdot 10^4$	$3.69 \cdot 10^4$	1.00	0.00

Table C.16: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model Telephony (state machine User_3 only). The Java code has been generated with the 'Random + Det' decision enabled and the results have been attained through logging-based measurements. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

C.2 Decision Mode Frequency Tables

C.2.1 Synthetic Test: CounterDistributor

Performance results for target model '`CounterDistributor`' ($n = 20, t = 30$, Random + Det)						
Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
`Counter.C`	$1.22 \cdot 10^8$	$2.27 \cdot 10^7$	$1.22 \cdot 10^8$	$2.27 \cdot 10^7$	1.00	0.00
Counter.C.C (p:0, id:0)	$1.22 \cdot 10^8$	$2.27 \cdot 10^7$	$1.22 \cdot 10^8$	$2.27 \cdot 10^7$	1.00	0.00
`Distributor.P`	$1.14 \cdot 10^8$	$1.52 \cdot 10^7$	$1.14 \cdot 10^8$	$1.52 \cdot 10^7$	1.00	0.00
Distributor.P.P (p:0, id:0)	$1.14 \cdot 10^8$	$1.52 \cdot 10^7$	$1.13 \cdot 10^7$	$1.53 \cdot 10^6$	$9.88 \cdot 10^{-2}$	$3.13 \cdot 10^{-3}$
Distributor.P.P (p:0, id:1)	$1.03 \cdot 10^8$	$1.37 \cdot 10^7$	$1.13 \cdot 10^7$	$1.45 \cdot 10^6$	$1.10 \cdot 10^{-1}$	$3.68 \cdot 10^{-3}$
Distributor.P.P (p:0, id:2)	$9.16 \cdot 10^7$	$1.23 \cdot 10^7$	$1.13 \cdot 10^7$	$1.48 \cdot 10^6$	$1.23 \cdot 10^{-1}$	$3.77 \cdot 10^{-3}$
Distributor.P.P (p:0, id:3)	$8.03 \cdot 10^7$	$1.09 \cdot 10^7$	$1.14 \cdot 10^7$	$1.54 \cdot 10^6$	$1.42 \cdot 10^{-1}$	$3.48 \cdot 10^{-3}$
Distributor.P.P (p:0, id:4)	$6.89 \cdot 10^7$	$9.35 \cdot 10^6$	$1.15 \cdot 10^7$	$1.55 \cdot 10^6$	$1.66 \cdot 10^{-1}$	$3.01 \cdot 10^{-3}$
Distributor.P.P (p:0, id:5)	$5.75 \cdot 10^7$	$7.82 \cdot 10^6$	$1.15 \cdot 10^7$	$1.57 \cdot 10^6$	$2.00 \cdot 10^{-1}$	$4.43 \cdot 10^{-3}$
Distributor.P.P (p:0, id:6)	$4.60 \cdot 10^7$	$6.26 \cdot 10^6$	$1.15 \cdot 10^7$	$1.55 \cdot 10^6$	$2.50 \cdot 10^{-1}$	$8.34 \cdot 10^{-3}$
Distributor.P.P (p:0, id:7)	$3.45 \cdot 10^7$	$4.78 \cdot 10^6$	$1.14 \cdot 10^7$	$1.60 \cdot 10^6$	$3.32 \cdot 10^{-1}$	$6.85 \cdot 10^{-3}$
Distributor.P.P (p:0, id:8)	$2.30 \cdot 10^7$	$3.20 \cdot 10^6$	$1.15 \cdot 10^7$	$1.56 \cdot 10^6$	$4.99 \cdot 10^{-1}$	$4.99 \cdot 10^{-3}$
Distributor.P.P (p:0, id:9)	$1.15 \cdot 10^7$	$1.65 \cdot 10^6$	$1.15 \cdot 10^7$	$1.65 \cdot 10^6$	1.00	0.00

Table C.17: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model `CounterDistributor`. The Java code has been generated with the 'Random + Det' decision mode enabled. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

Performance results for target model '`CounterDistributor`' ($n = 20, t = 30$, Sequential)						
Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
`Counter.C`	$1.26 \cdot 10^8$	$1.13 \cdot 10^7$	$1.26 \cdot 10^8$	$1.13 \cdot 10^7$	1.00	0.00
Counter.C.C (p:0, id:0)	$1.26 \cdot 10^8$	$1.13 \cdot 10^7$	$1.26 \cdot 10^8$	$1.13 \cdot 10^7$	1.00	0.00
`Distributor.P`	$1.02 \cdot 10^8$	$8.71 \cdot 10^6$	$1.02 \cdot 10^8$	$8.71 \cdot 10^6$	1.00	0.00
Distributor.P.P (p:0, id:0)	$1.02 \cdot 10^8$	$8.71 \cdot 10^6$	$1.00 \cdot 10^7$	$8.64 \cdot 10^5$	$9.83 \cdot 10^{-2}$	$2.45 \cdot 10^{-3}$
Distributor.P.P (p:0, id:1)	$9.22 \cdot 10^7$	$7.88 \cdot 10^6$	$1.02 \cdot 10^7$	$8.59 \cdot 10^5$	$1.11 \cdot 10^{-1}$	$2.71 \cdot 10^{-3}$
Distributor.P.P (p:0, id:2)	$8.20 \cdot 10^7$	$7.06 \cdot 10^6$	$1.02 \cdot 10^7$	$8.74 \cdot 10^5$	$1.25 \cdot 10^{-1}$	$3.08 \cdot 10^{-3}$
Distributor.P.P (p:0, id:3)	$7.18 \cdot 10^7$	$6.23 \cdot 10^6$	$1.03 \cdot 10^7$	$8.27 \cdot 10^5$	$1.44 \cdot 10^{-1}$	$4.01 \cdot 10^{-3}$
Distributor.P.P (p:0, id:4)	$6.14 \cdot 10^7$	$5.45 \cdot 10^6$	$1.03 \cdot 10^7$	$8.72 \cdot 10^5$	$1.69 \cdot 10^{-1}$	$4.60 \cdot 10^{-3}$
Distributor.P.P (p:0, id:5)	$5.11 \cdot 10^7$	$4.63 \cdot 10^6$	$1.03 \cdot 10^7$	$9.15 \cdot 10^5$	$2.01 \cdot 10^{-1}$	$4.64 \cdot 10^{-3}$
Distributor.P.P (p:0, id:6)	$4.08 \cdot 10^7$	$3.75 \cdot 10^6$	$1.02 \cdot 10^7$	$9.40 \cdot 10^5$	$2.51 \cdot 10^{-1}$	$5.52 \cdot 10^{-3}$
Distributor.P.P (p:0, id:7)	$3.06 \cdot 10^7$	$2.85 \cdot 10^6$	$1.02 \cdot 10^7$	$9.73 \cdot 10^5$	$3.33 \cdot 10^{-1}$	$4.68 \cdot 10^{-3}$
Distributor.P.P (p:0, id:8)	$2.04 \cdot 10^7$	$1.89 \cdot 10^6$	$1.02 \cdot 10^7$	$9.48 \cdot 10^5$	$4.99 \cdot 10^{-1}$	$5.21 \cdot 10^{-3}$
Distributor.P.P (p:0, id:9)	$1.02 \cdot 10^7$	$9.56 \cdot 10^5$	$1.02 \cdot 10^7$	$9.56 \cdot 10^5$	1.00	0.00

Table C.18: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model `CounterDistributor`. The Java code has been generated with the 'Sequential' decision mode enabled. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

Performance results for target model '`CounterDistributor`' ($n = 20, t = 30,$ Random)						
Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
`Counter.C`	$1.26 \cdot 10^8$	$1.75 \cdot 10^7$	$1.26 \cdot 10^8$	$1.75 \cdot 10^7$	1.00	0.00
Counter.C.C (p:0, id:0)	$1.26 \cdot 10^8$	$1.75 \cdot 10^7$	$1.26 \cdot 10^8$	$1.75 \cdot 10^7$	1.00	0.00
`Distributor.P`	$9.95 \cdot 10^7$	$1.59 \cdot 10^7$	$9.96 \cdot 10^6$	$1.59 \cdot 10^6$	$1.00 \cdot 10^{-1}$	$1.63 \cdot 10^{-4}$
Distributor.P.P (p:0, id:0)	$9.95 \cdot 10^6$	$1.59 \cdot 10^6$	$9.96 \cdot 10^5$	$1.60 \cdot 10^5$	$1.00 \cdot 10^{-1}$	$4.51 \cdot 10^{-4}$
Distributor.P.P (p:0, id:1)	$9.96 \cdot 10^6$	$1.59 \cdot 10^6$	$9.95 \cdot 10^5$	$1.58 \cdot 10^5$	$1.00 \cdot 10^{-1}$	$3.60 \cdot 10^{-4}$
Distributor.P.P (p:0, id:2)	$9.95 \cdot 10^6$	$1.59 \cdot 10^6$	$9.96 \cdot 10^5$	$1.58 \cdot 10^5$	$1.00 \cdot 10^{-1}$	$3.03 \cdot 10^{-4}$
Distributor.P.P (p:0, id:3)	$9.95 \cdot 10^6$	$1.59 \cdot 10^6$	$9.95 \cdot 10^5$	$1.59 \cdot 10^5$	$1.00 \cdot 10^{-1}$	$3.13 \cdot 10^{-4}$
Distributor.P.P (p:0, id:4)	$9.95 \cdot 10^6$	$1.59 \cdot 10^6$	$9.95 \cdot 10^5$	$1.58 \cdot 10^5$	$1.00 \cdot 10^{-1}$	$2.92 \cdot 10^{-4}$
Distributor.P.P (p:0, id:5)	$9.95 \cdot 10^6$	$1.59 \cdot 10^6$	$9.96 \cdot 10^5$	$1.59 \cdot 10^5$	$1.00 \cdot 10^{-1}$	$2.48 \cdot 10^{-4}$
Distributor.P.P (p:0, id:6)	$9.95 \cdot 10^6$	$1.59 \cdot 10^6$	$9.96 \cdot 10^5$	$1.59 \cdot 10^5$	$1.00 \cdot 10^{-1}$	$2.77 \cdot 10^{-4}$
Distributor.P.P (p:0, id:7)	$9.95 \cdot 10^6$	$1.59 \cdot 10^6$	$9.96 \cdot 10^5$	$1.59 \cdot 10^5$	$1.00 \cdot 10^{-1}$	$3.05 \cdot 10^{-4}$
Distributor.P.P (p:0, id:8)	$9.95 \cdot 10^6$	$1.59 \cdot 10^6$	$9.95 \cdot 10^5$	$1.59 \cdot 10^5$	$1.00 \cdot 10^{-1}$	$3.20 \cdot 10^{-4}$
Distributor.P.P (p:0, id:9)	$9.95 \cdot 10^6$	$1.59 \cdot 10^6$	$9.97 \cdot 10^5$	$1.59 \cdot 10^5$	$1.00 \cdot 10^{-1}$	$2.51 \cdot 10^{-4}$

Table C.19: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model `CounterDistributor`. The Java code has been generated with the 'Random' decision mode enabled. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

C.2.2 Synthetic Test: Tokens

Performance results for target model 'Tokens' ($n = 20, t = 30$, Random + Det)						
Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
`A.act`	$7.26 \cdot 10^7$	$8.96 \cdot 10^6$	$3.63 \cdot 10^7$	$4.49 \cdot 10^6$	$5.00 \cdot 10^{-1}$	$1.11 \cdot 10^{-4}$
A.act.act (p:0, id:0)	$3.63 \cdot 10^7$	$4.48 \cdot 10^6$	$2.70 \cdot 10^6$	$3.02 \cdot 10^5$	$7.48 \cdot 10^{-2}$	$8.51 \cdot 10^{-3}$
A.act.update (p:1, id:1)	$3.63 \cdot 10^7$	$4.48 \cdot 10^6$	$3.02 \cdot 10^7$	$3.91 \cdot 10^6$	$8.33 \cdot 10^{-1}$	$7.69 \cdot 10^{-3}$
A.act.wait (p:1, id:2)	$6.06 \cdot 10^6$	$6.25 \cdot 10^5$	$3.37 \cdot 10^6$	$4.35 \cdot 10^5$	$5.55 \cdot 10^{-1}$	$3.03 \cdot 10^{-2}$
`A.update`	$3.36 \cdot 10^7$	$4.35 \cdot 10^6$	$3.36 \cdot 10^7$	$4.35 \cdot 10^6$	1.00	0.00
A.update.act (p:0, id:0)	$3.36 \cdot 10^7$	$4.35 \cdot 10^6$	$3.36 \cdot 10^7$	$4.35 \cdot 10^6$	1.00	0.00
`A.wait`	$2.66 \cdot 10^7$	$4.52 \cdot 10^6$	$3.94 \cdot 10^6$	$4.79 \cdot 10^5$	$1.50 \cdot 10^{-1}$	$1.92 \cdot 10^{-2}$
A.wait.wait (p:0, id:0)	$1.33 \cdot 10^7$	$2.26 \cdot 10^6$	$5.79 \cdot 10^5$	$8.34 \cdot 10^4$	$4.39 \cdot 10^{-2}$	$5.44 \cdot 10^{-3}$
A.wait.update (p:1, id:1)	$1.33 \cdot 10^7$	$2.26 \cdot 10^6$	$3.37 \cdot 10^6$	$4.35 \cdot 10^5$	$2.57 \cdot 10^{-1}$	$3.59 \cdot 10^{-2}$
`B.act`	$9.05 \cdot 10^7$	$8.25 \cdot 10^6$	$4.53 \cdot 10^7$	$4.12 \cdot 10^6$	$5.00 \cdot 10^{-1}$	$7.88 \cdot 10^{-5}$
B.act.act (p:0, id:0)	$4.53 \cdot 10^7$	$4.12 \cdot 10^6$	$2.79 \cdot 10^6$	$3.99 \cdot 10^5$	$6.18 \cdot 10^{-2}$	$6.74 \cdot 10^{-3}$
B.act.update (p:1, id:1)	$4.53 \cdot 10^7$	$4.12 \cdot 10^6$	$3.82 \cdot 10^7$	$3.50 \cdot 10^6$	$8.44 \cdot 10^{-1}$	$6.09 \cdot 10^{-3}$
B.act.wait (p:1, id:2)	$7.05 \cdot 10^6$	$6.95 \cdot 10^5$	$4.25 \cdot 10^6$	$3.90 \cdot 10^5$	$6.04 \cdot 10^{-1}$	$2.77 \cdot 10^{-2}$
`B.update`	$4.25 \cdot 10^7$	$3.89 \cdot 10^6$	$4.25 \cdot 10^7$	$3.89 \cdot 10^6$	1.00	0.00
B.update.act (p:0, id:0)	$4.25 \cdot 10^7$	$3.89 \cdot 10^6$	$4.25 \cdot 10^7$	$3.89 \cdot 10^6$	1.00	0.00
`B.wait`	$2.82 \cdot 10^7$	$4.89 \cdot 10^6$	$4.83 \cdot 10^6$	$4.41 \cdot 10^5$	$1.74 \cdot 10^{-1}$	$2.33 \cdot 10^{-2}$
B.wait.wait (p:0, id:0)	$1.41 \cdot 10^7$	$2.45 \cdot 10^6$	$5.72 \cdot 10^5$	$6.80 \cdot 10^4$	$4.11 \cdot 10^{-2}$	$5.28 \cdot 10^{-3}$
B.wait.update (p:1, id:1)	$1.41 \cdot 10^7$	$2.45 \cdot 10^6$	$4.25 \cdot 10^6$	$3.90 \cdot 10^5$	$3.07 \cdot 10^{-1}$	$4.23 \cdot 10^{-2}$
`C.act`	$7.48 \cdot 10^7$	$7.25 \cdot 10^6$	$3.64 \cdot 10^7$	$3.56 \cdot 10^6$	$4.87 \cdot 10^{-1}$	$2.35 \cdot 10^{-3}$
C.act.act (p:0, id:0)	$3.74 \cdot 10^7$	$3.63 \cdot 10^6$	$3.74 \cdot 10^6$	$3.47 \cdot 10^5$	$1.01 \cdot 10^{-1}$	$1.02 \cdot 10^{-2}$
C.act.wait (p:1, id:1)	$3.74 \cdot 10^7$	$3.62 \cdot 10^6$	$3.28 \cdot 10^6$	$3.44 \cdot 10^5$	$8.75 \cdot 10^{-2}$	$1.30 \cdot 10^{-3}$
C.act.update (p:1, id:2)	$3.41 \cdot 10^7$	$3.28 \cdot 10^6$	$2.94 \cdot 10^7$	$3.09 \cdot 10^6$	$8.61 \cdot 10^{-1}$	$1.40 \cdot 10^{-2}$
`C.update`	$3.27 \cdot 10^7$	$3.43 \cdot 10^6$	$3.27 \cdot 10^7$	$3.43 \cdot 10^6$	1.00	0.00
C.update.act (p:0, id:0)	$3.27 \cdot 10^7$	$3.43 \cdot 10^6$	$3.27 \cdot 10^7$	$3.43 \cdot 10^6$	1.00	0.00
`C.wait`	$2.07 \cdot 10^7$	$2.71 \cdot 10^6$	$3.79 \cdot 10^6$	$3.26 \cdot 10^5$	$1.85 \cdot 10^{-1}$	$1.71 \cdot 10^{-2}$
C.wait.wait (p:0, id:0)	$1.03 \cdot 10^7$	$1.36 \cdot 10^6$	$5.10 \cdot 10^5$	$1.07 \cdot 10^5$	$4.97 \cdot 10^{-2}$	$1.01 \cdot 10^{-2}$
C.wait.update (p:1, id:1)	$1.03 \cdot 10^7$	$1.35 \cdot 10^6$	$3.28 \cdot 10^6$	$3.44 \cdot 10^5$	$3.20 \cdot 10^{-1}$	$3.40 \cdot 10^{-2}$

Table C.20: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model `Tokens`. The Java code has been generated with the 'Random + Det' decision mode enabled. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

Performance results for target model 'Tokens' ($n = 20, t = 30$, Sequential)						
Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
`A.act`	$4.47 \cdot 10^7$	$5.46 \cdot 10^6$	$4.47 \cdot 10^7$	$5.46 \cdot 10^6$	1.00	0.00
A.act.act (p:0, id:0)	$4.47 \cdot 10^7$	$5.46 \cdot 10^6$	$3.23 \cdot 10^6$	$5.00 \cdot 10^5$	$7.22 \cdot 10^{-2}$	$5.99 \cdot 10^{-3}$
A.act.update (p:1, id:1)	$4.15 \cdot 10^7$	$5.04 \cdot 10^6$	$3.73 \cdot 10^7$	$4.54 \cdot 10^6$	$9.00 \cdot 10^{-1}$	$4.15 \cdot 10^{-8}$
A.act.wait (p:1, id:2)	$4.16 \cdot 10^6$	$5.05 \cdot 10^5$	$4.16 \cdot 10^6$	$5.05 \cdot 10^5$	1.00	0.00
`A.update`	$4.15 \cdot 10^7$	$5.04 \cdot 10^6$	$4.15 \cdot 10^7$	$5.04 \cdot 10^6$	1.00	0.00
A.update.act (p:0, id:0)	$4.15 \cdot 10^7$	$5.04 \cdot 10^6$	$4.15 \cdot 10^7$	$5.04 \cdot 10^6$	1.00	0.00
`A.wait`	$1.75 \cdot 10^7$	$2.56 \cdot 10^6$	$5.04 \cdot 10^6$	$5.40 \cdot 10^5$	$2.93 \cdot 10^{-1}$	$4.69 \cdot 10^{-2}$
A.wait.wait (p:0, id:0)	$1.75 \cdot 10^7$	$2.56 \cdot 10^6$	$8.83 \cdot 10^5$	$1.18 \cdot 10^5$	$5.08 \cdot 10^{-2}$	$5.32 \cdot 10^{-3}$
A.wait.update (p:1, id:1)	$1.66 \cdot 10^7$	$2.47 \cdot 10^6$	$4.16 \cdot 10^6$	$5.05 \cdot 10^5$	$2.55 \cdot 10^{-1}$	$4.74 \cdot 10^{-2}$
`B.act`	$6.50 \cdot 10^7$	$6.77 \cdot 10^6$	$6.50 \cdot 10^7$	$6.77 \cdot 10^6$	1.00	0.00
B.act.act (p:0, id:0)	$6.50 \cdot 10^7$	$6.77 \cdot 10^6$	$3.41 \cdot 10^6$	$5.32 \cdot 10^5$	$5.24 \cdot 10^{-2}$	$5.93 \cdot 10^{-3}$
B.act.update (p:1, id:1)	$6.16 \cdot 10^7$	$6.42 \cdot 10^6$	$5.54 \cdot 10^7$	$5.77 \cdot 10^6$	$9.00 \cdot 10^{-1}$	$2.32 \cdot 10^{-8}$
B.act.wait (p:1, id:2)	$6.17 \cdot 10^6$	$6.43 \cdot 10^5$	$6.17 \cdot 10^6$	$6.43 \cdot 10^5$	1.00	0.00
`B.update`	$6.16 \cdot 10^7$	$6.42 \cdot 10^6$	$6.16 \cdot 10^7$	$6.42 \cdot 10^6$	1.00	0.00
B.update.act (p:0, id:0)	$6.16 \cdot 10^7$	$6.42 \cdot 10^6$	$6.16 \cdot 10^7$	$6.42 \cdot 10^6$	1.00	0.00
`B.wait`	$2.53 \cdot 10^7$	$4.56 \cdot 10^6$	$6.92 \cdot 10^6$	$6.58 \cdot 10^5$	$2.82 \cdot 10^{-1}$	$5.57 \cdot 10^{-2}$
B.wait.wait (p:0, id:0)	$2.53 \cdot 10^7$	$4.56 \cdot 10^6$	$7.47 \cdot 10^5$	$8.65 \cdot 10^4$	$3.02 \cdot 10^{-2}$	$4.63 \cdot 10^{-3}$
B.wait.update (p:1, id:1)	$2.45 \cdot 10^7$	$4.50 \cdot 10^6$	$6.17 \cdot 10^6$	$6.43 \cdot 10^5$	$2.60 \cdot 10^{-1}$	$5.49 \cdot 10^{-2}$
`C.act`	$4.55 \cdot 10^7$	$5.66 \cdot 10^6$	$4.55 \cdot 10^7$	$5.66 \cdot 10^6$	1.00	0.00
C.act.act (p:0, id:0)	$4.55 \cdot 10^7$	$5.66 \cdot 10^6$	$4.46 \cdot 10^6$	$5.48 \cdot 10^5$	$9.81 \cdot 10^{-2}$	$6.50 \cdot 10^{-3}$
C.act.wait (p:1, id:1)	$4.11 \cdot 10^7$	$5.21 \cdot 10^6$	$4.12 \cdot 10^6$	$5.22 \cdot 10^5$	$1.00 \cdot 10^{-1}$	$6.78 \cdot 10^{-8}$
C.act.update (p:1, id:2)	$3.70 \cdot 10^7$	$4.68 \cdot 10^6$	$3.70 \cdot 10^7$	$4.68 \cdot 10^6$	1.00	0.00
`C.update`	$4.11 \cdot 10^7$	$5.21 \cdot 10^6$	$4.11 \cdot 10^7$	$5.21 \cdot 10^6$	1.00	0.00
C.update.act (p:0, id:0)	$4.11 \cdot 10^7$	$5.21 \cdot 10^6$	$4.11 \cdot 10^7$	$5.21 \cdot 10^6$	1.00	0.00
`C.wait`	$1.84 \cdot 10^7$	$2.60 \cdot 10^6$	$5.83 \cdot 10^6$	$5.92 \cdot 10^5$	$3.21 \cdot 10^{-1}$	$4.79 \cdot 10^{-2}$
C.wait.wait (p:0, id:0)	$1.84 \cdot 10^7$	$2.60 \cdot 10^6$	$1.71 \cdot 10^6$	$2.18 \cdot 10^5$	$9.34 \cdot 10^{-2}$	$6.90 \cdot 10^{-3}$
C.wait.update (p:1, id:1)	$1.67 \cdot 10^7$	$2.41 \cdot 10^6$	$4.12 \cdot 10^6$	$5.22 \cdot 10^5$	$2.51 \cdot 10^{-1}$	$4.99 \cdot 10^{-2}$

Table C.21: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model `Tokens`. The Java code has been generated with the 'Sequential' decision mode enabled. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

Performance results for target model 'Tokens' ($n = 20, t = 30$, Random)						
Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
`A.act`	$7.42 \cdot 10^7$	$1.60 \cdot 10^7$	$2.47 \cdot 10^7$	$5.35 \cdot 10^6$	$3.33 \cdot 10^{-1}$	$7.57 \cdot 10^{-5}$
A.act.act (p:0, id:0)	$2.47 \cdot 10^7$	$5.35 \cdot 10^6$	$2.25 \cdot 10^6$	$3.56 \cdot 10^5$	$9.30 \cdot 10^{-2}$	$1.38 \cdot 10^{-2}$
A.act.update (p:1, id:1)	$2.47 \cdot 10^7$	$5.35 \cdot 10^6$	$2.02 \cdot 10^7$	$4.57 \cdot 10^6$	$8.16 \cdot 10^{-1}$	$1.24 \cdot 10^{-2}$
A.act.wait (p:1, id:2)	$2.47 \cdot 10^7$	$5.35 \cdot 10^6$	$2.25 \cdot 10^6$	$5.09 \cdot 10^5$	$9.09 \cdot 10^{-2}$	$1.39 \cdot 10^{-3}$
`A.update`	$2.25 \cdot 10^7$	$5.08 \cdot 10^6$	$2.25 \cdot 10^7$	$5.08 \cdot 10^6$	1.00	0.00
A.update.act (p:0, id:0)	$2.25 \cdot 10^7$	$5.08 \cdot 10^6$	$2.25 \cdot 10^7$	$5.08 \cdot 10^6$	1.00	0.00
`A.wait`	$1.08 \cdot 10^7$	$2.56 \cdot 10^6$	$2.56 \cdot 10^6$	$5.55 \cdot 10^5$	$2.38 \cdot 10^{-1}$	$1.59 \cdot 10^{-2}$
A.wait.wait (p:0, id:0)	$5.42 \cdot 10^6$	$1.28 \cdot 10^6$	$3.11 \cdot 10^5$	$5.92 \cdot 10^4$	$5.86 \cdot 10^{-2}$	$8.68 \cdot 10^{-3}$
A.wait.update (p:1, id:1)	$5.42 \cdot 10^6$	$1.28 \cdot 10^6$	$2.25 \cdot 10^6$	$5.09 \cdot 10^5$	$4.17 \cdot 10^{-1}$	$2.87 \cdot 10^{-2}$
`B.act`	$1.40 \cdot 10^8$	$1.89 \cdot 10^7$	$4.65 \cdot 10^7$	$6.31 \cdot 10^6$	$3.33 \cdot 10^{-1}$	$3.67 \cdot 10^{-5}$
B.act.act (p:0, id:0)	$4.65 \cdot 10^7$	$6.31 \cdot 10^6$	$1.87 \cdot 10^6$	$4.47 \cdot 10^5$	$4.02 \cdot 10^{-2}$	$7.64 \cdot 10^{-3}$
B.act.update (p:1, id:1)	$4.65 \cdot 10^7$	$6.31 \cdot 10^6$	$4.02 \cdot 10^7$	$5.45 \cdot 10^6$	$8.64 \cdot 10^{-1}$	$6.90 \cdot 10^{-3}$
B.act.wait (p:1, id:2)	$4.65 \cdot 10^7$	$6.31 \cdot 10^6$	$4.47 \cdot 10^6$	$6.07 \cdot 10^5$	$9.62 \cdot 10^{-2}$	$7.66 \cdot 10^{-4}$
`B.update`	$4.46 \cdot 10^7$	$6.06 \cdot 10^6$	$4.46 \cdot 10^7$	$6.06 \cdot 10^6$	1.00	0.00
B.update.act (p:0, id:0)	$4.46 \cdot 10^7$	$6.06 \cdot 10^6$	$4.46 \cdot 10^7$	$6.06 \cdot 10^6$	1.00	0.00
`B.wait`	$6.57 \cdot 10^7$	$1.06 \cdot 10^7$	$4.86 \cdot 10^6$	$6.31 \cdot 10^5$	$7.48 \cdot 10^{-2}$	$9.38 \cdot 10^{-3}$
B.wait.wait (p:0, id:0)	$3.28 \cdot 10^7$	$5.31 \cdot 10^6$	$3.82 \cdot 10^5$	$8.25 \cdot 10^4$	$1.19 \cdot 10^{-2}$	$3.22 \cdot 10^{-3}$
B.wait.update (p:1, id:1)	$3.28 \cdot 10^7$	$5.31 \cdot 10^6$	$4.47 \cdot 10^6$	$6.07 \cdot 10^5$	$1.38 \cdot 10^{-1}$	$1.70 \cdot 10^{-2}$
`C.act`	$8.90 \cdot 10^7$	$1.27 \cdot 10^7$	$2.96 \cdot 10^7$	$4.25 \cdot 10^6$	$3.32 \cdot 10^{-1}$	$2.66 \cdot 10^{-4}$
C.act.act (p:0, id:0)	$2.97 \cdot 10^7$	$4.25 \cdot 10^6$	$4.00 \cdot 10^6$	$5.27 \cdot 10^5$	$1.36 \cdot 10^{-1}$	$1.91 \cdot 10^{-2}$
C.act.wait (p:1, id:1)	$2.97 \cdot 10^7$	$4.25 \cdot 10^6$	$2.56 \cdot 10^6$	$4.05 \cdot 10^5$	$8.63 \cdot 10^{-2}$	$1.95 \cdot 10^{-3}$
C.act.update (p:1, id:2)	$2.97 \cdot 10^7$	$4.25 \cdot 10^6$	$2.30 \cdot 10^7$	$3.63 \cdot 10^6$	$7.75 \cdot 10^{-1}$	$1.76 \cdot 10^{-2}$
`C.update`	$2.56 \cdot 10^7$	$4.04 \cdot 10^6$	$2.56 \cdot 10^7$	$4.04 \cdot 10^6$	1.00	0.00
C.update.act (p:0, id:0)	$2.56 \cdot 10^7$	$4.04 \cdot 10^6$	$2.56 \cdot 10^7$	$4.04 \cdot 10^6$	1.00	0.00
`C.wait`	$1.90 \cdot 10^7$	$3.16 \cdot 10^6$	$3.04 \cdot 10^6$	$4.36 \cdot 10^5$	$1.61 \cdot 10^{-1}$	$1.14 \cdot 10^{-2}$
C.wait.wait (p:0, id:0)	$9.50 \cdot 10^6$	$1.58 \cdot 10^6$	$4.73 \cdot 10^5$	$9.81 \cdot 10^4$	$5.04 \cdot 10^{-2}$	$1.01 \cdot 10^{-2}$
C.wait.update (p:1, id:1)	$9.50 \cdot 10^6$	$1.58 \cdot 10^6$	$2.56 \cdot 10^6$	$4.05 \cdot 10^5$	$2.71 \cdot 10^{-1}$	$2.20 \cdot 10^{-2}$

Table C.22: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model `Tokens`. The Java code has been generated with the 'Random' decision mode enabled. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

C.2.3 Practical Test: Elevator

Performance results for target model '`Elevator`' ($n = 20, t = 30$, Random + Det)						
Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
`cabin.idle`	$1.98 \cdot 10^8$	$2.46 \cdot 10^7$	$7.11 \cdot 10^6$	$1.01 \cdot 10^6$	$3.61 \cdot 10^{-2}$	$5.03 \cdot 10^{-3}$
cabin.idle.mov (p:0, id:0)	$1.98 \cdot 10^8$	$2.46 \cdot 10^7$	$7.11 \cdot 10^6$	$1.01 \cdot 10^6$	$3.61 \cdot 10^{-2}$	$5.03 \cdot 10^{-3}$
`cabin.mov`	$1.25 \cdot 10^7$	$1.79 \cdot 10^6$	$1.25 \cdot 10^7$	$1.79 \cdot 10^6$	1.00	0.00
cabin.mov.open (p:0, id:0)	$1.25 \cdot 10^7$	$1.79 \cdot 10^6$	$7.11 \cdot 10^6$	$1.01 \cdot 10^6$	$5.70 \cdot 10^{-1}$	$5.68 \cdot 10^{-4}$
cabin.mov.mov (p:0, id:1)	$5.36 \cdot 10^6$	$7.74 \cdot 10^5$	$2.68 \cdot 10^6$	$3.87 \cdot 10^5$	$5.00 \cdot 10^{-1}$	$1.16 \cdot 10^{-7}$
cabin.mov.mov (p:0, id:2)	$2.68 \cdot 10^6$	$3.87 \cdot 10^5$	$2.68 \cdot 10^6$	$3.87 \cdot 10^5$	1.00	0.00
`cabin.open`	$7.11 \cdot 10^6$	$1.01 \cdot 10^6$	$7.11 \cdot 10^6$	$1.01 \cdot 10^6$	1.00	0.00
cabin.open.idle (p:0, id:0)	$7.11 \cdot 10^6$	$1.01 \cdot 10^6$	$7.11 \cdot 10^6$	$1.01 \cdot 10^6$	1.00	0.00
`controller.done`	$7.11 \cdot 10^6$	$1.01 \cdot 10^6$	$7.11 \cdot 10^6$	$1.01 \cdot 10^6$	1.00	0.00
controller.done.wait (p:0, id:0)	$7.11 \cdot 10^6$	$1.01 \cdot 10^6$	$7.11 \cdot 10^6$	$1.01 \cdot 10^6$	1.00	0.00
`controller.wait`	$5.66 \cdot 10^7$	$1.32 \cdot 10^7$	$8.93 \cdot 10^6$	$1.28 \cdot 10^6$	$1.63 \cdot 10^{-1}$	$2.61 \cdot 10^{-2}$
controller.wait.work (p:0, id:0)	$5.66 \cdot 10^7$	$1.32 \cdot 10^7$	$8.93 \cdot 10^6$	$1.28 \cdot 10^6$	$1.63 \cdot 10^{-1}$	$2.61 \cdot 10^{-2}$
`controller.work`	$9.12 \cdot 10^6$	$1.34 \cdot 10^6$	$9.12 \cdot 10^6$	$1.34 \cdot 10^6$	1.00	0.00
controller.work.wait (p:0, id:0)	$9.12 \cdot 10^6$	$1.34 \cdot 10^6$	$1.82 \cdot 10^6$	$2.68 \cdot 10^5$	$2.00 \cdot 10^{-1}$	$2.84 \cdot 10^{-8}$
controller.work.done (p:0, id:1)	$7.29 \cdot 10^6$	$1.07 \cdot 10^6$	$7.11 \cdot 10^6$	$1.01 \cdot 10^6$	$9.76 \cdot 10^{-1}$	$2.01 \cdot 10^{-2}$
controller.work.work (p:0, id:2)	$1.84 \cdot 10^5$	$1.69 \cdot 10^5$	$1.84 \cdot 10^5$	$1.69 \cdot 10^5$	1.00	0.00
`environment.read`	$3.14 \cdot 10^8$	$5.86 \cdot 10^7$	$7.11 \cdot 10^6$	$1.01 \cdot 10^6$	$2.31 \cdot 10^{-2}$	$3.84 \cdot 10^{-3}$
environment.read.read (p:0, id:0)	$7.85 \cdot 10^7$	$1.47 \cdot 10^7$	$1.77 \cdot 10^6$	$2.47 \cdot 10^5$	$2.30 \cdot 10^{-2}$	$3.78 \cdot 10^{-3}$
environment.read.read (p:0, id:1)	$7.85 \cdot 10^7$	$1.47 \cdot 10^7$	$1.79 \cdot 10^6$	$2.58 \cdot 10^5$	$2.32 \cdot 10^{-2}$	$3.88 \cdot 10^{-3}$
environment.read.read (p:0, id:2)	$7.85 \cdot 10^7$	$1.47 \cdot 10^7$	$1.79 \cdot 10^6$	$2.58 \cdot 10^5$	$2.32 \cdot 10^{-2}$	$3.88 \cdot 10^{-3}$
environment.read.read (p:0, id:3)	$7.85 \cdot 10^7$	$1.47 \cdot 10^7$	$1.77 \cdot 10^6$	$2.51 \cdot 10^5$	$2.31 \cdot 10^{-2}$	$3.80 \cdot 10^{-3}$

Table C.23: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model `Elevator`. The Java code has been generated with the 'Random + Det' decision mode enabled. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

Performance results for target model 'Elevator' ($n = 20, t = 30$, Sequential)						
Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
`cabin.idle`	$1.94 \cdot 10^8$	$3.28 \cdot 10^7$	$6.73 \cdot 10^6$	$7.94 \cdot 10^5$	$3.51 \cdot 10^{-2}$	$3.73 \cdot 10^{-3}$
cabin.idle.mov (p:0, id:0)	$1.94 \cdot 10^8$	$3.28 \cdot 10^7$	$6.73 \cdot 10^6$	$7.94 \cdot 10^5$	$3.51 \cdot 10^{-2}$	$3.73 \cdot 10^{-3}$
`cabin.mov`	$1.18 \cdot 10^7$	$1.39 \cdot 10^6$	$1.18 \cdot 10^7$	$1.39 \cdot 10^6$	1.00	0.00
cabin.mov.open (p:0, id:0)	$1.18 \cdot 10^7$	$1.39 \cdot 10^6$	$6.73 \cdot 10^6$	$7.94 \cdot 10^5$	$5.71 \cdot 10^{-1}$	$5.99 \cdot 10^{-4}$
cabin.mov.mov (p:0, id:1)	$5.07 \cdot 10^6$	$5.92 \cdot 10^5$	$2.53 \cdot 10^6$	$2.96 \cdot 10^5$	$5.00 \cdot 10^{-1}$	$1.00 \cdot 10^{-7}$
cabin.mov.mov (p:0, id:2)	$2.53 \cdot 10^6$	$2.96 \cdot 10^5$	$2.53 \cdot 10^6$	$2.96 \cdot 10^5$	1.00	0.00
`cabin.open`	$6.73 \cdot 10^6$	$7.94 \cdot 10^5$	$6.73 \cdot 10^6$	$7.94 \cdot 10^5$	1.00	0.00
cabin.open.idle (p:0, id:0)	$6.73 \cdot 10^6$	$7.94 \cdot 10^5$	$6.73 \cdot 10^6$	$7.94 \cdot 10^5$	1.00	0.00
`controller.done`	$6.73 \cdot 10^6$	$7.94 \cdot 10^5$	$6.73 \cdot 10^6$	$7.94 \cdot 10^5$	1.00	0.00
controller.done.wait (p:0, id:0)	$6.73 \cdot 10^6$	$7.94 \cdot 10^5$	$6.73 \cdot 10^6$	$7.94 \cdot 10^5$	1.00	0.00
`controller.wait`	$5.10 \cdot 10^7$	$1.15 \cdot 10^7$	$8.49 \cdot 10^6$	$9.86 \cdot 10^5$	$1.70 \cdot 10^{-1}$	$1.98 \cdot 10^{-2}$
controller.wait.work (p:0, id:0)	$5.10 \cdot 10^7$	$1.15 \cdot 10^7$	$8.49 \cdot 10^6$	$9.86 \cdot 10^5$	$1.70 \cdot 10^{-1}$	$1.98 \cdot 10^{-2}$
`controller.work`	$8.80 \cdot 10^6$	$9.96 \cdot 10^5$	$8.80 \cdot 10^6$	$9.96 \cdot 10^5$	1.00	0.00
controller.work.wait (p:0, id:0)	$8.80 \cdot 10^6$	$9.96 \cdot 10^5$	$1.76 \cdot 10^6$	$1.99 \cdot 10^5$	$2.00 \cdot 10^{-1}$	$2.86 \cdot 10^{-8}$
controller.work.done (p:0, id:1)	$7.04 \cdot 10^6$	$7.97 \cdot 10^5$	$6.73 \cdot 10^6$	$7.94 \cdot 10^5$	$9.56 \cdot 10^{-1}$	$3.32 \cdot 10^{-2}$
controller.work.work (p:0, id:2)	$3.10 \cdot 10^5$	$2.36 \cdot 10^5$	$3.10 \cdot 10^5$	$2.36 \cdot 10^5$	1.00	0.00
`environment.read`	$1.30 \cdot 10^8$	$2.10 \cdot 10^7$	$6.73 \cdot 10^6$	$7.94 \cdot 10^5$	$5.26 \cdot 10^{-2}$	$7.53 \cdot 10^{-3}$
environment.read.read (p:0, id:0)	$1.30 \cdot 10^8$	$2.10 \cdot 10^7$	$1.68 \cdot 10^6$	$1.99 \cdot 10^5$	$1.31 \cdot 10^{-2}$	$1.87 \cdot 10^{-3}$
environment.read.read (p:0, id:1)	$1.28 \cdot 10^8$	$2.09 \cdot 10^7$	$1.68 \cdot 10^6$	$1.98 \cdot 10^5$	$1.33 \cdot 10^{-2}$	$1.94 \cdot 10^{-3}$
environment.read.read (p:0, id:2)	$1.27 \cdot 10^8$	$2.08 \cdot 10^7$	$1.68 \cdot 10^6$	$1.98 \cdot 10^5$	$1.35 \cdot 10^{-2}$	$2.00 \cdot 10^{-3}$
environment.read.read (p:0, id:3)	$1.25 \cdot 10^8$	$2.06 \cdot 10^7$	$1.68 \cdot 10^6$	$1.99 \cdot 10^5$	$1.37 \cdot 10^{-2}$	$2.05 \cdot 10^{-3}$

Table C.24: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model `Elevator`. The Java code has been generated with the 'Sequential' decision mode enabled. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

Performance results for target model 'Elevator' ($n = 20, t = 30$, Random)						
Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
`cabin.idle`	$1.27 \cdot 10^8$	$2.08 \cdot 10^7$	$5.86 \cdot 10^6$	$5.91 \cdot 10^5$	$4.69 \cdot 10^{-2}$	$7.25 \cdot 10^{-3}$
cabin.idle.mov (p:0, id:0)	$1.27 \cdot 10^8$	$2.08 \cdot 10^7$	$5.86 \cdot 10^6$	$5.91 \cdot 10^5$	$4.69 \cdot 10^{-2}$	$7.25 \cdot 10^{-3}$
`cabin.mov`	$3.08 \cdot 10^7$	$3.11 \cdot 10^6$	$1.03 \cdot 10^7$	$1.03 \cdot 10^6$	$3.33 \cdot 10^{-1}$	$9.86 \cdot 10^{-5}$
cabin.mov.open (p:0, id:0)	$1.03 \cdot 10^7$	$1.04 \cdot 10^6$	$5.86 \cdot 10^6$	$5.91 \cdot 10^5$	$5.71 \cdot 10^{-1}$	$2.06 \cdot 10^{-4}$
cabin.mov.mov (p:0, id:1)	$1.03 \cdot 10^7$	$1.03 \cdot 10^6$	$2.20 \cdot 10^6$	$2.22 \cdot 10^5$	$2.14 \cdot 10^{-1}$	$1.31 \cdot 10^{-4}$
cabin.mov.mov (p:0, id:2)	$1.03 \cdot 10^7$	$1.04 \cdot 10^6$	$2.20 \cdot 10^6$	$2.22 \cdot 10^5$	$2.14 \cdot 10^{-1}$	$1.15 \cdot 10^{-4}$
`cabin.open`	$5.86 \cdot 10^6$	$5.91 \cdot 10^5$	$5.86 \cdot 10^6$	$5.91 \cdot 10^5$	1.00	0.00
cabin.open.idle (p:0, id:0)	$5.86 \cdot 10^6$	$5.91 \cdot 10^5$	$5.86 \cdot 10^6$	$5.91 \cdot 10^5$	1.00	0.00
`controller.done`	$5.86 \cdot 10^6$	$5.91 \cdot 10^5$	$5.86 \cdot 10^6$	$5.91 \cdot 10^5$	1.00	0.00
controller.done.wait (p:0, id:0)	$5.86 \cdot 10^6$	$5.91 \cdot 10^5$	$5.86 \cdot 10^6$	$5.91 \cdot 10^5$	1.00	0.00
`controller.wait`	$8.59 \cdot 10^7$	$1.48 \cdot 10^7$	$7.32 \cdot 10^6$	$7.39 \cdot 10^5$	$8.70 \cdot 10^{-2}$	$1.38 \cdot 10^{-2}$
controller.wait.work (p:0, id:0)	$8.59 \cdot 10^7$	$1.48 \cdot 10^7$	$7.32 \cdot 10^6$	$7.39 \cdot 10^5$	$8.70 \cdot 10^{-2}$	$1.38 \cdot 10^{-2}$
`controller.work`	$2.20 \cdot 10^7$	$2.22 \cdot 10^6$	$7.33 \cdot 10^6$	$7.39 \cdot 10^5$	$3.33 \cdot 10^{-1}$	$1.40 \cdot 10^{-4}$
controller.work.wait (p:0, id:0)	$7.33 \cdot 10^6$	$7.39 \cdot 10^5$	$1.47 \cdot 10^6$	$1.48 \cdot 10^5$	$2.00 \cdot 10^{-1}$	$1.13 \cdot 10^{-4}$
controller.work.done (p:0, id:1)	$7.33 \cdot 10^6$	$7.40 \cdot 10^5$	$5.86 \cdot 10^6$	$5.91 \cdot 10^5$	$7.99 \cdot 10^{-1}$	$6.55 \cdot 10^{-4}$
controller.work.work (p:0, id:2)	$7.33 \cdot 10^6$	$7.40 \cdot 10^5$	$9.79 \cdot 10^3$	$4.26 \cdot 10^3$	$1.35 \cdot 10^{-3}$	$6.12 \cdot 10^{-4}$
`environment.read`	$3.77 \cdot 10^8$	$6.07 \cdot 10^7$	$5.86 \cdot 10^6$	$5.91 \cdot 10^5$	$1.58 \cdot 10^{-2}$	$2.13 \cdot 10^{-3}$
environment.read.read (p:0, id:0)	$9.43 \cdot 10^7$	$1.52 \cdot 10^7$	$1.46 \cdot 10^6$	$1.47 \cdot 10^5$	$1.58 \cdot 10^{-2}$	$2.12 \cdot 10^{-3}$
environment.read.read (p:0, id:1)	$9.43 \cdot 10^7$	$1.52 \cdot 10^7$	$1.47 \cdot 10^6$	$1.48 \cdot 10^5$	$1.58 \cdot 10^{-2}$	$2.14 \cdot 10^{-3}$
environment.read.read (p:0, id:2)	$9.43 \cdot 10^7$	$1.52 \cdot 10^7$	$1.47 \cdot 10^6$	$1.48 \cdot 10^5$	$1.58 \cdot 10^{-2}$	$2.15 \cdot 10^{-3}$
environment.read.read (p:0, id:3)	$9.43 \cdot 10^7$	$1.52 \cdot 10^7$	$1.46 \cdot 10^6$	$1.48 \cdot 10^5$	$1.58 \cdot 10^{-2}$	$2.12 \cdot 10^{-3}$

Table C.25: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model `Elevator`. The Java code has been generated with the 'Random' decision mode enabled. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

C.2.4 Practical Test: ToadsAndFrogs

Performance results for target model '`ToadsAndFrogs`' ($n = 20, t = 30$, Random + Det)						
Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
`control.done`	$3.04 \cdot 10^{6}$	$1.79 \cdot 10^{5}$	$3.04 \cdot 10^{6}$	$1.79 \cdot 10^{5}$	1.00	0.00
control.done.success (p:0, id:0)	$3.04 \cdot 10^{6}$	$1.79 \cdot 10^{5}$	$9.00 \cdot 10^{-1}$	$9.68 \cdot 10^{-1}$	$2.98 \cdot 10^{-7}$	$3.20 \cdot 10^{-7}$
control.done.failure (p:0, id:1)	$3.04 \cdot 10^{6}$	$1.79 \cdot 10^{5}$	$3.04 \cdot 10^{6}$	$1.79 \cdot 10^{5}$	1.00	0.00
`control.failure`	$3.04 \cdot 10^{6}$	$1.79 \cdot 10^{5}$	$3.04 \cdot 10^{6}$	$1.79 \cdot 10^{5}$	1.00	0.00
control.failure.reset (p:0, id:0)	$3.04 \cdot 10^{6}$	$1.79 \cdot 10^{5}$	$3.04 \cdot 10^{6}$	$1.79 \cdot 10^{5}$	1.00	0.00
`control.reset`	$3.04 \cdot 10^{6}$	$1.79 \cdot 10^{5}$	$3.04 \cdot 10^{6}$	$1.79 \cdot 10^{5}$	1.00	0.00
control.reset.running (p:0, id:0)	$3.04 \cdot 10^{6}$	$1.79 \cdot 10^{5}$	$3.04 \cdot 10^{6}$	$1.79 \cdot 10^{5}$	1.00	0.00
`control.running`	$1.93 \cdot 10^{7}$	$1.08 \cdot 10^{6}$	$3.04 \cdot 10^{6}$	$1.79 \cdot 10^{5}$	$1.58 \cdot 10^{-1}$	$1.58 \cdot 10^{-2}$
control.running.done (p:0, id:0)	$1.93 \cdot 10^{7}$	$1.08 \cdot 10^{6}$	$1.76 \cdot 10^{6}$	$2.39 \cdot 10^{5}$	$9.14 \cdot 10^{-2}$	$1.46 \cdot 10^{-2}$
control.running.done (p:0, id:1)	$1.76 \cdot 10^{7}$	$1.17 \cdot 10^{6}$	$3.91 \cdot 10^{1}$	$1.04 \cdot 10^{1}$	$2.23 \cdot 10^{-6}$	$5.68 \cdot 10^{-7}$
control.running.done (p:0, id:2)	$1.76 \cdot 10^{7}$	$1.17 \cdot 10^{6}$	$3.92 \cdot 10^{1}$	$1.27 \cdot 10^{1}$	$2.22 \cdot 10^{-6}$	$6.51 \cdot 10^{-7}$
control.running.done (p:0, id:3)	$1.76 \cdot 10^{7}$	$1.17 \cdot 10^{6}$	$1.28 \cdot 10^{6}$	$2.48 \cdot 10^{5}$	$7.32 \cdot 10^{-2}$	$1.49 \cdot 10^{-2}$
control.running.done (p:0, id:4)	$1.63 \cdot 10^{7}$	$1.18 \cdot 10^{6}$	9.55	3.94	$5.89 \cdot 10^{-7}$	$2.39 \cdot 10^{-7}$
`control.success`	$9.00 \cdot 10^{-1}$	$9.68 \cdot 10^{-1}$	$9.00 \cdot 10^{-1}$	$9.68 \cdot 10^{-1}$	1.00	0.00
control.success.reset (p:0, id:0)	$9.00 \cdot 10^{-1}$	$9.68 \cdot 10^{-1}$	$9.00 \cdot 10^{-1}$	$9.68 \cdot 10^{-1}$	1.00	0.00
`frog.q`	$2.26 \cdot 10^{7}$	$1.08 \cdot 10^{6}$	$5.15 \cdot 10^{6}$	$9.91 \cdot 10^{5}$	$2.27 \cdot 10^{-1}$	$3.89 \cdot 10^{-2}$
frog.q.q (p:0, id:0)	$2.26 \cdot 10^{7}$	$1.08 \cdot 10^{6}$	$3.84 \cdot 10^{6}$	$7.44 \cdot 10^{5}$	$1.69 \cdot 10^{-1}$	$2.92 \cdot 10^{-2}$
frog.q.q (p:0, id:1)	$1.88 \cdot 10^{7}$	$8.55 \cdot 10^{5}$	$1.28 \cdot 10^{6}$	$2.48 \cdot 10^{5}$	$6.85 \cdot 10^{-2}$	$1.40 \cdot 10^{-2}$
frog.q.q (p:0, id:2)	$1.75 \cdot 10^{7}$	$9.10 \cdot 10^{5}$	$3.23 \cdot 10^{4}$	$6.68 \cdot 10^{3}$	$1.85 \cdot 10^{-3}$	$3.65 \cdot 10^{-4}$
frog.q.q (p:0, id:3)	$1.74 \cdot 10^{7}$	$9.08 \cdot 10^{5}$	$1.31 \cdot 10^{1}$	4.36	$7.56 \cdot 10^{-7}$	$2.58 \cdot 10^{-7}$
`toad.q`	$2.22 \cdot 10^{7}$	$1.11 \cdot 10^{6}$	$7.06 \cdot 10^{6}$	$9.54 \cdot 10^{5}$	$3.17 \cdot 10^{-1}$	$3.91 \cdot 10^{-2}$
toad.q.q (p:0, id:0)	$2.22 \cdot 10^{7}$	$1.11 \cdot 10^{6}$	$5.27 \cdot 10^{6}$	$7.16 \cdot 10^{5}$	$2.37 \cdot 10^{-1}$	$2.94 \cdot 10^{-2}$
toad.q.q (p:0, id:1)	$1.70 \cdot 10^{7}$	$1.04 \cdot 10^{6}$	$1.76 \cdot 10^{6}$	$2.39 \cdot 10^{5}$	$1.04 \cdot 10^{-1}$	$1.69 \cdot 10^{-2}$
toad.q.q (p:0, id:2)	$1.52 \cdot 10^{7}$	$1.12 \cdot 10^{6}$	$2.92 \cdot 10^{4}$	$4.96 \cdot 10^{3}$	$1.92 \cdot 10^{-3}$	$3.43 \cdot 10^{-4}$
toad.q.q (p:0, id:3)	$1.52 \cdot 10^{7}$	$1.12 \cdot 10^{6}$	$1.69 \cdot 10^{1}$	7.90	$1.13 \cdot 10^{-6}$	$5.75 \cdot 10^{-7}$

Table C.26: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model `ToadsAndFrogs`. The Java code has been generated with the 'Random + Det' decision mode enabled. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

Performance results for target model '`ToadsAndFrogs`' ($n = 20, t = 30$, Sequential)						
Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
`control.done`	$3.04 \cdot 10^{6}$	$1.39 \cdot 10^{5}$	$3.04 \cdot 10^{6}$	$1.39 \cdot 10^{5}$	1.00	0.00
control.done.success (p:0, id:0)	$3.04 \cdot 10^{6}$	$1.39 \cdot 10^{5}$	$9.00 \cdot 10^{-1}$	$9.68 \cdot 10^{-1}$	$2.95 \cdot 10^{-7}$	$3.15 \cdot 10^{-7}$
control.done.failure (p:0, id:1)	$3.04 \cdot 10^{6}$	$1.39 \cdot 10^{5}$	$3.04 \cdot 10^{6}$	$1.39 \cdot 10^{5}$	1.00	0.00
`control.failure`	$3.04 \cdot 10^{6}$	$1.39 \cdot 10^{5}$	$3.04 \cdot 10^{6}$	$1.39 \cdot 10^{5}$	1.00	0.00
control.failure.reset (p:0, id:0)	$3.04 \cdot 10^{6}$	$1.39 \cdot 10^{5}$	$3.04 \cdot 10^{6}$	$1.39 \cdot 10^{5}$	1.00	0.00
`control.reset`	$3.04 \cdot 10^{6}$	$1.39 \cdot 10^{5}$	$3.04 \cdot 10^{6}$	$1.39 \cdot 10^{5}$	1.00	0.00
control.reset.running (p:0, id:0)	$3.04 \cdot 10^{6}$	$1.39 \cdot 10^{5}$	$3.04 \cdot 10^{6}$	$1.39 \cdot 10^{5}$	1.00	0.00
`control.running`	$1.87 \cdot 10^{7}$	$1.61 \cdot 10^{6}$	$3.04 \cdot 10^{6}$	$1.39 \cdot 10^{5}$	$1.64 \cdot 10^{-1}$	$1.48 \cdot 10^{-2}$
control.running.done (p:0, id:0)	$1.87 \cdot 10^{7}$	$1.61 \cdot 10^{6}$	$1.72 \cdot 10^{6}$	$2.18 \cdot 10^{5}$	$9.29 \cdot 10^{-2}$	$1.57 \cdot 10^{-2}$
control.running.done (p:0, id:1)	$1.70 \cdot 10^{7}$	$1.70 \cdot 10^{6}$	$4.55 \cdot 10^{1}$	$1.35 \cdot 10^{1}$	$2.68 \cdot 10^{-6}$	$7.37 \cdot 10^{-7}$
control.running.done (p:0, id:2)	$1.70 \cdot 10^{7}$	$1.70 \cdot 10^{6}$	$4.26 \cdot 10^{1}$	$1.30 \cdot 10^{1}$	$2.55 \cdot 10^{-6}$	$8.42 \cdot 10^{-7}$
control.running.done (p:0, id:3)	$1.70 \cdot 10^{7}$	$1.70 \cdot 10^{6}$	$1.33 \cdot 10^{6}$	$2.31 \cdot 10^{5}$	$7.83 \cdot 10^{-2}$	$1.32 \cdot 10^{-2}$
control.running.done (p:0, id:4)	$1.56 \cdot 10^{7}$	$1.61 \cdot 10^{6}$	9.25	2.63	$5.97 \cdot 10^{-7}$	$1.81 \cdot 10^{-7}$
`control.success`	$9.00 \cdot 10^{-1}$	$9.68 \cdot 10^{-1}$	$9.00 \cdot 10^{-1}$	$9.68 \cdot 10^{-1}$	1.00	0.00
control.success.reset (p:0, id:0)	$9.00 \cdot 10^{-1}$	$9.68 \cdot 10^{-1}$	$9.00 \cdot 10^{-1}$	$9.68 \cdot 10^{-1}$	1.00	0.00
`frog.q`	$2.26 \cdot 10^{7}$	$1.19 \cdot 10^{6}$	$5.33 \cdot 10^{6}$	$9.25 \cdot 10^{5}$	$2.37 \cdot 10^{-1}$	$4.26 \cdot 10^{-2}$
frog.q.q (p:0, id:0)	$2.26 \cdot 10^{7}$	$1.19 \cdot 10^{6}$	$3.98 \cdot 10^{6}$	$6.93 \cdot 10^{5}$	$1.77 \cdot 10^{-1}$	$3.19 \cdot 10^{-2}$
frog.q.q (p:0, id:1)	$1.86 \cdot 10^{7}$	$1.35 \cdot 10^{6}$	$1.33 \cdot 10^{6}$	$2.31 \cdot 10^{5}$	$7.21 \cdot 10^{-2}$	$1.54 \cdot 10^{-2}$
frog.q.q (p:0, id:2)	$1.73 \cdot 10^{7}$	$1.47 \cdot 10^{6}$	$3.13 \cdot 10^{4}$	$6.00 \cdot 10^{3}$	$1.83 \cdot 10^{-3}$	$4.02 \cdot 10^{-4}$
frog.q.q (p:0, id:3)	$1.72 \cdot 10^{7}$	$1.47 \cdot 10^{6}$	$1.46 \cdot 10^{1}$	5.34	$8.49 \cdot 10^{-7}$	$3.18 \cdot 10^{-7}$
`toad.q`	$2.22 \cdot 10^{7}$	$1.44 \cdot 10^{6}$	$6.91 \cdot 10^{6}$	$8.69 \cdot 10^{5}$	$3.12 \cdot 10^{-1}$	$4.02 \cdot 10^{-2}$
toad.q.q (p:0, id:0)	$2.22 \cdot 10^{7}$	$1.44 \cdot 10^{6}$	$5.16 \cdot 10^{6}$	$6.52 \cdot 10^{5}$	$2.33 \cdot 10^{-1}$	$3.01 \cdot 10^{-2}$
toad.q.q (p:0, id:1)	$1.71 \cdot 10^{7}$	$1.48 \cdot 10^{6}$	$1.72 \cdot 10^{6}$	$2.18 \cdot 10^{5}$	$1.02 \cdot 10^{-1}$	$1.72 \cdot 10^{-2}$
toad.q.q (p:0, id:2)	$1.53 \cdot 10^{7}$	$1.56 \cdot 10^{6}$	$3.08 \cdot 10^{4}$	$4.93 \cdot 10^{3}$	$2.03 \cdot 10^{-3}$	$4.08 \cdot 10^{-4}$
toad.q.q (p:0, id:3)	$1.53 \cdot 10^{7}$	$1.56 \cdot 10^{6}$	$1.67 \cdot 10^{1}$	6.20	$1.10 \cdot 10^{-6}$	$4.38 \cdot 10^{-7}$

Table C.27: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model `ToadsAndFrogs`. The Java code has been generated with the 'Sequential' decision mode enabled. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

Performance results for target model '`ToadsAndFrogs`' ($n = 20, t = 30$, Random)						
Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
`control.done`	$4.85 \cdot 10^6$	$3.67 \cdot 10^5$	$2.42 \cdot 10^6$	$1.83 \cdot 10^5$	$5.00 \cdot 10^{-1}$	$2.75 \cdot 10^{-4}$
control.done.success (p:0, id:0)	$2.42 \cdot 10^6$	$1.84 \cdot 10^5$	$8.04 \cdot 10^2$	$1.92 \cdot 10^2$	$3.28 \cdot 10^{-4}$	$6.85 \cdot 10^{-5}$
control.done.failure (p:0, id:1)	$2.42 \cdot 10^6$	$1.83 \cdot 10^5$	$2.42 \cdot 10^6$	$1.83 \cdot 10^5$	1.00	$7.18 \cdot 10^{-5}$
`control.failure`	$2.42 \cdot 10^6$	$1.83 \cdot 10^5$	$2.42 \cdot 10^6$	$1.83 \cdot 10^5$	1.00	0.00
control.failure.reset (p:0, id:0)	$2.42 \cdot 10^6$	$1.83 \cdot 10^5$	$2.42 \cdot 10^6$	$1.83 \cdot 10^5$	1.00	0.00
`control.reset`	$2.42 \cdot 10^6$	$1.83 \cdot 10^5$	$2.42 \cdot 10^6$	$1.83 \cdot 10^5$	1.00	0.00
control.reset.running (p:0, id:0)	$2.42 \cdot 10^6$	$1.83 \cdot 10^5$	$2.42 \cdot 10^6$	$1.83 \cdot 10^5$	1.00	0.00
`control.running`	$5.82 \cdot 10^7$	$7.86 \cdot 10^6$	$2.42 \cdot 10^6$	$1.83 \cdot 10^5$	$4.22 \cdot 10^{-2}$	$4.64 \cdot 10^{-3}$
control.running.done (p:0, id:0)	$1.16 \cdot 10^7$	$1.57 \cdot 10^6$	$1.19 \cdot 10^6$	$8.94 \cdot 10^4$	$1.03 \cdot 10^{-1}$	$1.11 \cdot 10^{-2}$
control.running.done (p:0, id:1)	$1.16 \cdot 10^7$	$1.57 \cdot 10^6$	$3.90 \cdot 10^3$	$1.02 \cdot 10^3$	$3.34 \cdot 10^{-4}$	$8.79 \cdot 10^{-5}$
control.running.done (p:0, id:2)	$1.16 \cdot 10^7$	$1.57 \cdot 10^6$	$3.77 \cdot 10^3$	$1.13 \cdot 10^3$	$3.24 \cdot 10^{-4}$	$9.78 \cdot 10^{-5}$
control.running.done (p:0, id:3)	$1.16 \cdot 10^7$	$1.57 \cdot 10^6$	$1.22 \cdot 10^6$	$1.59 \cdot 10^5$	$1.06 \cdot 10^{-1}$	$1.59 \cdot 10^{-2}$
control.running.done (p:0, id:4)	$1.16 \cdot 10^7$	$1.57 \cdot 10^6$	$7.50 \cdot 10^3$	$2.09 \cdot 10^3$	$6.44 \cdot 10^{-4}$	$1.82 \cdot 10^{-4}$
`control.success`	$8.04 \cdot 10^2$	$1.92 \cdot 10^2$	$8.04 \cdot 10^2$	$1.92 \cdot 10^2$	1.00	0.00
control.success.reset (p:0, id:0)	$8.04 \cdot 10^2$	$1.92 \cdot 10^2$	$8.04 \cdot 10^2$	$1.92 \cdot 10^2$	1.00	0.00
`frog.q`	$7.73 \cdot 10^7$	$9.10 \cdot 10^6$	$5.09 \cdot 10^6$	$6.70 \cdot 10^5$	$6.69 \cdot 10^{-2}$	$1.25 \cdot 10^{-2}$
frog.q.q (p:0, id:0)	$1.93 \cdot 10^7$	$2.28 \cdot 10^6$	$3.69 \cdot 10^6$	$4.95 \cdot 10^5$	$1.94 \cdot 10^{-1}$	$3.67 \cdot 10^{-2}$
frog.q.q (p:0, id:1)	$1.93 \cdot 10^7$	$2.28 \cdot 10^6$	$1.23 \cdot 10^6$	$1.61 \cdot 10^5$	$6.47 \cdot 10^{-2}$	$1.20 \cdot 10^{-2}$
frog.q.q (p:0, id:2)	$1.93 \cdot 10^7$	$2.28 \cdot 10^6$	$1.58 \cdot 10^5$	$2.32 \cdot 10^4$	$8.31 \cdot 10^{-3}$	$1.64 \cdot 10^{-3}$
frog.q.q (p:0, id:3)	$1.93 \cdot 10^7$	$2.28 \cdot 10^6$	$6.13 \cdot 10^3$	$1.66 \cdot 10^3$	$3.26 \cdot 10^{-4}$	$9.86 \cdot 10^{-5}$
`toad.q`	$7.42 \cdot 10^7$	$7.06 \cdot 10^6$	$4.95 \cdot 10^6$	$3.79 \cdot 10^5$	$6.75 \cdot 10^{-2}$	$9.15 \cdot 10^{-3}$
toad.q.q (p:0, id:0)	$1.86 \cdot 10^7$	$1.76 \cdot 10^6$	$3.60 \cdot 10^6$	$2.73 \cdot 10^5$	$1.96 \cdot 10^{-1}$	$2.64 \cdot 10^{-2}$
toad.q.q (p:0, id:1)	$1.86 \cdot 10^7$	$1.77 \cdot 10^6$	$1.20 \cdot 10^6$	$9.05 \cdot 10^4$	$6.51 \cdot 10^{-2}$	$8.70 \cdot 10^{-3}$
toad.q.q (p:0, id:2)	$1.86 \cdot 10^7$	$1.77 \cdot 10^6$	$1.52 \cdot 10^5$	$3.11 \cdot 10^4$	$8.33 \cdot 10^{-3}$	$2.01 \cdot 10^{-3}$
toad.q.q (p:0, id:3)	$1.86 \cdot 10^7$	$1.77 \cdot 10^6$	$6.10 \cdot 10^3$	$1.57 \cdot 10^3$	$3.37 \cdot 10^{-4}$	$9.89 \cdot 10^{-5}$

Table C.28: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model `ToadsAndFrogs`. The Java code has been generated with the 'Random' decision mode enabled. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

C.2.5 Practical Test: Telephony

Performance results for target model 'Telephony' ($n = 20, t = 30$, Random, User_0)						
Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
User_0.busy	$1.56 \cdot 10^6$	$2.19 \cdot 10^6$	$1.56 \cdot 10^6$	$2.19 \cdot 10^6$	1.00	0.00
User_0.busy.idle (p:0, id:0)	$1.56 \cdot 10^6$	$2.19 \cdot 10^6$	$1.56 \cdot 10^6$	$2.19 \cdot 10^6$	1.00	0.00
User_0.calling	$1.76 \cdot 10^7$	$2.49 \cdot 10^7$	$3.12 \cdot 10^6$	$4.40 \cdot 10^6$	$1.99 \cdot 10^{-1}$	$7.78 \cdot 10^{-2}$
User_0.calling.busy (p:0, id:0)	$2.93 \cdot 10^6$	$4.15 \cdot 10^6$	$3.66 \cdot 10^5$	$5.16 \cdot 10^5$	$1.81 \cdot 10^{-1}$	$2.21 \cdot 10^{-1}$
User_0.calling.unobtainable (p:0, id:1)	$2.93 \cdot 10^6$	$4.15 \cdot 10^6$	$1.83 \cdot 10^5$	$2.58 \cdot 10^5$	$6.80 \cdot 10^{-2}$	$6.31 \cdot 10^{-2}$
User_0.calling.ringback (p:0, id:2)	$2.93 \cdot 10^6$	$4.15 \cdot 10^6$	$1.83 \cdot 10^5$	$2.58 \cdot 10^5$	$8.16 \cdot 10^{-2}$	$6.75 \cdot 10^{-2}$
User_0.calling.busy (p:0, id:3)	$2.93 \cdot 10^6$	$4.15 \cdot 10^6$	$1.19 \cdot 10^6$	$1.68 \cdot 10^6$	$3.99 \cdot 10^{-1}$	$9.92 \cdot 10^{-2}$
User_0.calling.calling (p:0, id:4)	$2.93 \cdot 10^6$	$4.15 \cdot 10^6$	$1.19 \cdot 10^6$	$1.70 \cdot 10^6$	$3.67 \cdot 10^{-1}$	$1.40 \cdot 10^{-1}$
User_0.calling.oalert (p:0, id:5)	$2.93 \cdot 10^6$	$4.15 \cdot 10^6$	1.75	1.74	$5.11 \cdot 10^{-2}$	$9.11 \cdot 10^{-2}$
User_0.dialing	$2.20 \cdot 10^6$	$3.10 \cdot 10^6$	$2.20 \cdot 10^6$	$3.10 \cdot 10^6$	1.00	0.00
User_0.dialing.idle (p:0, id:0)	$3.67 \cdot 10^5$	$5.17 \cdot 10^5$	$3.67 \cdot 10^5$	$5.17 \cdot 10^5$	1.00	0.00
User_0.dialing.calling (p:0, id:1)	$3.66 \cdot 10^5$	$5.16 \cdot 10^5$	$3.66 \cdot 10^5$	$5.16 \cdot 10^5$	1.00	0.00
User_0.dialing.calling (p:0, id:2)	$3.66 \cdot 10^5$	$5.16 \cdot 10^5$	$3.66 \cdot 10^5$	$5.16 \cdot 10^5$	1.00	0.00
User_0.dialing.calling (p:0, id:3)	$3.66 \cdot 10^5$	$5.16 \cdot 10^5$	$3.66 \cdot 10^5$	$5.16 \cdot 10^5$	1.00	0.00
User_0.dialing.calling (p:0, id:4)	$3.66 \cdot 10^5$	$5.16 \cdot 10^5$	$3.66 \cdot 10^5$	$5.16 \cdot 10^5$	1.00	0.00
User_0.dialing.calling (p:0, id:5)	$3.66 \cdot 10^5$	$5.16 \cdot 10^5$	$3.66 \cdot 10^5$	$5.16 \cdot 10^5$	1.00	0.00
User_0.dveoringout	1.60	1.54	1.60	1.54	1.00	0.00
User_0.dveoringout.idle (p:0, id:0)	1.60	1.54	1.60	1.54	1.00	0.00
User_0.idle	$3.27 \cdot 10^7$	$1.94 \cdot 10^7$	$1.64 \cdot 10^7$	$9.72 \cdot 10^6$	$5.00 \cdot 10^{-1}$	$1.03 \cdot 10^{-4}$
User_0.idle.dialing (p:0, id:0)	$1.64 \cdot 10^7$	$9.71 \cdot 10^6$	$2.20 \cdot 10^6$	$3.10 \cdot 10^6$	$3.50 \cdot 10^{-1}$	$4.89 \cdot 10^{-1}$
User_0.idle.qi (p:0, id:1)	$1.64 \cdot 10^7$	$9.72 \cdot 10^6$	$1.42 \cdot 10^7$	$1.23 \cdot 10^7$	$6.50 \cdot 10^{-1}$	$4.89 \cdot 10^{-1}$
User_0.oalert	4.80	7.43	1.75	1.74	$6.64 \cdot 10^{-1}$	$3.42 \cdot 10^{-1}$
User_0.oalert.oconnected (p:0, id:1)	1.65	2.91	$1.50 \cdot 10^{-1}$	$3.66 \cdot 10^{-1}$	$8.10 \cdot 10^{-2}$	$1.01 \cdot 10^{-1}$
User_0.oalert.dveoringout (p:0, id:2)	1.75	1.89	1.60	1.54	$9.71 \cdot 10^{-1}$	$1.11 \cdot 10^{-1}$
User_0.oconnected	$1.50 \cdot 10^{-1}$	$3.66 \cdot 10^{-1}$	$1.50 \cdot 10^{-1}$	$3.66 \cdot 10^{-1}$	1.00	0.00
User_0.oconnected.idle (p:0, id:0)	$1.50 \cdot 10^{-1}$	$3.66 \cdot 10^{-1}$	$1.50 \cdot 10^{-1}$	$3.66 \cdot 10^{-1}$	1.00	0.00
User_0.qi	$2.83 \cdot 10^7$	$2.45 \cdot 10^7$	$1.42 \cdot 10^7$	$1.23 \cdot 10^7$	$5.36 \cdot 10^{-1}$	$1.34 \cdot 10^{-1}$
User_0.qi.talert (p:0, id:0)	$1.42 \cdot 10^7$	$1.23 \cdot 10^7$	$5.00 \cdot 10^{-1}$	$6.07 \cdot 10^{-1}$	$7.14 \cdot 10^{-2}$	$2.67 \cdot 10^{-1}$
User_0.qi.idle (p:0, id:1)	$1.42 \cdot 10^7$	$1.23 \cdot 10^7$	$1.42 \cdot 10^7$	$1.23 \cdot 10^7$	1.00	$5.06 \cdot 10^{-8}$
User_0.ringback	$1.83 \cdot 10^5$	$2.58 \cdot 10^5$	$1.83 \cdot 10^5$	$2.58 \cdot 10^5$	1.00	0.00
User_0.ringback.idle (p:0, id:0)	$9.16 \cdot 10^4$	$1.29 \cdot 10^5$	$9.16 \cdot 10^4$	$1.29 \cdot 10^5$	1.00	0.00
User_0.ringback.calling (p:0, id:1)	$9.16 \cdot 10^4$	$1.29 \cdot 10^5$	$9.16 \cdot 10^4$	$1.29 \cdot 10^5$	1.00	0.00
User_0.talert	1.55	2.95	$5.00 \cdot 10^{-1}$	$6.07 \cdot 10^{-1}$	$6.11 \cdot 10^{-1}$	$3.84 \cdot 10^{-1}$
User_0.talert.tpickup (p:0, id:1)	$4.50 \cdot 10^{-1}$	$8.26 \cdot 10^{-1}$	$1.50 \cdot 10^{-1}$	$3.66 \cdot 10^{-1}$	$5.00 \cdot 10^{-1}$	$5.48 \cdot 10^{-1}$
User_0.talert.idle (p:0, id:2)	$4.50 \cdot 10^{-1}$	$7.59 \cdot 10^{-1}$	$3.50 \cdot 10^{-1}$	$4.89 \cdot 10^{-1}$	$9.05 \cdot 10^{-1}$	$2.52 \cdot 10^{-1}$
User_0.tpickup	$3.00 \cdot 10^{-1}$	$8.01 \cdot 10^{-1}$	$1.50 \cdot 10^{-1}$	$3.66 \cdot 10^{-1}$	$6.11 \cdot 10^{-1}$	$3.47 \cdot 10^{-1}$
User_0.tpickup.idle (p:0, id:1)	$1.50 \cdot 10^{-1}$	$3.66 \cdot 10^{-1}$	$1.50 \cdot 10^{-1}$	$3.66 \cdot 10^{-1}$	1.00	0.00
User_0.unobtainable	$1.83 \cdot 10^5$	$2.58 \cdot 10^5$	$1.83 \cdot 10^5$	$2.58 \cdot 10^5$	1.00	0.00
User_0.unobtainable.idle (p:0, id:0)	$1.83 \cdot 10^5$	$2.58 \cdot 10^5$	$1.83 \cdot 10^5$	$2.58 \cdot 10^5$	1.00	0.00

Table C.29: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model Telephony (state machine User_0 only). The Java code has been generated with the 'Random' decision mode enabled. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

Performance results for target model 'Telephony' ($n = 20, t = 30$, Random, User_1)						
Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
`User_1.busy`	$5.21 \cdot 10^5$	$1.64 \cdot 10^6$	$5.21 \cdot 10^5$	$1.64 \cdot 10^6$	1.00	0.00
User_1.busy.idle (p:0, id:0)	$5.21 \cdot 10^5$	$1.64 \cdot 10^6$	$5.21 \cdot 10^5$	$1.64 \cdot 10^6$	1.00	0.00
`User_1.calling`	$5.04 \cdot 10^6$	$1.58 \cdot 10^7$	$9.01 \cdot 10^5$	$2.82 \cdot 10^6$	$2.90 \cdot 10^{-1}$	$2.56 \cdot 10^{-1}$
User_1.calling.busy (p:0, id:0)	$8.40 \cdot 10^5$	$2.63 \cdot 10^6$	$2.58 \cdot 10^5$	$8.01 \cdot 10^5$	$3.64 \cdot 10^{-1}$	$2.12 \cdot 10^{-1}$
User_1.calling.unobtainable (p:0, id:1)	$8.40 \cdot 10^5$	$2.63 \cdot 10^6$	$6.13 \cdot 10^4$	$1.93 \cdot 10^5$	$1.62 \cdot 10^{-1}$	$3.11 \cdot 10^{-1}$
User_1.calling.ringback (p:0, id:2)	$8.39 \cdot 10^5$	$2.63 \cdot 10^6$	$6.13 \cdot 10^4$	$1.92 \cdot 10^5$	$1.23 \cdot 10^{-1}$	$2.34 \cdot 10^{-1}$
User_1.calling.busy (p:0, id:3)	$8.40 \cdot 10^5$	$2.63 \cdot 10^6$	$2.64 \cdot 10^5$	$8.39 \cdot 10^5$	$3.12 \cdot 10^{-1}$	$2.26 \cdot 10^{-1}$
User_1.calling.calling (p:0, id:4)	$8.40 \cdot 10^5$	$2.63 \cdot 10^6$	$2.58 \cdot 10^5$	$8.01 \cdot 10^5$	$3.49 \cdot 10^{-1}$	$1.63 \cdot 10^{-1}$
User_1.calling.oalert (p:0, id:5)	$8.39 \cdot 10^5$	$2.63 \cdot 10^6$	$7.50 \cdot 10^{-1}$	1.02	$5.85 \cdot 10^{-2}$	$1.05 \cdot 10^{-1}$
`User_1.dialing`	$7.36 \cdot 10^5$	$2.31 \cdot 10^6$	$7.36 \cdot 10^5$	$2.31 \cdot 10^6$	1.00	0.00
User_1.dialing.idle (p:0, id:0)	$1.23 \cdot 10^5$	$3.85 \cdot 10^5$	$1.23 \cdot 10^5$	$3.85 \cdot 10^5$	1.00	0.00
User_1.dialing.calling (p:0, id:1)	$1.23 \cdot 10^5$	$3.85 \cdot 10^5$	$1.23 \cdot 10^5$	$3.85 \cdot 10^5$	1.00	0.00
User_1.dialing.calling (p:0, id:2)	$1.23 \cdot 10^5$	$3.85 \cdot 10^5$	$1.23 \cdot 10^5$	$3.85 \cdot 10^5$	1.00	0.00
User_1.dialing.calling (p:0, id:3)	$1.23 \cdot 10^5$	$3.85 \cdot 10^5$	$1.23 \cdot 10^5$	$3.85 \cdot 10^5$	1.00	0.00
User_1.dialing.calling (p:0, id:4)	$1.23 \cdot 10^5$	$3.85 \cdot 10^5$	$1.23 \cdot 10^5$	$3.85 \cdot 10^5$	1.00	0.00
User_1.dialing.calling (p:0, id:5)	$1.23 \cdot 10^5$	$3.85 \cdot 10^5$	$1.23 \cdot 10^5$	$3.85 \cdot 10^5$	1.00	0.00
`User_1.dveoringout`	$7.00 \cdot 10^{-1}$	$8.65 \cdot 10^{-1}$	$7.00 \cdot 10^{-1}$	$8.65 \cdot 10^{-1}$	1.00	0.00
User_1.dveoringout.idle (p:0, id:0)	$7.00 \cdot 10^{-1}$	$8.65 \cdot 10^{-1}$	$7.00 \cdot 10^{-1}$	$8.65 \cdot 10^{-1}$	1.00	0.00
`User_1.idle`	$4.28 \cdot 10^7$	$1.81 \cdot 10^7$	$2.14 \cdot 10^7$	$9.03 \cdot 10^6$	$5.00 \cdot 10^{-1}$	$1.30 \cdot 10^{-4}$
User_1.idle.dialing (p:0, id:0)	$2.14 \cdot 10^7$	$9.04 \cdot 10^6$	$7.36 \cdot 10^5$	$2.31 \cdot 10^6$	$1.00 \cdot 10^{-1}$	$3.08 \cdot 10^{-1}$
User_1.idle.qi (p:0, id:1)	$2.14 \cdot 10^7$	$9.03 \cdot 10^6$	$2.06 \cdot 10^7$	$1.04 \cdot 10^7$	$9.00 \cdot 10^{-1}$	$3.08 \cdot 10^{-1}$
`User_1.oalert`	2.60	6.18	$7.50 \cdot 10^{-1}$	1.02	$5.39 \cdot 10^{-1}$	$3.42 \cdot 10^{-1}$
User_1.oalert.oconnected (p:0, id:1)	1.20	3.59	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	$1.25 \cdot 10^{-2}$	$2.80 \cdot 10^{-2}$
User_1.oalert.dveoringout (p:0, id:2)	$7.00 \cdot 10^{-1}$	$8.65 \cdot 10^{-1}$	$7.00 \cdot 10^{-1}$	$8.65 \cdot 10^{-1}$	1.00	0.00
`User_1.oconnected`	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	1.00	NaN
User_1.oconnected.idle (p:0, id:0)	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	1.00	NaN
`User_1.qi`	$4.13 \cdot 10^7$	$2.08 \cdot 10^7$	$2.06 \cdot 10^7$	$1.04 \cdot 10^7$	$5.26 \cdot 10^{-1}$	$1.15 \cdot 10^{-1}$
User_1.qi.talert (p:0, id:0)	$2.06 \cdot 10^7$	$1.04 \cdot 10^7$	$5.50 \cdot 10^{-1}$	$5.10 \cdot 10^{-1}$	$5.26 \cdot 10^{-2}$	$2.29 \cdot 10^{-1}$
User_1.qi.idle (p:0, id:1)	$2.06 \cdot 10^7$	$1.04 \cdot 10^7$	$2.06 \cdot 10^7$	$1.04 \cdot 10^7$	1.00	$7.07 \cdot 10^{-8}$
`User_1.ringback`	$6.13 \cdot 10^4$	$1.92 \cdot 10^5$	$6.13 \cdot 10^4$	$1.92 \cdot 10^5$	$8.89 \cdot 10^{-1}$	$2.69 \cdot 10^{-1}$
User_1.ringback.idle (p:0, id:0)	$3.06 \cdot 10^4$	$9.61 \cdot 10^4$	$3.06 \cdot 10^4$	$9.61 \cdot 10^4$	1.00	0.00
User_1.ringback.calling (p:0, id:1)	$3.07 \cdot 10^4$	$9.64 \cdot 10^4$	$3.07 \cdot 10^4$	$9.64 \cdot 10^4$	$7.50 \cdot 10^{-1}$	$4.63 \cdot 10^{-1}$
`User_1.talert`	1.80	2.63	$5.50 \cdot 10^{-1}$	$5.10 \cdot 10^{-1}$	$5.89 \cdot 10^{-1}$	$4.07 \cdot 10^{-1}$
User_1.talert.tpickup (p:0, id:1)	$7.00 \cdot 10^{-1}$	1.30	$1.00 \cdot 10^{-1}$	$3.08 \cdot 10^{-1}$	$2.86 \cdot 10^{-1}$	$4.88 \cdot 10^{-1}$
User_1.talert.idle (p:0, id:2)	$6.50 \cdot 10^{-1}$	$7.45 \cdot 10^{-1}$	$4.50 \cdot 10^{-1}$	$5.10 \cdot 10^{-1}$	$7.50 \cdot 10^{-1}$	$3.54 \cdot 10^{-1}$
`User_1.tpickup`	$1.00 \cdot 10^{-1}$	$3.08 \cdot 10^{-1}$	$1.00 \cdot 10^{-1}$	$3.08 \cdot 10^{-1}$	1.00	0.00
User_1.tpickup.idle (p:0, id:1)	$1.00 \cdot 10^{-1}$	$3.08 \cdot 10^{-1}$	$1.00 \cdot 10^{-1}$	$3.08 \cdot 10^{-1}$	1.00	0.00
`User_1.unobtainable`	$6.13 \cdot 10^4$	$1.93 \cdot 10^5$	$6.13 \cdot 10^4$	$1.93 \cdot 10^5$	1.00	0.00
User_1.unobtainable.idle (p:0, id:0)	$6.13 \cdot 10^4$	$1.93 \cdot 10^5$	$6.13 \cdot 10^4$	$1.93 \cdot 10^5$	1.00	0.00

Table C.30: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model `Telephony` (state machine `User_1` only). The Java code has been generated with the 'Random' decision mode enabled. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

Performance results for target model 'Telephony' ($n = 20, t = 30$, Random, User_2)						
Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
`User_2.busy`	$1.57 \cdot 10^{6}$	$2.25 \cdot 10^{6}$	$1.57 \cdot 10^{6}$	$2.25 \cdot 10^{6}$	1.00	0.00
User_2.busy.idle (p:0, id:0)	$1.57 \cdot 10^{6}$	$2.25 \cdot 10^{6}$	$1.57 \cdot 10^{6}$	$2.25 \cdot 10^{6}$	1.00	0.00
`User_2.calling`	$1.79 \cdot 10^{7}$	$2.57 \cdot 10^{7}$	$3.17 \cdot 10^{6}$	$4.55 \cdot 10^{6}$	$1.93 \cdot 10^{-1}$	$8.63 \cdot 10^{-2}$
User_2.calling.busy (p:0, id:0)	$2.98 \cdot 10^{6}$	$4.28 \cdot 10^{6}$	$1.18 \cdot 10^{6}$	$1.70 \cdot 10^{6}$	$4.09 \cdot 10^{-1}$	$1.84 \cdot 10^{-1}$
User_2.calling.unobtainable (p:0, id:1)	$2.98 \cdot 10^{6}$	$4.28 \cdot 10^{6}$	$1.85 \cdot 10^{5}$	$2.65 \cdot 10^{5}$	$1.06 \cdot 10^{-1}$	$2.27 \cdot 10^{-1}$
User_2.calling.ringback (p:0, id:2)	$2.98 \cdot 10^{6}$	$4.28 \cdot 10^{6}$	$1.85 \cdot 10^{5}$	$2.65 \cdot 10^{5}$	$1.06 \cdot 10^{-1}$	$2.27 \cdot 10^{-1}$
User_2.calling.busy (p:0, id:3)	$2.98 \cdot 10^{6}$	$4.28 \cdot 10^{6}$	$3.93 \cdot 10^{5}$	$5.58 \cdot 10^{5}$	$1.20 \cdot 10^{-1}$	$7.93 \cdot 10^{-2}$
User_2.calling.calling (p:0, id:4)	$2.98 \cdot 10^{6}$	$4.28 \cdot 10^{6}$	$1.22 \cdot 10^{6}$	$1.77 \cdot 10^{6}$	$3.53 \cdot 10^{-1}$	$1.72 \cdot 10^{-1}$
User_2.calling.oalert (p:0, id:5)	$2.98 \cdot 10^{6}$	$4.28 \cdot 10^{6}$	1.15	1.18	$2.11 \cdot 10^{-2}$	$3.65 \cdot 10^{-2}$
`User_2.dialing`	$2.22 \cdot 10^{6}$	$3.18 \cdot 10^{6}$	$2.22 \cdot 10^{6}$	$3.18 \cdot 10^{6}$	1.00	0.00
User_2.dialing.idle (p:0, id:0)	$3.70 \cdot 10^{5}$	$5.29 \cdot 10^{5}$	$3.70 \cdot 10^{5}$	$5.29 \cdot 10^{5}$	1.00	0.00
User_2.dialing.calling (p:0, id:1)	$3.70 \cdot 10^{5}$	$5.30 \cdot 10^{5}$	$3.70 \cdot 10^{5}$	$5.30 \cdot 10^{5}$	1.00	0.00
User_2.dialing.calling (p:0, id:2)	$3.70 \cdot 10^{5}$	$5.30 \cdot 10^{5}$	$3.70 \cdot 10^{5}$	$5.30 \cdot 10^{5}$	1.00	0.00
User_2.dialing.calling (p:0, id:3)	$3.70 \cdot 10^{5}$	$5.30 \cdot 10^{5}$	$3.70 \cdot 10^{5}$	$5.30 \cdot 10^{5}$	1.00	0.00
User_2.dialing.calling (p:0, id:4)	$3.70 \cdot 10^{5}$	$5.29 \cdot 10^{5}$	$3.70 \cdot 10^{5}$	$5.29 \cdot 10^{5}$	1.00	0.00
User_2.dialing.calling (p:0, id:5)	$3.70 \cdot 10^{5}$	$5.30 \cdot 10^{5}$	$3.70 \cdot 10^{5}$	$5.30 \cdot 10^{5}$	1.00	0.00
`User_2.dveoringout`	1.10	1.07	1.10	1.07	1.00	0.00
User_2.dveoringout.idle (p:0, id:0)	1.10	1.07	1.10	1.07	1.00	0.00
`User_2.idle`	$2.76 \cdot 10^{7}$	$2.07 \cdot 10^{7}$	$1.38 \cdot 10^{7}$	$1.04 \cdot 10^{7}$	$5.05 \cdot 10^{-1}$	$1.66 \cdot 10^{-2}$
User_2.idle.dialing (p:0, id:0)	$1.38 \cdot 10^{7}$	$1.04 \cdot 10^{7}$	$2.22 \cdot 10^{6}$	$3.18 \cdot 10^{6}$	$4.33 \cdot 10^{-1}$	$4.97 \cdot 10^{-1}$
User_2.idle.qi (p:0, id:1)	$1.38 \cdot 10^{7}$	$1.04 \cdot 10^{7}$	$1.16 \cdot 10^{7}$	$1.23 \cdot 10^{7}$	$5.77 \cdot 10^{-1}$	$4.91 \cdot 10^{-1}$
`User_2.oalert`	3.65	5.22	1.15	1.18	$5.42 \cdot 10^{-1}$	$3.36 \cdot 10^{-1}$
User_2.oalert.oconnected (p:0, id:1)	1.40	2.35	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	$2.78 \cdot 10^{-2}$	$8.33 \cdot 10^{-2}$
User_2.oalert.dveoringout (p:0, id:2)	1.10	1.07	1.10	1.07	1.00	0.00
`User_2.oconnected`	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	1.00	NaN
User_2.oconnected.idle (p:0, id:0)	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	1.00	NaN
`User_2.qi`	$2.31 \cdot 10^{7}$	$2.46 \cdot 10^{7}$	$1.16 \cdot 10^{7}$	$1.23 \cdot 10^{7}$	$5.83 \cdot 10^{-1}$	$1.72 \cdot 10^{-1}$
User_2.qi.talert (p:0, id:0)	$1.16 \cdot 10^{7}$	$1.23 \cdot 10^{7}$	$6.50 \cdot 10^{-1}$	1.04	$3.13 \cdot 10^{-1}$	$4.79 \cdot 10^{-1}$
User_2.qi.idle (p:0, id:1)	$1.16 \cdot 10^{7}$	$1.23 \cdot 10^{7}$	$1.16 \cdot 10^{7}$	$1.23 \cdot 10^{7}$	$7.86 \cdot 10^{-1}$	$4.26 \cdot 10^{-1}$
`User_2.ringback`	$1.85 \cdot 10^{5}$	$2.65 \cdot 10^{5}$	$1.85 \cdot 10^{5}$	$2.65 \cdot 10^{5}$	$9.76 \cdot 10^{-1}$	$8.91 \cdot 10^{-2}$
User_2.ringback.idle (p:0, id:0)	$9.26 \cdot 10^{4}$	$1.32 \cdot 10^{5}$	$9.26 \cdot 10^{4}$	$1.32 \cdot 10^{5}$	1.00	0.00
User_2.ringback.calling (p:0, id:1)	$9.26 \cdot 10^{4}$	$1.32 \cdot 10^{5}$	$9.26 \cdot 10^{4}$	$1.32 \cdot 10^{5}$	$9.17 \cdot 10^{-1}$	$2.89 \cdot 10^{-1}$
`User_2.talert`	1.30	3.16	$6.50 \cdot 10^{-1}$	1.04	$7.86 \cdot 10^{-1}$	$3.03 \cdot 10^{-1}$
User_2.talert.errorstate (p:0, id:0)	$5.00 \cdot 10^{-1}$	1.40	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	$2.50 \cdot 10^{-1}$	$5.00 \cdot 10^{-1}$
User_2.talert.tpickup (p:0, id:1)	$6.00 \cdot 10^{-1}$	1.39	$4.50 \cdot 10^{-1}$	$8.26 \cdot 10^{-1}$	$9.17 \cdot 10^{-1}$	$2.04 \cdot 10^{-1}$
User_2.talert.idle (p:0, id:2)	$2.00 \cdot 10^{-1}$	$5.23 \cdot 10^{-1}$	$1.50 \cdot 10^{-1}$	$3.66 \cdot 10^{-1}$	$8.33 \cdot 10^{-1}$	$2.89 \cdot 10^{-1}$
`User_2.tconnected`	$1.30 \cdot 10^{7}$	$5.80 \cdot 10^{7}$	$4.32 \cdot 10^{6}$	$1.93 \cdot 10^{7}$	$1.81 \cdot 10^{-1}$	$1.40 \cdot 10^{-1}$
User_2.tconnected.tconnected (p:0, id:0)	$4.32 \cdot 10^{6}$	$1.93 \cdot 10^{7}$	$2.16 \cdot 10^{6}$	$9.66 \cdot 10^{6}$	$1.97 \cdot 10^{-1}$	$2.66 \cdot 10^{-1}$
User_2.tconnected.tconnected (p:0, id:1)	$4.32 \cdot 10^{6}$	$1.93 \cdot 10^{7}$	$2.16 \cdot 10^{6}$	$9.66 \cdot 10^{6}$	$2.08 \cdot 10^{-1}$	$2.60 \cdot 10^{-1}$
User_2.tconnected.idle (p:0, id:2)	$4.32 \cdot 10^{6}$	$1.93 \cdot 10^{7}$	$1.50 \cdot 10^{-1}$	$3.66 \cdot 10^{-1}$	$1.37 \cdot 10^{-1}$	$1.32 \cdot 10^{-1}$
`User_2.tpickup`	$9.50 \cdot 10^{-1}$	1.93	$4.50 \cdot 10^{-1}$	$8.26 \cdot 10^{-1}$	$6.20 \cdot 10^{-1}$	$3.15 \cdot 10^{-1}$
User_2.tpickup.tconnected (p:0, id:0)	$2.50 \cdot 10^{-1}$	$4.44 \cdot 10^{-1}$	$2.00 \cdot 10^{-1}$	$4.10 \cdot 10^{-1}$	$8.00 \cdot 10^{-1}$	$4.47 \cdot 10^{-1}$
User_2.tpickup.idle (p:0, id:1)	$7.00 \cdot 10^{-1}$	1.63	$2.50 \cdot 10^{-1}$	$6.39 \cdot 10^{-1}$	$3.33 \cdot 10^{-1}$	$2.36 \cdot 10^{-1}$
`User_2.unobtainable`	$1.85 \cdot 10^{5}$	$2.65 \cdot 10^{5}$	$1.85 \cdot 10^{5}$	$2.65 \cdot 10^{5}$	1.00	0.00
User_2.unobtainable.idle (p:0, id:0)	$1.85 \cdot 10^{5}$	$2.65 \cdot 10^{5}$	$1.85 \cdot 10^{5}$	$2.65 \cdot 10^{5}$	1.00	0.00

Table C.31: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model `Telephony` (state machine `User_2` only). The Java code has been generated with the 'Random' decision mode enabled. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

Performance results for target model 'Telephony' ($n = 20, t = 30$, Random, User_3)						
Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
`User_3.busy`	$7.68 \cdot 10^5$	$1.65 \cdot 10^6$	$7.68 \cdot 10^5$	$1.65 \cdot 10^6$	1.00	0.00
User_3.busy.idle (p:0, id:0)	$7.68 \cdot 10^5$	$1.65 \cdot 10^6$	$7.68 \cdot 10^5$	$1.65 \cdot 10^6$	1.00	0.00
`User_3.calling`	$1.22 \cdot 10^7$	$2.63 \cdot 10^7$	$2.13 \cdot 10^6$	$4.58 \cdot 10^6$	$1.76 \cdot 10^{-1}$	$3.84 \cdot 10^{-2}$
User_3.calling.busy (p:0, id:0)	$2.04 \cdot 10^6$	$4.39 \cdot 10^6$	$7.68 \cdot 10^5$	$1.65 \cdot 10^6$	$3.76 \cdot 10^{-1}$	$1.03 \cdot 10^{-1}$
User_3.calling.unobtainable (p:0, id:1)	$2.04 \cdot 10^6$	$4.39 \cdot 10^6$	$9.03 \cdot 10^4$	$1.94 \cdot 10^5$	$4.24 \cdot 10^{-2}$	$5.77 \cdot 10^{-2}$
User_3.calling.ringback (p:0, id:2)	$2.04 \cdot 10^6$	$4.39 \cdot 10^6$	$9.04 \cdot 10^4$	$1.94 \cdot 10^5$	$8.21 \cdot 10^{-2}$	$1.04 \cdot 10^{-1}$
User_3.calling.calling (p:0, id:4)	$2.04 \cdot 10^6$	$4.39 \cdot 10^6$	$1.18 \cdot 10^6$	$2.55 \cdot 10^6$	$4.98 \cdot 10^{-1}$	$1.25 \cdot 10^{-1}$
User_3.calling.oalert (p:0, id:5)	$2.04 \cdot 10^6$	$4.39 \cdot 10^6$	$7.50 \cdot 10^{-1}$	$8.51 \cdot 10^{-1}$	$2.29 \cdot 10^{-2}$	$4.25 \cdot 10^{-2}$
`User_3.dialing`	$1.08 \cdot 10^6$	$2.33 \cdot 10^6$	$1.08 \cdot 10^6$	$2.33 \cdot 10^6$	1.00	0.00
User_3.dialing.idle (p:0, id:0)	$1.81 \cdot 10^5$	$3.88 \cdot 10^5$	$1.81 \cdot 10^5$	$3.88 \cdot 10^5$	1.00	0.00
User_3.dialing.calling (p:0, id:1)	$1.81 \cdot 10^5$	$3.88 \cdot 10^5$	$1.81 \cdot 10^5$	$3.88 \cdot 10^5$	1.00	0.00
User_3.dialing.calling (p:0, id:2)	$1.81 \cdot 10^5$	$3.88 \cdot 10^5$	$1.81 \cdot 10^5$	$3.88 \cdot 10^5$	1.00	0.00
User_3.dialing.calling (p:0, id:3)	$1.81 \cdot 10^5$	$3.88 \cdot 10^5$	$1.81 \cdot 10^5$	$3.88 \cdot 10^5$	1.00	0.00
User_3.dialing.calling (p:0, id:4)	$1.81 \cdot 10^5$	$3.88 \cdot 10^5$	$1.81 \cdot 10^5$	$3.88 \cdot 10^5$	1.00	0.00
User_3.dialing.calling (p:0, id:5)	$1.81 \cdot 10^5$	$3.88 \cdot 10^5$	$1.81 \cdot 10^5$	$3.88 \cdot 10^5$	1.00	0.00
`User_3.dveoringout`	$7.00 \cdot 10^{-1}$	$8.65 \cdot 10^{-1}$	$7.00 \cdot 10^{-1}$	$8.65 \cdot 10^{-1}$	1.00	0.00
User_3.dveoringout.idle (p:0, id:0)	$7.00 \cdot 10^{-1}$	$8.65 \cdot 10^{-1}$	$7.00 \cdot 10^{-1}$	$8.65 \cdot 10^{-1}$	1.00	0.00
`User_3.idle`	$3.71 \cdot 10^7$	$1.81 \cdot 10^7$	$1.86 \cdot 10^7$	$9.05 \cdot 10^6$	$5.00 \cdot 10^{-1}$	$1.16 \cdot 10^{-4}$
User_3.idle.dialing (p:0, id:0)	$1.86 \cdot 10^7$	$9.05 \cdot 10^6$	$1.08 \cdot 10^6$	$2.33 \cdot 10^6$	$2.00 \cdot 10^{-1}$	$4.10 \cdot 10^{-1}$
User_3.idle.qi (p:0, id:1)	$1.86 \cdot 10^7$	$9.05 \cdot 10^6$	$1.75 \cdot 10^7$	$1.08 \cdot 10^7$	$8.00 \cdot 10^{-1}$	$4.10 \cdot 10^{-1}$
`User_3.oalert`	1.70	2.00	$7.50 \cdot 10^{-1}$	$8.51 \cdot 10^{-1}$	$5.67 \cdot 10^{-1}$	$3.14 \cdot 10^{-1}$
User_3.oalert.oconnected (p:0, id:1)	$3.50 \cdot 10^{-1}$	$9.33 \cdot 10^{-1}$	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	$2.50 \cdot 10^{-1}$	$5.00 \cdot 10^{-1}$
User_3.oalert.dveoringout (p:0, id:2)	$7.00 \cdot 10^{-1}$	$8.65 \cdot 10^{-1}$	$7.00 \cdot 10^{-1}$	$8.65 \cdot 10^{-1}$	1.00	0.00
`User_3.oconnected`	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	1.00	NaN
User_3.oconnected.idle (p:0, id:0)	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	1.00	NaN
`User_3.qi`	$3.49 \cdot 10^7$	$2.16 \cdot 10^7$	$1.75 \cdot 10^7$	$1.08 \cdot 10^7$	$5.00 \cdot 10^{-1}$	$8.25 \cdot 10^{-5}$
User_3.qi.talert (p:0, id:0)	$1.75 \cdot 10^7$	$1.08 \cdot 10^7$	$7.50 \cdot 10^{-1}$	1.02	$5.00 \cdot 10^{-8}$	$5.86 \cdot 10^{-8}$
User_3.qi.idle (p:0, id:1)	$1.75 \cdot 10^7$	$1.08 \cdot 10^7$	$1.75 \cdot 10^7$	$1.08 \cdot 10^7$	1.00	$7.80 \cdot 10^{-8}$
`User_3.ringback`	$9.04 \cdot 10^4$	$1.94 \cdot 10^5$	$9.04 \cdot 10^4$	$1.94 \cdot 10^5$	$9.11 \cdot 10^{-1}$	$1.91 \cdot 10^{-1}$
User_3.ringback.idle (p:0, id:0)	$4.53 \cdot 10^4$	$9.73 \cdot 10^4$	$4.53 \cdot 10^4$	$9.73 \cdot 10^4$	1.00	0.00
User_3.ringback.calling (p:0, id:1)	$4.51 \cdot 10^4$	$9.70 \cdot 10^4$	$4.51 \cdot 10^4$	$9.70 \cdot 10^4$	$7.99 \cdot 10^{-1}$	$3.72 \cdot 10^{-1}$
`User_3.talert`	1.50	2.26	$7.50 \cdot 10^{-1}$	1.02	$6.26 \cdot 10^{-1}$	$3.12 \cdot 10^{-1}$
User_3.talert.tpickup (p:0, id:1)	$5.00 \cdot 10^{-1}$	$9.46 \cdot 10^{-1}$	$4.50 \cdot 10^{-1}$	$8.87 \cdot 10^{-1}$	$9.00 \cdot 10^{-1}$	$2.24 \cdot 10^{-1}$
User_3.talert.idle (p:0, id:2)	$5.00 \cdot 10^{-1}$	$7.61 \cdot 10^{-1}$	$3.00 \cdot 10^{-1}$	$4.70 \cdot 10^{-1}$	$6.43 \cdot 10^{-1}$	$3.78 \cdot 10^{-1}$
`User_3.tconnected`	$1.22 \cdot 10^1$	$3.85 \cdot 10^1$	$3.00 \cdot 10^{-1}$	$8.01 \cdot 10^{-1}$	$4.52 \cdot 10^{-2}$	$3.61 \cdot 10^{-2}$
User_3.tconnected.tconnected (p:0, id:0)	4.05	$1.26 \cdot 10^1$	$1.00 \cdot 10^{-1}$	$3.08 \cdot 10^{-1}$	$5.16 \cdot 10^{-2}$	$4.51 \cdot 10^{-2}$
User_3.tconnected.tconnected (p:0, id:1)	4.65	$1.43 \cdot 10^1$	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	$2.38 \cdot 10^{-2}$	$4.12 \cdot 10^{-2}$
User_3.tconnected.idle (p:0, id:2)	3.55	$1.16 \cdot 10^1$	$1.50 \cdot 10^{-1}$	$3.66 \cdot 10^{-1}$	$7.39 \cdot 10^{-2}$	$4.81 \cdot 10^{-2}$
`User_3.tpickup`	$9.00 \cdot 10^{-1}$	1.94	$4.50 \cdot 10^{-1}$	$8.87 \cdot 10^{-1}$	$6.32 \cdot 10^{-1}$	$3.37 \cdot 10^{-1}$
User_3.tpickup.tconnected (p:0, id:0)	$4.00 \cdot 10^{-1}$	$8.83 \cdot 10^{-1}$	$1.50 \cdot 10^{-1}$	$3.66 \cdot 10^{-1}$	$4.58 \cdot 10^{-1}$	$4.17 \cdot 10^{-1}$
User_3.tpickup.idle (p:0, id:1)	$5.00 \cdot 10^{-1}$	1.24	$3.00 \cdot 10^{-1}$	$6.57 \cdot 10^{-1}$	$7.25 \cdot 10^{-1}$	$3.20 \cdot 10^{-1}$
`User_3.unobtainable`	$9.03 \cdot 10^4$	$1.94 \cdot 10^5$	$9.03 \cdot 10^4$	$1.94 \cdot 10^5$	1.00	0.00
User_3.unobtainable.idle (p:0, id:0)	$9.03 \cdot 10^4$	$1.94 \cdot 10^5$	$9.03 \cdot 10^4$	$1.94 \cdot 10^5$	1.00	0.00

Table C.32: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model `Telephony` (state machine `User_3` only). The Java code has been generated with the 'Random' decision mode enabled. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

C.3 Locking Mode Frequency Tables

C.3.1 Synthetic Test: CounterDistributor

Performance results for target model '`CounterDistributor`' ($n = 20, t = 30$, Variable)						
Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
`Counter.C`	$1.21 \cdot 10^8$	$1.61 \cdot 10^7$	$1.21 \cdot 10^8$	$1.61 \cdot 10^7$	1.00	0.00
Counter.C.C (p:0, id:0)	$1.21 \cdot 10^8$	$1.61 \cdot 10^7$	$1.21 \cdot 10^8$	$1.61 \cdot 10^7$	1.00	0.00
`Distributor.P`	$1.15 \cdot 10^8$	$1.66 \cdot 10^7$	$1.15 \cdot 10^8$	$1.66 \cdot 10^7$	1.00	0.00
Distributor.P.P (p:0, id:0)	$1.15 \cdot 10^8$	$1.66 \cdot 10^7$	$1.13 \cdot 10^7$	$1.61 \cdot 10^6$	$9.80 \cdot 10^{-2}$	$2.37 \cdot 10^{-3}$
Distributor.P.P (p:0, id:1)	$1.04 \cdot 10^8$	$1.50 \cdot 10^7$	$1.13 \cdot 10^7$	$1.58 \cdot 10^6$	$1.09 \cdot 10^{-1}$	$2.59 \cdot 10^{-3}$
Distributor.P.P (p:0, id:2)	$9.24 \cdot 10^7$	$1.34 \cdot 10^7$	$1.14 \cdot 10^7$	$1.62 \cdot 10^6$	$1.23 \cdot 10^{-1}$	$2.42 \cdot 10^{-3}$
Distributor.P.P (p:0, id:3)	$8.10 \cdot 10^7$	$1.18 \cdot 10^7$	$1.15 \cdot 10^7$	$1.65 \cdot 10^6$	$1.42 \cdot 10^{-1}$	$3.02 \cdot 10^{-3}$
Distributor.P.P (p:0, id:4)	$6.95 \cdot 10^7$	$1.02 \cdot 10^7$	$1.14 \cdot 10^7$	$1.67 \cdot 10^6$	$1.65 \cdot 10^{-1}$	$2.73 \cdot 10^{-3}$
Distributor.P.P (p:0, id:5)	$5.81 \cdot 10^7$	$8.56 \cdot 10^6$	$1.15 \cdot 10^7$	$1.69 \cdot 10^6$	$1.98 \cdot 10^{-1}$	$3.99 \cdot 10^{-3}$
Distributor.P.P (p:0, id:6)	$4.66 \cdot 10^7$	$6.89 \cdot 10^6$	$1.15 \cdot 10^7$	$1.76 \cdot 10^6$	$2.47 \cdot 10^{-1}$	$7.27 \cdot 10^{-3}$
Distributor.P.P (p:0, id:7)	$3.51 \cdot 10^7$	$5.18 \cdot 10^6$	$1.16 \cdot 10^7$	$1.76 \cdot 10^6$	$3.32 \cdot 10^{-1}$	$5.37 \cdot 10^{-3}$
Distributor.P.P (p:0, id:8)	$2.34 \cdot 10^7$	$3.43 \cdot 10^6$	$1.17 \cdot 10^7$	$1.77 \cdot 10^6$	$4.99 \cdot 10^{-1}$	$4.55 \cdot 10^{-3}$
Distributor.P.P (p:0, id:9)	$1.17 \cdot 10^7$	$1.67 \cdot 10^6$	$1.17 \cdot 10^7$	$1.67 \cdot 10^6$	1.00	0.00

Table C.33: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model `CounterDistributor`. The Java code has been generated with the 'Variable' locking mode enabled. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

Performance results for target model '`CounterDistributor`' ($n = 20, t = 30$, Statement)						
Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
`Counter.C`	$1.20 \cdot 10^8$	$1.51 \cdot 10^7$	$1.20 \cdot 10^8$	$1.51 \cdot 10^7$	1.00	0.00
Counter.C.C (p:0, id:0)	$1.20 \cdot 10^8$	$1.51 \cdot 10^7$	$1.20 \cdot 10^8$	$1.51 \cdot 10^7$	1.00	0.00
`Distributor.P`	$1.13 \cdot 10^8$	$1.14 \cdot 10^7$	$1.13 \cdot 10^8$	$1.14 \cdot 10^7$	1.00	0.00
Distributor.P.P (p:0, id:0)	$1.13 \cdot 10^8$	$1.14 \cdot 10^7$	$1.12 \cdot 10^7$	$1.10 \cdot 10^6$	$9.84 \cdot 10^{-2}$	$2.46 \cdot 10^{-3}$
Distributor.P.P (p:0, id:1)	$1.02 \cdot 10^8$	$1.03 \cdot 10^7$	$1.13 \cdot 10^7$	$1.07 \cdot 10^6$	$1.11 \cdot 10^{-1}$	$3.25 \cdot 10^{-3}$
Distributor.P.P (p:0, id:2)	$9.10 \cdot 10^7$	$9.29 \cdot 10^6$	$1.13 \cdot 10^7$	$1.14 \cdot 10^6$	$1.25 \cdot 10^{-1}$	$2.76 \cdot 10^{-3}$
Distributor.P.P (p:0, id:3)	$7.96 \cdot 10^7$	$8.18 \cdot 10^6$	$1.14 \cdot 10^7$	$1.13 \cdot 10^6$	$1.43 \cdot 10^{-1}$	$3.41 \cdot 10^{-3}$
Distributor.P.P (p:0, id:4)	$6.83 \cdot 10^7$	$7.08 \cdot 10^6$	$1.14 \cdot 10^7$	$1.15 \cdot 10^6$	$1.67 \cdot 10^{-1}$	$2.64 \cdot 10^{-3}$
Distributor.P.P (p:0, id:5)	$5.69 \cdot 10^7$	$5.94 \cdot 10^6$	$1.14 \cdot 10^7$	$1.18 \cdot 10^6$	$2.00 \cdot 10^{-1}$	$3.91 \cdot 10^{-3}$
Distributor.P.P (p:0, id:6)	$4.55 \cdot 10^7$	$4.79 \cdot 10^6$	$1.14 \cdot 10^7$	$1.13 \cdot 10^6$	$2.50 \cdot 10^{-1}$	$5.21 \cdot 10^{-3}$
Distributor.P.P (p:0, id:7)	$3.42 \cdot 10^7$	$3.69 \cdot 10^6$	$1.15 \cdot 10^7$	$1.24 \cdot 10^6$	$3.36 \cdot 10^{-1}$	$6.05 \cdot 10^{-3}$
Distributor.P.P (p:0, id:8)	$2.27 \cdot 10^7$	$2.48 \cdot 10^6$	$1.14 \cdot 10^7$	$1.28 \cdot 10^6$	$5.04 \cdot 10^{-1}$	$6.80 \cdot 10^{-3}$
Distributor.P.P (p:0, id:9)	$1.13 \cdot 10^7$	$1.22 \cdot 10^6$	$1.13 \cdot 10^7$	$1.22 \cdot 10^6$	1.00	0.00

Table C.34: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model `CounterDistributor`. The Java code has been generated with the 'Statement' locking mode enabled. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

C.3.2 Synthetic Test: Tokens

Performance results for target model 'Tokens' ($n = 20, t = 30$, Variable)						
Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
`A.act`	$2.96 \cdot 10^7$	$2.13 \cdot 10^6$	$2.96 \cdot 10^7$	$2.13 \cdot 10^6$	1.00	0.00
A.act.act (p:0, id:0)	$2.96 \cdot 10^7$	$2.13 \cdot 10^6$	$1.08 \cdot 10^6$	$1.82 \cdot 10^5$	$3.64 \cdot 10^{-2}$	$6.58 \cdot 10^{-3}$
A.act.update (p:1, id:1)	$2.86 \cdot 10^7$	$2.13 \cdot 10^6$	$2.57 \cdot 10^7$	$1.91 \cdot 10^6$	$9.00 \cdot 10^{-1}$	$6.86 \cdot 10^{-8}$
A.act.wait (p:1, id:2)	$2.86 \cdot 10^6$	$2.13 \cdot 10^5$	$2.86 \cdot 10^6$	$2.13 \cdot 10^5$	1.00	0.00
`A.update`	$2.86 \cdot 10^7$	$2.13 \cdot 10^6$	$2.86 \cdot 10^7$	$2.13 \cdot 10^6$	1.00	0.00
A.update.act (p:0, id:0)	$2.86 \cdot 10^7$	$2.13 \cdot 10^6$	$2.86 \cdot 10^7$	$2.13 \cdot 10^6$	1.00	0.00
`A.wait`	$7.07 \cdot 10^7$	$1.18 \cdot 10^7$	$4.72 \cdot 10^6$	$3.19 \cdot 10^5$	$6.87 \cdot 10^{-2}$	$1.25 \cdot 10^{-2}$
A.wait.wait (p:0, id:0)	$7.07 \cdot 10^7$	$1.18 \cdot 10^7$	$1.86 \cdot 10^6$	$1.92 \cdot 10^5$	$2.71 \cdot 10^{-2}$	$5.70 \cdot 10^{-3}$
A.wait.update (p:1, id:1)	$6.88 \cdot 10^7$	$1.19 \cdot 10^7$	$2.86 \cdot 10^6$	$2.13 \cdot 10^5$	$4.28 \cdot 10^{-2}$	$7.64 \cdot 10^{-3}$
`B.act`	$3.04 \cdot 10^7$	$2.31 \cdot 10^6$	$3.04 \cdot 10^7$	$2.31 \cdot 10^6$	1.00	0.00
B.act.act (p:0, id:0)	$3.04 \cdot 10^7$	$2.31 \cdot 10^6$	$9.59 \cdot 10^5$	$1.53 \cdot 10^5$	$3.17 \cdot 10^{-2}$	$5.14 \cdot 10^{-3}$
B.act.update (p:1, id:1)	$2.95 \cdot 10^7$	$2.30 \cdot 10^6$	$2.65 \cdot 10^7$	$2.07 \cdot 10^6$	$9.00 \cdot 10^{-1}$	$6.58 \cdot 10^{-8}$
B.act.wait (p:1, id:2)	$2.95 \cdot 10^6$	$2.30 \cdot 10^5$	$2.95 \cdot 10^6$	$2.30 \cdot 10^5$	1.00	0.00
`B.update`	$2.95 \cdot 10^7$	$2.30 \cdot 10^6$	$2.95 \cdot 10^7$	$2.30 \cdot 10^6$	1.00	0.00
B.update.act (p:0, id:0)	$2.95 \cdot 10^7$	$2.30 \cdot 10^6$	$2.95 \cdot 10^7$	$2.30 \cdot 10^6$	1.00	0.00
`B.wait`	$7.16 \cdot 10^7$	$7.43 \cdot 10^6$	$4.85 \cdot 10^6$	$3.75 \cdot 10^5$	$6.82 \cdot 10^{-2}$	$6.59 \cdot 10^{-3}$
B.wait.wait (p:0, id:0)	$7.16 \cdot 10^7$	$7.43 \cdot 10^6$	$1.90 \cdot 10^6$	$1.97 \cdot 10^5$	$2.68 \cdot 10^{-2}$	$3.25 \cdot 10^{-3}$
B.wait.update (p:1, id:1)	$6.97 \cdot 10^7$	$7.38 \cdot 10^6$	$2.95 \cdot 10^6$	$2.30 \cdot 10^5$	$4.26 \cdot 10^{-2}$	$4.15 \cdot 10^{-3}$
`C.act`	$3.03 \cdot 10^7$	$2.18 \cdot 10^6$	$3.03 \cdot 10^7$	$2.18 \cdot 10^6$	1.00	0.00
C.act.act (p:0, id:0)	$3.03 \cdot 10^7$	$2.18 \cdot 10^6$	$9.84 \cdot 10^5$	$1.79 \cdot 10^5$	$3.26 \cdot 10^{-2}$	$5.78 \cdot 10^{-3}$
C.act.wait (p:1, id:1)	$2.93 \cdot 10^7$	$2.15 \cdot 10^6$	$2.94 \cdot 10^6$	$2.16 \cdot 10^5$	$1.00 \cdot 10^{-1}$	$1.05 \cdot 10^{-7}$
C.act.update (p:1, id:2)	$2.64 \cdot 10^7$	$1.94 \cdot 10^6$	$2.64 \cdot 10^7$	$1.94 \cdot 10^6$	1.00	0.00
`C.update`	$2.93 \cdot 10^7$	$2.15 \cdot 10^6$	$2.93 \cdot 10^7$	$2.15 \cdot 10^6$	1.00	0.00
C.update.act (p:0, id:0)	$2.93 \cdot 10^7$	$2.15 \cdot 10^6$	$2.93 \cdot 10^7$	$2.15 \cdot 10^6$	1.00	0.00
`C.wait`	$7.72 \cdot 10^7$	$1.18 \cdot 10^7$	$4.90 \cdot 10^6$	$3.56 \cdot 10^5$	$6.47 \cdot 10^{-2}$	$9.51 \cdot 10^{-3}$
C.wait.wait (p:0, id:0)	$7.72 \cdot 10^7$	$1.18 \cdot 10^7$	$1.97 \cdot 10^6$	$1.78 \cdot 10^5$	$2.60 \cdot 10^{-2}$	$4.37 \cdot 10^{-3}$
C.wait.update (p:1, id:1)	$7.52 \cdot 10^7$	$1.19 \cdot 10^7$	$2.94 \cdot 10^6$	$2.16 \cdot 10^5$	$3.98 \cdot 10^{-2}$	$5.64 \cdot 10^{-3}$

Table C.35: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model `Tokens`. The Java code has been generated with the 'Variable' locking mode enabled. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

Performance results for target model 'Tokens' ($n = 20, t = 30$, Statement)						
Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
`A.act`	$2.74 \cdot 10^7$	$2.22 \cdot 10^6$	$2.74 \cdot 10^7$	$2.22 \cdot 10^6$	1.00	0.00
A.act.act (p:0, id:0)	$2.74 \cdot 10^7$	$2.22 \cdot 10^6$	$8.86 \cdot 10^5$	$1.49 \cdot 10^5$	$3.25 \cdot 10^{-2}$	$5.92 \cdot 10^{-3}$
A.act.update (p:1, id:1)	$2.65 \cdot 10^7$	$2.22 \cdot 10^6$	$2.39 \cdot 10^7$	$1.99 \cdot 10^6$	$9.00 \cdot 10^{-1}$	$6.54 \cdot 10^{-8}$
A.act.wait (p:1, id:2)	$2.66 \cdot 10^6$	$2.22 \cdot 10^5$	$2.66 \cdot 10^6$	$2.22 \cdot 10^5$	1.00	0.00
`A.update`	$2.65 \cdot 10^7$	$2.22 \cdot 10^6$	$2.65 \cdot 10^7$	$2.22 \cdot 10^6$	1.00	0.00
A.update.act (p:0, id:0)	$2.65 \cdot 10^7$	$2.22 \cdot 10^6$	$2.65 \cdot 10^7$	$2.22 \cdot 10^6$	1.00	0.00
`A.wait`	$7.68 \cdot 10^7$	$1.87 \cdot 10^7$	$4.57 \cdot 10^6$	$2.98 \cdot 10^5$	$6.25 \cdot 10^{-2}$	$1.39 \cdot 10^{-2}$
A.wait.wait (p:0, id:0)	$7.68 \cdot 10^7$	$1.87 \cdot 10^7$	$1.91 \cdot 10^6$	$1.61 \cdot 10^5$	$2.63 \cdot 10^{-2}$	$6.76 \cdot 10^{-3}$
A.wait.update (p:1, id:1)	$7.49 \cdot 10^7$	$1.88 \cdot 10^7$	$2.66 \cdot 10^6$	$2.22 \cdot 10^5$	$3.72 \cdot 10^{-2}$	$7.80 \cdot 10^{-3}$
`B.act`	$2.81 \cdot 10^7$	$2.22 \cdot 10^6$	$2.81 \cdot 10^7$	$2.22 \cdot 10^6$	1.00	0.00
B.act.act (p:0, id:0)	$2.81 \cdot 10^7$	$2.22 \cdot 10^6$	$7.49 \cdot 10^5$	$1.81 \cdot 10^5$	$2.69 \cdot 10^{-2}$	$7.38 \cdot 10^{-3}$
B.act.update (p:1, id:1)	$2.73 \cdot 10^7$	$2.24 \cdot 10^6$	$2.46 \cdot 10^7$	$2.02 \cdot 10^6$	$9.00 \cdot 10^{-1}$	$5.98 \cdot 10^{-8}$
B.act.wait (p:1, id:2)	$2.74 \cdot 10^6$	$2.25 \cdot 10^5$	$2.74 \cdot 10^6$	$2.25 \cdot 10^5$	1.00	0.00
`B.update`	$2.73 \cdot 10^7$	$2.24 \cdot 10^6$	$2.73 \cdot 10^7$	$2.24 \cdot 10^6$	1.00	0.00
B.update.act (p:0, id:0)	$2.73 \cdot 10^7$	$2.24 \cdot 10^6$	$2.73 \cdot 10^7$	$2.24 \cdot 10^6$	1.00	0.00
`B.wait`	$8.10 \cdot 10^7$	$1.54 \cdot 10^7$	$4.65 \cdot 10^6$	$3.89 \cdot 10^5$	$5.90 \cdot 10^{-2}$	$1.05 \cdot 10^{-2}$
B.wait.wait (p:0, id:0)	$8.10 \cdot 10^7$	$1.54 \cdot 10^7$	$1.91 \cdot 10^6$	$1.97 \cdot 10^5$	$2.42 \cdot 10^{-2}$	$4.71 \cdot 10^{-3}$
B.wait.update (p:1, id:1)	$7.91 \cdot 10^7$	$1.54 \cdot 10^7$	$2.74 \cdot 10^6$	$2.25 \cdot 10^5$	$3.56 \cdot 10^{-2}$	$6.29 \cdot 10^{-3}$
`C.act`	$2.88 \cdot 10^7$	$1.86 \cdot 10^6$	$2.88 \cdot 10^7$	$1.86 \cdot 10^6$	1.00	0.00
C.act.act (p:0, id:0)	$2.88 \cdot 10^7$	$1.86 \cdot 10^6$	$8.73 \cdot 10^5$	$1.48 \cdot 10^5$	$3.03 \cdot 10^{-2}$	$5.26 \cdot 10^{-3}$
C.act.wait (p:1, id:1)	$2.79 \cdot 10^7$	$1.83 \cdot 10^6$	$2.80 \cdot 10^6$	$1.83 \cdot 10^5$	$1.00 \cdot 10^{-1}$	$9.86 \cdot 10^{-8}$
C.act.update (p:1, id:2)	$2.51 \cdot 10^7$	$1.65 \cdot 10^6$	$2.51 \cdot 10^7$	$1.65 \cdot 10^6$	1.00	0.00
`C.update`	$2.79 \cdot 10^7$	$1.83 \cdot 10^6$	$2.79 \cdot 10^7$	$1.83 \cdot 10^6$	1.00	0.00
C.update.act (p:0, id:0)	$2.79 \cdot 10^7$	$1.83 \cdot 10^6$	$2.79 \cdot 10^7$	$1.83 \cdot 10^6$	1.00	0.00
`C.wait`	$8.89 \cdot 10^7$	$2.29 \cdot 10^7$	$4.66 \cdot 10^6$	$3.38 \cdot 10^5$	$5.54 \cdot 10^{-2}$	$1.32 \cdot 10^{-2}$
C.wait.wait (p:0, id:0)	$8.89 \cdot 10^7$	$2.29 \cdot 10^7$	$1.87 \cdot 10^6$	$2.02 \cdot 10^5$	$2.22 \cdot 10^{-2}$	$5.92 \cdot 10^{-3}$
C.wait.update (p:1, id:1)	$8.70 \cdot 10^7$	$2.29 \cdot 10^7$	$2.80 \cdot 10^6$	$1.83 \cdot 10^5$	$3.40 \cdot 10^{-2}$	$7.86 \cdot 10^{-3}$

Table C.36: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model `Tokens`. The Java code has been generated with the 'Statement' locking mode enabled. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

C.3.3 Practical Test: Elevator

Performance results for target model '`Elevator`' ($n = 20, t = 30$, Variable)						
Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
`cabin.idle`	$2.10 \cdot 10^{8}$	$4.36 \cdot 10^{7}$	$6.40 \cdot 10^{6}$	$1.09 \cdot 10^{6}$	$3.13 \cdot 10^{-2}$	$6.04 \cdot 10^{-3}$
cabin.idle.mov (p:0, id:0)	$2.10 \cdot 10^{8}$	$4.36 \cdot 10^{7}$	$6.40 \cdot 10^{6}$	$1.09 \cdot 10^{6}$	$3.13 \cdot 10^{-2}$	$6.04 \cdot 10^{-3}$
`cabin.mov`	$1.12 \cdot 10^{7}$	$1.91 \cdot 10^{6}$	$1.12 \cdot 10^{7}$	$1.91 \cdot 10^{6}$	1.00	0.00
cabin.mov.open (p:0, id:0)	$1.12 \cdot 10^{7}$	$1.91 \cdot 10^{6}$	$6.40 \cdot 10^{6}$	$1.09 \cdot 10^{6}$	$5.71 \cdot 10^{-1}$	$8.10 \cdot 10^{-5}$
cabin.mov.mov (p:0, id:1)	$4.81 \cdot 10^{6}$	$8.20 \cdot 10^{5}$	$2.40 \cdot 10^{6}$	$4.10 \cdot 10^{5}$	$5.00 \cdot 10^{-1}$	$1.18 \cdot 10^{-7}$
cabin.mov.mov (p:0, id:2)	$2.40 \cdot 10^{6}$	$4.10 \cdot 10^{5}$	$2.40 \cdot 10^{6}$	$4.10 \cdot 10^{5}$	1.00	0.00
`cabin.open`	$6.40 \cdot 10^{6}$	$1.09 \cdot 10^{6}$	$6.40 \cdot 10^{6}$	$1.09 \cdot 10^{6}$	1.00	0.00
cabin.open.idle (p:0, id:0)	$6.40 \cdot 10^{6}$	$1.09 \cdot 10^{6}$	$6.40 \cdot 10^{6}$	$1.09 \cdot 10^{6}$	1.00	0.00
`controller.done`	$6.40 \cdot 10^{6}$	$1.09 \cdot 10^{6}$	$6.40 \cdot 10^{6}$	$1.09 \cdot 10^{6}$	1.00	0.00
controller.done.wait (p:0, id:0)	$6.40 \cdot 10^{6}$	$1.09 \cdot 10^{6}$	$6.40 \cdot 10^{6}$	$1.09 \cdot 10^{6}$	1.00	0.00
`controller.wait`	$5.67 \cdot 10^{7}$	$1.71 \cdot 10^{7}$	$8.02 \cdot 10^{6}$	$1.37 \cdot 10^{6}$	$1.49 \cdot 10^{-1}$	$3.13 \cdot 10^{-2}$
controller.wait.work (p:0, id:0)	$5.67 \cdot 10^{7}$	$1.71 \cdot 10^{7}$	$8.02 \cdot 10^{6}$	$1.37 \cdot 10^{6}$	$1.49 \cdot 10^{-1}$	$3.13 \cdot 10^{-2}$
`controller.work`	$8.08 \cdot 10^{6}$	$1.38 \cdot 10^{6}$	$8.08 \cdot 10^{6}$	$1.38 \cdot 10^{6}$	1.00	0.00
controller.work.wait (p:0, id:0)	$8.08 \cdot 10^{6}$	$1.38 \cdot 10^{6}$	$1.62 \cdot 10^{6}$	$2.77 \cdot 10^{5}$	$2.00 \cdot 10^{-1}$	$3.27 \cdot 10^{-8}$
controller.work.done (p:0, id:1)	$6.46 \cdot 10^{6}$	$1.11 \cdot 10^{6}$	$6.40 \cdot 10^{6}$	$1.09 \cdot 10^{6}$	$9.91 \cdot 10^{-1}$	$6.82 \cdot 10^{-3}$
controller.work.work (p:0, id:2)	$6.09 \cdot 10^{4}$	$4.64 \cdot 10^{4}$	$6.09 \cdot 10^{4}$	$4.64 \cdot 10^{4}$	1.00	0.00
`environment.read`	$4.37 \cdot 10^{8}$	$8.45 \cdot 10^{7}$	$6.40 \cdot 10^{6}$	$1.09 \cdot 10^{6}$	$1.50 \cdot 10^{-2}$	$2.68 \cdot 10^{-3}$
environment.read.read (p:0, id:0)	$4.37 \cdot 10^{8}$	$8.45 \cdot 10^{7}$	$1.60 \cdot 10^{6}$	$2.73 \cdot 10^{5}$	$3.75 \cdot 10^{-3}$	$6.70 \cdot 10^{-4}$
environment.read.read (p:0, id:1)	$4.35 \cdot 10^{8}$	$8.44 \cdot 10^{7}$	$1.60 \cdot 10^{6}$	$2.73 \cdot 10^{5}$	$3.77 \cdot 10^{-3}$	$6.76 \cdot 10^{-4}$
environment.read.read (p:0, id:2)	$4.33 \cdot 10^{8}$	$8.43 \cdot 10^{7}$	$1.60 \cdot 10^{6}$	$2.73 \cdot 10^{5}$	$3.78 \cdot 10^{-3}$	$6.81 \cdot 10^{-4}$
environment.read.read (p:0, id:3)	$4.32 \cdot 10^{8}$	$8.42 \cdot 10^{7}$	$1.60 \cdot 10^{6}$	$2.73 \cdot 10^{5}$	$3.80 \cdot 10^{-3}$	$6.85 \cdot 10^{-4}$

Table C.37: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model `Elevator`. The Java code has been generated with the 'Variable' locking mode enabled. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

Performance results for target model 'Elevator' ($n = 20, t = 30$, Statement)						
Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
`cabin.idle`	$1.45 \cdot 10^8$	$1.91 \cdot 10^7$	$3.11 \cdot 10^6$	$3.00 \cdot 10^5$	$2.19 \cdot 10^{-2}$	$4.03 \cdot 10^{-3}$
cabin.idle.mov (p:0, id:0)	$1.45 \cdot 10^8$	$1.91 \cdot 10^7$	$3.11 \cdot 10^6$	$3.00 \cdot 10^5$	$2.19 \cdot 10^{-2}$	$4.03 \cdot 10^{-3}$
`cabin.mov`	$5.70 \cdot 10^6$	$5.60 \cdot 10^5$	$5.70 \cdot 10^6$	$5.60 \cdot 10^5$	1.00	0.00
cabin.mov.open (p:0, id:0)	$5.70 \cdot 10^6$	$5.60 \cdot 10^5$	$3.11 \cdot 10^6$	$3.00 \cdot 10^5$	$5.46 \cdot 10^{-1}$	$5.08 \cdot 10^{-3}$
cabin.mov.mov (p:0, id:1)	$2.59 \cdot 10^6$	$2.62 \cdot 10^5$	$1.30 \cdot 10^6$	$1.31 \cdot 10^5$	$5.00 \cdot 10^{-1}$	$2.08 \cdot 10^{-7}$
cabin.mov.mov (p:0, id:2)	$1.30 \cdot 10^6$	$1.31 \cdot 10^5$	$1.30 \cdot 10^6$	$1.31 \cdot 10^5$	1.00	0.00
`cabin.open`	$3.11 \cdot 10^6$	$3.00 \cdot 10^5$	$3.11 \cdot 10^6$	$3.00 \cdot 10^5$	1.00	0.00
cabin.open.idle (p:0, id:0)	$3.11 \cdot 10^6$	$3.00 \cdot 10^5$	$3.11 \cdot 10^6$	$3.00 \cdot 10^5$	1.00	0.00
`controller.done`	$3.11 \cdot 10^6$	$3.00 \cdot 10^5$	$3.11 \cdot 10^6$	$3.00 \cdot 10^5$	1.00	0.00
controller.done.wait (p:0, id:0)	$3.11 \cdot 10^6$	$3.00 \cdot 10^5$	$3.11 \cdot 10^6$	$3.00 \cdot 10^5$	1.00	0.00
`controller.wait`	$1.49 \cdot 10^8$	$1.71 \cdot 10^7$	$3.98 \cdot 10^6$	$3.86 \cdot 10^5$	$2.71 \cdot 10^{-2}$	$4.59 \cdot 10^{-3}$
controller.wait.work (p:0, id:0)	$1.49 \cdot 10^8$	$1.71 \cdot 10^7$	$3.98 \cdot 10^6$	$3.86 \cdot 10^5$	$2.71 \cdot 10^{-2}$	$4.59 \cdot 10^{-3}$
`controller.work`	$4.32 \cdot 10^6$	$4.36 \cdot 10^5$	$4.32 \cdot 10^6$	$4.36 \cdot 10^5$	1.00	0.00
controller.work.wait (p:0, id:0)	$4.32 \cdot 10^6$	$4.36 \cdot 10^5$	$8.64 \cdot 10^5$	$8.73 \cdot 10^4$	$2.00 \cdot 10^{-1}$	$5.30 \cdot 10^{-8}$
controller.work.done (p:0, id:1)	$3.46 \cdot 10^6$	$3.49 \cdot 10^5$	$3.11 \cdot 10^6$	$3.00 \cdot 10^5$	$9.01 \cdot 10^{-1}$	$1.84 \cdot 10^{-2}$
controller.work.work (p:0, id:2)	$3.44 \cdot 10^5$	$8.13 \cdot 10^4$	$3.44 \cdot 10^5$	$8.13 \cdot 10^4$	1.00	0.00
`environment.read`	$1.43 \cdot 10^8$	$1.26 \cdot 10^7$	$3.11 \cdot 10^6$	$3.00 \cdot 10^5$	$2.19 \cdot 10^{-2}$	$2.72 \cdot 10^{-3}$
environment.read.read (p:0, id:0)	$1.43 \cdot 10^8$	$1.26 \cdot 10^7$	$6.93 \cdot 10^5$	$6.68 \cdot 10^4$	$4.88 \cdot 10^{-3}$	$6.27 \cdot 10^{-4}$
environment.read.read (p:0, id:1)	$1.42 \cdot 10^8$	$1.26 \cdot 10^7$	$8.64 \cdot 10^5$	$8.73 \cdot 10^4$	$6.11 \cdot 10^{-3}$	$7.61 \cdot 10^{-4}$
environment.read.read (p:0, id:2)	$1.41 \cdot 10^8$	$1.26 \cdot 10^7$	$8.64 \cdot 10^5$	$8.73 \cdot 10^4$	$6.15 \cdot 10^{-3}$	$7.70 \cdot 10^{-4}$
environment.read.read (p:0, id:3)	$1.41 \cdot 10^8$	$1.26 \cdot 10^7$	$6.92 \cdot 10^5$	$6.72 \cdot 10^4$	$4.96 \cdot 10^{-3}$	$6.50 \cdot 10^{-4}$

Table C.38: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model `Elevator`. The Java code has been generated with the 'Statement' locking mode enabled. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

C.3.4 Practical Test: ToadsAndFrogs

Performance results for target model '`ToadsAndFrogs`' ($n = 20, t = 30$, Variable)						
Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
`control.done`	$2.76 \cdot 10^{6}$	$1.66 \cdot 10^{5}$	$2.76 \cdot 10^{6}$	$1.66 \cdot 10^{5}$	1.00	0.00
control.done.success (p:0, id:0)	$2.76 \cdot 10^{6}$	$1.66 \cdot 10^{5}$	2.65	2.28	$9.58 \cdot 10^{-7}$	$8.33 \cdot 10^{-7}$
control.done.failure (p:0, id:1)	$2.76 \cdot 10^{6}$	$1.66 \cdot 10^{5}$	$2.76 \cdot 10^{6}$	$1.66 \cdot 10^{5}$	1.00	0.00
`control.failure`	$2.76 \cdot 10^{6}$	$1.66 \cdot 10^{5}$	$2.76 \cdot 10^{6}$	$1.66 \cdot 10^{5}$	1.00	0.00
control.failure.reset (p:0, id:0)	$2.76 \cdot 10^{6}$	$1.66 \cdot 10^{5}$	$2.76 \cdot 10^{6}$	$1.66 \cdot 10^{5}$	1.00	0.00
`control.reset`	$2.76 \cdot 10^{6}$	$1.66 \cdot 10^{5}$	$2.76 \cdot 10^{6}$	$1.66 \cdot 10^{5}$	1.00	0.00
control.reset.running (p:0, id:0)	$2.76 \cdot 10^{6}$	$1.66 \cdot 10^{5}$	$2.76 \cdot 10^{6}$	$1.66 \cdot 10^{5}$	1.00	0.00
`control.running`	$5.98 \cdot 10^{7}$	$5.84 \cdot 10^{6}$	$2.76 \cdot 10^{6}$	$1.66 \cdot 10^{5}$	$4.65 \cdot 10^{-2}$	$4.62 \cdot 10^{-3}$
control.running.done (p:0, id:0)	$5.98 \cdot 10^{7}$	$5.84 \cdot 10^{6}$	$1.41 \cdot 10^{6}$	$1.57 \cdot 10^{5}$	$2.38 \cdot 10^{-2}$	$3.54 \cdot 10^{-3}$
control.running.done (p:0, id:1)	$5.84 \cdot 10^{7}$	$5.88 \cdot 10^{6}$	$7.62 \cdot 10^{1}$	$2.89 \cdot 10^{1}$	$1.30 \cdot 10^{-6}$	$4.69 \cdot 10^{-7}$
control.running.done (p:0, id:2)	$5.84 \cdot 10^{7}$	$5.88 \cdot 10^{6}$	$7.53 \cdot 10^{1}$	$2.54 \cdot 10^{1}$	$1.29 \cdot 10^{-6}$	$4.14 \cdot 10^{-7}$
control.running.done (p:0, id:3)	$5.84 \cdot 10^{7}$	$5.88 \cdot 10^{6}$	$1.35 \cdot 10^{6}$	$1.84 \cdot 10^{5}$	$2.33 \cdot 10^{-2}$	$3.38 \cdot 10^{-3}$
control.running.done (p:0, id:4)	$5.70 \cdot 10^{7}$	$5.84 \cdot 10^{6}$	$1.08 \cdot 10^{1}$	4.58	$1.90 \cdot 10^{-7}$	$7.89 \cdot 10^{-8}$
`control.success`	2.65	2.28	2.65	2.28	1.00	0.00
control.success.reset (p:0, id:0)	2.65	2.28	2.65	2.28	1.00	0.00
`frog.q`	$6.02 \cdot 10^{7}$	$6.78 \cdot 10^{6}$	$5.52 \cdot 10^{6}$	$7.32 \cdot 10^{5}$	$9.26 \cdot 10^{-2}$	$1.38 \cdot 10^{-2}$
frog.q.q (p:0, id:0)	$6.02 \cdot 10^{7}$	$6.78 \cdot 10^{6}$	$4.05 \cdot 10^{6}$	$5.44 \cdot 10^{5}$	$6.80 \cdot 10^{-2}$	$1.02 \cdot 10^{-2}$
frog.q.q (p:0, id:1)	$5.61 \cdot 10^{7}$	$6.80 \cdot 10^{6}$	$1.35 \cdot 10^{6}$	$1.84 \cdot 10^{5}$	$2.43 \cdot 10^{-2}$	$3.92 \cdot 10^{-3}$
frog.q.q (p:0, id:2)	$5.48 \cdot 10^{7}$	$6.81 \cdot 10^{6}$	$1.18 \cdot 10^{5}$	$4.82 \cdot 10^{4}$	$2.18 \cdot 10^{-3}$	$9.23 \cdot 10^{-4}$
frog.q.q (p:0, id:3)	$5.46 \cdot 10^{7}$	$6.81 \cdot 10^{6}$	$3.50 \cdot 10^{1}$	$1.13 \cdot 10^{1}$	$6.50 \cdot 10^{-7}$	$2.19 \cdot 10^{-7}$
`toad.q`	$6.31 \cdot 10^{7}$	$4.79 \cdot 10^{6}$	$5.76 \cdot 10^{6}$	$6.26 \cdot 10^{5}$	$9.19 \cdot 10^{-2}$	$1.28 \cdot 10^{-2}$
toad.q.q (p:0, id:0)	$6.31 \cdot 10^{7}$	$4.79 \cdot 10^{6}$	$4.22 \cdot 10^{6}$	$4.57 \cdot 10^{5}$	$6.73 \cdot 10^{-2}$	$9.18 \cdot 10^{-3}$
toad.q.q (p:0, id:1)	$5.88 \cdot 10^{7}$	$4.89 \cdot 10^{6}$	$1.41 \cdot 10^{6}$	$1.57 \cdot 10^{5}$	$2.42 \cdot 10^{-2}$	$3.64 \cdot 10^{-3}$
toad.q.q (p:0, id:2)	$5.74 \cdot 10^{7}$	$4.94 \cdot 10^{6}$	$1.26 \cdot 10^{5}$	$4.46 \cdot 10^{4}$	$2.24 \cdot 10^{-3}$	$9.05 \cdot 10^{-4}$
toad.q.q (p:0, id:3)	$5.73 \cdot 10^{7}$	$4.96 \cdot 10^{6}$	$3.41 \cdot 10^{1}$	$1.22 \cdot 10^{1}$	$5.99 \cdot 10^{-7}$	$2.28 \cdot 10^{-7}$

Table C.39: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model `ToadsAndFrogs`. The Java code has been generated with the 'Variable' locking mode enabled. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

Performance results for target model '`ToadsAndFrogs`' ($n = 20, t = 30$, Statement)						
Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
`control.done`	$3.29 \cdot 10^{6}$	$2.34 \cdot 10^{5}$	$3.29 \cdot 10^{6}$	$2.34 \cdot 10^{5}$	1.00	0.00
control.done.success (p:0, id:0)	$3.29 \cdot 10^{6}$	$2.34 \cdot 10^{5}$	$2.39 \cdot 10^{1}$	$1.21 \cdot 10^{1}$	$7.30 \cdot 10^{-6}$	$3.65 \cdot 10^{-6}$
control.done.failure (p:0, id:1)	$3.29 \cdot 10^{6}$	$2.34 \cdot 10^{5}$	$3.29 \cdot 10^{6}$	$2.34 \cdot 10^{5}$	1.00	0.00
`control.failure`	$3.29 \cdot 10^{6}$	$2.34 \cdot 10^{5}$	$3.29 \cdot 10^{6}$	$2.34 \cdot 10^{5}$	1.00	0.00
control.failure.reset (p:0, id:0)	$3.29 \cdot 10^{6}$	$2.34 \cdot 10^{5}$	$3.29 \cdot 10^{6}$	$2.34 \cdot 10^{5}$	1.00	0.00
`control.reset`	$3.29 \cdot 10^{6}$	$2.34 \cdot 10^{5}$	$3.29 \cdot 10^{6}$	$2.34 \cdot 10^{5}$	1.00	0.00
control.reset.running (p:0, id:0)	$3.29 \cdot 10^{6}$	$2.34 \cdot 10^{5}$	$3.29 \cdot 10^{6}$	$2.34 \cdot 10^{5}$	1.00	0.00
`control.running`	$1.26 \cdot 10^{8}$	$1.06 \cdot 10^{7}$	$3.29 \cdot 10^{6}$	$2.34 \cdot 10^{5}$	$2.63 \cdot 10^{-2}$	$2.50 \cdot 10^{-3}$
control.running.done (p:0, id:0)	$1.26 \cdot 10^{8}$	$1.06 \cdot 10^{7}$	$1.63 \cdot 10^{6}$	$2.01 \cdot 10^{5}$	$1.31 \cdot 10^{-2}$	$2.10 \cdot 10^{-3}$
control.running.done (p:0, id:1)	$1.24 \cdot 10^{8}$	$1.07 \cdot 10^{7}$	$1.86 \cdot 10^{3}$	$1.89 \cdot 10^{3}$	$1.46 \cdot 10^{-5}$	$1.36 \cdot 10^{-5}$
control.running.done (p:0, id:2)	$1.24 \cdot 10^{8}$	$1.07 \cdot 10^{7}$	$1.81 \cdot 10^{3}$	$1.02 \cdot 10^{3}$	$1.43 \cdot 10^{-5}$	$6.91 \cdot 10^{-6}$
control.running.done (p:0, id:3)	$1.24 \cdot 10^{8}$	$1.07 \cdot 10^{7}$	$1.65 \cdot 10^{6}$	$2.31 \cdot 10^{5}$	$1.33 \cdot 10^{-2}$	$1.82 \cdot 10^{-3}$
control.running.done (p:0, id:4)	$1.22 \cdot 10^{8}$	$1.06 \cdot 10^{7}$	$2.30 \cdot 10^{2}$	$2.36 \cdot 10^{2}$	$1.83 \cdot 10^{-6}$	$1.77 \cdot 10^{-6}$
`control.success`	$2.39 \cdot 10^{1}$	$1.21 \cdot 10^{1}$	$2.39 \cdot 10^{1}$	$1.21 \cdot 10^{1}$	1.00	0.00
control.success.reset (p:0, id:0)	$2.39 \cdot 10^{1}$	$1.21 \cdot 10^{1}$	$2.39 \cdot 10^{1}$	$1.21 \cdot 10^{1}$	1.00	0.00
`frog.q`	$1.46 \cdot 10^{8}$	$1.74 \cdot 10^{7}$	$6.74 \cdot 10^{6}$	$9.33 \cdot 10^{5}$	$4.70 \cdot 10^{-2}$	$9.50 \cdot 10^{-3}$
frog.q.q (p:0, id:0)	$1.46 \cdot 10^{8}$	$1.74 \cdot 10^{7}$	$4.93 \cdot 10^{6}$	$7.02 \cdot 10^{5}$	$3.44 \cdot 10^{-2}$	$7.07 \cdot 10^{-3}$
frog.q.q (p:0, id:1)	$1.41 \cdot 10^{8}$	$1.79 \cdot 10^{7}$	$1.65 \cdot 10^{6}$	$2.31 \cdot 10^{5}$	$1.19 \cdot 10^{-2}$	$2.48 \cdot 10^{-3}$
frog.q.q (p:0, id:2)	$1.40 \cdot 10^{8}$	$1.81 \cdot 10^{7}$	$1.53 \cdot 10^{5}$	$9.67 \cdot 10^{4}$	$1.12 \cdot 10^{-3}$	$7.14 \cdot 10^{-4}$
frog.q.q (p:0, id:3)	$1.40 \cdot 10^{8}$	$1.81 \cdot 10^{7}$	$8.32 \cdot 10^{2}$	$3.72 \cdot 10^{2}$	$6.09 \cdot 10^{-6}$	$2.97 \cdot 10^{-6}$
`toad.q`	$1.39 \cdot 10^{8}$	$1.65 \cdot 10^{7}$	$6.70 \cdot 10^{6}$	$8.01 \cdot 10^{5}$	$4.89 \cdot 10^{-2}$	$9.67 \cdot 10^{-3}$
toad.q.q (p:0, id:0)	$1.39 \cdot 10^{8}$	$1.65 \cdot 10^{7}$	$4.93 \cdot 10^{6}$	$6.19 \cdot 10^{5}$	$3.60 \cdot 10^{-2}$	$7.30 \cdot 10^{-3}$
toad.q.q (p:0, id:1)	$1.34 \cdot 10^{8}$	$1.67 \cdot 10^{7}$	$1.64 \cdot 10^{6}$	$2.00 \cdot 10^{5}$	$1.24 \cdot 10^{-2}$	$2.50 \cdot 10^{-3}$
toad.q.q (p:0, id:2)	$1.33 \cdot 10^{8}$	$1.67 \cdot 10^{7}$	$1.27 \cdot 10^{5}$	$6.16 \cdot 10^{4}$	$9.90 \cdot 10^{-4}$	$4.90 \cdot 10^{-4}$
toad.q.q (p:0, id:3)	$1.33 \cdot 10^{8}$	$1.67 \cdot 10^{7}$	$9.65 \cdot 10^{2}$	$8.82 \cdot 10^{2}$	$7.51 \cdot 10^{-6}$	$7.43 \cdot 10^{-6}$

Table C.40: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model `ToadsAndFrogs`. The Java code has been generated with the 'Statement' locking mode enabled. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

C.3.5 Practical Test: Telephony

Performance results for target model 'Telephony' ($n = 20, t = 30$, Random + Det, Variable, User_0)						
Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
User_0.busy	$2.28 \cdot 10^6$	$4.05 \cdot 10^6$	$2.28 \cdot 10^6$	$4.05 \cdot 10^6$	1.00	0.00
User_0.busy.idle (p:0, id:0)	$2.28 \cdot 10^6$	$4.05 \cdot 10^6$	$2.28 \cdot 10^6$	$4.05 \cdot 10^6$	1.00	0.00
User_0.calling	$4.58 \cdot 10^6$	$8.16 \cdot 10^6$	$4.58 \cdot 10^6$	$8.16 \cdot 10^6$	1.00	0.00
User_0.calling.busy (p:0, id:0)	$4.58 \cdot 10^6$	$8.16 \cdot 10^6$	$5.42 \cdot 10^5$	$9.66 \cdot 10^5$	$1.43 \cdot 10^{-1}$	$1.59 \cdot 10^{-1}$
User_0.calling.unobtainable (p:0, id:1)	$2.71 \cdot 10^5$	$4.83 \cdot 10^5$	$2.71 \cdot 10^5$	$4.83 \cdot 10^5$	1.00	0.00
User_0.calling.ringback (p:0, id:2)	$2.71 \cdot 10^5$	$4.83 \cdot 10^5$	$2.71 \cdot 10^5$	$4.83 \cdot 10^5$	1.00	0.00
User_0.calling.busy (p:0, id:3)	$3.49 \cdot 10^6$	$6.23 \cdot 10^6$	$1.73 \cdot 10^6$	$3.09 \cdot 10^6$	$4.13 \cdot 10^{-1}$	$2.01 \cdot 10^{-1}$
User_0.calling.calling (p:0, id:4)	$1.76 \cdot 10^6$	$3.14 \cdot 10^6$	$1.76 \cdot 10^6$	$3.14 \cdot 10^6$	$8.29 \cdot 10^{-1}$	$3.44 \cdot 10^{-1}$
User_0.calling.oalert (p:0, id:5)	1.25	1.55	1.25	1.55	1.00	0.00
User_0.dialing	$3.25 \cdot 10^6$	$5.80 \cdot 10^6$	$3.25 \cdot 10^6$	$5.80 \cdot 10^6$	1.00	0.00
User_0.dialing.idle (p:0, id:0)	$5.42 \cdot 10^5$	$9.65 \cdot 10^5$	$5.42 \cdot 10^5$	$9.65 \cdot 10^5$	1.00	0.00
User_0.dialing.calling (p:0, id:1)	$5.42 \cdot 10^5$	$9.66 \cdot 10^5$	$5.42 \cdot 10^5$	$9.66 \cdot 10^5$	1.00	0.00
User_0.dialing.calling (p:0, id:2)	$5.42 \cdot 10^5$	$9.65 \cdot 10^5$	$5.42 \cdot 10^5$	$9.65 \cdot 10^5$	1.00	0.00
User_0.dialing.calling (p:0, id:3)	$5.42 \cdot 10^5$	$9.66 \cdot 10^5$	$5.42 \cdot 10^5$	$9.66 \cdot 10^5$	1.00	0.00
User_0.dialing.calling (p:0, id:4)	$5.43 \cdot 10^5$	$9.66 \cdot 10^5$	$5.43 \cdot 10^5$	$9.66 \cdot 10^5$	1.00	0.00
User_0.dialing.calling (p:0, id:5)	$5.42 \cdot 10^5$	$9.66 \cdot 10^5$	$5.42 \cdot 10^5$	$9.66 \cdot 10^5$	1.00	0.00
User_0.dveoringout	1.05	1.32	1.05	1.32	1.00	0.00
User_0.dveoringout.idle (p:0, id:0)	1.05	1.32	1.05	1.32	1.00	0.00
User_0.idle	$2.30 \cdot 10^7$	$1.11 \cdot 10^7$	$2.30 \cdot 10^7$	$1.11 \cdot 10^7$	1.00	0.00
User_0.idle.dialing (p:0, id:0)	$2.30 \cdot 10^7$	$1.11 \cdot 10^7$	$3.25 \cdot 10^6$	$5.80 \cdot 10^6$	$3.36 \cdot 10^{-1}$	$4.72 \cdot 10^{-1}$
User_0.idle.qi (p:0, id:1)	$1.97 \cdot 10^7$	$1.50 \cdot 10^7$	$1.97 \cdot 10^7$	$1.50 \cdot 10^7$	1.00	0.00
User_0.oalert	1.25	1.55	1.25	1.55	1.00	0.00
User_0.oalert.oconnected (p:0, id:1)	1.25	1.55	$2.00 \cdot 10^{-1}$	$4.10 \cdot 10^{-1}$	$1.75 \cdot 10^{-1}$	$3.13 \cdot 10^{-1}$
User_0.oalert.dveoringout (p:0, id:2)	1.05	1.32	1.05	1.32	1.00	0.00
User_0.oconnected	$2.00 \cdot 10^{-1}$	$4.10 \cdot 10^{-1}$	$2.00 \cdot 10^{-1}$	$4.10 \cdot 10^{-1}$	1.00	0.00
User_0.oconnected.idle (p:0, id:0)	$2.00 \cdot 10^{-1}$	$4.10 \cdot 10^{-1}$	$2.00 \cdot 10^{-1}$	$4.10 \cdot 10^{-1}$	1.00	0.00
User_0.qi	$1.97 \cdot 10^7$	$1.50 \cdot 10^7$	$1.97 \cdot 10^7$	$1.50 \cdot 10^7$	1.00	0.00
User_0.qi.talert (p:0, id:0)	$1.97 \cdot 10^7$	$1.50 \cdot 10^7$	$6.00 \cdot 10^{-1}$	$5.03 \cdot 10^{-1}$	$2.35 \cdot 10^{-1}$	$4.37 \cdot 10^{-1}$
User_0.qi.idle (p:0, id:1)	$1.97 \cdot 10^7$	$1.50 \cdot 10^7$	$1.97 \cdot 10^7$	$1.50 \cdot 10^7$	1.00	0.00
User_0.ringback	$3.28 \cdot 10^5$	$6.33 \cdot 10^5$	$2.71 \cdot 10^5$	$4.83 \cdot 10^5$	$8.00 \cdot 10^{-1}$	$2.58 \cdot 10^{-1}$
User_0.ringback.idle (p:0, id:0)	$1.64 \cdot 10^5$	$3.16 \cdot 10^5$	$1.64 \cdot 10^5$	$3.16 \cdot 10^5$	1.00	0.00
User_0.ringback.calling (p:0, id:1)	$1.64 \cdot 10^5$	$3.16 \cdot 10^5$	$1.07 \cdot 10^5$	$2.20 \cdot 10^5$	$6.50 \cdot 10^{-1}$	$4.74 \cdot 10^{-1}$
User_0.talert	1.50	2.16	$6.00 \cdot 10^{-1}$	$5.03 \cdot 10^{-1}$	$6.97 \cdot 10^{-1}$	$3.85 \cdot 10^{-1}$
User_0.talert.tpickup (p:0, id:1)	$6.00 \cdot 10^{-1}$	$5.03 \cdot 10^{-1}$	$2.00 \cdot 10^{-1}$	$4.10 \cdot 10^{-1}$	$3.33 \cdot 10^{-1}$	$4.92 \cdot 10^{-1}$
User_0.talert.idle (p:0, id:2)	$4.00 \cdot 10^{-1}$	$5.03 \cdot 10^{-1}$	$4.00 \cdot 10^{-1}$	$5.03 \cdot 10^{-1}$	1.00	0.00
User_0.tconnected	$1.01 \cdot 10^7$	$3.15 \cdot 10^7$	$1.01 \cdot 10^7$	$3.15 \cdot 10^7$	1.00	0.00
User_0.tconnected.tconnected (p:0, id:0)	$1.01 \cdot 10^7$	$3.15 \cdot 10^7$	$5.06 \cdot 10^6$	$1.58 \cdot 10^7$	$5.00 \cdot 10^{-1}$	$3.05 \cdot 10^{-9}$
User_0.tconnected.tconnected (p:0, id:1)	$5.06 \cdot 10^6$	$1.58 \cdot 10^7$	$5.06 \cdot 10^6$	$1.58 \cdot 10^7$	1.00	0.00
User_0.tpickup	$2.00 \cdot 10^{-1}$	$4.10 \cdot 10^{-1}$	$2.00 \cdot 10^{-1}$	$4.10 \cdot 10^{-1}$	1.00	0.00
User_0.tpickup.tconnected (p:0, id:0)	$2.00 \cdot 10^{-1}$	$4.10 \cdot 10^{-1}$	$1.00 \cdot 10^{-1}$	$3.08 \cdot 10^{-1}$	$5.00 \cdot 10^{-1}$	$5.77 \cdot 10^{-1}$
User_0.tpickup.idle (p:0, id:1)	$1.00 \cdot 10^{-1}$	$3.08 \cdot 10^{-1}$	$1.00 \cdot 10^{-1}$	$3.08 \cdot 10^{-1}$	1.00	0.00
User_0.unobtainable	$2.71 \cdot 10^5$	$4.83 \cdot 10^5$	$2.71 \cdot 10^5$	$4.83 \cdot 10^5$	1.00	0.00
User_0.unobtainable.idle (p:0, id:0)	$2.71 \cdot 10^5$	$4.83 \cdot 10^5$	$2.71 \cdot 10^5$	$4.83 \cdot 10^5$	1.00	0.00

Table C.41: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model Telephony (state machine User_0 only). The Java code has been generated with the 'Random + Det' decision mode and 'Variable' locking mode enabled. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

Performance results for target model 'Telephony' ($n = 20, t = 30$, Random + Det, Variable, User_1)						
Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
`User_1.busy`	$4.57 \cdot 10^{6}$	$5.23 \cdot 10^{6}$	$4.57 \cdot 10^{6}$	$5.23 \cdot 10^{6}$	1.00	0.00
User_1.busy.idle (p:0, id:0)	$4.57 \cdot 10^{6}$	$5.23 \cdot 10^{6}$	$4.57 \cdot 10^{6}$	$5.23 \cdot 10^{6}$	1.00	0.00
`User_1.calling`	$7.98 \cdot 10^{6}$	$9.14 \cdot 10^{6}$	$7.98 \cdot 10^{6}$	$9.14 \cdot 10^{6}$	1.00	0.00
User_1.calling.busy (p:0, id:0)	$7.98 \cdot 10^{6}$	$9.14 \cdot 10^{6}$	$2.27 \cdot 10^{6}$	$2.60 \cdot 10^{6}$	$2.99 \cdot 10^{-1}$	$1.47 \cdot 10^{-1}$
User_1.calling.unobtainable (p:0, id:1)	$5.38 \cdot 10^{5}$	$6.15 \cdot 10^{5}$	$5.38 \cdot 10^{5}$	$6.15 \cdot 10^{5}$	1.00	0.00
User_1.calling.ringback (p:0, id:2)	$5.37 \cdot 10^{5}$	$6.14 \cdot 10^{5}$	$5.37 \cdot 10^{5}$	$6.14 \cdot 10^{5}$	1.00	0.00
User_1.calling.busy (p:0, id:3)	$4.64 \cdot 10^{6}$	$5.32 \cdot 10^{6}$	$2.30 \cdot 10^{6}$	$2.64 \cdot 10^{6}$	$5.18 \cdot 10^{-1}$	$2.22 \cdot 10^{-1}$
User_1.calling.calling (p:0, id:4)	$2.34 \cdot 10^{6}$	$2.69 \cdot 10^{6}$	$2.34 \cdot 10^{6}$	$2.69 \cdot 10^{6}$	$9.82 \cdot 10^{-1}$	$4.73 \cdot 10^{-2}$
User_1.calling.oalert (p:0, id:5)	1.15	1.31	1.15	1.31	1.00	0.00
`User_1.dialing`	$6.45 \cdot 10^{6}$	$7.38 \cdot 10^{6}$	$6.45 \cdot 10^{6}$	$7.38 \cdot 10^{6}$	1.00	0.00
User_1.dialing.idle (p:0, id:0)	$1.07 \cdot 10^{6}$	$1.23 \cdot 10^{6}$	$1.07 \cdot 10^{6}$	$1.23 \cdot 10^{6}$	1.00	0.00
User_1.dialing.calling (p:0, id:1)	$1.08 \cdot 10^{6}$	$1.23 \cdot 10^{6}$	$1.08 \cdot 10^{6}$	$1.23 \cdot 10^{6}$	1.00	0.00
User_1.dialing.calling (p:0, id:2)	$1.08 \cdot 10^{6}$	$1.23 \cdot 10^{6}$	$1.08 \cdot 10^{6}$	$1.23 \cdot 10^{6}$	1.00	0.00
User_1.dialing.calling (p:0, id:3)	$1.07 \cdot 10^{6}$	$1.23 \cdot 10^{6}$	$1.07 \cdot 10^{6}$	$1.23 \cdot 10^{6}$	1.00	0.00
User_1.dialing.calling (p:0, id:4)	$1.08 \cdot 10^{6}$	$1.23 \cdot 10^{6}$	$1.08 \cdot 10^{6}$	$1.23 \cdot 10^{6}$	1.00	0.00
User_1.dialing.calling (p:0, id:5)	$1.08 \cdot 10^{6}$	$1.23 \cdot 10^{6}$	$1.08 \cdot 10^{6}$	$1.23 \cdot 10^{6}$	1.00	0.00
`User_1.dveoringout`	1.15	1.31	1.15	1.31	1.00	0.00
User_1.dveoringout.idle (p:0, id:0)	1.15	1.31	1.15	1.31	1.00	0.00
`User_1.idle`	$2.23 \cdot 10^{7}$	$1.03 \cdot 10^{7}$	$2.23 \cdot 10^{7}$	$1.03 \cdot 10^{7}$	1.00	0.00
User_1.idle.dialing (p:0, id:0)	$2.23 \cdot 10^{7}$	$1.03 \cdot 10^{7}$	$6.45 \cdot 10^{6}$	$7.38 \cdot 10^{6}$	$4.97 \cdot 10^{-1}$	$5.10 \cdot 10^{-1}$
User_1.idle.qi (p:0, id:1)	$1.59 \cdot 10^{7}$	$1.64 \cdot 10^{7}$	$1.59 \cdot 10^{7}$	$1.64 \cdot 10^{7}$	1.00	0.00
`User_1.oalert`	1.15	1.31	1.15	1.31	1.00	0.00
User_1.oalert.dveoringout (p:0, id:2)	1.15	1.31	1.15	1.31	1.00	0.00
`User_1.qi`	$1.59 \cdot 10^{7}$	$1.64 \cdot 10^{7}$	$1.59 \cdot 10^{7}$	$1.64 \cdot 10^{7}$	1.00	0.00
User_1.qi.talert (p:0, id:0)	$1.59 \cdot 10^{7}$	$1.64 \cdot 10^{7}$	$4.50 \cdot 10^{-1}$	$6.05 \cdot 10^{-1}$	$2.31 \cdot 10^{-1}$	$4.39 \cdot 10^{-1}$
User_1.qi.idle (p:0, id:1)	$1.59 \cdot 10^{7}$	$1.64 \cdot 10^{7}$	$1.59 \cdot 10^{7}$	$1.64 \cdot 10^{7}$	1.00	0.00
`User_1.ringback`	$5.37 \cdot 10^{5}$	$6.14 \cdot 10^{5}$	$5.37 \cdot 10^{5}$	$6.14 \cdot 10^{5}$	1.00	0.00
User_1.ringback.idle (p:0, id:0)	$2.69 \cdot 10^{5}$	$3.07 \cdot 10^{5}$	$2.69 \cdot 10^{5}$	$3.07 \cdot 10^{5}$	1.00	0.00
User_1.ringback.calling (p:0, id:1)	$2.69 \cdot 10^{5}$	$3.07 \cdot 10^{5}$	$2.69 \cdot 10^{5}$	$3.07 \cdot 10^{5}$	1.00	0.00
`User_1.talert`	$7.50 \cdot 10^{-1}$	1.07	$4.50 \cdot 10^{-1}$	$6.05 \cdot 10^{-1}$	$7.08 \cdot 10^{-1}$	$3.18 \cdot 10^{-1}$
User_1.talert.tpickup (p:0, id:1)	$4.50 \cdot 10^{-1}$	$6.05 \cdot 10^{-1}$	$2.00 \cdot 10^{-1}$	$5.23 \cdot 10^{-1}$	$3.75 \cdot 10^{-1}$	$5.18 \cdot 10^{-1}$
User_1.talert.idle (p:0, id:2)	$2.50 \cdot 10^{-1}$	$4.44 \cdot 10^{-1}$	$2.50 \cdot 10^{-1}$	$4.44 \cdot 10^{-1}$	1.00	0.00
`User_1.tconnected`	$4.74 \cdot 10^{6}$	$2.12 \cdot 10^{7}$	$4.74 \cdot 10^{6}$	$2.12 \cdot 10^{7}$	$8.15 \cdot 10^{-1}$	$3.21 \cdot 10^{-1}$
User_1.tconnected.tconnected (p:0, id:0)	$4.74 \cdot 10^{6}$	$2.12 \cdot 10^{7}$	$2.37 \cdot 10^{6}$	$1.06 \cdot 10^{7}$	$3.74 \cdot 10^{-1}$	$1.41 \cdot 10^{-1}$
User_1.tconnected.tconnected (p:0, id:1)	$2.37 \cdot 10^{6}$	$1.06 \cdot 10^{7}$	$2.37 \cdot 10^{6}$	$1.06 \cdot 10^{7}$	$6.03 \cdot 10^{-1}$	$4.32 \cdot 10^{-1}$
User_1.tconnected.idle (p:0, id:2)	$6.00 \cdot 10^{-1}$	1.70	$1.50 \cdot 10^{-1}$	$3.66 \cdot 10^{-1}$	$4.56 \cdot 10^{-1}$	$4.72 \cdot 10^{-1}$
`User_1.tpickup`	$2.00 \cdot 10^{-1}$	$5.23 \cdot 10^{-1}$	$2.00 \cdot 10^{-1}$	$5.23 \cdot 10^{-1}$	1.00	0.00
User_1.tpickup.tconnected (p:0, id:0)	$2.00 \cdot 10^{-1}$	$5.23 \cdot 10^{-1}$	$2.00 \cdot 10^{-1}$	$5.23 \cdot 10^{-1}$	1.00	0.00
`User_1.unobtainable`	$5.38 \cdot 10^{5}$	$6.15 \cdot 10^{5}$	$5.38 \cdot 10^{5}$	$6.15 \cdot 10^{5}$	1.00	0.00
User_1.unobtainable.idle (p:0, id:0)	$5.38 \cdot 10^{5}$	$6.15 \cdot 10^{5}$	$5.38 \cdot 10^{5}$	$6.15 \cdot 10^{5}$	1.00	0.00

Table C.42: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model `Telephony` (state machine `User_1` only). The Java code has been generated with the 'Random + Det' decision mode and 'Variable' locking mode enabled. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

Performance results for target model 'Telephony' ($n = 20, t = 30$, Random + Det, Variable, User_2)						
Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
`User_2.busy`	$1.05 \cdot 10^{6}$	$3.24 \cdot 10^{6}$	$1.05 \cdot 10^{6}$	$3.24 \cdot 10^{6}$	1.00	0.00
User_2.busy.idle (p:0, id:0)	$1.05 \cdot 10^{6}$	$3.24 \cdot 10^{6}$	$1.05 \cdot 10^{6}$	$3.24 \cdot 10^{6}$	1.00	0.00
`User_2.calling`	$2.04 \cdot 10^{6}$	$6.28 \cdot 10^{6}$	$2.04 \cdot 10^{6}$	$6.28 \cdot 10^{6}$	1.00	0.00
User_2.calling.busy (p:0, id:0)	$2.04 \cdot 10^{6}$	$6.28 \cdot 10^{6}$	$7.41 \cdot 10^{5}$	$2.28 \cdot 10^{6}$	$3.34 \cdot 10^{-1}$	$9.97 \cdot 10^{-2}$
User_2.calling.unobtainable (p:0, id:1)	$1.24 \cdot 10^{5}$	$3.81 \cdot 10^{5}$	$1.24 \cdot 10^{5}$	$3.81 \cdot 10^{5}$	1.00	0.00
User_2.calling.ringback (p:0, id:2)	$1.24 \cdot 10^{5}$	$3.81 \cdot 10^{5}$	$1.24 \cdot 10^{5}$	$3.81 \cdot 10^{5}$	1.00	0.00
User_2.calling.busy (p:0, id:3)	$1.05 \cdot 10^{6}$	$3.24 \cdot 10^{6}$	$3.09 \cdot 10^{5}$	$9.52 \cdot 10^{5}$	$2.45 \cdot 10^{-1}$	$1.94 \cdot 10^{-1}$
User_2.calling.calling (p:0, id:4)	$7.41 \cdot 10^{5}$	$2.28 \cdot 10^{6}$	$7.41 \cdot 10^{5}$	$2.28 \cdot 10^{6}$	$8.73 \cdot 10^{-1}$	$1.80 \cdot 10^{-1}$
User_2.calling.oalert (p:0, id:5)	$5.50 \cdot 10^{-1}$	$6.86 \cdot 10^{-1}$	$5.50 \cdot 10^{-1}$	$6.86 \cdot 10^{-1}$	1.00	0.00
`User_2.dialing`	$1.48 \cdot 10^{6}$	$4.57 \cdot 10^{6}$	$1.48 \cdot 10^{6}$	$4.57 \cdot 10^{6}$	1.00	0.00
User_2.dialing.idle (p:0, id:0)	$2.47 \cdot 10^{5}$	$7.62 \cdot 10^{5}$	$2.47 \cdot 10^{5}$	$7.62 \cdot 10^{5}$	1.00	0.00
User_2.dialing.calling (p:0, id:1)	$2.47 \cdot 10^{5}$	$7.61 \cdot 10^{5}$	$2.47 \cdot 10^{5}$	$7.61 \cdot 10^{5}$	1.00	0.00
User_2.dialing.calling (p:0, id:2)	$2.47 \cdot 10^{5}$	$7.61 \cdot 10^{5}$	$2.47 \cdot 10^{5}$	$7.61 \cdot 10^{5}$	1.00	0.00
User_2.dialing.calling (p:0, id:3)	$2.47 \cdot 10^{5}$	$7.62 \cdot 10^{5}$	$2.47 \cdot 10^{5}$	$7.62 \cdot 10^{5}$	1.00	0.00
User_2.dialing.calling (p:0, id:4)	$2.47 \cdot 10^{5}$	$7.61 \cdot 10^{5}$	$2.47 \cdot 10^{5}$	$7.61 \cdot 10^{5}$	1.00	0.00
User_2.dialing.calling (p:0, id:5)	$2.47 \cdot 10^{5}$	$7.62 \cdot 10^{5}$	$2.47 \cdot 10^{5}$	$7.62 \cdot 10^{5}$	1.00	0.00
`User_2.dveoringout`	$5.00 \cdot 10^{-1}$	$6.07 \cdot 10^{-1}$	$5.00 \cdot 10^{-1}$	$6.07 \cdot 10^{-1}$	1.00	0.00
User_2.dveoringout.idle (p:0, id:0)	$5.00 \cdot 10^{-1}$	$6.07 \cdot 10^{-1}$	$5.00 \cdot 10^{-1}$	$6.07 \cdot 10^{-1}$	1.00	0.00
`User_2.idle`	$2.85 \cdot 10^{7}$	$8.88 \cdot 10^{6}$	$2.85 \cdot 10^{7}$	$8.88 \cdot 10^{6}$	1.00	0.00
User_2.idle.dialing (p:0, id:0)	$2.85 \cdot 10^{7}$	$8.88 \cdot 10^{6}$	$1.48 \cdot 10^{6}$	$4.57 \cdot 10^{6}$	$1.00 \cdot 10^{-1}$	$3.08 \cdot 10^{-1}$
User_2.idle.qi (p:0, id:1)	$2.70 \cdot 10^{7}$	$1.19 \cdot 10^{7}$	$2.70 \cdot 10^{7}$	$1.19 \cdot 10^{7}$	1.00	0.00
`User_2.oalert`	$5.50 \cdot 10^{-1}$	$6.86 \cdot 10^{-1}$	$5.50 \cdot 10^{-1}$	$6.86 \cdot 10^{-1}$	1.00	0.00
User_2.oalert.oconnected (p:0, id:1)	$5.50 \cdot 10^{-1}$	$6.86 \cdot 10^{-1}$	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	$5.56 \cdot 10^{-2}$	$1.67 \cdot 10^{-1}$
User_2.oalert.dveoringout (p:0, id:2)	$5.00 \cdot 10^{-1}$	$6.07 \cdot 10^{-1}$	$5.00 \cdot 10^{-1}$	$6.07 \cdot 10^{-1}$	1.00	0.00
`User_2.oconnected`	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	1.00	NaN
User_2.oconnected.idle (p:0, id:0)	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	1.00	NaN
`User_2.qi`	$2.70 \cdot 10^{7}$	$1.19 \cdot 10^{7}$	$2.70 \cdot 10^{7}$	$1.19 \cdot 10^{7}$	1.00	0.00
User_2.qi.talert (p:0, id:0)	$2.70 \cdot 10^{7}$	$1.19 \cdot 10^{7}$	$3.00 \cdot 10^{-1}$	$4.70 \cdot 10^{-1}$	$5.56 \cdot 10^{-2}$	$2.36 \cdot 10^{-1}$
User_2.qi.idle (p:0, id:1)	$2.70 \cdot 10^{7}$	$1.19 \cdot 10^{7}$	$2.70 \cdot 10^{7}$	$1.19 \cdot 10^{7}$	1.00	0.00
`User_2.ringback`	$1.24 \cdot 10^{5}$	$3.81 \cdot 10^{5}$	$1.24 \cdot 10^{5}$	$3.81 \cdot 10^{5}$	1.00	0.00
User_2.ringback.idle (p:0, id:0)	$6.18 \cdot 10^{4}$	$1.90 \cdot 10^{5}$	$6.18 \cdot 10^{4}$	$1.90 \cdot 10^{5}$	1.00	0.00
User_2.ringback.calling (p:0, id:1)	$6.19 \cdot 10^{4}$	$1.91 \cdot 10^{5}$	$6.19 \cdot 10^{4}$	$1.91 \cdot 10^{5}$	1.00	0.00
`User_2.talert`	$5.00 \cdot 10^{-1}$	$8.27 \cdot 10^{-1}$	$3.00 \cdot 10^{-1}$	$4.70 \cdot 10^{-1}$	$6.67 \cdot 10^{-1}$	$2.58 \cdot 10^{-1}$
User_2.talert.tpickup (p:0, id:1)	$3.00 \cdot 10^{-1}$	$4.70 \cdot 10^{-1}$	$1.00 \cdot 10^{-1}$	$3.08 \cdot 10^{-1}$	$3.33 \cdot 10^{-1}$	$5.16 \cdot 10^{-1}$
User_2.talert.idle (p:0, id:2)	$2.00 \cdot 10^{-1}$	$4.10 \cdot 10^{-1}$	$2.00 \cdot 10^{-1}$	$4.10 \cdot 10^{-1}$	1.00	0.00
`User_2.tconnected`	$4.99 \cdot 10^{6}$	$2.23 \cdot 10^{7}$	$4.99 \cdot 10^{6}$	$2.23 \cdot 10^{7}$	$5.37 \cdot 10^{-1}$	$6.55 \cdot 10^{-1}$
User_2.tconnected.tconnected (p:0, id:0)	$4.99 \cdot 10^{6}$	$2.23 \cdot 10^{7}$	$2.50 \cdot 10^{6}$	$1.12 \cdot 10^{7}$	$2.62 \cdot 10^{-1}$	$3.36 \cdot 10^{-1}$
User_2.tconnected.tconnected (p:0, id:1)	$2.50 \cdot 10^{6}$	$1.12 \cdot 10^{7}$	$2.50 \cdot 10^{6}$	$1.12 \cdot 10^{7}$	$5.12 \cdot 10^{-1}$	$6.89 \cdot 10^{-1}$
User_2.tconnected.idle (p:0, id:2)	1.95	8.72	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	$2.56 \cdot 10^{-2}$	NaN
`User_2.tpickup`	$1.00 \cdot 10^{-1}$	$3.08 \cdot 10^{-1}$	$1.00 \cdot 10^{-1}$	$3.08 \cdot 10^{-1}$	1.00	0.00
User_2.tpickup.tconnected (p:0, id:0)	$1.00 \cdot 10^{-1}$	$3.08 \cdot 10^{-1}$	$1.00 \cdot 10^{-1}$	$3.08 \cdot 10^{-1}$	1.00	0.00
`User_2.unobtainable`	$1.24 \cdot 10^{5}$	$3.81 \cdot 10^{5}$	$1.24 \cdot 10^{5}$	$3.81 \cdot 10^{5}$	1.00	0.00
User_2.unobtainable.idle (p:0, id:0)	$1.24 \cdot 10^{5}$	$3.81 \cdot 10^{5}$	$1.24 \cdot 10^{5}$	$3.81 \cdot 10^{5}$	1.00	0.00

Table C.43: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model `Telephony` (state machine `User_2` only). The Java code has been generated with the 'Random + Det' decision mode and 'Variable' locking mode enabled. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

Performance results for target model 'Telephony' ($n = 20, t = 30$, Random + Det, Variable, User_3)						
Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
`User_3.busy`	$1.50 \cdot 10^{6}$	$3.08 \cdot 10^{6}$	$1.50 \cdot 10^{6}$	$3.08 \cdot 10^{6}$	1.00	0.00
User_3.busy.idle (p:0, id:0)	$1.50 \cdot 10^{6}$	$3.08 \cdot 10^{6}$	$1.50 \cdot 10^{6}$	$3.08 \cdot 10^{6}$	1.00	0.00
`User_3.calling`	$4.17 \cdot 10^{6}$	$8.56 \cdot 10^{6}$	$4.17 \cdot 10^{6}$	$8.56 \cdot 10^{6}$	1.00	0.00
User_3.calling.busy (p:0, id:0)	$4.17 \cdot 10^{6}$	$8.56 \cdot 10^{6}$	$1.50 \cdot 10^{6}$	$3.08 \cdot 10^{6}$	$3.25 \cdot 10^{-1}$	$1.18 \cdot 10^{-1}$
User_3.calling.unobtainable (p:0, id:1)	$1.82 \cdot 10^{5}$	$3.74 \cdot 10^{5}$	$1.82 \cdot 10^{5}$	$3.74 \cdot 10^{5}$	1.00	0.00
User_3.calling.ringback (p:0, id:2)	$1.82 \cdot 10^{5}$	$3.74 \cdot 10^{5}$	$1.82 \cdot 10^{5}$	$3.74 \cdot 10^{5}$	1.00	0.00
User_3.calling.calling (p:0, id:4)	$2.30 \cdot 10^{6}$	$4.72 \cdot 10^{6}$	$2.30 \cdot 10^{6}$	$4.72 \cdot 10^{6}$	$8.68 \cdot 10^{-1}$	$2.62 \cdot 10^{-1}$
User_3.calling.oalert (p:0, id:5)	$8.00 \cdot 10^{-1}$	$8.34 \cdot 10^{-1}$	$8.00 \cdot 10^{-1}$	$8.34 \cdot 10^{-1}$	1.00	0.00
`User_3.dialing`	$2.19 \cdot 10^{6}$	$4.49 \cdot 10^{6}$	$2.19 \cdot 10^{6}$	$4.49 \cdot 10^{6}$	1.00	0.00
User_3.dialing.idle (p:0, id:0)	$3.64 \cdot 10^{5}$	$7.48 \cdot 10^{5}$	$3.64 \cdot 10^{5}$	$7.48 \cdot 10^{5}$	1.00	0.00
User_3.dialing.calling (p:0, id:1)	$3.64 \cdot 10^{5}$	$7.48 \cdot 10^{5}$	$3.64 \cdot 10^{5}$	$7.48 \cdot 10^{5}$	1.00	0.00
User_3.dialing.calling (p:0, id:2)	$3.65 \cdot 10^{5}$	$7.48 \cdot 10^{5}$	$3.65 \cdot 10^{5}$	$7.48 \cdot 10^{5}$	1.00	0.00
User_3.dialing.calling (p:0, id:3)	$3.64 \cdot 10^{5}$	$7.48 \cdot 10^{5}$	$3.64 \cdot 10^{5}$	$7.48 \cdot 10^{5}$	1.00	0.00
User_3.dialing.calling (p:0, id:4)	$3.64 \cdot 10^{5}$	$7.48 \cdot 10^{5}$	$3.64 \cdot 10^{5}$	$7.48 \cdot 10^{5}$	1.00	0.00
User_3.dialing.calling (p:0, id:5)	$3.65 \cdot 10^{5}$	$7.48 \cdot 10^{5}$	$3.65 \cdot 10^{5}$	$7.48 \cdot 10^{5}$	1.00	0.00
`User_3.dveoringout`	$8.00 \cdot 10^{-1}$	$8.34 \cdot 10^{-1}$	$8.00 \cdot 10^{-1}$	$8.34 \cdot 10^{-1}$	1.00	0.00
User_3.dveoringout.idle (p:0, id:0)	$8.00 \cdot 10^{-1}$	$8.34 \cdot 10^{-1}$	$8.00 \cdot 10^{-1}$	$8.34 \cdot 10^{-1}$	1.00	0.00
`User_3.idle`	$2.26 \cdot 10^{7}$	$1.29 \cdot 10^{7}$	$2.26 \cdot 10^{7}$	$1.29 \cdot 10^{7}$	1.00	0.00
User_3.idle.dialing (p:0, id:0)	$2.26 \cdot 10^{7}$	$1.29 \cdot 10^{7}$	$2.19 \cdot 10^{6}$	$4.49 \cdot 10^{6}$	$2.81 \cdot 10^{-1}$	$4.43 \cdot 10^{-1}$
User_3.idle.qi (p:0, id:1)	$2.04 \cdot 10^{7}$	$1.55 \cdot 10^{7}$	$2.04 \cdot 10^{7}$	$1.55 \cdot 10^{7}$	1.00	0.00
`User_3.oalert`	$8.00 \cdot 10^{-1}$	$8.34 \cdot 10^{-1}$	$8.00 \cdot 10^{-1}$	$8.34 \cdot 10^{-1}$	1.00	0.00
User_3.oalert.dveoringout (p:0, id:2)	$8.00 \cdot 10^{-1}$	$8.34 \cdot 10^{-1}$	$8.00 \cdot 10^{-1}$	$8.34 \cdot 10^{-1}$	1.00	0.00
`User_3.qi`	$2.04 \cdot 10^{7}$	$1.55 \cdot 10^{7}$	$2.04 \cdot 10^{7}$	$1.55 \cdot 10^{7}$	1.00	0.00
User_3.qi.talert (p:0, id:0)	$2.04 \cdot 10^{7}$	$1.55 \cdot 10^{7}$	$6.00 \cdot 10^{-1}$	$5.98 \cdot 10^{-1}$	$1.88 \cdot 10^{-1}$	$4.03 \cdot 10^{-1}$
User_3.qi.idle (p:0, id:1)	$2.04 \cdot 10^{7}$	$1.55 \cdot 10^{7}$	$2.04 \cdot 10^{7}$	$1.55 \cdot 10^{7}$	1.00	0.00
`User_3.ringback`	$2.74 \cdot 10^{5}$	$6.03 \cdot 10^{5}$	$1.82 \cdot 10^{5}$	$3.74 \cdot 10^{5}$	$8.06 \cdot 10^{-1}$	$2.92 \cdot 10^{-1}$
User_3.ringback.idle (p:0, id:0)	$1.37 \cdot 10^{5}$	$3.02 \cdot 10^{5}$	$1.37 \cdot 10^{5}$	$3.02 \cdot 10^{5}$	1.00	0.00
User_3.ringback.calling (p:0, id:1)	$1.37 \cdot 10^{5}$	$3.02 \cdot 10^{5}$	$4.51 \cdot 10^{4}$	$1.39 \cdot 10^{5}$	$5.83 \cdot 10^{-1}$	$5.15 \cdot 10^{-1}$
`User_3.talert`	$8.00 \cdot 10^{-1}$	$8.34 \cdot 10^{-1}$	$6.00 \cdot 10^{-1}$	$5.98 \cdot 10^{-1}$	$8.18 \cdot 10^{-1}$	$2.52 \cdot 10^{-1}$
User_3.talert.tpickup (p:0, id:1)	$6.00 \cdot 10^{-1}$	$5.98 \cdot 10^{-1}$	$3.50 \cdot 10^{-1}$	$5.87 \cdot 10^{-1}$	$5.45 \cdot 10^{-1}$	$5.22 \cdot 10^{-1}$
User_3.talert.idle (p:0, id:2)	$2.50 \cdot 10^{-1}$	$4.44 \cdot 10^{-1}$	$2.50 \cdot 10^{-1}$	$4.44 \cdot 10^{-1}$	1.00	0.00
`User_3.tconnected`	$1.46 \cdot 10^{7}$	$3.60 \cdot 10^{7}$	$1.46 \cdot 10^{7}$	$3.60 \cdot 10^{7}$	1.00	$6.83 \cdot 10^{-9}$
User_3.tconnected.tconnected (p:0, id:0)	$1.46 \cdot 10^{7}$	$3.60 \cdot 10^{7}$	$7.32 \cdot 10^{6}$	$1.80 \cdot 10^{7}$	$5.00 \cdot 10^{-1}$	$8.75 \cdot 10^{-9}$
User_3.tconnected.tconnected (p:0, id:1)	$7.32 \cdot 10^{6}$	$1.80 \cdot 10^{7}$	$7.32 \cdot 10^{6}$	$1.80 \cdot 10^{7}$	1.00	$2.73 \cdot 10^{-8}$
User_3.tconnected.idle (p:0, id:2)	$1.00 \cdot 10^{-1}$	$4.47 \cdot 10^{-1}$	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	$5.00 \cdot 10^{-1}$	NaN
`User_3.tpickup`	$3.50 \cdot 10^{-1}$	$5.87 \cdot 10^{-1}$	$3.50 \cdot 10^{-1}$	$5.87 \cdot 10^{-1}$	1.00	0.00
User_3.tpickup.tconnected (p:0, id:0)	$3.50 \cdot 10^{-1}$	$5.87 \cdot 10^{-1}$	$2.00 \cdot 10^{-1}$	$5.23 \cdot 10^{-1}$	$5.00 \cdot 10^{-1}$	$5.48 \cdot 10^{-1}$
User_3.tpickup.idle (p:0, id:1)	$1.50 \cdot 10^{-1}$	$3.66 \cdot 10^{-1}$	$1.50 \cdot 10^{-1}$	$3.66 \cdot 10^{-1}$	1.00	0.00
`User_3.unobtainable`	$1.82 \cdot 10^{5}$	$3.74 \cdot 10^{5}$	$1.82 \cdot 10^{5}$	$3.74 \cdot 10^{5}$	1.00	0.00
User_3.unobtainable.idle (p:0, id:0)	$1.82 \cdot 10^{5}$	$3.74 \cdot 10^{5}$	$1.82 \cdot 10^{5}$	$3.74 \cdot 10^{5}$	1.00	0.00

Table C.44: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model `Telephony` (state machine `User_3` only). The Java code has been generated with the 'Random + Det' decision mode and 'Variable' locking mode enabled. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

Performance results for target model 'Telephony' ($n = 20, t = 30$, Random + Det, Statement, User_0)						
Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
User_0.busy	$6.71 \cdot 10^{6}$	$6.92 \cdot 10^{6}$	$6.71 \cdot 10^{6}$	$6.92 \cdot 10^{6}$	1.00	0.00
User_0.busy.idle (p:0, id:0)	$6.71 \cdot 10^{6}$	$6.92 \cdot 10^{6}$	$6.71 \cdot 10^{6}$	$6.92 \cdot 10^{6}$	1.00	0.00
User_0.calling	$1.35 \cdot 10^{7}$	$1.39 \cdot 10^{7}$	$1.35 \cdot 10^{7}$	$1.39 \cdot 10^{7}$	1.00	0.00
User_0.calling.busy (p:0, id:0)	$1.35 \cdot 10^{7}$	$1.39 \cdot 10^{7}$	$1.59 \cdot 10^{6}$	$1.64 \cdot 10^{6}$	$1.22 \cdot 10^{-1}$	$1.25 \cdot 10^{-1}$
User_0.calling.unobtainable (p:0, id:1)	$7.94 \cdot 10^{5}$	$8.19 \cdot 10^{5}$	$7.94 \cdot 10^{5}$	$8.19 \cdot 10^{5}$	1.00	0.00
User_0.calling.ringback (p:0, id:2)	$7.94 \cdot 10^{5}$	$8.19 \cdot 10^{5}$	$7.94 \cdot 10^{5}$	$8.19 \cdot 10^{5}$	1.00	0.00
User_0.calling.busy (p:0, id:3)	$1.03 \cdot 10^{7}$	$1.07 \cdot 10^{7}$	$5.12 \cdot 10^{6}$	$5.28 \cdot 10^{6}$	$4.43 \cdot 10^{-1}$	$2.31 \cdot 10^{-1}$
User_0.calling.calling (p:0, id:4)	$5.20 \cdot 10^{6}$	$5.39 \cdot 10^{6}$	$5.20 \cdot 10^{6}$	$5.39 \cdot 10^{6}$	$7.72 \cdot 10^{-1}$	$4.03 \cdot 10^{-1}$
User_0.calling.oalert (p:0, id:5)	1.60	1.50	1.60	1.50	1.00	0.00
User_0.dialing	$9.53 \cdot 10^{6}$	$9.83 \cdot 10^{6}$	$9.53 \cdot 10^{6}$	$9.83 \cdot 10^{6}$	1.00	0.00
User_0.dialing.idle (p:0, id:0)	$1.59 \cdot 10^{6}$	$1.64 \cdot 10^{6}$	$1.59 \cdot 10^{6}$	$1.64 \cdot 10^{6}$	1.00	0.00
User_0.dialing.calling (p:0, id:1)	$1.59 \cdot 10^{6}$	$1.64 \cdot 10^{6}$	$1.59 \cdot 10^{6}$	$1.64 \cdot 10^{6}$	1.00	0.00
User_0.dialing.calling (p:0, id:2)	$1.59 \cdot 10^{6}$	$1.64 \cdot 10^{6}$	$1.59 \cdot 10^{6}$	$1.64 \cdot 10^{6}$	1.00	0.00
User_0.dialing.calling (p:0, id:3)	$1.59 \cdot 10^{6}$	$1.64 \cdot 10^{6}$	$1.59 \cdot 10^{6}$	$1.64 \cdot 10^{6}$	1.00	0.00
User_0.dialing.calling (p:0, id:4)	$1.59 \cdot 10^{6}$	$1.64 \cdot 10^{6}$	$1.59 \cdot 10^{6}$	$1.64 \cdot 10^{6}$	1.00	0.00
User_0.dialing.calling (p:0, id:5)	$1.59 \cdot 10^{6}$	$1.64 \cdot 10^{6}$	$1.59 \cdot 10^{6}$	$1.64 \cdot 10^{6}$	1.00	0.00
User_0.dveoringout	1.40	1.35	1.40	1.35	1.00	0.00
User_0.dveoringout.idle (p:0, id:0)	1.40	1.35	1.40	1.35	1.00	0.00
User_0.idle	$2.84 \cdot 10^{7}$	$1.69 \cdot 10^{7}$	$2.84 \cdot 10^{7}$	$1.69 \cdot 10^{7}$	1.00	0.00
User_0.idle.dialing (p:0, id:0)	$2.84 \cdot 10^{7}$	$1.69 \cdot 10^{7}$	$9.53 \cdot 10^{6}$	$9.83 \cdot 10^{6}$	$5.88 \cdot 10^{-1}$	$4.94 \cdot 10^{-1}$
User_0.idle.qi (p:0, id:1)	$1.88 \cdot 10^{7}$	$2.38 \cdot 10^{7}$	$1.88 \cdot 10^{7}$	$2.38 \cdot 10^{7}$	1.00	0.00
User_0.oalert	1.60	1.50	1.60	1.50	1.00	0.00
User_0.oalert.oconnected (p:0, id:1)	1.60	1.50	$2.00 \cdot 10^{-1}$	$5.23 \cdot 10^{-1}$	$1.17 \cdot 10^{-1}$	$2.81 \cdot 10^{-1}$
User_0.oalert.dveoringout (p:0, id:2)	1.40	1.35	1.40	1.35	1.00	0.00
User_0.oconnected	$2.00 \cdot 10^{-1}$	$5.23 \cdot 10^{-1}$	$2.00 \cdot 10^{-1}$	$5.23 \cdot 10^{-1}$	1.00	0.00
User_0.oconnected.idle (p:0, id:0)	$2.00 \cdot 10^{-1}$	$5.23 \cdot 10^{-1}$	$2.00 \cdot 10^{-1}$	$5.23 \cdot 10^{-1}$	1.00	0.00
User_0.qi	$1.88 \cdot 10^{7}$	$2.38 \cdot 10^{7}$	$1.88 \cdot 10^{7}$	$2.38 \cdot 10^{7}$	1.00	0.00
User_0.qi.talert (p:0, id:0)	$1.88 \cdot 10^{7}$	$2.38 \cdot 10^{7}$	$3.00 \cdot 10^{-1}$	$4.70 \cdot 10^{-1}$	$3.33 \cdot 10^{-1}$	$4.92 \cdot 10^{-1}$
User_0.qi.idle (p:0, id:1)	$1.88 \cdot 10^{7}$	$2.38 \cdot 10^{7}$	$1.88 \cdot 10^{7}$	$2.38 \cdot 10^{7}$	1.00	0.00
User_0.ringback	$8.78 \cdot 10^{5}$	$9.84 \cdot 10^{5}$	$7.94 \cdot 10^{5}$	$8.19 \cdot 10^{5}$	$9.17 \cdot 10^{-1}$	$1.95 \cdot 10^{-1}$
User_0.ringback.idle (p:0, id:0)	$4.39 \cdot 10^{5}$	$4.92 \cdot 10^{5}$	$4.39 \cdot 10^{5}$	$4.92 \cdot 10^{5}$	1.00	0.00
User_0.ringback.calling (p:0, id:1)	$4.39 \cdot 10^{5}$	$4.92 \cdot 10^{5}$	$3.55 \cdot 10^{5}$	$4.05 \cdot 10^{5}$	$8.18 \cdot 10^{-1}$	$4.05 \cdot 10^{-1}$
User_0.talert	$4.00 \cdot 10^{-1}$	$6.81 \cdot 10^{-1}$	$3.00 \cdot 10^{-1}$	$4.70 \cdot 10^{-1}$	$8.33 \cdot 10^{-1}$	$2.58 \cdot 10^{-1}$
User_0.talert.tpickup (p:0, id:1)	$3.00 \cdot 10^{-1}$	$4.70 \cdot 10^{-1}$	$2.00 \cdot 10^{-1}$	$4.10 \cdot 10^{-1}$	$6.67 \cdot 10^{-1}$	$5.16 \cdot 10^{-1}$
User_0.talert.idle (p:0, id:2)	$1.00 \cdot 10^{-1}$	$3.08 \cdot 10^{-1}$	$1.00 \cdot 10^{-1}$	$3.08 \cdot 10^{-1}$	1.00	0.00
User_0.tconnected	$8.39 \cdot 10^{6}$	$2.59 \cdot 10^{7}$	$8.39 \cdot 10^{6}$	$2.59 \cdot 10^{7}$	$7.12 \cdot 10^{-1}$	$4.99 \cdot 10^{-1}$
User_0.tconnected.tconnected (p:0, id:0)	$8.39 \cdot 10^{6}$	$2.59 \cdot 10^{7}$	$4.19 \cdot 10^{6}$	$1.29 \cdot 10^{7}$	$3.48 \cdot 10^{-1}$	$2.62 \cdot 10^{-1}$
User_0.tconnected.tconnected (p:0, id:1)	$4.19 \cdot 10^{6}$	$1.29 \cdot 10^{7}$	$4.19 \cdot 10^{6}$	$1.29 \cdot 10^{7}$	$6.83 \cdot 10^{-1}$	$5.50 \cdot 10^{-1}$
User_0.tconnected.idle (p:0, id:2)	1.00	4.47	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	$5.00 \cdot 10^{-2}$	NaN
User_0.tpickup	$2.00 \cdot 10^{-1}$	$4.10 \cdot 10^{-1}$	$2.00 \cdot 10^{-1}$	$4.10 \cdot 10^{-1}$	1.00	0.00
User_0.tpickup.tconnected (p:0, id:0)	$2.00 \cdot 10^{-1}$	$4.10 \cdot 10^{-1}$	$1.50 \cdot 10^{-1}$	$3.66 \cdot 10^{-1}$	$7.50 \cdot 10^{-1}$	$5.00 \cdot 10^{-1}$
User_0.tpickup.idle (p:0, id:1)	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	1.00	NaN
User_0.unobtainable	$7.94 \cdot 10^{5}$	$8.19 \cdot 10^{5}$	$7.94 \cdot 10^{5}$	$8.19 \cdot 10^{5}$	1.00	0.00
User_0.unobtainable.idle (p:0, id:0)	$7.94 \cdot 10^{5}$	$8.19 \cdot 10^{5}$	$7.94 \cdot 10^{5}$	$8.19 \cdot 10^{5}$	1.00	0.00

Table C.45: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model Telephony (state machine User_0 only). The Java code has been generated with the 'Random + Det' decision mode and 'Statement' locking mode enabled. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

Performance results for target model 'Telephony' ($n = 20, t = 30$, Random + Det, Statement, User_1)						
Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
`User_1.busy`	$2.45 \cdot 10^{6}$	$5.99 \cdot 10^{6}$	$2.45 \cdot 10^{6}$	$5.99 \cdot 10^{6}$	1.00	0.00
User_1.busy.idle (p:0, id:0)	$2.45 \cdot 10^{6}$	$5.99 \cdot 10^{6}$	$2.45 \cdot 10^{6}$	$5.99 \cdot 10^{6}$	1.00	0.00
`User_1.calling`	$4.30 \cdot 10^{6}$	$1.05 \cdot 10^{7}$	$4.30 \cdot 10^{6}$	$1.05 \cdot 10^{7}$	1.00	0.00
User_1.calling.busy (p:0, id:0)	$4.30 \cdot 10^{6}$	$1.05 \cdot 10^{7}$	$1.25 \cdot 10^{6}$	$3.06 \cdot 10^{6}$	$2.28 \cdot 10^{-1}$	$1.29 \cdot 10^{-1}$
User_1.calling.unobtainable (p:0, id:1)	$3.00 \cdot 10^{5}$	$7.33 \cdot 10^{5}$	$3.00 \cdot 10^{5}$	$7.33 \cdot 10^{5}$	1.00	0.00
User_1.calling.ringback (p:0, id:2)	$3.00 \cdot 10^{5}$	$7.33 \cdot 10^{5}$	$3.00 \cdot 10^{5}$	$7.33 \cdot 10^{5}$	1.00	0.00
User_1.calling.busy (p:0, id:3)	$2.45 \cdot 10^{6}$	$5.99 \cdot 10^{6}$	$1.20 \cdot 10^{6}$	$2.93 \cdot 10^{6}$	$4.50 \cdot 10^{-1}$	$2.32 \cdot 10^{-1}$
User_1.calling.calling (p:0, id:4)	$1.25 \cdot 10^{6}$	$3.06 \cdot 10^{6}$	$1.25 \cdot 10^{6}$	$3.06 \cdot 10^{6}$	$6.71 \cdot 10^{-1}$	$4.24 \cdot 10^{-1}$
User_1.calling.oalert (p:0, id:5)	$8.00 \cdot 10^{-1}$	$7.68 \cdot 10^{-1}$	$8.00 \cdot 10^{-1}$	$7.68 \cdot 10^{-1}$	1.00	0.00
`User_1.dialing`	$3.60 \cdot 10^{6}$	$8.80 \cdot 10^{6}$	$3.60 \cdot 10^{6}$	$8.80 \cdot 10^{6}$	1.00	0.00
User_1.dialing.idle (p:0, id:0)	$6.00 \cdot 10^{5}$	$1.47 \cdot 10^{6}$	$6.00 \cdot 10^{5}$	$1.47 \cdot 10^{6}$	1.00	0.00
User_1.dialing.calling (p:0, id:1)	$6.00 \cdot 10^{5}$	$1.47 \cdot 10^{6}$	$6.00 \cdot 10^{5}$	$1.47 \cdot 10^{6}$	1.00	0.00
User_1.dialing.calling (p:0, id:2)	$6.00 \cdot 10^{5}$	$1.47 \cdot 10^{6}$	$6.00 \cdot 10^{5}$	$1.47 \cdot 10^{6}$	1.00	0.00
User_1.dialing.calling (p:0, id:3)	$6.00 \cdot 10^{5}$	$1.47 \cdot 10^{6}$	$6.00 \cdot 10^{5}$	$1.47 \cdot 10^{6}$	1.00	0.00
User_1.dialing.calling (p:0, id:4)	$6.00 \cdot 10^{5}$	$1.47 \cdot 10^{6}$	$6.00 \cdot 10^{5}$	$1.47 \cdot 10^{6}$	1.00	0.00
User_1.dialing.calling (p:0, id:5)	$6.00 \cdot 10^{5}$	$1.47 \cdot 10^{6}$	$6.00 \cdot 10^{5}$	$1.47 \cdot 10^{6}$	1.00	0.00
`User_1.dveoringout`	$7.50 \cdot 10^{-1}$	$7.86 \cdot 10^{-1}$	$7.50 \cdot 10^{-1}$	$7.86 \cdot 10^{-1}$	1.00	0.00
User_1.dveoringout.idle (p:0, id:0)	$7.50 \cdot 10^{-1}$	$7.86 \cdot 10^{-1}$	$7.50 \cdot 10^{-1}$	$7.86 \cdot 10^{-1}$	1.00	0.00
`User_1.idle`	$4.47 \cdot 10^{7}$	$1.99 \cdot 10^{7}$	$4.47 \cdot 10^{7}$	$1.99 \cdot 10^{7}$	1.00	0.00
User_1.idle.dialing (p:0, id:0)	$4.47 \cdot 10^{7}$	$1.99 \cdot 10^{7}$	$3.60 \cdot 10^{6}$	$8.80 \cdot 10^{6}$	$2.41 \cdot 10^{-1}$	$4.28 \cdot 10^{-1}$
User_1.idle.qi (p:0, id:1)	$4.11 \cdot 10^{7}$	$2.51 \cdot 10^{7}$	$4.11 \cdot 10^{7}$	$2.51 \cdot 10^{7}$	1.00	0.00
`User_1.oalert`	$8.00 \cdot 10^{-1}$	$7.68 \cdot 10^{-1}$	$8.00 \cdot 10^{-1}$	$7.68 \cdot 10^{-1}$	1.00	0.00
User_1.oalert.oconnected (p:0, id:1)	$8.00 \cdot 10^{-1}$	$7.68 \cdot 10^{-1}$	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	$8.33 \cdot 10^{-2}$	$2.89 \cdot 10^{-1}$
User_1.oalert.dveoringout (p:0, id:2)	$7.50 \cdot 10^{-1}$	$7.86 \cdot 10^{-1}$	$7.50 \cdot 10^{-1}$	$7.86 \cdot 10^{-1}$	1.00	0.00
`User_1.oconnected`	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	1.00	NaN
User_1.oconnected.idle (p:0, id:0)	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	1.00	NaN
`User_1.qi`	$4.11 \cdot 10^{7}$	$2.51 \cdot 10^{7}$	$4.11 \cdot 10^{7}$	$2.51 \cdot 10^{7}$	1.00	0.00
User_1.qi.talert (p:0, id:0)	$4.11 \cdot 10^{7}$	$2.51 \cdot 10^{7}$	$5.00 \cdot 10^{-1}$	$6.07 \cdot 10^{-1}$	$1.67 \cdot 10^{-1}$	$3.83 \cdot 10^{-1}$
User_1.qi.idle (p:0, id:1)	$4.11 \cdot 10^{7}$	$2.51 \cdot 10^{7}$	$4.11 \cdot 10^{7}$	$2.51 \cdot 10^{7}$	1.00	0.00
`User_1.ringback`	$5.01 \cdot 10^{5}$	$1.28 \cdot 10^{6}$	$3.00 \cdot 10^{5}$	$7.33 \cdot 10^{5}$	$8.91 \cdot 10^{-1}$	$2.02 \cdot 10^{-1}$
User_1.ringback.idle (p:0, id:0)	$2.51 \cdot 10^{5}$	$6.42 \cdot 10^{5}$	$2.51 \cdot 10^{5}$	$6.42 \cdot 10^{5}$	1.00	0.00
User_1.ringback.calling (p:0, id:1)	$2.51 \cdot 10^{5}$	$6.42 \cdot 10^{5}$	$4.95 \cdot 10^{4}$	$2.21 \cdot 10^{5}$	$6.11 \cdot 10^{-1}$	$4.91 \cdot 10^{-1}$
`User_1.talert`	$8.00 \cdot 10^{-1}$	1.06	$5.00 \cdot 10^{-1}$	$6.07 \cdot 10^{-1}$	$7.41 \cdot 10^{-1}$	$3.13 \cdot 10^{-1}$
User_1.talert.tpickup (p:0, id:1)	$5.00 \cdot 10^{-1}$	$6.07 \cdot 10^{-1}$	$3.50 \cdot 10^{-1}$	$4.89 \cdot 10^{-1}$	$7.22 \cdot 10^{-1}$	$4.41 \cdot 10^{-1}$
User_1.talert.idle (p:0, id:2)	$1.50 \cdot 10^{-1}$	$3.66 \cdot 10^{-1}$	$1.50 \cdot 10^{-1}$	$3.66 \cdot 10^{-1}$	1.00	0.00
`User_1.tconnected`	$1.02 \cdot 10^{7}$	$3.15 \cdot 10^{7}$	$1.02 \cdot 10^{7}$	$3.15 \cdot 10^{7}$	$6.73 \cdot 10^{-1}$	$4.48 \cdot 10^{-1}$
User_1.tconnected.tconnected (p:0, id:0)	$1.02 \cdot 10^{7}$	$3.15 \cdot 10^{7}$	$5.11 \cdot 10^{6}$	$1.57 \cdot 10^{7}$	$3.31 \cdot 10^{-1}$	$2.32 \cdot 10^{-1}$
User_1.tconnected.tconnected (p:0, id:1)	$5.11 \cdot 10^{6}$	$1.57 \cdot 10^{7}$	$5.11 \cdot 10^{6}$	$1.57 \cdot 10^{7}$	$5.76 \cdot 10^{-1}$	$4.77 \cdot 10^{-1}$
User_1.tconnected.idle (p:0, id:2)	2.10	6.40	$1.50 \cdot 10^{-1}$	$3.66 \cdot 10^{-1}$	$3.67 \cdot 10^{-1}$	$5.48 \cdot 10^{-1}$
`User_1.tpickup`	$3.50 \cdot 10^{-1}$	$4.89 \cdot 10^{-1}$	$3.50 \cdot 10^{-1}$	$4.89 \cdot 10^{-1}$	1.00	0.00
User_1.tpickup.tconnected (p:0, id:0)	$3.50 \cdot 10^{-1}$	$4.89 \cdot 10^{-1}$	$2.50 \cdot 10^{-1}$	$4.44 \cdot 10^{-1}$	$7.14 \cdot 10^{-1}$	$4.88 \cdot 10^{-1}$
User_1.tpickup.idle (p:0, id:1)	$1.00 \cdot 10^{-1}$	$3.08 \cdot 10^{-1}$	$1.00 \cdot 10^{-1}$	$3.08 \cdot 10^{-1}$	1.00	0.00
`User_1.unobtainable`	$3.00 \cdot 10^{5}$	$7.33 \cdot 10^{5}$	$3.00 \cdot 10^{5}$	$7.33 \cdot 10^{5}$	1.00	0.00
User_1.unobtainable.idle (p:0, id:0)	$3.00 \cdot 10^{5}$	$7.33 \cdot 10^{5}$	$3.00 \cdot 10^{5}$	$7.33 \cdot 10^{5}$	1.00	0.00

Table C.46: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model `Telephony` (state machine `User_1` only). The Java code has been generated with the 'Random + Det' decision mode and 'Statement' locking mode enabled. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

Performance results for target model 'Telephony' ($n = 20, t = 30$, Random + Det, Statement, User_2)						
Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
User_2.busy	$3.83 \cdot 10^{6}$	$6.81 \cdot 10^{6}$	$3.83 \cdot 10^{6}$	$6.81 \cdot 10^{6}$	1.00	0.00
User_2.busy.idle (p:0, id:0)	$3.83 \cdot 10^{6}$	$6.81 \cdot 10^{6}$	$3.83 \cdot 10^{6}$	$6.81 \cdot 10^{6}$	1.00	0.00
User_2.calling	$7.62 \cdot 10^{6}$	$1.36 \cdot 10^{7}$	$7.62 \cdot 10^{6}$	$1.36 \cdot 10^{7}$	1.00	0.00
User_2.calling.busy (p:0, id:0)	$7.62 \cdot 10^{6}$	$1.36 \cdot 10^{7}$	$2.88 \cdot 10^{6}$	$5.12 \cdot 10^{6}$	$2.97 \cdot 10^{-1}$	$1.64 \cdot 10^{-1}$
User_2.calling.unobtainable (p:0, id:1)	$4.56 \cdot 10^{5}$	$8.11 \cdot 10^{5}$	$4.56 \cdot 10^{5}$	$8.11 \cdot 10^{5}$	1.00	0.00
User_2.calling.ringback (p:0, id:2)	$4.56 \cdot 10^{5}$	$8.11 \cdot 10^{5}$	$4.56 \cdot 10^{5}$	$8.11 \cdot 10^{5}$	1.00	0.00
User_2.calling.busy (p:0, id:3)	$3.83 \cdot 10^{6}$	$6.81 \cdot 10^{6}$	$9.55 \cdot 10^{5}$	$1.70 \cdot 10^{6}$	$2.58 \cdot 10^{-1}$	$2.46 \cdot 10^{-1}$
User_2.calling.calling (p:0, id:4)	$2.88 \cdot 10^{6}$	$5.12 \cdot 10^{6}$	$2.88 \cdot 10^{6}$	$5.12 \cdot 10^{6}$	$7.54 \cdot 10^{-1}$	$3.63 \cdot 10^{-1}$
User_2.calling.oalert (p:0, id:5)	1.00	1.30	1.00	1.30	1.00	0.00
User_2.dialing	$5.47 \cdot 10^{6}$	$9.74 \cdot 10^{6}$	$5.47 \cdot 10^{6}$	$9.74 \cdot 10^{6}$	1.00	0.00
User_2.dialing.idle (p:0, id:0)	$9.12 \cdot 10^{5}$	$1.62 \cdot 10^{6}$	$9.12 \cdot 10^{5}$	$1.62 \cdot 10^{6}$	1.00	0.00
User_2.dialing.calling (p:0, id:1)	$9.12 \cdot 10^{5}$	$1.62 \cdot 10^{6}$	$9.12 \cdot 10^{5}$	$1.62 \cdot 10^{6}$	1.00	0.00
User_2.dialing.calling (p:0, id:2)	$9.12 \cdot 10^{5}$	$1.62 \cdot 10^{6}$	$9.12 \cdot 10^{5}$	$1.62 \cdot 10^{6}$	1.00	0.00
User_2.dialing.calling (p:0, id:3)	$9.12 \cdot 10^{5}$	$1.62 \cdot 10^{6}$	$9.12 \cdot 10^{5}$	$1.62 \cdot 10^{6}$	1.00	0.00
User_2.dialing.calling (p:0, id:4)	$9.12 \cdot 10^{5}$	$1.62 \cdot 10^{6}$	$9.12 \cdot 10^{5}$	$1.62 \cdot 10^{6}$	1.00	0.00
User_2.dialing.calling (p:0, id:5)	$9.12 \cdot 10^{5}$	$1.62 \cdot 10^{6}$	$9.12 \cdot 10^{5}$	$1.62 \cdot 10^{6}$	1.00	0.00
User_2.dveoringout	$9.50 \cdot 10^{-1}$	1.19	$9.50 \cdot 10^{-1}$	1.19	1.00	0.00
User_2.dveoringout.idle (p:0, id:0)	$9.50 \cdot 10^{-1}$	1.19	$9.50 \cdot 10^{-1}$	1.19	1.00	0.00
User_2.idle	$3.69 \cdot 10^{7}$	$2.11 \cdot 10^{7}$	$3.69 \cdot 10^{7}$	$2.11 \cdot 10^{7}$	1.00	0.00
User_2.idle.dialing (p:0, id:0)	$3.69 \cdot 10^{7}$	$2.11 \cdot 10^{7}$	$5.47 \cdot 10^{6}$	$9.74 \cdot 10^{6}$	$3.26 \cdot 10^{-1}$	$4.61 \cdot 10^{-1}$
User_2.idle.qi (p:0, id:1)	$3.15 \cdot 10^{7}$	$2.67 \cdot 10^{7}$	$3.15 \cdot 10^{7}$	$2.67 \cdot 10^{7}$	1.00	0.00
User_2.oalert	1.00	1.30	1.00	1.30	1.00	0.00
User_2.oalert.oconnected (p:0, id:1)	1.00	1.30	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	$2.27 \cdot 10^{-2}$	$7.54 \cdot 10^{-2}$
User_2.oalert.dveoringout (p:0, id:2)	$9.50 \cdot 10^{-1}$	1.19	$9.50 \cdot 10^{-1}$	1.19	1.00	0.00
User_2.oconnected	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	1.00	NaN
User_2.oconnected.idle (p:0, id:0)	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	$5.00 \cdot 10^{-2}$	$2.24 \cdot 10^{-1}$	1.00	NaN
User_2.qi	$3.15 \cdot 10^{7}$	$2.67 \cdot 10^{7}$	$3.15 \cdot 10^{7}$	$2.67 \cdot 10^{7}$	1.00	0.00
User_2.qi.talert (p:0, id:0)	$3.15 \cdot 10^{7}$	$2.67 \cdot 10^{7}$	$5.00 \cdot 10^{-1}$	$8.27 \cdot 10^{-1}$	$2.50 \cdot 10^{-1}$	$4.47 \cdot 10^{-1}$
User_2.qi.idle (p:0, id:1)	$3.15 \cdot 10^{7}$	$2.67 \cdot 10^{7}$	$3.15 \cdot 10^{7}$	$2.67 \cdot 10^{7}$	1.00	0.00
User_2.ringback	$5.50 \cdot 10^{5}$	$1.05 \cdot 10^{6}$	$4.56 \cdot 10^{5}$	$8.11 \cdot 10^{5}$	$9.50 \cdot 10^{-1}$	$1.58 \cdot 10^{-1}$
User_2.ringback.idle (p:0, id:0)	$2.75 \cdot 10^{5}$	$5.27 \cdot 10^{5}$	$2.75 \cdot 10^{5}$	$5.27 \cdot 10^{5}$	1.00	0.00
User_2.ringback.calling (p:0, id:1)	$2.75 \cdot 10^{5}$	$5.27 \cdot 10^{5}$	$1.81 \cdot 10^{5}$	$3.73 \cdot 10^{5}$	$8.00 \cdot 10^{-1}$	$4.47 \cdot 10^{-1}$
User_2.talert	$7.50 \cdot 10^{-1}$	1.29	$5.00 \cdot 10^{-1}$	$8.27 \cdot 10^{-1}$	$7.86 \cdot 10^{-1}$	$3.04 \cdot 10^{-1}$
User_2.talert.tpickup (p:0, id:1)	$5.00 \cdot 10^{-1}$	$8.27 \cdot 10^{-1}$	$4.00 \cdot 10^{-1}$	$7.54 \cdot 10^{-1}$	$7.86 \cdot 10^{-1}$	$3.93 \cdot 10^{-1}$
User_2.talert.idle (p:0, id:2)	$1.00 \cdot 10^{-1}$	$3.08 \cdot 10^{-1}$	$1.00 \cdot 10^{-1}$	$3.08 \cdot 10^{-1}$	1.00	0.00
User_2.tconnected	$1.79 \cdot 10^{7}$	$4.39 \cdot 10^{7}$	$1.79 \cdot 10^{7}$	$4.39 \cdot 10^{7}$	$8.46 \cdot 10^{-1}$	$3.77 \cdot 10^{-1}$
User_2.tconnected.tconnected (p:0, id:0)	$1.79 \cdot 10^{7}$	$4.39 \cdot 10^{7}$	$8.97 \cdot 10^{6}$	$2.19 \cdot 10^{7}$	$3.38 \cdot 10^{-1}$	$2.52 \cdot 10^{-1}$
User_2.tconnected.tconnected (p:0, id:1)	$8.97 \cdot 10^{6}$	$2.19 \cdot 10^{7}$	$8.97 \cdot 10^{6}$	$2.19 \cdot 10^{7}$	$6.29 \cdot 10^{-1}$	$4.87 \cdot 10^{-1}$
User_2.tconnected.idle (p:0, id:2)	1.95	8.26	$1.50 \cdot 10^{-1}$	$3.66 \cdot 10^{-1}$	$6.76 \cdot 10^{-1}$	$5.62 \cdot 10^{-1}$
User_2.tpickup	$4.00 \cdot 10^{-1}$	$7.54 \cdot 10^{-1}$	$4.00 \cdot 10^{-1}$	$7.54 \cdot 10^{-1}$	1.00	0.00
User_2.tpickup.tconnected (p:0, id:0)	$4.00 \cdot 10^{-1}$	$7.54 \cdot 10^{-1}$	$3.00 \cdot 10^{-1}$	$4.70 \cdot 10^{-1}$	$8.89 \cdot 10^{-1}$	$2.72 \cdot 10^{-1}$
User_2.tpickup.idle (p:0, id:1)	$1.00 \cdot 10^{-1}$	$4.47 \cdot 10^{-1}$	$1.00 \cdot 10^{-1}$	$4.47 \cdot 10^{-1}$	1.00	NaN
User_2.unobtainable	$4.56 \cdot 10^{5}$	$8.11 \cdot 10^{5}$	$4.56 \cdot 10^{5}$	$8.11 \cdot 10^{5}$	1.00	0.00
User_2.unobtainable.idle (p:0, id:0)	$4.56 \cdot 10^{5}$	$8.11 \cdot 10^{5}$	$4.56 \cdot 10^{5}$	$8.11 \cdot 10^{5}$	1.00	0.00

Table C.47: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model Telephony (state machine User_2 only). The Java code has been generated with the 'Random + Det' decision mode and 'Statement' locking mode enabled. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.

Performance results for target model 'Telephony' ($n = 20, t = 30$, Random + Det, Statement, User_3)						
Target	$\mu(e)$	$\sigma(e)$	$\mu(se)$	$\sigma(se)$	$\mu(sr)$	$\sigma(sr)$
User_3.busy	$1.45 \cdot 10^{6}$	$4.49 \cdot 10^{6}$	$1.45 \cdot 10^{6}$	$4.49 \cdot 10^{6}$	1.00	0.00
User_3.busy.idle (p:0, id:0)	$1.45 \cdot 10^{6}$	$4.49 \cdot 10^{6}$	$1.45 \cdot 10^{6}$	$4.49 \cdot 10^{6}$	1.00	0.00
User_3.calling	$3.99 \cdot 10^{6}$	$1.24 \cdot 10^{7}$	$3.99 \cdot 10^{6}$	$1.24 \cdot 10^{7}$	1.00	0.00
User_3.calling.busy (p:0, id:0)	$3.99 \cdot 10^{6}$	$1.24 \cdot 10^{7}$	$1.45 \cdot 10^{6}$	$4.49 \cdot 10^{6}$	$3.12 \cdot 10^{-1}$	$1.65 \cdot 10^{-1}$
User_3.calling.unobtainable (p:0, id:1)	$1.75 \cdot 10^{5}$	$5.41 \cdot 10^{5}$	$1.75 \cdot 10^{5}$	$5.41 \cdot 10^{5}$	1.00	0.00
User_3.calling.ringback (p:0, id:2)	$1.75 \cdot 10^{5}$	$5.41 \cdot 10^{5}$	$1.75 \cdot 10^{5}$	$5.41 \cdot 10^{5}$	1.00	0.00
User_3.calling.calling (p:0, id:4)	$2.20 \cdot 10^{6}$	$6.83 \cdot 10^{6}$	$2.20 \cdot 10^{6}$	$6.83 \cdot 10^{6}$	$6.71 \cdot 10^{-1}$	$3.90 \cdot 10^{-1}$
User_3.calling.oalert (p:0, id:5)	$9.00 \cdot 10^{-1}$	$9.12 \cdot 10^{-1}$	$9.00 \cdot 10^{-1}$	$9.12 \cdot 10^{-1}$	1.00	0.00
User_3.dialing	$2.10 \cdot 10^{6}$	$6.50 \cdot 10^{6}$	$2.10 \cdot 10^{6}$	$6.50 \cdot 10^{6}$	1.00	0.00
User_3.dialing.idle (p:0, id:0)	$3.50 \cdot 10^{5}$	$1.08 \cdot 10^{6}$	$3.50 \cdot 10^{5}$	$1.08 \cdot 10^{6}$	1.00	0.00
User_3.dialing.calling (p:0, id:1)	$3.50 \cdot 10^{5}$	$1.08 \cdot 10^{6}$	$3.50 \cdot 10^{5}$	$1.08 \cdot 10^{6}$	1.00	0.00
User_3.dialing.calling (p:0, id:2)	$3.50 \cdot 10^{5}$	$1.08 \cdot 10^{6}$	$3.50 \cdot 10^{5}$	$1.08 \cdot 10^{6}$	1.00	0.00
User_3.dialing.calling (p:0, id:3)	$3.50 \cdot 10^{5}$	$1.08 \cdot 10^{6}$	$3.50 \cdot 10^{5}$	$1.08 \cdot 10^{6}$	1.00	0.00
User_3.dialing.calling (p:0, id:4)	$3.50 \cdot 10^{5}$	$1.08 \cdot 10^{6}$	$3.50 \cdot 10^{5}$	$1.08 \cdot 10^{6}$	1.00	0.00
User_3.dialing.calling (p:0, id:5)	$3.50 \cdot 10^{5}$	$1.08 \cdot 10^{6}$	$3.50 \cdot 10^{5}$	$1.08 \cdot 10^{6}$	1.00	0.00
User_3.dveoringout	$7.50 \cdot 10^{-1}$	$7.16 \cdot 10^{-1}$	$7.50 \cdot 10^{-1}$	$7.16 \cdot 10^{-1}$	1.00	0.00
User_3.dveoringout.idle (p:0, id:0)	$7.50 \cdot 10^{-1}$	$7.16 \cdot 10^{-1}$	$7.50 \cdot 10^{-1}$	$7.16 \cdot 10^{-1}$	1.00	0.00
User_3.idle	$4.28 \cdot 10^{7}$	$2.22 \cdot 10^{7}$	$4.28 \cdot 10^{7}$	$2.22 \cdot 10^{7}$	1.00	0.00
User_3.idle.dialing (p:0, id:0)	$4.28 \cdot 10^{7}$	$2.22 \cdot 10^{7}$	$2.10 \cdot 10^{6}$	$6.50 \cdot 10^{6}$	$1.69 \cdot 10^{-1}$	$3.60 \cdot 10^{-1}$
User_3.idle.qi (p:0, id:1)	$4.07 \cdot 10^{7}$	$2.51 \cdot 10^{7}$	$4.07 \cdot 10^{7}$	$2.51 \cdot 10^{7}$	1.00	0.00
User_3.oalert	$9.00 \cdot 10^{-1}$	$9.12 \cdot 10^{-1}$	$9.00 \cdot 10^{-1}$	$9.12 \cdot 10^{-1}$	1.00	0.00
User_3.oalert.oconnected (p:0, id:1)	$9.00 \cdot 10^{-1}$	$9.12 \cdot 10^{-1}$	$1.50 \cdot 10^{-1}$	$3.66 \cdot 10^{-1}$	$1.11 \cdot 10^{-1}$	$2.05 \cdot 10^{-1}$
User_3.oalert.dveoringout (p:0, id:2)	$7.50 \cdot 10^{-1}$	$7.16 \cdot 10^{-1}$	$7.50 \cdot 10^{-1}$	$7.16 \cdot 10^{-1}$	1.00	0.00
User_3.oconnected	$1.50 \cdot 10^{-1}$	$3.66 \cdot 10^{-1}$	$1.50 \cdot 10^{-1}$	$3.66 \cdot 10^{-1}$	1.00	0.00
User_3.oconnected.idle (p:0, id:0)	$1.50 \cdot 10^{-1}$	$3.66 \cdot 10^{-1}$	$1.50 \cdot 10^{-1}$	$3.66 \cdot 10^{-1}$	1.00	0.00
User_3.qi	$4.07 \cdot 10^{7}$	$2.51 \cdot 10^{7}$	$4.07 \cdot 10^{7}$	$2.51 \cdot 10^{7}$	1.00	0.00
User_3.qi.talert (p:0, id:0)	$4.07 \cdot 10^{7}$	$2.51 \cdot 10^{7}$	$4.00 \cdot 10^{-1}$	$5.03 \cdot 10^{-1}$	$2.11 \cdot 10^{-1}$	$4.19 \cdot 10^{-1}$
User_3.qi.idle (p:0, id:1)	$4.07 \cdot 10^{7}$	$2.51 \cdot 10^{7}$	$4.07 \cdot 10^{7}$	$2.51 \cdot 10^{7}$	1.00	0.00
User_3.ringback	$2.54 \cdot 10^{5}$	$8.08 \cdot 10^{5}$	$1.75 \cdot 10^{5}$	$5.41 \cdot 10^{5}$	$8.03 \cdot 10^{-1}$	$3.24 \cdot 10^{-1}$
User_3.ringback.idle (p:0, id:0)	$1.27 \cdot 10^{5}$	$4.04 \cdot 10^{5}$	$1.27 \cdot 10^{5}$	$4.04 \cdot 10^{5}$	1.00	0.00
User_3.ringback.calling (p:0, id:1)	$1.27 \cdot 10^{5}$	$4.04 \cdot 10^{5}$	$4.80 \cdot 10^{4}$	$2.14 \cdot 10^{5}$	$6.67 \cdot 10^{-1}$	$4.71 \cdot 10^{-1}$
User_3.talert	$5.50 \cdot 10^{-1}$	$8.26 \cdot 10^{-1}$	$4.00 \cdot 10^{-1}$	$5.03 \cdot 10^{-1}$	$8.54 \cdot 10^{-1}$	$2.74 \cdot 10^{-1}$
User_3.talert.tpickup (p:0, id:1)	$4.00 \cdot 10^{-1}$	$5.03 \cdot 10^{-1}$	$4.00 \cdot 10^{-1}$	$5.03 \cdot 10^{-1}$	1.00	0.00
User_3.tconnected	$1.77 \cdot 10^{7}$	$4.38 \cdot 10^{7}$	$1.77 \cdot 10^{7}$	$4.38 \cdot 10^{7}$	$8.10 \cdot 10^{-1}$	$3.25 \cdot 10^{-1}$
User_3.tconnected.tconnected (p:0, id:0)	$1.77 \cdot 10^{7}$	$4.38 \cdot 10^{7}$	$8.85 \cdot 10^{6}$	$2.19 \cdot 10^{7}$	$3.80 \cdot 10^{-1}$	$2.17 \cdot 10^{-1}$
User_3.tconnected.tconnected (p:0, id:1)	$8.85 \cdot 10^{6}$	$2.19 \cdot 10^{7}$	$8.85 \cdot 10^{6}$	$2.19 \cdot 10^{7}$	$6.67 \cdot 10^{-1}$	$4.71 \cdot 10^{-1}$
User_3.tconnected.idle (p:0, id:2)	$3.00 \cdot 10^{-1}$	$9.79 \cdot 10^{-1}$	$1.00 \cdot 10^{-1}$	$3.08 \cdot 10^{-1}$	$3.75 \cdot 10^{-1}$	$1.77 \cdot 10^{-1}$
User_3.tpickup	$4.00 \cdot 10^{-1}$	$5.03 \cdot 10^{-1}$	$4.00 \cdot 10^{-1}$	$5.03 \cdot 10^{-1}$	1.00	0.00
User_3.tpickup.tconnected (p:0, id:0)	$4.00 \cdot 10^{-1}$	$5.03 \cdot 10^{-1}$	$2.50 \cdot 10^{-1}$	$4.44 \cdot 10^{-1}$	$6.25 \cdot 10^{-1}$	$5.18 \cdot 10^{-1}$
User_3.tpickup.idle (p:0, id:1)	$1.50 \cdot 10^{-1}$	$3.66 \cdot 10^{-1}$	$1.50 \cdot 10^{-1}$	$3.66 \cdot 10^{-1}$	1.00	0.00
User_3.unobtainable	$1.75 \cdot 10^{5}$	$5.41 \cdot 10^{5}$	$1.75 \cdot 10^{5}$	$5.41 \cdot 10^{5}$	1.00	0.00
User_3.unobtainable.idle (p:0, id:0)	$1.75 \cdot 10^{5}$	$5.41 \cdot 10^{5}$	$1.75 \cdot 10^{5}$	$5.41 \cdot 10^{5}$	1.00	0.00

Table C.48: A table containing statistics on the number of executions (e), number of successful executions (se) and success ratio (sr) measured during the execution of the target model Telephony (state machine User_3 only). The Java code has been generated with the 'Random + Det' decision mode and 'Statement' locking mode enabled. The results have been measured over a time span of 30 seconds, where each entry is represented by measurements taken over 20 trials.