Using a DSL and fine-grained model : transformations to explore the boundaries of model verification

Using a DSL and Fine-grained Model Transformations to Explore the Boundaries of Model Verification

M.F. van Amstel, M.G.J. van den Brand, L.J.P. Engelen

Department of Mathematics and Computer Science
Eindhoven University of Technology
P.O. Box 513, 5600 MB, Eindhoven, The Netherlands
{M.F.v.Amstel | M.G.J.v.d.Brand | L.J.P.Engelen}@tue.nl

Abstract. Traditionally, the state-space explosion problem in model checking is handled by applying abstractions and simplifications to the model that needs to be verified. In this paper, we propose a model-driven engineering approach that works the other way around. Instead of making a concrete model more abstract, we propose to refine an abstract model to make it more concrete. We propose to use fine-grained model transformations to enable model checking of models that are as close to the implementation model as possible. We applied our approach in a case study. The results show that it is possible to validate models that are more concrete when fine-grained transformations are applied.

1 Introduction

Model-driven engineering (MDE) is a software engineering paradigm in which models play a central role throughout the entire development process [1]. MDE combines domain-specific languages (DSLs) for modeling at a higher level of abstraction and model transformations for the automated generation of various artifacts, such as code from these models. Our goal is to generate reliable code from models specified using a DSL. To increase the reliability of generated code, formal methods such as verification can be used. Model checking is an automated verification technique that checks whether a formally specified property holds for a model of a system [2]. An exhaustive state space search is performed by an automated model checker to determine whether a property holds in a finite state model of a system. Often, this state space is huge and model checking is no longer a feasible approach for verification. Traditionally, abstractions and simplifications are applied to the model to enable model checking in such cases [3–5]. We propose an MDE approach to enable model checking that works the other way around. Instead of starting with a large model and iteratively simplifying it, we start with a small model and iteratively refine it.

In a typical MDE development process domain-specific models are iteratively refined using model transformations until a model is acquired with enough details to implement a system [6]. To increase the reliability of the final system,

model checking can be employed. Because of the aforementioned state-space explosion problem, model checking the final system may be infeasible. Therefore, we propose to define a model transformation that transforms the domain-specific models to models suitable for model checking. Using this model transformation, model checking can be applied on the domain-specific models in every stage of the refinement process. While model checking the final system may be infeasible, using this approach intermediate models close to the implementation can be model checked.

In this paper, we demonstrate this approach using a domain-specific language (DSL) for modeling systems consisting of concurrent, communicating objects. This DSL has an intuitive graphical syntax to model the structure and behavior of a system, and offers constructs such as synchronous communication over lossless channels to make models concise. To execute models, we implemented a chain of transformations that transform models specified using our DSL to a restricted version of C [7]. The semantic properties of this implementation platform differ from those of our DSL, which means that some construct that are available in our DSL have no direct counterparts on the implementation platform. This platform, for instance, does not offer constructs such as synchronous communication and lossless channels. Instead, communication on the implementation platform is asynchronous and takes place over a lossy channel. To enable transformation from our DSL to the implementation platform, the semantic gaps between the two platforms need to be bridged [8]. Therefore, we added a number of constructs to our DSL and implemented a number of transformations that can be used to stepwise refine models to align the semantic properties of the DSL with the implementation platform. These transformations replace the constructs in a model that are not offered by the implementation platform by constructs that it does offer, while preserving the observable behavior of the model. A final transformation transforms the resulting model to executable code.

We also implemented a model transformation from our DSL to Spin [9], to enable model checking of the (intermediate) domain-specific models. Our first experiments showed that verification of the models generated by the refining transformations using Spin was infeasible due to state-space explosion. We concluded that the change induced on the models by the transformations was too large, i.e., the transformations were too coarse-grained. Therefore, we split up the coarse-grained transformations into more fine-grained ones. The impact of such a fine-grained transformation on a model is smaller, i.e., the model does not change drastically. This is reflected in the increase of the state space size that is searched by Spin. Using this approach, intermediate models that are generated by the fine-grained transformations can be model checked almost all the way up to the models that can be executed, because the state space stays within reasonable bounds.

The remainder of this paper is structured as follows. Our approach to enable model checking of models almost all the way up to the implementation model is discussed in Section 2. Section 3 describes the transformations that can be used to refine the models created using the DSL as well as transformations for

transforming them to different formalisms. The experiments we conducted are presented in Section 4. In Section 5 we reflect on our work. Section 6 describes related work. Conclusions and directions for further research are given in Section 7.

2 Approach

Our goal is to generate reliable code from models specified using a DSL. To increase the reliability of generated code, formal methods such as verification can be used. To ensure that the same model is verified and executed, models specified using the DSL should automatically be transformed to models suitable for these purposes. In this way, these models do not have to be created by hand. This enables the use of formal methods without having to create models suitable for that purpose separately. This has the advantage that engineers do not have to learn the syntax and semantics of different languages. Moreover, manual transformation is a slow and error-prone task.

Often, the DSL and the implementation platform have different semantical characteristics. Therefore, the semantic gap between the two formalisms needs to be bridged [8]. We propose to use model transformations to refine a DSL model in such a way that the semantic properties of the DSL and the implementation platform are aligned. In this way, the abstract DSL model becomes concrete and transformation from the refined (concrete) DSL model to the implementation model is merely a syntactical transformation.

To enable verification of a DSL model, a transformation from the DSL to a formalism for verification, e.g., a model checking formalism, should be implemented. Using this transformation, it is possible to verify whether both the abstract and the concrete DSL models fulfill their requirements. From the experiments presented in Section 4, we concluded that verification of an abstract model poses no problems. However, verification of a concrete model is infeasible. The verification takes too much time and needs too many resources.

The transformations used to refine the abstract DSL models produce intermediate models. These models can also be transformed to a verification formalism. By verifying the intermediate DSL models, it is possible to verify models that are more concrete. This approach is schematically depicted in the top half of Figure 1. The check marks indicate models that can be verified, whereas the crosses indicate models that cannot be verified. Our experiments showed that it is possible to verify some of the intermediate models. However, the most concrete model that can be verified is still not concrete enough. The reason for that is that the change induced on the models by the transformations is too large, i.e., the transformations are too coarse-grained. Therefore, we propose to use more fine-grained transformations to enable verification of more concrete models. This can be achieved by splitting existing transformations into smaller parts. In this way, more intermediate models are generated that can be verified. This approach is schematically depicted in the bottom half of Figure 1. Using this approach, it is possible to verify models that are closer to the concrete model. By replac-

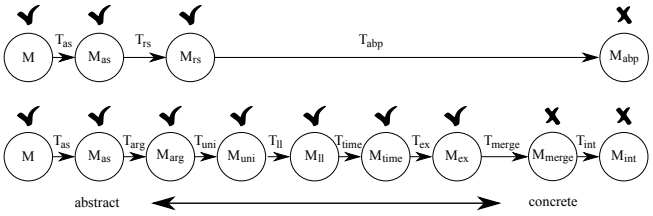


Fig. 1. Verification of intermediate models

ing the coarse-grained transformations T_{rs} and T_{abp} from Figure 1 by the more fine-grained transformations T_{arg}, T_{uni}, T_{ll}, T_{time}, T_{ex}, T_{merge}, and T_{int}, for instance, the state space of the intermediate model M_{ex} can be explored, instead of that of the less concrete model M_{rs}. These transformations are explained in Sections 3.2 and 3.3. The example shown in Figure 1 is an illustration of one of the experiments presented in Section 4. In different cases, the transformation steps as well as which of them can be verified will vary.

The most concrete model that can be model checked may still not be close enough to the implementation model. An attempt can be made to split the transformations into even smaller parts. If this is not possible anymore, another possibility is to apply the model transformation to part of the model only. Since the refinement, in this case, is applied to a small part of the model, this will most likely result in models that give rise to smaller state spaces. Using partial refinement, the boundaries of what can be verified using model checking can be explored even further.

Using more fine-grained transformations has some positive side-effects. Since fine-grained transformations tend to be smaller than course-grained ones, it is easier to locate defects in them. Also, fine-grained transformations have proven to be more reusable than course-grained ones during our experiments. Another advantage of having fine-grained transformations is that it enables shuffling the order in which they are applied. This order affects the output model, i.e., some sequences of transformations lead to more efficient implementations than others.

3 Case Architecture

In this section, we first introduce our DSL, the language used for execution, and the differences between them. Then, we discuss both coarse-grained and fine-grained transformations that overcome these differences by refining abstract models to concrete models. We do not describe the transformation that transforms the resulting concrete models specified using our DSL to the execution language. Because this transformation is only applied after the platform characteristics have been aligned, this transformation merely transforms syntax. This transformation is described in [10]. Finally, we describe the language used for verification and the transformation that transforms models specified using our DSL to the verification language. This transformation is used in Section 4 to

illustrate the difference between the coarse-grained and fine-grained transformations.

3.1 DSL and Execution Language

Simple Language of Communicating Objects We designed a domain-specific language called *Simple Language of Communicating Objects* (SLCO) [10]. It provides constructs for specifying the structure and behavior of systems consisting of concurrent, communicating objects.

An SLCO model consists of a number of classes, instances of these classes (objects), and channels. Channels are used to connect a pair of objects such that they can send signals to each other. An example of this is shown in Figure 2. Two objects, $a1$ and $a2$, that are instances of the same class, A, can communicate over a channel, c. A class has ports and variables that define the structure of

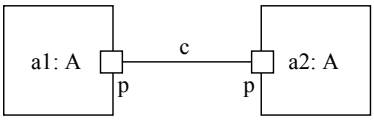


Fig. 2. Two objects connected by a channel

its instances, and state machines that describe their behavior. Ports are used to connect channels to objects. Figure 2 shows that both instances of class A have a port p connecting them to channel c. Channels in SLCO are either synchronous or asynchronous, and unidirectional or bidirectional. Furthermore, asynchronous channels can be lossless or lossy. A state machine consists of variables, states, and transitions. A transition has a source and a target state, and possibly a guard, a trigger, an effect, or a combination of these. A guard is a boolean expression that must hold to enable the transition from source state to target state. There are two types of triggers: a delay and a signal reception. If the amount of time specified by a delay has passed or if a signal is received, the transition that has such a trigger is enabled. When a transition is made from one state into another state, the statements that constitute the effect of the transition are executed. There are statements for assigning values to variables and for sending signals over channels. Figure 3 shows an example of two state machines.

Execution We use the Lego Mindstorms [11] platform for the execution of SLCO models. The key part of this platform is a programmable Lego brick, called RCX. This RCX has an infrared port for communication and is connected by wires to sensors and motors for interaction with its environment. We deliberately opted for the outdated RCX brick instead of the newer and more advanced NXT brick to investigate the strength of our transformational approach when dealing with a very primitive execution platform. The language we use to program these

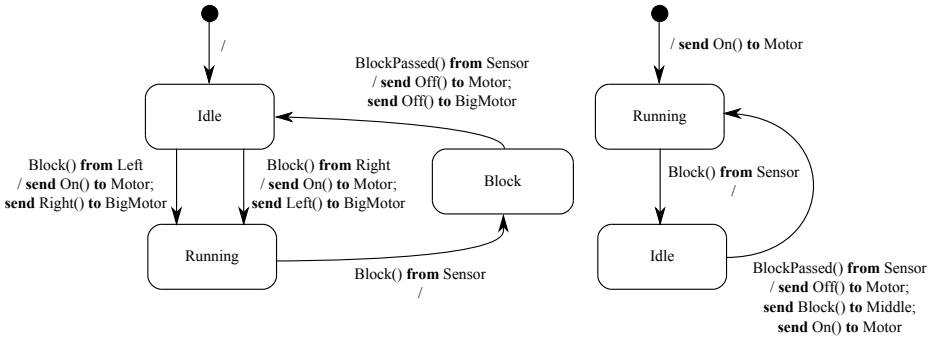


Fig. 3. Two state machines in SLCO

programmable bricks is called Not Quite C (NQC) [7]. NQC is a restricted version of C, combined with an API that provides access to the various features of the Lego Mindstorms platform such as sensors, outputs, timers, and the infrared port. To execute SLCO models, we defined a transformation from SLCO models to NQC code.

Language Characteristics The discussed platforms and languages have different characteristics. These differences are shown in Table 1. The first column

	(A)synchronous communication	Lossless/lossy communication	Concurrent objects	Variable and parameter types	Connectivity for communication
SLCO	both	both	∞	Integer, Boolean String	Point-to-point
NQC	asynchronous	lossy	limited	Integer	Broadcast

Table 1. Platform characteristics

lists the languages. The second column indicates whether communication is synchronous or asynchronous on the corresponding platform. In case communication is synchronous, both sender and receiver need to be available before a signal can be sent. In this way, sender and receiver synchronize on communication. In case communication is asynchronous, a sender can send a signal and proceed with its execution even though the receiver is not yet ready to receive the signal. The third column indicates whether channels are lossless or lossy. In case a channel is lossless, a signal that is sent will always arrive at the receiving end. In case a channel is lossy, a signal that is sent may get lost. The fourth column lists the amount of objects that can be instantiated simultaneously. In SLCO, this amount is unlimited. For Lego Mindstorms, however, this number is limited in practice. Because every object should be deployed on an RCX, the amount

of concurrent objects is bounded by the available number of RCX bricks. The fifth column shows the datatypes that are available on the corresponding platforms. The sixth column shows whether signals are broadcasted or sent using point-to-point communication. When signals are broadcasted, each signal can be received by multiple objects. In the case of point-to-point communication, however, signals are sent from one object to exactly one other object.

3.2 Coarse-Grained Model Transformations

In Section 3.1, we explained that the characteristics of the platforms differ. To execute SLCO models, these semantic platform gaps need to be bridged. Therefore, we defined a number of transformations that transform an SLCO model to a refined SLCO model with equivalent observable behavior. Each of these transformations eliminates one of the platform gaps. An SLCO model that uses synchronous communication only, for example, can be transformed to an equivalent SLCO model that uses asynchronous communication only.

First, we discuss two coarse-grained transformations that bridge platform gaps, as well as a transformation that ensures that the precondition of one of those other transformations is met. Afterwards, we discuss the more fine-grained versions of those transformations.

The coarse-grained transformations deal with only two of the five platform gaps. The modeler is responsible for creating input models that do not introduce problems concerning the other three gaps. These transformations do not introduce objects and there is no transformation that can be used to reduce the number of objects, so the modeler is responsible for creating input models that contain as much objects as can be deployed. The coarse-grained transformations also do not introduce datatypes that can not be used in NQC. If the input model does not use these datatypes, the transformations will result in a deployable model. Because there is no transformation that deals with the problem of identifying the sender of a message that has been broadcasted, only input models with two communicating parties are allowed.

Synchronized Communication over Asynchronous Channels The transformation that replaces communication using synchronous signals by communication using asynchronous signals ensures that the behavior of the model is still as desired by adding acknowledgment signals for synchronization. Whenever a signal is sent, the receiving party sends an acknowledgement indicating that the signal has been received. The sending party waits until the acknowledgement has been received. In this way, synchronization is achieved.

Lossless Communication over Lossy Channels Lossless communication over lossy channels is implemented using a variant of the alternating bit protocol (ABP) [12].This protocol ensures that each signal that is sent, is eventually received, assuming that not all signals get lost. This transformation adds the ABP to a model by adding new state machines implementing the protocol to objects

that communicate over a lossy channel. These new state machines communicate with the existing state machines in these objects using shared variables.

Exclusive Access to Ports To ensure that a model meets the precondition of the previous transformation, we use a third transformation. When multiple state machines communicate over the same port, the previous transformation may only be applied if at most one of the state machines sends a message over this port at the same time. The transformation that ensures exclusive access to ports adds a token server to ensure that this is the case. This token server is implemented as an additional state machine that is added to the objects directly. The token server and the existing state machines pass information using shared variables.

3.3 Fine-Grained Model Transformations

To minimize the influence of each transformation on the size of the state space, we implemented a number of more fine-grained transformations. In contrast to the coarse-grained transformations, there is a fine-grained transformation to deal with each of the platform gaps. The transformation that replaces synchronous signals by asynchronous signals is left unchanged. The other transformations are described below.

Lossless communication over a lossy channel Lossless communication over lossy channels is again implemented using the ABP. In this case, the ABP that is added by this transformation is implemented as a number of concurrent objects that are connected to the communicating objects using lossless channels. In contrast, the coarse-grained version of this transformation adds the ABP as a number of state machines to the communicating objects.

Reducing the number of objects The transformation that reduces the number of objects merges objects by creating a new object that contains all the variables, ports and state machines contained in the original objects. If two state machines that were originally contained in two different objects communicate over a lossless, unidirectional, synchronous channel, this form of communication is replaced by communication using shared variables. A four-phase handshake is used to ensure synchronized communication between the two state machines.

Replacing strings by integers Because strings are unavailable in NQC, we implemented a transformation that replaces all string constants by integer constants.

Making the sender of a signal explicit When multiple objects broadcast signals with the same name and number of arguments over the same medium,

the receiving object cannot determine the origin of such a signal. This situation arises when multiple RCX controllers communicate with each other, because they communicate using infrared. To enable a receiving controller to determine the origin of each signal it receives, a number identifying the sending object is appended to the names of each signal.

Making all signal names equal To keep the transformation that adds the ABP as simple as possible, our implementation of the ABP takes signals with a fixed name as input.Before this instance of the ABP can be used to substitute an asynchronous, lossless, unidirectional channel, the signal names that are sent over this channel have to be changed into the fixed name it uses. We implemented a transformation that changes the names of signals and ensures that the original names of the signals are passed as a parameter of the signals that replace them.

Replacing a bidirectional channel by two unidirectional channels Our implementation of the ABP can only substitute asynchronous, lossless, unidirectional channels. In some cases, therefore, a transformation is needed that replaces communication over a bidirectional channel by communication over two unidirectional channels.

Duplicating a channel for each state machine that uses it The four-phase handshaking we employ when merging objects does not work properly if more than one state machine sends information over the same port at the same time. When two objects are connected by a unidirectional channel and multiple state machines within one of the objects send signals over this channel, the channel must be replaced by multiple channels before these objects can be merged. We implemented a transformation that introduces a new channel with the same properties as the original channel for each state machine that sends signals over this channel.

Reducing the number of channels When two objects are connected by more than one channel, these channels can be merged into one. Therefore, we implemented a transformation that changes the names of the signals that are sent over the new channel, to distinguish between identical signals that were previously sent over different channels. Merging channels can be used to optimize a model, because it is a way to reduce the number of instances of the ABP that need to be added.

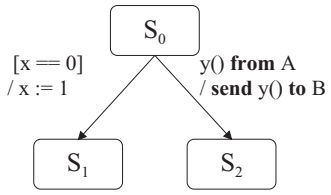
Adding delays to transitions To prevent objects from sending signals continuously, we implemented a transformation that adds delay triggers to the transitions that send signals as part of their effect. This transformation also optimizes a model, because it reduces the number of messages that are being sent, which in turn reduces the number of collisions between messages sent via infrared.

3.4 Verification

To investigate the influence of the coarse-grained and fine-grained transformations on the size of the state space of models, we use a model checker and a transformation that transforms SLCO models to models readable by this model checker.

Promela Model checking is an automated verification technique that checks whether a formally specified property holds for a model of a system [2]. We use the model checker Spin [9] for verifying our models. The input language for Spin is Promela. Promela has constructs for modeling selections and loops, based on Dijkstra's guarded commands, and primitives for message passing between processes over channels. The syntax of expressions and assignments in the Promela language is similar to that of C.

Transforming SLCO to Promela The transformation from SLCO to Promela transforms every state machine describing the behavior of an object in an SLCO model to a Promela proctype. Channels between objects are transformed to channels between proctypes. State machines can be implemented using an imperative programming style in multiple ways. We chose to implement them as jump tables using goto statements. An example of the transformation is shown in Figure 4. State S_0 depicted in Figure 4(a) is transformed to the code depicted in Figure 4(b). A state is transformed to a labeled selection statement.



(a) State machine

```
S0:
 if
 ::x == 0 -> x = 1;
   goto S1;
 ::A?y -> B!y
   goto S2;
 fi;
```

(b) Promela code

Fig. 4. Transforming SLCO to Promela

Every outgoing transition of state S_0 is transformed to an alternative of the selection statement. The semantics of the selection statement is such that it will non-deterministically execute one of the alternatives for which the guard holds and it will block if none of the guards hold. The guard is the first statement or expression of the alternative. In case the guard is an expression, it holds if it evaluates to `true`. In case the guard is a statement, it holds if the statement is executable. When the guard holds, the statements following it can be executed. The guard and statements of a transition are transformed to Promela code in

a straightforward way. After execution, the transition to a state has been completed and the state machine is in the target state of the transition. Therefore, a goto statement that jumps to the label representing the target state of a transition is added after the transformed statements in the code. Because we use a version of Spin that does not support time, we abstract from time by transforming delay triggers to skip statements. A signal reception trigger is transformed to a receive statement, which blocks until it is able to receive a message over a channel.

4 Experiments

We performed a number of experiments to determine the size of the state space of intermediate models generated by our chains of refining transformations. By transforming intermediate SLCO models to Promela models, we obtain models whose state space can be explored using Spin. For the experiments described below, we configured Spin to explore the state-spaces by means of a depth-first search with a maximum search depth of $1 \cdot 10^8$ transitions and using at most $4 \cdot 10^4$ megabytes of memory. After describing the models that serve as inputs in our experiment, we show that an approach using coarse-grained transformations quickly leads to models with very large state spaces. Then, we present the results of our experiments using fine-grained transformations. Finally, we discuss how applying transformations to a part of the applicable model elements only can also be used to explore the state space of less abstract versions of models.

4.1 Cases

We apply the refining transformations described in Section 3 to three different models. The first model consists of one object that repeatedly sends synchronous signals over a port (the producer) and one object that is always able to receive signals over a ports (the consumer). A channel connects the ports of the producer to the ports of the consumer. The model is depicted in Figure 5.

The second model describes the behavior of a system consisting of three interoperating conveyor belts. The leftmost part of Figure 3 shows the behavior of one of the three components in this system. The behavior of another component is specified using two instances of the state machine shown in the rightmost part of Figure 3. The third component models the environment of the system and is not described in this paper.

The third model consists of two objects that repeatedly send synchronous signals over a port (the producers) and one object that is always able to receive signals over two ports (the consumer). Two channels connect the ports of each of the producers to the two ports of the consumer. The model is depicted in Figure 6.

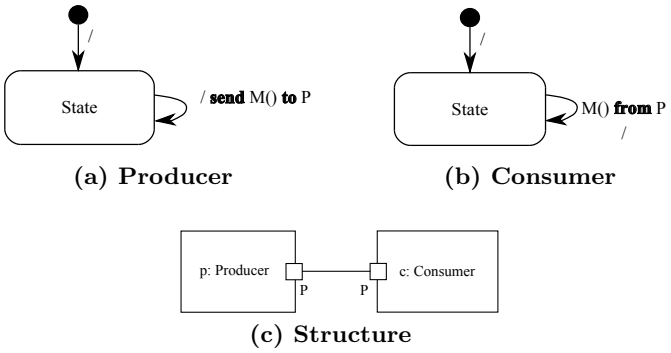


Fig. 5. One producer and one consumer

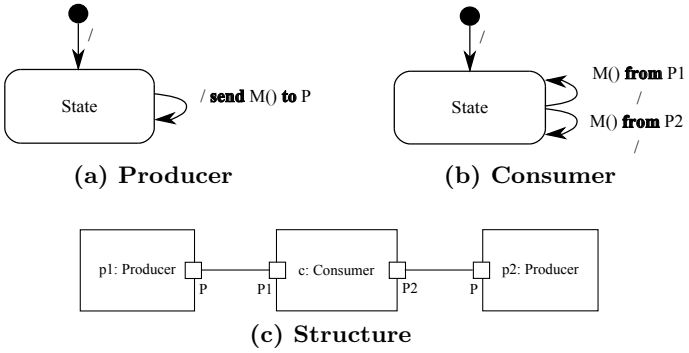


Fig. 6. Two producers and one consumer

4.2 Results

Applying coarse-grained transformations to the model of the producer and consumer leads to the state space sizes shown in Table 2.

Replacing synchronous communication by asynchronous communication approximately doubles the size of the state space. Adding a number of state machines that implement the ABP to each of the two objects, however, leads to a significant increase of the size of the state space. Although the resulting state space of the most concrete model is much larger than the one of the intermediate model, it is still small enough for verification given the aforementioned configuration of Spin.

To further illustrate the effects of the coarse-grained transformations on the size of the state space, we applied them to the second model, which is slightly more complex than the first. One of the components in the system of the three conveyor belts consists of two instances of the same state machine. Both instances communicate over the same port, which means that a token server must be added when refining this model, before the transformation that adds the ABP can be employed. Table 3 shows that adding a token server leads to a state space

	# States	# Transitions
Original	4	6
Asynchronous signals	8	11
Lossless communication	76 066 432	542 196 960

Table 2. One producer and one consumer - coarse-grained transformations

that can still be model checked. The final row in the table indicates that it is impossible to explore the entire state space before the search depth is exceeded or all available memory is used. This shows that the output of this transformation is not suited for model checking, even though the input model is still relatively small.

	# States	# Transitions
Original	494	1 294
Asynchronous signals	748	1 980
Token server	10 090	33 820
Lossless communication	–	–

Table 3. Interoperating conveyor belts - coarse-grained transformations

The results of these experiments lead us to implement the more fine-grained versions of the transformations presented in Section 3.3. Table 4 shows the effect of the fine-grained transformations on the size of the state spaces of the intermediate models in the case of the producer and the consumer, and Table 5 shows the effect in the case of the three interoperating conveyor belts. The transformations that ensure that all signals have a fixed name, replace bidirectional channels by two unidirectional channels, ensure that each state machine within an object communicates with the ABP over an exclusive channel, and replace strings by integers have no effect on the size of the state space.

In the case of the producer and the consumer, each intermediate model has a state-space that can be explored given the aforementioned configuration of Spin. In the case of the conveyor belts, however, merging objects leads to a state-space that is too large to explore. Even though the most concrete model is still unsuited for model checking, fine-grained transformations made it possible to explore an intermediate model that is more concrete than the ones produced using coarse-grained transformations.

4.3 Exploring the Boundaries

In both of the cases mentioned above, only two instances of the ABP are added, because communication takes place in two directions between one pair of objects.

	# States	# Transitions
Original	4	6
Asynchronous signals	8	11
Fixed signal names	8	11
Unidirectional channels	8	11
Lossless communication	114 388	596 367
Delays	1 009 856	5 902 673
Merged objects	83 251 840	592 242 910
Integers instead of strings	83 251 840	592 242 910

Table 4. One producer and one consumer - fine-grained transformations

	# States	# Transitions
Original	494	1 294
Asynchronous signals	748	1 980
Fixed signal names	748	1 980
Unidirectional channels	748	1 980
Lossless communication	19 148 872	141 049 260
Delays	167 466 690	1 334 614 400
Exclusive channels	167 466 690	1 334 614 400
Merged objects	–	–

Table 5. Interoperating conveyor belts - fine-grained transformations

Table 6 shows the results for the model consisting of two producers and one consumer. To achieve lossless communication over a lossy channel in this case, four instances of the ABP have to be added, because communication takes place in two directions between two pairs of objects.

Adding four instances of the objects that implement the ABP leads to an explosion of the state space. This makes it very hard to verify properties of this model using state space exploration. Table 7 shows the effect of adding an instance of the ABP to respectively one, two, and three channels in the model of two producers and one consumer, while leaving the other channels untouched.

By replacing communication over only a subset of the four channels in the model by communication via the ABP, a model is obtained with a state space that is significantly smaller than the state space corresponding to the model in which communication over all channels is replaced. In this way, verification of a model that resembles the implementation more closely than the original, more abstract, model is possible. The same approach can be used to merge only some of the objects in the model of the interoperating conveyor belts. In general, applying a refining transformation to a part of the applicable elements in the model only can be used to model check intermediate models that resemble the

	# States	# Transitions
Original	8	17
Asynchronous signals	33	68
Fixed signal names	33	68
Unidirectional channels	33	68
Lossless communication	–	–

Table 6. Two producers and one consumer - fine-grained transformations

	# States	# Transitions
Original	8	17
Asynchronous signals	33	68
Fixed signal names	33	68
Unidirectional channels	33	68
one ABP instance	5 188	21 335
two ABP instances	527 108	3 224 435
three ABP instances	105 715 260	879 085 750

Table 7. Two producers and one consumer - fine-grained transformations

implementation as close as possible, in cases where it is impossible to model check the completely refined model.

5 Discussion

In Section 4, we used the model checker Spin to illustrate the effect of both coarse-grained and fine-grained transformations on state spaces. However, our approach is not limited to one particular model checker. The refining transformations we implemented take SLCO models as input and produce SLCO models as output. Support for another model checker or a similar tool can be added by implementing a single transformation from SLCO to the formalism supported by that tool. In fact, we implemented such a transformation to a formalism for performance analysis and simulation [10].

To clearly show the influence of our refining transformations, we used no additional reduction or abstraction techniques. However, our approach can be combined with such techniques in practical situations. Using one of the standard state vector compression modes offered by Spin [13], for instance, it is possible to explore larger state spaces. Using this compression method and the configuration described in Section 4, the state space of the timed version of the model of the three conveyor belts can be explored using approximately $15 \cdot 10^3$ megabytes, instead of $31 \cdot 10^3$ megabytes.

Typically, model checking is used to verify whether a property holds for a model of a system. Because the refining transformations modify the model, properties under investigation may have to change as well. After adding communication via the ABP to a model, for example, there are unfair traces in the state space representing the behavior that all signals are discarded by the lossy channel. To consider only the fair traces, a fairness constraint has to be added to the property.

6 Related Work

Multiple proposals are presented in literature to enable model checking of huge specifications. Clarke et al. suggest four different abstraction techniques and demonstrate their practicality on a number of examples [4]. Another possibility, applied by Chan et al., is to model check only a part of the system [3]. They also applied simplifications to the model to avoid constructs that could not be handled properly by their model checker. Wing and Vaziri-Farahani enabled quick verification in a case study by applying abstractions to both the model and the verification properties [5]. They state that the choice of what abstractions to apply takes some 'good' judgement. All of the aforementioned approaches work by applying abstraction and simplification to concrete models. Our approach works the other way around, we refine an abstract model to a more refined one. Note that our approach does not preclude the use of abstractions and simplifications on the (intermediate) models. The B-method [14] is developed as a means to refine abstract specifications into implementations. By fulfilling a number of proof obligations and thus proving that each refinement step is sound, it can be proven that an implementation adheres to the corresponding initial specification. Using the B-method, reliable code is derived starting from one initial specification, whereas our approach focusses on automatically generating reliable code from every possible model that can be described using our DSL.

7 Conclusions and Future Work

7.1 Conclusions

In this paper, we proposed an approach using model checking to increase the reliability of code generated from models specified in a domain-specific language. A model transformation from the domain-specific language to a language suitable for model checking can be defined to enable model checking of domain-specific models. Using this model transformation, model checking can be applied on the domain-specific models in every stage of the refinement process. To deal with the state-explosion problem we advocate to use fine-grained model transformations to stepwise refine domain-specific models. Since fine-grained transformations tend to be smaller than coarse-grained transformations, it is easier to locate defects in them. Another advantage of fine-grained transformations is that they

may be more reusable than coarse-grained ones. When fine-grained transformations do not allow model checking of concrete enough models, it may be helpful to apply a transformation to part of a model only.

We conducted experiments to validate our approach on multiple cases with a DSL we defined. The results show that it is possible to validate models that are more concrete when fine-grained transformations are applied.

7.2 Future Work

As discussed in Section 5, reduction techniques such as partial order reduction and state vector compression can be applied to a verification model. Reduction may also be applied to domain-specific models. Models in our DSL consist of state machines. Therefore, algorithms for state machine composition, such as presented in [15], may be applicable. Reducing the number of state machines in a model may lead to smaller state-spaces.

The cases on which we applied our technique are rather small. Also the chain of fine-grained transformations is not large. We believe these small examples already show the advantages of the proposed approach. However, a point we see for future work is applying the approach to larger models and more complex transformation chains.

Model checking is one way of increasing the reliability of systems created in an MDE process. Another way to do this is using formal correctness proofs. When correctness of model transformations can be formally proven, model checking is no longer required to validate intermediate results. It would then suffice to validate the initial model only. Formally proving model transformations requires that the semantics of source and target language are formally defined. Since a lot of DSLs have an informal semantics only, the correctness of a model transformation cannot be proven. Therefore, model checking intermediate models may still be required.

Acknowledgements

We would like to thank Anton Wijs and Dragan Bošnački for their valuable feedback on our work.

This work has been carried out as part of the FALCON project under the responsibility of the Embedded Systems Institute with Vanderlande Industries as the industrial partner. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Embedded Systems Institute (BSIK03021) program.

This work has been carried out as part of the KWR 09124 project LithoSysSL.

References

1. Schmidt, D.C.: Model-Driven Engineering. Computer **39**(2) (February 2006) 25–31
2. Clarke Jr., E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press (1999)

3. Chan, W., Anderson, R.J., Beame, P., Burns, S., Modugno, F., Notkin, D., Reese, J.D.: Model Checking Large Software Specifications. IEEE Transactions on Software Engineering **24**(7) (July 1998) 498–520
4. Clarke Jr., E.M., Grumberg, O., Long, D.E.: Model Checking and Abstraction. ACM Transactions on Programming Languages and Systems **16**(5) (September 1994) 1512–1542
5. Wing, J.M., Vaziri-Farahani, M.: Model Checking Software Systems: A Case Study. SIGSOFT Software Engineering Notes **20**(4) (October 1995) 128–139
6. Kurtev, I., van den Berg, K., Akşit, M.: UML to XML-Schema Transformation: a Case Study in Managing Alternative Model Transformations in MDA. In: Forum on Specification and Design Languages (FDL'03), Frankfurt, Germany (September 2003)
7. Baum, D.: NQC Programmer's Guide. (2003)
8. van Amstel, M.F., van den Brand, M.G.J., Protić, Z., Verhoeff, T.: Transforming Process Algebra Models into UML State Machines: Bridging a Semantic Gap? In: Proceedings of the First International Conference on Model Transformation (ICMT'08). Volume 5063 of Lecture Notes in Computer Science., Zürich, Switserland, Springer (July 2008) 61–75
9. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley (2003)
10. van Amstel, M.F., van den Brand, M.G.J., Engelen, L.J.P.: An Exercise in Iterative Domain-Specific Language Design. In: Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE), Antwerp, Belgium, ACM (September 2010) 48–57
11. : Lego Mindstorms website. `http://www.lego.com/eng/education/mindstorms/home.asp`, viewed February 2010
12. Baeten, J., Middelburg, C.A.: Process Algebra with Timing. Monographs in Theoretical Computer Science An EATCS Series. Springer (2002)
13. Holzmann, G.J.: State Compression in SPIN: Recursive Indexing and Compression Training Runs. In: Proceedings of the Third International Spin Workshop, Enschede, The Netherlands (April 1997)
14. Abrial, J.R., Lee, M.K.O., Neilson, D., Scharbach, P.N., Sørensen, I.H.: The B-Method. In Prehn, S., Toetenel, W.J., eds.: Proceedings of the Fourth International Symposium of VDM Europe. Volume 552 of Lecture Notes in Computer Science., Noordwijkerhout, The Netherlands, Springer (October 1991) 398–405
15. Holzmann, G.J.: Design and Validation of Computer Protocols. Prentice-Hall Software Series. Prentice-Hall (1991)