Defining and validating the behavior of component interfaces in navigation software

Josquin, G.J.

Award date:
2014

Link to publication
Master’s thesis

Defining and validating the behavior of component interfaces in navigation software

November 9, 2012

Guido Josquin
Eindhoven University of Technology
Computer Science and Engineering
g.j.josquin@student.tue.nl
Partners

TomTom

Steering Group

Suzana Andova
Robert Lukassen

Duration

April - October 2012
Abstract

In component-based software, interfaces describe the syntax and semantics of messages that are exchanged between components. The semantics refer to the order of messages and the content of their parameters. Due to the informal nature of the current specifications, the interface semantics can be misinterpreted and bugs are introduced into the software.

A new specification method is desired to reduce the number of ambiguities in specifications and to support the usage of automated testing tools. The available options for languages are evaluated and compared. The chosen solution uses a domain-specific language based on UML statecharts. The details of this DSL are explained and a toolchain is presented that generates runnable code from models. The generated code is inserted into a framework that can automatically validate the correct use of component interfaces.
# Contents

1 Introduction 6

2 Problem domain 7
   2.1 Communication protocols 7
   2.2 Interface specifications 7
      2.2.1 Syntactical specification 7
      2.2.2 Behavioral description 8
   2.3 Problem statement 9
      2.3.1 Stakeholders 9
      2.3.2 List of requirements 9
      2.3.3 Feasibility 10
   2.4 Considerations for languages and tools 10
      2.4.1 Specification language 10
      2.4.2 Specification tools 10
      2.4.3 Validation 10

3 Related work 13
   3.1 Assertional methods 13
   3.2 Process algebras 13
   3.3 Petri nets 13
   3.4 UML 13
   3.5 Domain-specific solutions 13
   3.6 Interface specification languages 14

4 Solution domain 15
   4.1 Assertional methods 15
   4.2 Process algebras 15
   4.3 Petri nets 16
   4.4 Statecharts 17
   4.5 Domain-specific languages 19

5 Solution architecture 20
   5.1 Base language 20
   5.2 Metamodeling toolkit 21
   5.3 Graphical editor 21
   5.4 Model execution 21
   5.5 Solution environment 21

6 Reflection statechart specification 23
   6.1 Statechart extensions 23
      6.1.1 Model properties 23
      6.1.2 Hierarchical states 24
      6.1.3 Transition extensions 25
      6.1.4 Embedded code 25
      6.1.5 Final state 26
      6.1.6 Choice operator 26
      6.1.7 History state 27
      6.1.8 Orthogonal regions 28
   6.2 Metamodel 29
      6.2.1 States 29
      6.2.2 Transitions 29
      6.2.3 History 29
      6.2.4 Assumptions 29
   6.3 Statechart semantics 30
      6.3.1 Definitions 30
      6.3.2 Model initialization 31
      6.3.3 Transition selection 32
6.3.4 Transition execution .............................................. 33

7 C++ code framework .............................................. 36
   7.1 Data structures .................................................. 37
      7.1.1 States and Transitions ...................................... 37
      7.1.2 Configurations ............................................... 37
      7.1.3 Exceptions .................................................. 37
      7.1.4 Reflection framework ....................................... 37
   7.2 Generated code .................................................. 37
      7.2.1 Model objects file ......................................... 38
      7.2.2 State configurations file ................................... 38
      7.2.3 Transition configurations file .............................. 39
      7.2.4 NewConfiguration file ..................................... 40
   7.3 Trace checker .................................................... 41

8 Conclusion ............................................................ 42

9 Future work ........................................................... 43

A C++ data structures ................................................ 44

B Xpand template ....................................................... 49

C C++ trace checker ................................................... 54
1 Introduction

In most large-scale software solutions, features are split up in components to improve modularity and maintainability. The communication between these components may be implicitly or explicitly restricted by a protocol. This protocol, also called the communication interface, consists of two parts. The first part is the syntax of the interface, which defines the messages that may be sent between the components. The second part contains the behavioral properties of the interface, which define how and when the messages may be sent.

For example, a very simple interface syntax may support the messages Request(int userId), Deny() and Accept(). A possible behavior for this interface is that a Request(1) should always be followed by an Accept(), but any request with userId ≠ 1 should be followed by a Deny().

In their portable navigation software, TomTom use a domain-specific language called Reflection to specify the syntax used in interfaces. A Reflection Interface Compiler is available to parse these specifications and translate them into various programming languages. The resulting code is used by components to define the necessary Reflection messages, parameters and types.

The semantic side of interfaces is described in Reflection Interface Documents (RIDs). These documents use English text, UML State Machine Diagrams (also known as statecharts) and UML Sequence Diagrams to describe the behavior of communicating components. The documents are meant to be easily understandable for component developers and testers. However, due to the informal nature of the specification, interface behavior is occasionally misinterpreted. As a result, components use subtly different protocols and the communication between them may fail.

Bugs introduced this way are very hard to detect. Since the behavioral specifications are written in natural language, a manual comparison is required between the specification and the actual implementation. Testers at TomTom do this by inspecting communication logs and flagging invalid traces as protocol violations.

TomTom desire to improve the current development and testing environment by specifying interface semantics in a more formal way. Semi-formal behavioral semantics would remove many of the unclarities in specifications. Additionally, tools could interpret the specification and automatically validate the correctness of a communication trace. The goal of this project is to provide a method to specify interface behavior and to set up a toolset that can automatically validate the behavior of communicating components.

Many different methods are available for the specification of behavior. To determine which method is the most appropriate, the problem domain and requirements are analyzed in Section 2. Next, the related work with respect to this problem is discussed in Section 3. The different possible languages and tools are compared in Section 4.

The chosen language for the specification of interface behavior is a domain-specific language based on statecharts. This language, named Reflection Statecharts, is formally defined by a metamodel in the Eclipse Modeling Framework. A separate graphical editor is used to draw models. A code generation template in Xpand transforms the models into a format suitable for trace checking.

The architecture of the solution environment is explained and motivated in Section 5. The used language is defined in Section 6. The methods used for code generation and trace checking are explained in Section 7. Finally, a conclusion is provided in Section 8 and possibilities for future work are discussed in Section 9.
2 Problem domain

The most common type of interface between components is the class interface. In such an interface, each component is a class that contains a number of public member variables and functions. When two classes communicate, they simply call each other’s public functions with a list of parameters and receive a result. The syntax of class members can often be defined separately in a header. Some languages combine the syntax definitions and implementations into a single file. Classes typically do not specify the behavior of their interfaces.

In centralized software, class interfaces are often sufficient. However, software in TomTom navigation systems is distributed in nature. Components may run on different processors, different hardware and may be written in different languages. For example, the TomTom navigation core can be embedded in automotive solutions with a third-party frontend. In many situations, components do not share memory. This means a channel is required for communication. Physically, a channel can be anything from a copper line to a Bluetooth connection. In order to ensure that the components interpret each other’s messages correctly, a protocol is required that describes both the syntax and the semantics of communication.

2.1 Communication protocols

Reflection messages are exchanged according to the Reflection protocol, which assumes the channel is lossless and asynchronous. Currently, the Reflection protocol makes use of the Transmission Control Protocol (TCP) and the Serial Line Internet Protocol (SLIP).

Since all Reflection interfaces deal with exactly two components, the components are usually referred to as the male and female peers. For interfaces with a client-host architecture, the client is usually the male peer and the host is the female peer.

In order to pass information across channels, messages and parameter objects are serialized by the sending peer and deserialized by the receiving peer. Reflection interfaces use a special serialization method called Tiny eXternal Data Representation, which is an extension of the eXternal Data Representation. XDR was developed by Sun Microsystems [27].

2.2 Interface specifications

A Reflection interface specification consists of two parts: a syntactical specification and a behavioral description.

2.2.1 Syntactical specification

Components communicating over a channel are not aware of each other’s types and functions. This is why syntax specifications are created for each interface. These specifications are written in the language Reflection, which is specified by a grammar in Extended Backus-Naur Form [2].

Types in Reflection are similar to those in C. Examples of types are integers, strings, booleans, fixed size arrays, lists of variable length and structures. Function calls have a name, a peer type (male or female) and a number of typed parameters. Note that there are no return types: if a message needs to be returned after a function call, it can only be done by calling a matching function in the opposite peer.

A fragment of an interface syntax specification is shown in Listing 1. This fragment defines a Request function for one peer and a Result function for the other. The parameter types specific for this interface are also included. The types int8 and utf8string are defined in the Reflection specification.
// Authentication result codes
enum unsigned int8 TiAuthenticationResult
{
    EiAuthenticationResultSuccess  = 0,
    EiAuthenticationResultUnexpectedMessage = 1,
    EiAuthenticationResultUnknownIdentity = 2,
    EiAuthenticationResultRejectedProof = 3,
    EiAuthenticationResultNotReady = 4
};

typedef struct
{
    utf8string tag< KiAuthenticationMaxTagLength >;
    utf8string value< KiAuthenticationMaxValueLength >;
} TiAuthenticationIdentityPair;

typedef TiAuthenticationIdentityPair
    TiAuthenticationIdentity< KiAuthenticationMaxPairs >;

interface iAuthentication = 67
{
    female:
        // Request authentication
        // aId: identification of Requester
        Request(TiAuthenticationIdentity aId) = 1;

    male:
        // Provide authentication result
        // aResult: success or reason of not accepting authentication
        Result(TiAuthenticationResult aResult) = 4;
};

Listing 1: A partial syntactical specification in Reflection for an authentication interface.

### 2.2.2 Behavioral description

The behavioral descriptions of interfaces are separate from their syntactical specifications. Although we use the term *interface behavior*, the interface itself is not an active process. The behavior of an interface refers to the *observable* behavior of the components *connected to this interface*. The only observable behavior at an interface is the content of exchanged messages and the order in which they were exchanged. Currently, the behavioral specifications are described by English text in combination with UML statecharts and UML sequence diagrams.

Most behavioral descriptions dictate not only what should happen in a regular communication, but also what should happen when an invalid message is received. For example, this could be a call that is done before certain information is ready or a call containing invalid parameters. Each interface may handle invalid messages differently. Therefore, error handling is part of the behavioral specification.

In the event of an invalid message, the peer sending the invalid call is violating the specification. However, the peer receiving the invalid call can still show correct behavior by returning the appropriate error response. As a consequence, behavior that is accepted for one peer may not be accepted for the other peer. To completely describe interface behavior, the specification treats both peers separately. This construct adds to the robustness of the system and allows testers to determine which peer is at fault.
2.3 Problem statement

Interface semantics are described in natural language so that they are easy to read and understand for component developers and testers. The downside is that the descriptions can also be misread or misunderstood, leading to implementations that exhibit unexpected behavior. Currently, these faulty components can only be detected manually by testers.

TomTom would like to improve their interface specifications by using a non-ambiguous behavioral specification method. A toolset can then use the created specifications to validate the behavior of peers automatically. The new specification method and testing tools should improve the quality of components’ implementations and reduce the testing effort.

The desired specification and validation method should satisfy certain constraints imposed by TomTom and the environment. Within these constraints, the choice of languages and tools is free. Many different options can be considered, and these options have to be compared to find the most suitable solution. We composed a list of requirements to clarify the constraints imposed on the language and the tools.

2.3.1 Stakeholders

The stakeholders for this project are divided into three groups. These groups are involved in the interface development cycle at TomTom. Their interests overlap for a large part, but some aspects are more important to a particular group.

When two components that were not previously connected require communication, a new specification is created by interface developers. Once the necessary types, functions and parameters have been agreed upon, a new RID (Reflection Interface Document) is created that includes the syntactical specification and the expected behavior of the peers.

Once the interface document has been created, the peers can be implemented by component developers. These implementations should adhere to the specified syntax and behavior. Unfortunately, this is not always the case. In order to verify the correctness of implementations, testers run the peer implementations. By using the packet sniffer tool Wireshark [23], a sequence of messages is extracted from the communication channel between peers. This trace is then manually inspected to find behavioral violations. If a violation is found, adjustments have to be made to either the component implementation or the interface specification.

The interface developers are mostly interested in having a user friendly tool for the creation of specifications. The component developers do not need to create specifications, but they do need to read them. For this group it is important that created specifications have an intuitive representation. Finally, the main concern of the testers is that the validation tools are reliable and that any occurring violations are easily traceable.

2.3.2 List of requirements

The following list of requirements has been composed in a discussion with the stakeholders at TomTom.

R5 The language must be sufficiently powerful to express Reflection data types.
R10 The language must support highly expressive user-defined functions.
R15 The tools must use a plain-text file format.
R20 The tool for designing interfaces must be easy and efficient in use, so that interface developers can focus on the design with minimal hindrance from the tool. A graphical user interface is highly preferred over a textual editor because it allows for a more intuitive design process.
R25 The designed specifications must have some representation that is easy to understand for component developers and testers, provided they have knowledge of the language.
R30 The toolset must be capable of checking the validity of an input trace with respect to the specification. The input trace is a sequence of Reflection messages extracted from a communication channel.
Preferably, the checker should also work in real-time. This means it can check the validity of individual input messages as they are being observed on the channel.

2.3.3 Feasibility

The requirements for the language and tools are very restrictive, making it uncertain if any one choice will satisfy all the constraints. Because of this, some of the requirements (R20, R25, R35) are soft, meaning they should be satisfied as much as possible by the chosen solution. Trade-offs can be made within preferences and qualitative properties such as usability. With these trade-offs it was considered feasible to find a sufficient solution to the problem.

2.4 Considerations for languages and tools

Before comparing the possible languages and toolsets, we will first take a look at the implications of the requirements. Formal software specification is becoming more and more popular, and the number of available languages and tools is astronomical. To narrow down the number of options, the capabilities of the language should be considered first. Since most graphical specification tools make use of models, the terms specification and model will be used interchangeably.

2.4.1 Specification language

The types used in Reflection communications are different for each interface. Some of the interfaces use dynamic size lists and custom structures. The chosen language should therefore at least support these constructs.

The behavior of interfaces is often dependent on the content of parameters. The more complicated the constraints are, the more expressive the language has to be. Since there is no limit to the complexity of interface behavior, the target language should ideally be as powerful as a generic programming language.

The risk of using such a powerful language is that the specification may resemble an implementation, while it is supposed to be on a higher level of abstraction. This is a necessary risk, since a less expressive language cannot capture the more complex behavioral properties. Exactly because these properties are the most complex, they are also the most likely cause of errors in the implementation. If the language is not powerful enough, the validation of interface behavior cannot be fully automated.

Ideally, the target language should support user defined functions and constraints, but separate them from the rest of the behavior in such a way that they do not obscure the specification. Fortunately, this kind of separation is present in many specification tools. These tools use a third-party function language to complement the specification. Popular languages to express functions in specifications are the Object Constraint Language [31], λ-calculus [1] or even regular programming languages.

2.4.2 Specification tools

The requirements for tools are soft, but should be satisfied sufficiently to come to a practical solution. Ease of use is a very important aspect which many formal specification tools lack. Often, the provided user interfaces (graphical or not) are not intuitive and require a lot of experience. Although the requirements do not enforce a graphical environment, it does seem to be the most suitable option. For interface designers, component developers and testers, a graphical representation will greatly reduce the effort required to write and read specifications.

Creating an new user interface for existing tools is a time consuming process and requires a lot of testing. On top of the existing requirements, this is not feasible. Therefore the focus will be primarily on tools that already have a usable interface.

2.4.3 Validation

Requirement R30 states that the toolset should be able to check whether or not observed behavior is correct. Model checking is a general term for checking the adherence of a model
to certain properties. These properties are typically written in languages like LTL, CTL* or μ-calculus [3, 36].

In model checking terms, a model \( m \) simulates a model \( m' \) if every action available in a state of \( m' \) can also be done in a similar state in \( m \). Trace checking is a special case of model checking where the model is checked for a specific sequence of actions. A trace check returns true if and only if the model can simulate the trace [7].

Traces are usually constructed by executing software with a specific set of parameters and logging the sequence of observed actions. In navigation software however, many components have ongoing processes that do not stop until the device is turned off. It is not always clear when a trace is complete, and violations of the specification may only occur after a large number of exchanged messages.

Once a violation is found, the behavior from that point on is undefined and the rest of the trace becomes unusable. Rather than checking full traces, it would be more efficient to check behavior in real-time. This way, there is no need to terminate the software in order to construct a trace and violations can be caught immediately when they occur.

This is the reason why real-time trace checking is introduced. In Figure 1, a monitor is attached to the communication channel between two peers. This monitor listens on the channel and passes observed messages on to the trace checker. The trace checker then validates the trace up until that point. For an ongoing sequence of \( n \) messages, this could be implemented by doing \( n \) trace checks, where the trace grows longer with every observed message. This is obviously inefficient as the input increases with every message.

Fortunately, many trace checkers use a recursive algorithm. This means that for a trace \( t \) and an action \( a \), the combined trace check \( \text{check}(\text{append}(t, a)) \) makes use of \( \text{check}(t) \). By saving the result of the previous trace check, a new action \( a \) can be validated without performing a complete trace check.

Although real-time trace checking is fairly simple on a technical level, the available tools may not allow the user to provide input in real-time. This is another aspect that should be taken into consideration when choosing a toolset.

Finally, it is worth noting that in deterministic models, each transition is triggered by a distinct message. For any state in the system, an incoming message uniquely determines the next state. If no transition can be taken, this means the message is not accepted by the model. Since the system only executes a single path, trace checking in a deterministic model can be done by simply executing it step by step. This opens up the possibility of using simulation tools that do not have full model checking capabilities.

Forcing specifications to be deterministic poses a restriction on the expressiveness of the language. To see whether or not this is an acceptable restriction, we have to look at the domain of interface specification.

Components that use interfaces make certain assumptions on the state of communication. Since an interface is not an active entity, there are no internal events that change the state of the interface. The state is only changed whenever a message is sent (or received) over the communication channel. Furthermore, it should be clear what the new state is when a specific message occurs. If the new state is ambiguous (due to non-determinism in the specification),

---

**Figure 1:** A setup for real-time monitoring and trace checking
components cannot determine whether or not their next message will be valid. Components would have to guess the state of the interface in order to send the correct message. This kind of non-determinism is undesirable, so the interface specification should be made to avoid these ambiguities.

In general, non-determinism in an interface specification should be kept to a minimum. Although the choice for deterministic models does somewhat restrict the language, it also has clear benefits. Therefore, it is considered an acceptable compromise.
3 Related work

Tools for the specification and validation of software are widely available on today’s market. The tools can be divided among several different specification methods. Each method has its own benefits and drawbacks, and researchers are continuously working to improve the power and applicability of these methods.

3.1 Assertional methods

Perhaps the most common approach is based on assertional methods. This approach uses preconditions and postconditions to reason about the input and output of a (partial) program. A study of assertional methods was presented in ‘Program correctness: on inductive assertion methods’ by J.C. King [22]. The limitation of assertional methods is that they focus only on individual components, but not the relation between them.

3.2 Process algebras

A different approach is taken by process algebras, which are more about the order of operations and the communication between processes. All process algebras describe communicating finite automata, which have a convenient graphical representation with nodes and transitions. The general aspects of process algebras are described in ‘Process algebra: an algebraic theory of concurrency’ by W. Fokkink [13].

The two most extensive toolsets for process algebras are mCRL2 by J.F. Groote et al. [15] and Lotos by T. Bolognesi and E. Brinksma [5]. These highly developed process algebras offer extensions to the generic formalism. They come with toolsets that combine formal semantics with high expressive power.

3.3 Petri nets

The main reason for the popularity of Petri nets is their compact graphical representation and their ability to express unbounded behavior. Since it is difficult to apply model checking techniques to models with unbounded behavior, most Petri net tools focus on simulation. An overview of the language was written by T. Murata in ‘Petri nets: properties, analysis and applications’ [30].

Colored Petri nets expand the formalism by adding types, variables and functions to the language. Most languages that use colored Petri nets are tightly coupled with a specific toolset. The most prominent toolset in this area is CPN tools [19], which has a graphical environment for drawing nets, syntax checking and (manual or automatic) simulation.

3.4 UML

In the industry, less formal tools are often used for specification purposes. The most common language in this area is the Unified Modeling Language [16]. UML contains a collection of languages such as class diagrams, sequence diagrams and statecharts that are convenient for the specification of object-oriented software. The downside of UML is that there are some unclarities in the semantics, making it difficult to develop model checking tools. The popularity of UML in the industry is largely due to its focus on practicality and usability. Efforts have been made for various software domains to analyze and simulate UML statecharts [10, 28, 29, 25, 8].

3.5 Domain-specific solutions

Of course, many other specification and validation tools exist. The tools that do not fall in any of the above categories are typically domain-specific tools using their own language.
One of these domain-specific languages is the recently developed Simple Language of Communicating Systems [38]. SLCO uses state machines with variables and constraints to model concurrent processes. The language is defined by a metamodel in the Eclipse Modeling Framework [35], and Xpand [9] is used to generate runnable code from these models. SLCO served as an inspiration and example for some of the decisions made later on in the project.

There are too many DSL-based tools to list them all, but several of them will be covered in Section 4.

### 3.6 Interface specification languages

Some tools have been created for the specific purpose of specifying interface behavior. The languages used by these tools are known as **Behavioral Interface Specification Languages**. An overview of BISLs was presented by J. Hatcliff et al. [18]. Since most of the BISLs were created for a specific problem domain, their expressiveness and level of abstraction vary.

An example of a language that relates to Reflection interfaces is Lime [21]. Lime is a process algebra resembling language within Java that is specifically meant for the specification of interface behavior. Specifications in Lime can be used in a run-time monitoring tool to check the correctness of implementations. Alternatives to Lime also exist for other programming languages, for example ANSI C [4].
Solution domain

To narrow down the solution domain, we will first take a look at the different specification methods. Little research has been done regarding the specification of interface behavior, so instead we will look at behavioral specification in general. Most of the behavioral specification tools are based on a pre-existing formalism like assertional methods, process algebras, Petri nets or statecharts. Some of the smaller tools define their own formalism and use that as a language.

The formalisms and the capabilities of tools are compared in this section. In Section 5, the most suitable approach will be selected and motivated.

For several methods, example models are included to illustrate how interface behavior can be described using these methods. The example interface throughout this document will be the *iAuthentication* interface. This interface was chosen because it is relatively simple and captures the most important aspects of the language.

The *iAuthentication* interface is used when one component needs to verify the identity of another component. The *male* peer in this interface may send an authentication *Request*. With this request it supplies an ‘identity’, which is a list of tags and values. Several rules apply to this identity, i.e. it should contain one tag named ‘root’ and duplicate tags are not allowed. The *female* peer can respond to a valid request by sending a *Challenge*. The male peer uses the parameters in the challenge message to construct a *Proof*. The female peer determines whether or not the proof is valid and sends the appropriate *Result* message.

### 4.1 Assertional methods

Assertional methods [22] use pre-conditions and post-conditions to describe the intended effect of a certain procedure. The goal of this approach is to ensure the correct behavior of implementations. For interface behavior, it is necessary to reason about the system state and the relations between operations. This cannot be done by only describing individual operations. For this reason, assertional methods are not applicable to interface behavior specification.

### 4.2 Process algebras

Contrary to assertional methods, process algebras focus on the order of actions and the relations between them. Because these are exactly the properties that should be expressed in interface specifications, process algebras seem to be a good choice.

Whether or not a process algebra is sufficiently expressive, depends largely on the language that is used. The two most prominent process algebra languages are mCRL2 [15] and *Lotos* [5]. They support message parameters, custom types and user-defined functions. All the types and expressions in Reflection should be expressible in this language. For mCRL2, an extensive toolset is available that can simulate, verify and transform models. This would be more than sufficient to do trace checking on models. Real-time trace checking is not a supported feature, but extending the toolkit may be feasible.

The downside of process algebra tools is that they are often moderately usable. No graphical user interface is currently available for creating mCRL2 or Lotos specifications. More importantly, the expressions in these languages are very low-level, making them difficult to read and even more difficult to write.

It will require a significant time investment from interface designers to get started with a process algebra tool. Once they are familiar with it, there would still be a large overhead caused by the impracticalities of the language. Although every property can technically be expressed, it is often inconvenient to do so.

An example of these impracticalities in mCRL2 is the lack of characters and strings. A character could be defined as an enumeration type with all 256 characters. Certain characters are reserved however, such as the bar ('|') or equals ('='') signs. These characters have to be converted to reserved names such as 'charbar' and 'charequals'. A string can be defined as a sequence of characters. String literals using quotation marks ('example literal') cannot
be used in mCRL2, so literals have to be expressed explicitly as a sequence of characters. Functions on strings (e.g. find, replace) will also have to be created. Defining all these types and functions will require significant time and effort that developers would rather spend on the actual interface specification.

Another problem is the lack of modularity within specifications. Since there are no include statements in mCRL2 or Lotos, the given type definitions have to be included separately in each interface file along with every other generic Reflection type. This would clutter the specifications and make them very hard to maintain.

Aside from the problems with model creation, there is also an issue for component developers and testers: there is no convenient representation for process algebra models with unbounded behavior. Currently, labeled transition systems (LTS) are the only supported visual representation for mCRL2. If a Reflection function uses an unbounded parameter type, such as a string or a float, no LTS can be generated until the actual value of these parameters is known. So the only way to read the full specification of an interface is to look at the textual representation. Even for someone experienced with process algebras, it can be hard to read a specification and deduce exactly what is meant. It is unrealistic to assume that component developers and testers will be able to efficiently read and understand interface specifications in mCRL2.

In conclusion, while process algebras suffice on a technical level, they come with too many practical problems to be usable as a specification language for Reflection interfaces.

4.3 Petri nets

Petri nets is a popular modeling language that can describe distributed systems. The method is a balance between expressive power and analyzability. The modeling objects for Petri nets are places and transitions. Places are typically visualized as ovals; transitions as rectangles. Arrows between places and transitions are called arcs.

Places are containers that can hold a number of objects called tokens. The collection of tokens in a place is called a marking, and the collection of markings in a model is the state. Transitions are active components that can consume tokens from places and generate new tokens in other places. Arcs determine from which places tokens are consumed and in which places tokens are generated. When a transition is activated (or fired), a token is consumed in each input place and a token is generated in each output place.

Petri nets by themselves are not very expressive because there is only one type of token: ‘unit’ or ‘void’. Colored Petri nets extend the formalism by assigning types and values to tokens, and adding constraint functions to transitions.

An important benefit of Petri nets is that they can deal with unbounded behavior. With respect to interface behavior, tokens are a very natural way to deal with messages. For example, a RequestRoute(A, B) message can be stored as a token of type RouteRequest which has properties source and destination. This way, every observed Reflection message could be added to the Petri net as a token. Transitions can regulate the behavior of the interface by consuming input messages and producing output messages (e.g. ResultRoute(R)).

Traces can be validated by checking if, after inserting a message token, a number of transitions can be fired that create the appropriate response token. This requires a reachability analysis on Petri nets. Although various algorithms are available for this purpose, research [11] has shown that the complexity of these algorithms is NP-Complete. For a practical application with potentially large models, this will not suffice.

As mentioned in Section 2, trace checking on deterministic models can be done by means of simulation. Many Petri net tools support manual and automatic simulation [30]. Suppose that a model is built in such a way that only a single transition can be fired for each step in the trace. The constraints on transitions ensure that invalid tokens cannot be processed. As long as the next message is accepted, a transition is available. By using the simulation tools on this model, trace checking can be performed.
An advantage of the Petri net formalism is the focus on visually presentable models. A large list of Petri net tools is offered by the university of Hamburg [33]. However, many of these tools have limited functionality. According to the requirements, the tool must support custom types and functions, and have a good user interface. Additionally, a simulator must be available that can accept input traces without manual intervention.

After applying these restrictions to the list of available tools, the only remaining toolset is CPN Tools. The other tools either lack streamlined user interfaces, are not compatible with popular operating systems, are not expressive enough or cannot be extended to provide the expected functionality. CPN Tools offers a high quality user interface and a Java framework to interact with the tools programmatically [41, 40]. The toolkit uses a powerful functional language (Standard ML) to express types, variables and functions. Other advantages are the automatic syntax checking and graphical layout features.

To provide a proof-of-concept, an example net was created for the iAuthentication interface (see Figure 2). All messages of the interface are present in the model as places and transitions. The resulting model is quite intuitive, so the method seems suitable. However, upon extending the model with error handlers, a problem was discovered. Error handlers define what should happen when an invalid message is received. For example, if a Challenge message is received before a Request, the receiving party should return an UnexpectedResult message and return to its initial state. Expressing this property in Petri nets is problematic.

In the example, the difference between a valid and an invalid communication is the presence of a Request token. Checking for the presence (or absence) of a token is called an empty check. In most tools, transitions can only consume tokens and return tokens. For empty checking, a special type of arc is required. This so-called inhibitor arc is not supported by most Petri net tools.

There are some ways to circumvent the empty check problem in CPN tools. However, these ways effectively double the number of places and arcs in the model. Additionally, it makes arc constraints much less readable.

An even more inconvenient property of error handlers is that they typically return the system to the initial state. Since the state is determined by the collection of all tokens in all places, going back to the initial state implies clearing every place and restoring the tokens that were initially there. In order to clear a place, an empty check is required. This means that, for every error handler in the specification, we need arcs from and to every state in the model. The resulting figure is a spaghetti monster that is impossible to unravel.

Since no efficient way could be found to specify error handlers in Petri nets, the language was disqualified.

4.4 Statecharts

Currently, Reflection interface specifications make use of statecharts to show the expected behavior of a component. One of the models for the iAuthentication interface can be seen in Figure 3. Statecharts have gained popularity in software engineering because of their ease of writing and understandability. They resemble labeled transition systems, adding a number of extensions that make the language more expressive and more convenient to use.

In general, statecharts are graphically focused and aim at providing an overview rather than fully describing a system’s behavior. That said, many tools do allow constraints and code within models to enhance the expressive power. With these constraints, statecharts could be used to describe the behavior of interfaces. Since most developers are already familiar with the language, it would be a very convenient and practical solution.

Statecharts do not have standardized semantics, leading to certain ambiguities and inconsistencies between different versions of the language [12, 24]. Parts of the language have been formalized in research studies. In particular, Harel [17] has defined semantics for his version of statecharts, which includes hierarchy, orthogonal regions and global variables. UML State Machines further extend Harel’s work by adding entry and exit code to states. These language extensions will be further discussed in Section 5.
Due to the popularity of the language, many tools are available to model statecharts. Because of their focus on practicality, the tools often have well designed graphical user interfaces. The purpose of the tools and the available language extensions vary greatly. Some tools are merely meant for drawing statecharts: the models have no purpose beyond their graphical representation. An example of such a tool is QP Modeler [26]. It has drawing tools for states, initial states and transitions. Any constraint or code written in the model is treated as plain text.
Other tools (such as VisSim [39]) go further and restrict the code in the model to a certain programming language. The code can be compiled and the model can be simulated. Some tools can also generate partial implementations from the models. Similar to Petri nets, if the original model is deterministic, these features could be used to perform trace checking.

The drawback of statechart tools is that they come in many different variants, each of which is tailored for a specific domain. Very few tools support all the extensions available in UML state machines, and even fewer have good simulation features. Advanced statechart tools are typically commercial and closed-source, meaning they cannot be easily extended. Although statecharts would be an appropriate way to model interface behavior, the tools available on the market do not match the requirements for our domain.

4.5 Domain-specific languages

Many of the specification tools that do not belong in the previous categories, focus on a particular problem domain. Most of these tools also come with their own domain-specific language. Examples are Declare, Promela, Poosl and TorX [32, 6, 14, 37]. Each of these languages has its own specific purpose and level of formality. While some of these languages are very expressive, they are not directly applicable to the specification of interface behavior. For example, Declare constrains the order of events but cannot deal with message parameters. Poosl focusses specifically on the communication between components, but it visualizes the physical channels rather than the exchanged messages. These and many other domain-specific tools were not developed with interface behavior in mind.

A few domain-specific languages were specifically created for the specification of interface behavior. For example, Lime [21] is an interface specification language that aims at describing the order of messages using a variation of process algebras. However, this language is entirely textual and does not have sufficient expressive power to constrain message parameters.

In order to make a domain-specific solution work, it should be created specifically for the specification of Reflection interfaces. Creating a new DSL and a corresponding toolset would relieve us from the limitations of existing tools. Furthermore, a DSL can easily be maintained when there is a need for new features.
5 Solution architecture

The most popular specification methods were covered in the previous section. Due to the large solution domain, there are undoubtedly some specification languages and toolkits that were left undiscussed. However, a common trend is visible. Existing languages are either too limited, tools are lacking in usability or they cannot be extended to provide the necessary features. The existing tools are deemed unfit for the specification of Reflection interfaces. To achieve a solution for this problem, we decided to create a new domain-specific language. The language is domain-specific in the sense that it supports only those features that were considered relevant for Reflection interfaces. Choices made in the development of the language were influenced by TomTom’s current software environment.

The development of a new language and corresponding tools can be time-consuming. To aid with the time and resource constraints in this project, the new language is to be based on one of the existing formalisms. By choosing an existing base language, syntax and (partial) semantics do not have to be written from scratch. Some of the existing tools or algorithms can be used to speed up the development of a toolset.

Several choices have to be made in the construction of a new environment. First of all, a base language is chosen for the DSL in Section 5.1. To express the new DSL, a metamodeling toolkit is chosen in Section 5.2. In Section 5.3, a graphical editor is presented which matches the chosen language. Next, the code generation step is discussed in Section 5.4. The resulting solution architecture is shown in Section 5.5.

5.1 Base language

Each specification language discussed in Section 4 has some limitations. Some of these limitations can be overcome in a DSL. For example, process algebras could be made more readable by adding a graphical language that lets developers draw a model rather than writing a specification. Petri nets could be extended with zero checking and error handlers. Both of these ideas add a new layer on top of the language that makes it more convenient to use.

The drawback of creating a new graphical layer is the amount of work involved. Research is currently in progress on how to create an efficient user interface for mCRL2. Creating a good user interface is a time-consuming task that often takes many years of iteration. For example, CPN tools has been in development since 2000. With the time constraints for this project, creating a full new user interface is not feasible.

Contrary to process algebras and Petri nets, statecharts are not limited by problems with user interfaces. Many graphical editors are available for this language. Some of these editors do not put any restrictions on the code and expressions used within models. We can thus choose an expression language that is suitable for describing Reflection types and constraints, and use this language in an existing drawing tool.

Statechart drawing tools are not aware of any semantics. New tools will have to be written to enable model execution and trace checking. This will be the main challenge in creating a DSL based on statecharts.

Statecharts are familiar for developers at TomTom and partial models have already been created for Reflection (see Figure 3). These models use an informal syntax that is convenient for the description of Reflection interfaces. The new DSL can borrow concepts from the existing models. For these reasons, statecharts were chosen as a base language. Models in the new DSL will be referred to as Reflection Statecharts or RSCs.

Several questions are raised in the creation of the DSL. For example, exactly which extensions to statecharts should be supported? Which language should be used for code within the model? These questions were discussed with the stakeholders, and the desired properties of the language are discussed in Section 6.
5.2 Metamodeling toolkit

A new modeling language is typically specified by a metamodel. Metamodels describe the model elements of a language as well as their properties and relations. The most popular metamodeling toolkits for graphical languages are the Eclipse Modeling Framework (EMF) and DSL Tools. EMF is integrated in the Eclipse IDE; DSL tools is integrated in Microsoft’s Visual Studio.

A full comparison of these two toolkits was done in 2007 by Turhan Özgür [42]. The toolkits were also tested for subjective qualities such as user experience. Both toolkits let users create new languages by drawing a metamodel. Furthermore, models can be converted to code by using an integrated templating language. The toolkits can also generate customizable graphical interfaces for created languages. The graphical editors depend on the toolkits and have some limitations. In both cases, they are quite slow and a high amount of effort is required to customize them.

In conclusion, the necessary features for RSCs are available in both toolkits. Although both Visual Studio and Eclipse are used within TomTom, developers are much more familiar with Eclipse due to its compatibility with other platforms. Because of previous experience with EMF and its use in a related study (SLCO [38]), it was chosen as a metamodeling toolkit for RSCs.

5.3 Graphical editor

Graphical editors created in Eclipse are quite limited and slow. For a better user experience, a third-party tool is used to create RSC models. After a brief study, a compatible and user-friendly tool for drawing RSCs was found in QP Modeler [26].

Both Eclipse and QP Modeler use an XML-based file format. These formats are not compatible, so it is not possible to create an Eclipse metamodel that describes a QP model. Therefore, a translation step is necessary.

Unfortunately, the XML-based format used in QP Modeler is not at all specified. The format can be reverse engineered to obtain a metamodel, but this is an inexact process. Future versions of QP Modeler may change the file format slightly, and each version will require a new metamodel. Because of the uncertain nature of the source language, we did not attempt to use formal methods for this transformation.

A Python script was written to transform QP models to objects. The same script then uses the objects to construct an output RSC model. Python was chosen for its ease of XML parsing and its flexible data formats. If TomTom desire to use a different graphical editor in the future, only the interpretation of input models to objects will have to be adjusted.

5.4 Model execution

Verifying properties in models with embedded code is difficult. However, checking traces with deterministic models is much less complex. In Section 2.4.3, it was explained why non-determinism is often an undesired property in interface behavior. Trace checking in deterministic statecharts is equivalent to stepwise execution or simulation.

In model-driven engineering, models are often first converted into a generic programming language. This way, an existing compiler can be used to execute the model. The same strategy is applied for Reflection Statecharts. A templating engine takes a model and a template as input and produces generated code. The most popular templating engine in EMF is the Xpand SDK. We will use Xpand to generate code from RSCs.

5.5 Solution environment

Figure 4 shows the model-to-code environment at three levels of abstraction. QP models are the input of the tool chain. These models use an unspecified version of XML. A Python script converts the QP models to EMF models that respect the metamodel for Reflection
Statecharts. Finally, an Xpand template is used to convert the EMF models to code.

The generated code files contain classes and objects that represent the original models. The generated code is combined with the RSC framework, which defines the underlying data types and the trace checker. Finally, TomTom’s Reflection framework provides type definitions for interfaces and deserialization methods for input messages. The complete setup for real-time trace checking is shown in Figure 5. Code generation and the created RSC framework are further explained in Section 7.

![Figure 4: The environment for model-to-code transformation](image)

![Figure 5: The trace checking environment](image)
6 Reflection statechart specification

Existing languages based on statecharts offer various extensions to make models more powerful. These extensions include hierarchy, embedded code and history states among others. Some of these extensions are useful for modeling interfaces, whereas others may be superfluous. Each of the extensions is discussed separately in Section 6.1 to determine whether or not it should be included in Reflection Statecharts.

Next, an EMF metamodel is presented and motivated in Section 6.2. The semantics of the language are discussed in Section 6.3.

6.1 Statechart extensions

The simplest version of a statechart resembles a labeled transition system. An example of a simple statechart is shown in Figure 6. The model contains named states and transitions. One of the states is marked as the initial state. Often, the initial state is drawn as a black dot pointing towards a state. This black dot is not a state: it is a state reference, also known as a pseudo-state.

![Figure 6: Example of a simple statechart](image)

Transitions in the model are directional connectors between two states. Each transition has an annotation. Several different interpretations of this annotation are available. Typically, the annotation is composed of an event and an action. Events are input messages and actions are output messages. Both events and actions are optional. When an event occurs in the environment, the system takes a transition from the current state that matches this event. Next, the action is executed and the system enters the target state.

Transitions without an event or action are internal transitions. Internal transitions should be avoided in interface specifications, since they may cause non-determinism.

With just these model elements, the language is not very powerful. Extensions were added to statecharts in order to enhance the expressive power and reduce the number of states required in models. Most of the popular extensions to statecharts are included in UML State Machines. Not all of these extensions have well specified semantics [12] and some of them are not feasible to implement. A subset of the UML extensions is chosen to be part of the DSL.

6.1.1 Model properties

A Reflection statechart has three properties that concern the model as a whole. These properties are assigned to the root element.

First, an RSC has a name that uniquely identifies it. Secondly, a model references the Reflection interface that it describes. With this information, the code generator can automatically include the type definitions for the interface.

Finally, the model should specify which peer of the interface is modeled (male or female). This allows us to determine whether an input message is an (incoming) event or an (outgoing) action. If a message is treated as an event by one peer, it is treated as an action by the other.
6.1.2 Hierarchical states

One of the most important extensions to statecharts is hierarchy. In a nested statechart, there is an implicit distinction between simple states and container states. Container states contain at least one substate, a number of transitions and an initial state.

The states in a hierarchical statechart can be viewed as a tree. The root of this tree is the model itself. The model has a number of child states. Each child in turn can have child states, and so on. An example statechart and its corresponding tree are shown in Figure 7. The leaves (end nodes) of this tree are simple states, whereas the other nodes are container states.

When the system in Figure 7 starts, it enters the initial state $s_1$. Note that the content of $s_1$ is a statechart in itself. The system recursively descends until it reaches a simple state. In this case, the system enters the initial substate $s_{11}$. Since this is a simple state, the startup process is completed and the system will remain idle until a transition is triggered.

Note that the system is now in two states at the same time: $s_1$ and $s_{11}$. This set of states is the current scope of the system. By definition, the scope is equivalent to a single path in the tree, going from the root node to a leaf node. Since this path is unique for every simple state, the scope is often omitted and the system is said to be in state $s_{11}$.

When an event token $a$ is received, the transition is triggered. First, the system leaves state $s_{11}$. Then the event token is consumed, a response message $b$ is sent, and the system enters $s_{12}$. While executing the transition, the system always remains in $s_1$. Hence, the scope of the transition is $s_1$. Once the system has entered the target state, it is once again idle, waiting for input. In this case, $s_{12}$ is a deadlock state so the system cannot continue.

Note that it is possible to use container states as the source of transitions. For example, say a transition was added from $s_1$ to $s_{12}$. For both $s_{11}$ and $s_{12}, s_1$ is in the scope, so the added transition is available. In general, transitions leaving from a container state are available to all simple states within it.

Similarly, it is also possible to use container states as the target of transitions. When such a transition is executed, the system first enters the transition’s target state and then recursively descends through initial states until it reaches a simple state. For example, if we take a transition from $s_{12}$ to $s_1$, the system first exits $s_{12}$ and then enters the initial state $s_{11}$.

When simple states share certain properties or transitions, they can be grouped into a container state. This modular approach reduces the number of transitions and makes models more intuitive. Therefore, hierarchical states are included in Reflection statecharts.

Figure 7: Example of a hierarchical statechart (left) and its tree representation (right)
6.1.3 Transition extensions

Statecharts often allow for parameters in events and actions. An example of a parameterized event is \texttt{Divide}(x,y). When a \texttt{Divide} message is received, the parameters from the input message are bound to variables \( x \) and \( y \). These are local variables that are only available in the context of the transition.

As a response to the \texttt{Divide} event, we could have an action \texttt{Result}(x/y). Note that action parameters are expressions, not variables. The action parameters determine what an appropriate response is for a certain input event.

Transitions are often extended with guards. Guards are boolean expressions that can make use of event parameters. A transition can only be taken if its guard evaluates to \texttt{true}. The event in combination with a guard is known as a trigger. A useful guard in the previous example would be \( y \neq 0 \). When an input message attempts to divide by zero, the guard evaluates to false and the message is rejected.

Finally, instructions can be added after a transition action; these instructions are executed after the action is processed.

Each of the mentioned transition extensions are available in most statechart-based tools. Parameters, guards and code on transitions make state changes much more flexible. They are also necessary for Reflection interfaces and are therefore included in the DSL. By convention, the properties of a transition are written as a string with separator characters. The complete syntax for a transition is shown in Equation 1. Any of the four parts in this equation may be omitted, but there must be at least one event or one action to ensure determinism.

\[
\text{trigger} \quad \text{event} \quad \text{condition} \quad \text{action} \quad \text{code} \\
\text{Message(variables)} \ [\text{expression}] / \text{Message(expressions)} \ \{\text{statements}\}
\] (1)

6.1.4 Embedded code

To further increase the expressiveness of statecharts, code fragments are introduced into the model. States are extended with \texttt{entry code} and \texttt{exit code}; transitions already have \texttt{transition code}. The entry code of a state is executed whenever the system enters that state. Similarly, the exit code is executed when a transition causes the system to leave the state. Transition code is executed after an action is completed, but before the system enters the target state.

Code embedded in RSCs should be capable of expressing complex constraints and functions. To provide as much expressiveness as possible, we use a general purpose programming language. Most of TomTom’s software related to Reflection is written in C++, so this will be the language of choice.

Embedded code is of limited use without the presence of variables. Other statechart tools let users declare a number of \texttt{global variables}. Transition guards and action parameters may use these variables to further constrain the behavior.

The introduction of variables causes a significant change in the interpretation of statecharts. When a model is in a certain state, the behavior may be different depending on the values of global variables. Thus, states in the model no longer represent a unique situation in the system.

One of the advantages of hierarchical statecharts is their modular approach. Features can be grouped into container states that describe a specific part of the system. With the use of global variables, this advantage is lost. Even if the system leaves a state, information from that state may still be available through global variables. The reason why tools only use global variables has to do with the ease of implementation.

To provide better modularity in models, Reflection statecharts allow local variables in states and event parameters. The scope of a state or transition determines which variables
can be accessed. This way, separate parts of the system cannot interfere with each other. Additionally, variable names can be re-used in various transitions and states.

Reflection interfaces often use functions in constraints. For example, several transitions in the `iAuthentication` interface will check the correctness of an `identity` parameter. States in an RSC should be able to define functions as well as variables.

By now, the following properties have been assigned to a state.

- A name
- Entry code
- Exit code
- Functions
- Variables

This signature is somewhat familiar to that of a class in a programming language. Entry code and exit code can be interpreted as the constructor and destructor. We will make use of these similarities for model execution in Section 7.

6.1.5 Final state

In certain systems, there is a difference between successful termination and a deadlock. For this reason, the final state was introduced in UML. A final state is semantically equivalent to a simple state without outgoing transitions.

For Reflection interfaces, no distinction is made between successful termination and a deadlock. Therefore, it is not necessary to introduce this component in the DSL.

6.1.6 Choice operator

The choice operator is a forking construct in UML where a transition with a single event splits into multiple paths with mutually exclusive guards. This notation can further reduce the number of transitions in the model, but does introduce a new element into the language. Instead, a choice can also be expressed by using separate transitions for each guard. Since the added value of this operator is low, it will not be included in Reflection Statecharts.
6.1.7 History state

UML state machines also define history pseudo-states. When a container state contains a history state, this means its last visited substate will be stored. Transitions may target this history state to resume the most recent substate. Additionally, UML also defines recursive history states. When a transition targets the recursive history of a container state \( s \), this means the system will resume the most recent descendant of \( s \).

History states may be used in situations where a part of the behavior can be temporarily disabled, but has to be restored later. There are debates whether or not history states really belong in the language, since they capture dynamic information about a state when the system is not currently in that state. While this is true, there are situations where it can express behavior that is otherwise very inconvenient to model. An example of such a situation is given in Figure 8. If the history state were omitted, the unpause function cannot return to the correct state unless a separate pause state is defined for each other state.

Since history states can drastically reduce the complexity of some models, they are included in Reflection Statecharts.

![Figure 8: Example of a statechart using a history state](image)
6.1.8 Orthogonal regions

Finally, some statechart tools support orthogonal regions within container states. In an orthogonal region, two statecharts operate simultaneously.

According to the semantics defined by Harel [17], if both parts of an orthogonal region can execute the incoming event, two transitions are executed. Furthermore, an action in one region may trigger an event in the other. In this definition, the behavior of regions is no longer orthogonal, since they depend on one another. Despite this fact, they are still referred to as orthogonal regions by both Harel and UML.

The concept of interdependant ‘orthogonal’ regions raises many semantical questions, such as what happens when two transitions evoke each other, or what happens when two parallel transitions access the same variable. Harel addressed these issues by determining from an input event a consistent set of enabled transitions. Consistent here means that the transitions in the set do not interfere with each other. The chosen set of transitions is executed.

The choices made by Harel solve some of the semantic problems, but cause non-determinism and unbounded sequences of exchanged messages. The combination of history states and orthogonal states yields additional ambiguities [12]. Because of the counter-intuitive definition of orthogonal regions and the issues with their semantics, they will not be included in Reflection Statecharts.

Figure 9: Example of orthogonal regions
6.2 Metamodel

In Figure 10, an EMF metamodel is presented for Reflection Statecharts.

6.2.1 States

In the metamodel, the root object is an RSC. Since the root of the model shares many properties with a state, an RSC is defined as a specialization of a state. Each state can contain a number of child states, transitions and an initial pseudo-state. Every transition is owned by its surrounding state.

6.2.2 Transitions

Transitions have a source and a target state. Note that initial states are modeled as references, and history states are modeled as boolean properties. Thus, these pseudo-states cannot be the source or target of a transition.

Transitions have an event message, a condition, an action message and some code. Both types of messages come with a number of parameters. In order to provide unique names for transitions, transitions also have a hashed identifier \( id \).

6.2.3 History

History states are pseudo-states. Unlike an initial state, the target of a history state is dynamic. Upon initialization, it points to the same state as the initial state. Therefore, no references are necessary in the metamodel. The properties of a state determine whether or not it has a (recursive) history state. When a transition targets a container state, the property \( targetsHistory \) determines whether or not the transition should activate the target state’s history.

6.2.4 Assumptions

The aim of the metamodel is to provide a simple and sufficient representation for RSCs. It does however allow models that are not well-formed. For example, the metamodel allows transitions that use the root as their source and target. To restrict ourselves to valid statecharts, the following assumptions are made.

- A model has only one \( ReflectionStatechart \), which is the root.
• The root cannot be the source or target of transitions.
• All states except the root have a parent. The root does not.
• An initialSubState always points to a child state.
• The state that owns a transition is the least common ancestor of the source and target states of that transition.
• A transition can only have targetsHistory set to true if its target state has history.

The metamodel is an intermediate step in the tool chain to provide a more formal basis for code generation. In practice, developers will use the graphical editor to create models. The graphical editor, in this case QP Modeler, is only capable of drawing valid statecharts. Provided that the conversion from QP models to RSCs is correct, we can be assured that the resulting RSC is a valid statechart. The above assumptions will therefore always hold.

6.3 Statechart semantics

The structure and semantics of UML statecharts were researched by Y. Jin et al. [20] These semantical definitions were used as a starting point for defining RSCs.

The primary difference between Reflection Statecharts and UML is the interpretation of events and actions in the model. UML statecharts typically model a component implementation, where events are incoming messages from the environment and actions are responses that can always be executed. RSCs operate from the interface point of view, where actions are also incoming messages: the only difference between an event and an action is the direction of the message on the channel. Transitions with both an event and an action need to process two input messages in order to complete their execution.

A second difference between UML and RSCs is the presence of hierarchical variables and functions. These variables and functions can be used in state entry code, exit code and transition code. Due to the complexity of C++, we cannot realistically provide semantics for embedded code. For instance, it cannot be computed whether or not code terminates [34]. It is also impossible to determine if any part of the code causes non-deterministic behavior.

Since no guarantees can be made that embedded code is correct, the created toolset does not parse or in any way interpret embedded code. It is up to the interface designers to ensure that embedded code does not threaten the validity of the model. Only when the final transformation to C++ code is complete, the embedded code is compiled and executed along with the generated code. If errors are discovered at this point, developers can manually adjust the code to find a solution for the problem. When a solution is found, the code in the source model can be corrected.

Even though embedded code is not formally defined, we can make certain assumptions on their effect. For example, a C++ expression in a condition should always yield a boolean value. By defining a condition as a boolean value, semantics can be provided without considering how the value was calculated. Similarly, the validity of parameters is also treated as a boolean value. By abstracting from expressions, variables and functions are no longer relevant to the behavior of statecharts and full semantics can be provided.

First, definitions will be given to fully describe a Reflection Statechart. Next, the necessary operational semantics are given that determine the resulting state after executing a transition.

6.3.1 Definitions

In order to define the structure of a Reflection Statechart, the following set types are introduced.

• $M$ contains all possible message names (for events and actions). It also includes the empty message $\emptyset$.
• $P$ contains all lists of event parameters $P_e$ and action parameters $P_a$.
• $C$ contains all lists of executable statements (for code blocks).
• $E$ contains all possible boolean expressions (for conditions).
The structure of a Reflection Statechart is defined by a tuple <\( \Sigma, \Theta, S, T > \), where:

- \( \Sigma \) is the set of states in the model. The root of the model is also a state.
- \( \Theta \) is the set of transitions in the model.
- \( S \) is a group of functions:
  - \( S^{entry} : \Sigma \rightarrow \mathbb{C} \) and \( S^{exit} : \Sigma \rightarrow \mathbb{C} \). These functions map a state to its entry and exit code.
  - \( S^{init} : \Theta \rightarrow (\Sigma \cup \emptyset) \), mapping a state to the initial state within it, or \( \emptyset \) if no initial substate is defined. Every container state must have exactly one initial substate.
  - \( S^h : \Sigma \rightarrow \mathbb{B} \) and \( S^{rh} : \Sigma \rightarrow \mathbb{B} \), mapping a state to boolean values that indicate whether or not the state has a (recursive) history state. In a valid model, it is assumed that \( S^{rh}(s) \Rightarrow S^h(s) \) for every state \( s \in \Sigma \).
  - \( S^{parent} : \Sigma \rightarrow (\Sigma \cup \emptyset) \), mapping a state to its parent, or \( \emptyset \) in case of the root.
- \( T \) is a group of functions that map a transition to its properties:
  - \( T^{src} : \Theta \rightarrow \Sigma \) and \( T^{trg} : \Theta \rightarrow \Sigma \) map a transition to its source and target state.
  - \( T^{lea} : \Theta \rightarrow \Sigma \) maps a transition to its innermost containing state: the least common ancestor of the source and target.
  - \( T^{evt} : \Theta \rightarrow M \) and \( T^{act} : \Theta \rightarrow M \) map a transition to its event and action message.
  - \( T^{pars} : \Theta \rightarrow \mathbb{P} \) and \( T^{apars} : \Theta \rightarrow \mathbb{P} \) map a transition to its event and action parameters.
  - \( T^{cond} : \Theta \rightarrow \mathbb{E} \) maps a transition to its guard condition.
  - \( T^{code} : \Theta \rightarrow \mathbb{C} \) maps a transition to its associated code block.
  - \( T^{hist} : \Theta \rightarrow \mathbb{B} \) determines whether or not the target of a transition is a history state. In a valid model, each transition \( t \in \Theta \) has \( T^{hist}(t) \Rightarrow S^h(T^{rh}(t)) \).

### 6.3.2 Model initialization

Before messages can be processed, the statechart must be initialized. In addition to the model definitions, several operations are used:

- The system stores a scope consisting of currently active states. Whenever a state is entered or exited, that state is added to or removed from the scope. Since states are always entered and exited in order of depth, the scope is defined as \( \Delta : \text{Stack}(\Sigma) \), initially \( \Delta = \emptyset \).
- The operation \( \text{push} : \text{Stack}(\Sigma) \times \Sigma \rightarrow \emptyset \) adds a state to a state stack.
- The operation \( \text{pop} : \text{Stack}(\Sigma) \rightarrow \emptyset \) removes the top value from a state stack.
- The operation \( \text{top} : \text{Stack}(\Sigma) \rightarrow \Sigma \) returns the top value of a state stack. When the system is idle, that is, not executing a transition, \( \text{top}(\Delta) \) is the current simple state of the system.
- The function \( \text{evaluate} : \mathbb{E} \rightarrow \mathbb{B} \) evaluates the validity of a condition.
- The function \( \text{checkparams} : \mathbb{P} \times \mathbb{P} \rightarrow \mathbb{B} \) checks whether or not the observed parameters match those described by the model. The first list of parameters contains observed values; the second list of parameters contains variables or expressions defined by the model.
- The function \( \text{execute} : \mathbb{C} \rightarrow \emptyset \) executes state or transition code.
- The function \( \text{historysubstate} : \Sigma \rightarrow (\Sigma \cup \emptyset) \) is a dynamic function from a state to its most recently visited substate. Initially for each state \( s \), \( \text{historysubstate}(s) = S^{init}(s) \).
- The \( \text{historysubstate} \) function can be updated using \( \text{updatehistory} : (\Sigma \times \Sigma) \rightarrow \emptyset \). It holds that \( \text{updatehistory}(s, t) \Rightarrow (\text{historysubstate}(s) = t) \).

The initialization operation is shown in Algorithms 1, 2, 3 and 4. Upon initialization, the system enters the root state of the model. Upon state entry, the state is added to the scope and the entry code is executed. The \( \text{historysubstate} \) function is also updated to store the
newly visited state in the parent. For the root state, the parent is $\emptyset$ so the history does not need to be updated.

Once the system has entered the root state, it descends through the model by entering initial substates. When no more initial substate is available, the system has reached a simple state and the initialization terminates.

**Algorithm 1** InitializeStatechart(root: $\Sigma$)

EnterState(root)
DescendInitial()

**Algorithm 2** EnterState(s : $\Sigma$)

push($\Delta$, s)
execute($S^{entry}(s)$)

if $S^{parent}(s) \neq \emptyset$ then
  updatehistory($S^{parent}(s)$, s)
end if

**Algorithm 3** LeaveState()

execute($S^{exit}(\text{top}(\Delta))$)
pop($\Delta$)

6.3.3 Transition selection

In UML, statecharts represent active components that take an input event and produce an output action. This is different for Reflection statecharts, which represent the behavior of a component from an interface point of view. For a channel monitor, events and actions are both input messages. A message is an action for the sending peer and an event for the receiving peer. For example, a Request is typically an action for male peers and an event for female peers.

When a Reflection statechart receives input, there are three types of transitions that may be enabled. The first case is a transition that has an event, but no action. Once an event $(m, p) : M \times P_e$ occurs, the system determines which transitions are enabled as follows: $\text{enabled}_{\text{evt}}(m, p) \equiv \{ t \in \Theta | T^{src}(t) \in \Delta \land T^{evt}(t) = m \land \text{checkparams}(p, T^{pars}(t)) \land \text{evaluate}(T^{end}(t)) \}$. A valid input event should enable exactly one transition. If $\text{enabled}_{\text{evt}}$ is empty, the observed event is not supported by the model. Thus, the component that is being modeled has violated the protocol. If $\text{enabled}_{\text{evt}}$ has more than one element, this means the model is non-deterministic, which violates one of the assumptions. In each case, an error can be returned.

The second case is a transition has both an event and an action. In this case, the system can be in an intermediate state when the event of a transition has been processed, but the corresponding action $(m, p) \in M \times P_a$ has not yet been observed. The system must wait for this action to occur before it can proceed to the target state. In this case, $\text{enabled}_{\text{intermediate}}(m, p) \equiv T^{act}(t) = m \land \text{checkparams}(p, T^{pars}(t))$.

In the third case, the transition has an action without a preceding event. At least one such transition is required, since the first outgoing message in a conversation is not preceded by an event. The set of transitions enabled by action is then defined as $\text{enabled}_{\text{act}}(m, p) \equiv \{ t \in \Theta | T^{src}(t) \in \Delta \land T^{act}(t) = m \land \text{checkparams}(p, T^{pars}(t)) \land \text{evaluate}(T^{end}) \}$.

32 Guido Josquin
Algorithm 4 DescendInitial()

\[
\text{while } S^{\text{init}}(\text{top}(\Delta)) \neq \emptyset \text{ do} \\
\quad \text{EnterState}(S^{\text{init}}(\text{top}(\Delta))) \\
\text{end while}
\]

6.3.4 Transition execution

After initialization, the system is in a certain scope $\Delta$, awaiting an event or an action. When an event or action occurs, a transition is selected based on the type of message and the set of enabled transitions. Pseudo-code for this step is presented in Algorithm 5.

Algorithm 5 SelectTransition(msg: $M$, parameters: $P$, isEvent: $B$)

\[
\text{if } \text{isEvent} \text{ then} \\
\quad \text{enabled} := \text{enabled}_{\text{evt}}(\text{msg}, \text{parameters}) \\
\text{else} \\
\quad \text{enabled} := \text{enabled}_{\text{act}}(\text{msg}, \text{parameters}) \\
\text{end if} \\
\text{if } \text{size}(\text{enabled}) \neq 1 \text{ then} \\
\quad \text{return } \text{error}() \\
\text{else} \\
\quad t := \text{singleton}(\text{enabled}) \\
\quad \text{ExecuteTransition}(t, \text{isEvent}) \\
\text{end if}
\]

Once a transition $t$ is selected, the system will start executing it. To find the correct order of execution for a transition, the model is best viewed as a tree. Note that any two states in a tree are connected by a unique path. When a transition is executed, the system will traverse the path from the current state to a new state. States on the ascending path will be removed from the scope and states on the descending path will be removed. The topmost state in the path is the least common ancestor (LCA) state of $T^{\text{src}}(t)$ and $T^{\text{trg}}(t)$. This state ($T^{\text{lca}}(t)$) will remain in the scope during execution.

![Figure 11](image_url)

Figure 11: A tree representation of a statechart, highlighting the transition $s_3 \rightarrow s_7$.

An example state tree is given in Figure 11. Before executing the transition $s_3 \rightarrow s_7$, the system is in simple state $s = s_5$. This means that the current scope $\Delta = [\text{root}, s_2, s_3, s_5]$. 

Guido Josquin
During execution, the system ascends through the tree, removing states from the scope. On the descending path, new states are added to the scope. After execution, the system will end up in a new simple state $s' = s_9$ with scope $\Delta = \{\text{root}, s_2, s_4, s_7, s_9\}$.

The path from $s$ (the current simple state) to $s'$ (the resulting simple state) consists of four segments. The first segment is the ascending path from $s$ to $T_{src}(t)$. The second segment is the ascending path from $T_{src}(t)$ to $T_{lca}(t)$. The third path descends from $T_{lca}(t)$ to $T_{trg}(t)$. Finally, the last path descends from $T_{trg}(t)$ to $s'$. This path is determined by following either initial states or history states.

The first two segments of the path are traversed using Algorithm 6. This operation simply leaves states until the given state is reached. The third segment of the path is traversed by Algorithms 7 and 8. These operations follow the descending path from the LCA to the transition’s target. For the final segment of the path, a differentiation is made between transitions that target a history state and ‘normal’ transitions. Algorithms 4 and 9 show the operations for traversing the path from the transition’s target to the resulting state.

Algorithm 10 shows the operation for executing the transition as a whole. First, the system ascends until the LCA state is reached. If a transition has both an event and an action, the system must then wait for the action to occur. Finally, the system executes the transition code and descends to the resulting state.

**Algorithm 6 AscendTo($s : \Sigma$)**

```plaintext
while top($\Delta$) \(\neq s\) do
    LeaveState()
end while
```

**Algorithm 7 DescendTo($s : \Sigma$)**

```plaintext
while top($\Delta$) \(\neq s\) do
    next := NextOnDescendingPath(top($\Delta$), s)
    EnterState(next)
end while
```

**Algorithm 8 NextOnDescendingPath($x : \Sigma$, $y : \Sigma$)**

```plaintext
while $S_{parent}(y) \neq x$ do
    $y := S_{parent}(y)$
end while
return $y$
```
Algorithm 9 DescendHistory()

if $S^h$ (top($\Delta$)) then
  while historysubstate(top($\Delta$)) $\neq \emptyset$ do
    EnterState(historysubstate(top($\Delta$)))
  end while
else
  EnterState(historysubstate(top($\Delta$)))
  DescendInitial()
end if

Algorithm 10 ExecuteTransition($t : \Theta, isEvent : \mathbb{B}$)

AscendTo$(T^{src}(t))$
AscendTo$(T^{loc}(t))$
if isEvent $\land T^{act} \neq \emptyset$ then
  Wait for an action $(m, p) : M \times P_a$
  if $\neg enabled_{intermediate}(m, p)$ then
    return error()
  end if
end if
execute$(T^{code}(t))$
DescendTo$(T^{trag}(t))$
if $T^{hist}(t)$ then
  DescendHistory()
else
  DescendInitial()
end if
7 C++ code framework

Once a QP model has been converted to a valid RSC, we can generate code from the model. Two different methods were suggested for the creation of a trace checker.

The first method is a direct conversion from an RSC to a trace checker. This approach is direct because there is no intermediate data structure: the flow of execution is captured in a large number of if-then-else statements. Child states and transitions can be represented by nested code blocks. Whenever a message is observed, the guards in the if-then-else ladder will determine which state is chosen and which transition is executed.

There are several problems with the direct approach. First of all, functions in C++ cannot be declared within if-statements. Additionally, generated code would be very long and nearly impossible to read or debug. While the direct approach may be applicable for simpler modeling languages, it is not appropriate here.

The chosen method is an indirect conversion, where an RSC is transformed into C++ objects and classes. The objects have a predefined data structure, which serves as the ‘metamodel’ in C++. Since the structure of the generated code is fixed, the trace checker can be generalized to work for any RSC. This method was chosen because it is more modular and therefore easier to maintain.

As shown in Figure 12, the trace checker consists of a static part and a generated part. The static part consists of the Reflection framework and the RSC framework. The Reflection framework is provided by TomTom; it defines the data types used in interfaces and converts input data to a C++ compatible format. The RSC framework defines the trace checker and the underlying data types for generated code. The trace checker uses the input messages from the Reflection framework to command and manipulate the generated code objects.

This code generation method was inspired by SLCO [38], a state-based modeling language developed at the Eindhoven University of Technology. The fundamental difference between code generation in SLCO and statecharts is the presence of hierarchical states and localized code.

First, the used data structures are explained in Section 7.1. The structure of generated code is shown in Section 7.2. Finally, the features and operations of the checker framework will be described in Section 7.3.
7.1 Data structures

Several classes are defined in the C++ framework that serve as base types for generated objects. Each data structure has a header file for declarations and a .cpp file for definitions. The listings for these classes are included in Appendix A. The class ReflectionStatechart describes the root of a model. The model can be asked to execute a step by calling DoTransition with an input message. A ReflectionStatechart can also list all transitions available from the current scope.

7.1.1 States and Transitions

The data structures for states and transitions are separated into a static and a dynamic part. The static datatypes define the structure of a statechart as described by the EMF metamodel. The structure of the model does not change during the execution. States can calculate their own depth in the hierarchy and calculate a path to a target state. Transitions can calculate their least common ancestor, which is the state that contains them.

7.1.2 Configurations

The dynamic datatypes are called configurations. A StateConfiguration contains the code embedded in a certain state. This includes variable declarations, functions, entry code and exit code. This information is dynamic because the values stored in variables may change during the execution. Whenever the system leaves a state, the dynamic information within that state is lost. When the system enters a state, its variables are again declared and initialized.

A StateConfiguration is essentially an instantiation of a State where all variables are assigned a value. However, a C++ object instance cannot declare its own variables and functions. Therefore, a StateConfiguration must be stored as a class. Upon the entry of a state, an object of type StateConfiguration is created to store the evaluations of variables. The constructor of the class will then call the entry function. By default, this function is empty. When the state is exited, the destructor of the StateConfiguration object invokes the exit function.

TransitionConfigurations are similar to StateConfigurations, since they declare a number of variables in event parameters. These variables are assigned a value upon execution of the transition. A TransitionConfiguration is only accessible while attempting to execute a transition. Objects of type TransitionConfiguration can check the validity of a transition guard, evaluate and check the action parameters and execute the transition code.

7.1.3 Exceptions

Several exception types are defined for the trace checker, so that it can give meaningful feedback when a model fails to process an event or action.

7.1.4 Reflection framework

A ‘Reflection Interface Compiler’ is provided by TomTom that converts the Reflection Interface Documents to C++ headers. These headers are included by the Reflection framework to extract messages and parameters from a binary input stream. When a message occurs on the channel, the Reflection framework deserializes the data and calls the appropriate event handler. From here, the message can be passed to the trace checker.

The compiled Reflection headers are also included by the generated code, so that models have access to interface-specific types.

7.2 Generated code

To generate code, we use the templating language Xpand. This language is integrated in EMF and is capable of creating text output files based on a model.

Xpand templates can use expressions to define output files, sub-templates, loops and if-statements. Commands in Xpand are surrounded by French quotes (‘ and ’). Expressions
in a command can access objects and properties of the input model. Any text outside of commands will appear as-is in the output file.

The Xpand template used to generate code is included in Appendix B. The input of the transformation is a single RSC model. The output consists of four files. A generated code sample is included for each file.

### 7.2.1 Model objects file

The first file contains a class that extends `ReflectionStatechart`. This class creates new objects that contain the static information of the model. This includes states, transitions and references between them. It also creates a trace checker object, which will be discussed in Section 7.3.

```cpp
class iAuthentication_Female : public ReflectionStatechart {
    private:
        State* root;
        State* Not_Authenticated;
        Transition* trans2571664;
        Transition* trans2571676;
        Transition* trans2571688;

    public:
        // Constructor
        iAuthentication_Female () {
            isMalePeer = false;
            root = new State("root");
            root->hasHistory = false;
            root->hasRecursiveHistory = false;
            Not_Authenticated = new State("Not_Authenticated");
            Not_Authenticated->hasHistory = false;
            Not_Authenticated->hasRecursiveHistory = false;
            trans2571564 = new Transition(2571564, Not_Authenticated, "Abort", 
                                            ":reason", ":", ":", ":", ":", false);
            Not_Authenticated->transitions.push_back(trans2571564);
            trans2571576 = new Transition(2571576, Not_Authenticated, "Proof", 
                                            ":salt", ":proof", ":", ":Result", 
                                            ":(EiAuthenticationResultUnexpectedMessage)" , ":", false);
            Not_Authenticated->transitions.push_back(trans2571576);
            trans2571588 = new Transition(2571588, Not_Authenticated, "Request", 
                                            "(someId)", 
                                            ":root.isValidateIdentity(someId)", ":Result", 
                                            ":(EiAuthenticationResultUnknownIdentity)" , ":", false);
            Not_Authenticated->transitions.push_back(trans2571588);
            root->children.push_back(Not_Authenticated);
            Not_Authenticated->parent = root;
            root->initialSubState = Not_Authenticated;

            _scc = new StatechartChecker(iAuthentication, this->isMalePeer);
            _scc->Initialize ();

            // End of fragment
        }

Listing 2: Fragment of iAuthentication_Female.cpp
```

### 7.2.2 State configurations file

The second file defines a configuration class for each state in the model. These classes extend the abstract `StateConfiguration` class. In a specific `StateConfiguration`, the following elements are defined.

- A list of references to the ancestors of the state.
- The embedded code of the state taken from the input model.
• A constructor which initializes the references to all ancestor states and calls the entry code.
• A destructor which calls the exit code.

```cpp
// Generated class to contain the variables and functions of state Not_Authenticated
class State_iAuthenticationFemale_Not_Authenticated_Configuration : public StateConfiguration{
public:
    // List of ancestors (order irrelevant)
    State_iAuthenticationFemale_root_Configuration &root;

    // State code
    State_iAuthenticationFemale_Not_Authenticated_Configuration (State* s, std::vector<StateConfiguration*> &scope) :
        root(*((State_iAuthenticationFemale_root_Configuration*)scope[0])),
        this->state = s;
    this->entry();

    ~State_iAuthenticationFemale_Not_Authenticated_Configuration () {
        this->exit();
    }
};
```

Listing 3: Fragment of iAuthentication_Female_StateConfigurations.cpp

7.2.3 Transition configurations file

The third file defines a TransitionConfiguration for each transition in the input model. The structure of this file is very similar to that of StateConfigurations. Instead of state code, it declares the variables that are present in the transition event. These variables are initialized in the constructor with a list of input parameters. The transition code, condition expression and action expressions are included in separate functions that can be called by the trace checker.

```cpp
// Generated class to contain the variables and functions of transition Not_Authenticated
class Transition_iAuthenticationFemale_2312084_Configuration : public TransitionConfiguration{
public:
    // List of ancestors (order irrelevant)
    State_iAuthenticationFemale_Not_Authenticated_Configuration &Not_Authenticated;
    State_iAuthenticationFemale_root_Configuration &root;

    // Event variables
    TiAuthenticationIdentity& someId;

    // Constructor
    Transition_iAuthenticationFemale_2312084_Configuration (Transition* t, std::vector<StateConfiguration*> &scope, std::vector<void*> &params) :
        Not_Authenticated(*((State_iAuthenticationFemale_Not_Authenticated_Configuration*)scope[1])),
        root(*((State_iAuthenticationFemale_root_Configuration*)scope[0])),
        someId(*((TiAuthenticationIdentity*)params[0])) {
        transition = t;
    }

    // Destructor
    ~Transition_iAuthenticationFemale_2312084_Configuration () {
    }
};
```

Guido Josquin
// Executes a transition in this state
void ExecuteTransitionCode()
{
    // This code is attached to the transition in the model
}

// Evaluates a transition condition in this state
bool CheckTransitionEnabled()
{
    // This code is attached to the transition in the model
    return !rootstate.isValidIdentity(someId);
}

// Checks if the supplied parameters correspond to the parameters of a transition action
bool CheckActionParameters(std::vector<void*> &params)
{
    bool parsCorrect = true;
    // This code is attached to the transition in the model
    if (params.size() == 1)
    {
        parsCorrect = parsCorrect && (EiAuthenticationResultUnknownIdentity==((TiAuthenticationResult*)(params[0])));
    }
    else
    {
        parsCorrect = false;
    }
    return parsCorrect;
}

Listing 4: Fragment of iAuthentication_Female_TransitionConfigurations.cpp

7.2.4 NewConfiguration file

The last file contains code for the initialization of new configurations. Since the toolset must be capable of using code generated from multiple interfaces, the generated classes have an interface- and peer-specific name. In order to link the static trace checker to the dynamically named generated files, a NewConfiguration function was created in StatechartChecker.cpp that creates a new configuration depending on the name of an interface, its peer type and either the name of a state or the hash identifier of a transition.

//BEGIN_STATE_CONFIGURATION_INITIALIZERS:iAuthentication_Female
else if(root->name == "iAuthentication" && !isMalePeer)
{
    if(false){}
}
else if(s->name == "iAuthentication")
{
    sconf = new State_iAuthenticationFemale_iAuthentication_Configuration(s, scope);
}
else if(s->name == "femaleStatechart")
{
    sconf = new State_iAuthenticationFemale_femaleStatechart_Configuration(s, scope);
}
else if(s->name == "Not_Authenticated")
{
    sconf = new State_iAuthenticationFemale_Not_Authenticated_Configuration(s, scope);
}
//END_STATE_CONFIGURATION_INITIALIZERS

//BEGIN_TRANSITION_CONFIGURATION_INITIALIZERS:iAuthentication_Female
else if(root->name == "iAuthentication" && !isMalePeer)
{
    if(false){}
}
else if(t->id == 2571564)
{
Listing 5: Fragment of StatechartChecker_iAuthentication_Female_NewConfiguration.txt

An Xpand template template can only use one input model. In order to create a single function that links all states and transitions to their respective configurations, the generated NewConfiguration files from all interfaces have to be combined into a single file. A Python script was created to do this. Markers were added as comments to identify the appropriate sections.

7.3 Trace checker

Each ReflectionStatechart creates a StatechartChecker object. This is a generic trace checker for RSCs. The functions in StatechartChecker.cpp were implemented according to the operational semantics in Section 6.3.

The checker initializes the model with the Initialize function. A message can be passed to the checker using the DoTransition function. If the system is in an intermediate state (between an event and an action), the current transition is stored until the appropriate action is received. For debugging purposes, the checker also has a ListTransitions function that lists all transitions available from the current scope.
8 Conclusion

TomTom use explicit interfaces to regulate communication between software components. Although the syntax for these interfaces is formally defined, the semantical properties were not.

To improve the quality of software, a new interface specification method was presented to model interface behavior. In the creation of this method, many choices were made regarding the use of languages and tools. A list of requirements was set and followed. Although existing solutions seemed promising, it became clear that a domain-specific approach was necessary to meet the requirements.

The created language ‘Reflection Statecharts’ is based on UML Statecharts. The language is a balance between the semantical properties of a formal model and the expressiveness of code. A tool chain was implemented to provide a modular and maintainable solution. First, users can draw RSCs in a graphical modeling tool. These models are then transformed into EMF-compatible models. Next, an Xpand template transforms the models into executable C++ code. Finally, the generated code is inserted in a trace checker framework that automatically validates communications between components.

The toolset has been tested successfully in a realistic setup, using TomTom’s authentication interface. The tools are ready to be deployed in a live environment. At this point, we cannot yet provide evidence of how effective the new method is. We expect it to be a valuable addition to the development and testing environment at TomTom, by making the behavior of interfaces clear for component developers and by reducing the effort required to find protocol violations.
9 Future work

To fully exploit the possibilities of the new specification method, Reflection Statechart models should be created for each of TomTom’s interfaces. Existing and new implementations can then be tested with these models to find mistakes in either the specification or the implementation.

For the creation of new interfaces, TomTom may desire to add additional features to the toolset. For example, future versions of RSCs could support orthogonal regions. In this project, there was insufficient time to fully explore the semantical properties of orthogonal regions in statecharts.

The ease of model creation could be further improved by allowing transitions with multiple events or actions in disjunction. This way, several transitions with the same source and target states could be combined into one. With model transformation techniques in EMF, this extension could be implemented without making changes to the RSC framework.

An other interesting research topic is support for limited non-determinism in models. For example, the current trace checker cannot deterministically choose between two transitions that have the same trigger but different actions. However, the model could be transformed by adding an in-between state with a single incoming transition and two outgoing transitions with different actions. The new model would be deterministic. Alternatively, exhaustive simulation techniques could be used to explore all possible paths for a given trace.

Finally, RSCs could be used in the future for purposes other than testing. For example, component stubs could be generated from models. These stubs could help component developers to implement communication correctly.
A C++ data structures

The following listings show the data structures used by generated code and the C++ checker.

```
#ifndef _REFLECTIONSTATECHART_H
#define _REFLECTIONSTATECHART_H

#include <vector>
#include <string>

// Forward declaration
class StatechartChecker;

// Class to represent a Statechart
class ReflectionStatechart{
protected:
    StatechartChecker* _scc;
    bool _isMalePeer;
public:
    // Executes a transition based on a message and parameters
    std::string DoTransition(std::string msg, std::vector<void*> &params, bool peer);
    // Returns all available transitions from the current scope
    std::vector<std::string> ListOptions();
};
#endif
```

Listing 6: ReflectionStatechart.h

```
#include "ReflectionStatechart.h"
#include "StatechartChecker.h"

// Statechart functions
std::string ReflectionStatechart::DoTransition(std::string msg, std::vector<void*> &params, bool peer){
    // The message is an event if the statechart is not the source of the message
    // The message is an action if the statechart is the source of the message
    return this->_scc->DoTransition(msg, params, peer!=this->_isMalePeer);
}
std::vector<std::string> ReflectionStatechart::ListOptions(){
    std::vector<std::string> v;
    return this->_scc->ListTransitions();
}
```

Listing 7: ReflectionStatechart.cpp

```
#ifndef _STATE_H
#define _STATE_H

#include <vector>
#include <string>

// Forward declaration
class Transition;

// Class to represent the model element State
class State{
public:
    // Array containing the sub-states of this state
    std::vector<State*> children;
    // Array containing those transitions where the scope (least common ancestor) is this state
    std::vector<Transition*> transitions;
    // A reference to the parent state, null if this is the root State* parent;
};
```

Listing 8: State.h
// A reference to the initial substate, which is one of the children of this state
State* initialSubState;
// The state name
std::string name;

// Indicates if this is a history state
bool hasHistory;
bool hasRecursiveHistory;
// A reference to the substate the history state points to
State* historySubState;

// Constructor
State(std::string statename);

// Calculates the depth of this state, where root = 0
int GetDepth();

// Gets the path from this state to a descendant state
// The returned value is a vector of child indices, as follows:
// [1][0] indicates State* this->children[1]->children[0]
std::vector<int> GetPath(State* target);

#endif

Listing 8: State.h

#include "State.h"

// State functions
State::State(std::string statename){
    this->name = statename;
    this->parent = NULL;
    this->initialSubState = NULL;
    this->hasHistory = false;
    this->hasRecursiveHistory = false;
    this->historySubState = NULL;
}
int State::GetDepth(){
    return parent?parent->GetDepth() + 1:0;
}
std::vector<int> State::GetPath(State* target){
    std::vector<int> vi;
    State* tmp = target;
    while(tmp != this){
        for(unsigned int i=0; i<tmp->parent->children.size(); i++){
            if(tmp->parent->children[i] == tmp){
                vi.push_back(i);
                break;
            }
        }
        tmp = tmp->parent;
    }
    return vi;
}

Listing 9: State.cpp

#ifndef TRANSITION_H
#define TRANSITION_H

#include <string>

// Forward declaration of State
class State;

#include "State.h"

Listing 8: State.h

#include "State.h"

// State functions
State::State(std::string statename){
    this->name = statename;
    this->parent = NULL;
    this->initialSubState = NULL;
    this->hasHistory = false;
    this->hasRecursiveHistory = false;
    this->historySubState = NULL;
}
int State::GetDepth(){
    return parent?parent->GetDepth() + 1:0;
}
std::vector<int> State::GetPath(State* target){
    std::vector<int> vi;
    State* tmp = target;
    while(tmp != this){
        for(unsigned int i=0; i<tmp->parent->children.size(); i++){
            if(tmp->parent->children[i] == tmp){
                vi.push_back(i);
                break;
            }
        }
        tmp = tmp->parent;
    }
    return vi;
}

Listing 9: State.cpp

#ifndef TRANSITION_H
#define TRANSITION_H

#include <string>

// Forward declaration of State
class State;

#include "State.h"

Listing 8: State.h

#include "State.h"

// State functions
State::State(std::string statename){
    this->name = statename;
    this->parent = NULL;
    this->initialSubState = NULL;
    this->hasHistory = false;
    this->hasRecursiveHistory = false;
    this->historySubState = NULL;
}
int State::GetDepth(){
    return parent?parent->GetDepth() + 1:0;
}
std::vector<int> State::GetPath(State* target){
    std::vector<int> vi;
    State* tmp = target;
    while(tmp != this){
        for(unsigned int i=0; i<tmp->parent->children.size(); i++){
            if(tmp->parent->children[i] == tmp){
                vi.push_back(i);
                break;
            }
        }
        tmp = tmp->parent;
    }
    return vi;
}

Listing 9: State.cpp

#ifndef TRANSITION_H
#define TRANSITION_H

#include <string>

// Forward declaration of State
class State;

#include "State.h"

Listing 8: State.h

#include "State.h"

// State functions
State::State(std::string statename){
    this->name = statename;
    this->parent = NULL;
    this->initialSubState = NULL;
    this->hasHistory = false;
    this->hasRecursiveHistory = false;
    this->historySubState = NULL;
}
int State::GetDepth(){
    return parent?parent->GetDepth() + 1:0;
}
std::vector<int> State::GetPath(State* target){
    std::vector<int> vi;
    State* tmp = target;
    while(tmp != this){
        for(unsigned int i=0; i<tmp->parent->children.size(); i++){
            if(tmp->parent->children[i] == tmp){
                vi.push_back(i);
                break;
            }
        }
        tmp = tmp->parent;
    }
    return vi;
}

Listing 9: State.cpp

#ifndef TRANSITION_H
#define TRANSITION_H

#include <string>

// Forward declaration of State
class State;

#include "State.h"

Listing 8: State.h

#include "State.h"

// State functions
State::State(std::string statename){
    this->name = statename;
    this->parent = NULL;
    this->initialSubState = NULL;
    this->hasHistory = false;
    this->hasRecursiveHistory = false;
    this->historySubState = NULL;
}
int State::GetDepth(){
    return parent?parent->GetDepth() + 1:0;
}
std::vector<int> State::GetPath(State* target){
    std::vector<int> vi;
    State* tmp = target;
    while(tmp != this){
        for(unsigned int i=0; i<tmp->parent->children.size(); i++){
            if(tmp->parent->children[i] == tmp){
                vi.push_back(i);
                break;
            }
        }
        tmp = tmp->parent;
    }
    return vi;
}

Listing 9: State.cpp

#ifndef TRANSITION_H
#define TRANSITION_H

#include <string>

// Forward declaration of State
class State;

#include "State.h"

Listing 8: State.h

#include "State.h"

// State functions
State::State(std::string statename){
    this->name = statename;
    this->parent = NULL;
    this->initialSubState = NULL;
    this->hasHistory = false;
    this->hasRecursiveHistory = false;
    this->historySubState = NULL;
}
int State::GetDepth(){
    return parent?parent->GetDepth() + 1:0;
}
std::vector<int> State::GetPath(State* target){
    std::vector<int> vi;
    State* tmp = target;
    while(tmp != this){
        for(unsigned int i=0; i<tmp->parent->children.size(); i++){
            if(tmp->parent->children[i] == tmp){
                vi.push_back(i);
                break;
            }
        }
        tmp = tmp->parent;
    }
    return vi;
}

Listing 9: State.cpp

#ifndef TRANSITION_H
#define TRANSITION_H

#include <string>

// Forward declaration of State
class State;

#include "State.h"

Listing 8: State.h

#include "State.h"

// State functions
State::State(std::string statename){
    this->name = statename;
    this->parent = NULL;
    this->initialSubState = NULL;
    this->hasHistory = false;
    this->hasRecursiveHistory = false;
    this->historySubState = NULL;
}
int State::GetDepth(){
    return parent?parent->GetDepth() + 1:0;
}
std::vector<int> State::GetPath(State* target){
    std::vector<int> vi;
    State* tmp = target;
    while(tmp != this){
        for(unsigned int i=0; i<tmp->parent->children.size(); i++){
            if(tmp->parent->children[i] == tmp){
                vi.push_back(i);
                break;
            }
        }
        tmp = tmp->parent;
    }
    return vi;
}

Listing 9: State.cpp

#ifndef TRANSITION_H
#define TRANSITION_H

#include <string>

// Forward declaration of State
class State;

#include "State.h"

Listing 8: State.h

#include "State.h"

// State functions
State::State(std::string statename){
    this->name = statename;
    this->parent = NULL;
    this->initialSubState = NULL;
    this->hasHistory = false;
    this->hasRecursiveHistory = false;
    this->historySubState = NULL;
}
int State::GetDepth(){
    return parent?parent->GetDepth() + 1:0;
}
std::vector<int> State::GetPath(State* target){
    std::vector<int> vi;
    State* tmp = target;
    while(tmp != this){
        for(unsigned int i=0; i<tmp->parent->children.size(); i++){
            if(tmp->parent->children[i] == tmp){
                vi.push_back(i);
                break;
            }
        }
        tmp = tmp->parent;
    }
    return vi;
}

Listing 9: State.cpp

#ifndef TRANSITION_H
#define TRANSITION_H

#include <string>

// Forward declaration of State
class State;

#include "State.h"

Listing 8: State.h

#include "State.h"

// State functions
State::State(std::string statename){
    this->name = statename;
    this->parent = NULL;
    this->initialSubState = NULL;
    this->hasHistory = false;
    this->hasRecursiveHistory = false;
    this->historySubState = NULL;
}
int State::GetDepth(){
    return parent?parent->GetDepth() + 1:0;
}
std::vector<int> State::GetPath(State* target){
    std::vector<int> vi;
    State* tmp = target;
    while(tmp != this){
        for(unsigned int i=0; i<tmp->parent->children.size(); i++){
            if(tmp->parent->children[i] == tmp){
                vi.push_back(i);
                break;
            }
        }
        tmp = tmp->parent;
    }
    return vi;
}

Listing 9: State.cpp

#ifndef TRANSITION_H
#define TRANSITION_H

#include <string>

// Forward declaration of State
class State;

#include "State.h"

Listing 8: State.h

#include "State.h"

// State functions
State::State(std::string statename){
    this->name = statename;
    this->parent = NULL;
    this->initialSubState = NULL;
    this->hasHistory = false;
    this->hasRecursiveHistory = false;
    this->historySubState = NULL;
}
int State::GetDepth(){
    return parent?parent->GetDepth() + 1:0;
}
std::vector<int> State::GetPath(State* target){
    std::vector<int> vi;
    State* tmp = target;
    while(tmp != this){
        for(unsigned int i=0; i<tmp->parent->children.size(); i++){
            if(tmp->parent->children[i] == tmp){
                vi.push_back(i);
                break;
            }
        }
        tmp = tmp->parent;
    }
    return vi;
}

Listing 9: State.cpp

#ifndef TRANSITION_H
#define TRANSITION_H

#include <string>

// Forward declaration of State
class State;
// Class to represent the model element Transition
class Transition{
    private:
        State* LCA(State* source, State* target);
    public:
        std::string id;
        State* source;
        State* target;
        std::string _event;
        std::string _action;
        bool targetsHistory;
        // The following strings are solely for the purpose of printing/viewing
        std::string _eventparams;
        std::string _actionparams;
        std::string _condition;
        std::string _code;

    // Constructor with a source and a target state
    Transition(std::string tId , State* src , State* trg , std::string evt ,
               std::string evtParams , std::string condition , std::string act , std::string actParams ,
               std::string transCode , bool trgHistory);
    // Returns the least common ancestor of the source and target
    // This LCA (and all states above) define the scope for this transition
    State* GetLCA();
};

Listing 10: Transition.h

#include "Transition.h"
#include "State.h"

// Transition functions
Transition::Transition(std::string tId , State* src , State* trg , std::string evt ,
               std::string evtParams , std::string condition , std::string act , std::string actParams ,
               std::string transCode , bool trgHistory){
    this->id = tId;
    this->source = src;
    this->target = trg;
    this->_event = evt;
    this->_eventparams = evtParams;
    this->_condition = condition;
    this->_actionparams = actParams;
    this->_action = act;
    this->_code = transCode;
    this->TargetsHistory = trgHistory;
}

State* Transition::LCA(State* source, State* target){
    if(source == target){
        return source;
    }
    else{
        int ds = source->GetDepth();
        int dt = target->GetDepth();
        if(ds > dt)
            return this->LCA(source->parent, target);
        else if (dt > ds)
            return this->LCA(source, target->parent);
        else
            return this->LCA(source->parent, target->parent);
    }
    State* LCA(source->parent, target->parent);
}

Listing 11: Transition.cpp
Listing 11: Transition.cpp

```cpp
#ifndef _STATECONFIGURATION_H
#define _STATECONFIGURATION_H

#include <vector>
#include <string>

// Forward declarations
class State;

// An abstract base class representing a StateConfiguration
class StateConfiguration{
protected:
    std::string typeDescription;
public:
    State* state;
    void entry();
    void exit();
};
#endif
```

Listing 12: StateConfiguration.h

```cpp
#include "StateConfiguration.h"

// An abstract base class representing a StateConfiguration
void StateConfiguration::entry(){
}

void StateConfiguration::exit(){
}
```

Listing 13: StateConfiguration.cpp

```cpp
#ifndef _TRANSITIONCONFIGURATION_H
#define _TRANSITIONCONFIGURATION_H

#include <vector>

// Forward declarations
class Transition;

// An abstract base class representing a StateConfiguration
class TransitionConfiguration{
public:
    Transition* transition;
    virtual void ExecuteTransitionCode();
    virtual bool CheckTransitionEnabled();
    virtual bool CheckActionParameters(std::vector<void*> &params);
};
#endif
```

Listing 14: TransitionConfiguration.h

```cpp
#include "TransitionConfiguration.h"

// An abstract base class representing a TransitionConfiguration
```
void TransitionConfiguration :: ExecuteTransitionCode () {
}

bool TransitionConfiguration :: CheckTransitionEnabled () {
    return false;
}

bool TransitionConfiguration :: CheckActionParameters (std :: vector < void * > & params) {
    return false;
}

Listing 15: TransitionConfiguration.cpp

/***********************************************************/
#include <stdexcept>

class InvalidActionMessage : public std :: runtime_error {
public:
    InvalidActionMessage (std :: string expectedAction) :
        std :: runtime_error( "Input message does not match the expected action " +
            expectedAction ) {};
};

class MultipleAcceptingTransitions : public std :: runtime_error {
public:
    MultipleAcceptingTransitions () :
        std :: runtime_error( "Multiple transitions match the input. Non-determinism is not supported." ) {};
};

class NoAcceptingTransitions : public std :: runtime_error {
public:
    NoAcceptingTransitions () :
        std :: runtime_error( "Could not find an accepting transition!" ) {};
};

class InvalidEventParameters : public std :: runtime_error {
public:
    InvalidEventParameters () :
        std :: runtime_error( "The number of event parameters is incorrect." ) {};
};

class InvalidActionParameters : public std :: runtime_error {
public:
    InvalidActionParameters () :
        std :: runtime_error( "The entered action parameters do not satisfy the constraints." ) {};
};

class ExpectedAction : public std :: runtime_error {
public:
    ExpectedAction () :
        std :: runtime_error( "Expected an action, got an event. Please check the message type." ) {};
};

Listing 16: StatechartExceptions.h
B Xpand template

The following listings show the Xpand template used for code generation. Each listing concerns one of the generated files. Whitespace was added to the template for a more readable layout.

```cpp
«FILE name="-+(isMale?"Male":"Female")-+.cpp"»
2  /* Generated CPP code from input StateChart "this.name", */
* which is the model for the "isMale?"Male":"Female"" peer of Reflection ←
interface "this.interfaceName" */
* This file contains the classes that declare and define pieces of code ←
specific for this Statechart */
* Code generation template v1.1
* (C) TomTom 2012
* State and Transition objects contain the static information from the model.
* This information does not change during the execution.
*/
#include "../../StatechartChecker.h"
#include "../../ReflectionStatechart.h"
#include "../../State.h"
#include "../../Transition.h"

class «name+"_(isMale?"Male":"Female")»: public ReflectionStatechart {
private:

22 /* Constructor
26 «name+"_(isMale?"Male":"Female")]() {
154  isMalePeer = «(isMale?"true":"false")»;
258  «EXPAND generateModelObjectDefinitions FOR this»
262  _scc = new StatechartChecker("this.name", this→isMalePeer);
266  _scc→Initialize();
270 };
30 «ENDFILE»

«DEFINE generateModelObjectDeclarations FOR State»
State* «this.name»;
«EXPAND generateModelObjectDeclarations FOREACH this.children»
«FOREACH this.ownedTransitions AS t»
Transition* trans«t.id»;
«ENDFOREACH»
«ENDDEFINE»

«DEFINE generateModelObjectDefinitions FOR State»
«this.name» = new State("this.names");
«this.name»→hasHistory = «IF this.hasHistory»true«ELSE»false«ENDIF»;
«this.name»→hasRecursiveHistory = «IF «this.hasRecursiveHistory»true«ELSE»false«ENDIF»;
«FOREACH this.children AS s»
«EXPAND generateModelObjectDefinitions FOR s»
«this.name»→children.push_back("s.name");
«s.name»→parent = "this.name";
«ENDFOREACH»
«FOREACH this.ownedTransitions AS t»
trans«t.id» = new Transition("t.id", "t.source.name", "t.target.name", "t.events", "(
    «FOREACH t.eventParameters AS p SEPARATOR ', ' »p.value» «ENDFOREACH»
  )", "t.conditions", "t.actions", "(
    «FOREACH t.actionParameters AS p SEPARATOR ', ' »p.value» «ENDFOREACH»
  )", "t.code", "if t.targetsHistory» true «ELSE» false «ENDIF»);
«this.name»→transitions.push_back(trans«t.id»);
«ENDFOREACH»
```
Listing 17: Template for the generated objects

```cpp
#include "../../State.h"
#include "../../StateConfiguration.h"
#include "../../Reflection/"this.interfaceName".h"

#define generateStateConfigurations(rscname, depth) FOR State-

// Generated class to contain the variables and functions of state "this.name"
class State"_rscname_"_this.name"_Configuration : public StateConfiguration{
    public:
        // List of ancestors (order irrelevant)
        #if this.parent!=null
            #define generateAncestorConfigurationDeclarationList(rscname) FOR this.parent
            #ifdef
        #endif
        // State code
        #define code
        #code
        // Constructor
        State"_rscname_"_this.name"_Configuration (State* s, std::vector<StateConfiguration*>&scope)
            #if this.parent!=null
                #define generateAncestorConfigurationDefinitionList(depth-1, rscname) FOR this.parent
                #endif
        #endif
        this->state = s;
        this->entry();
    }
    "State"_rscname_"_this.name"_Configuration (){
        this->exit();
    }
};

#define generateStateConfigurations(rscname, depth+1) FOREACH this.children

```

Guido Josquin
Each transition has a TransitionConfiguration class which defines the accessible variables in that transition. TransitionConfiguration instances have specific evaluations of these variables and may change in the transition code.

```
#include "../../Transition.h"
#include "../../TransitionConfiguration.h"
#include "name+"+(isMale?"Male":"Female")+_StateConfigurations.cpp"

#define generateTransitionConfigurations (String rscname, int depth) FOR State this
   EXPAND generateTransitionConfiguration (rscname, this, depth) FOREACH this.ownedTransitions
   EXPAND generateTransitionConfigurations (rscname, depth+1) FOREACH this.children
ENDDEFINE

#define generateTransitionConfiguration (String rscname, State s, int depth) FOR Transition this
   // Generated class to contain the variables and functions of transition «this.source.name»−−»this.target.name»
   class Transition"+_rscname_+"+this.id+_Configuration : public TransitionConfiguration{
   public:
      // List of ancestors (order irrelevant)
      EXPAND generateAncestorConfigurationDeclarationList (rscname) FOR s
      // Event variables
      FOREACH this.eventParameters AS par
         par.type=IF par.type.endsWith("&")&ENDIF par.value;
      ENDFOREACH
      // Constructor
      Transition"+_rscname_+"+this.id+_Configuration (Transition t, std::vector<StateConfiguration*> &scope, std::vector<void*> &params) :
         FOREACH this.eventParameters AS par ITERATOR j,
            IF par.type.endsWith("&")
               par.value=(*(par.type.replaceFirst("&", "*"))(params[«j.counter0»]))
            ELSE
               par.value=*(par.type*)(params[«j.counter0»])
            ENDIF
         ENDFOREACH
      transition = t;
   }
   // Destructor
   Transition"+_rscname_+"+this.id+_Configuration () {
   }
   // Executes a transition in this state
   void ExecuteTransitionCode()
   {
      // This code is attached to the transition in the model
      this.code
   }
   // Evaluates a transition condition in this state
```
bool CheckTransitionEnabled(){
    //This code is attached to the transition in the model
    if (this.condition!=null && this.condition!="")
        return this.condition;
    return true;
}

// Checks if the supplied parameters correspond to the parameters of a transition action
bool CheckActionParameters(std::vector<void*> &params){
    bool parsCorrect = true;
    if (params.size()==this.actionParameters.size()){
        for (int i=0; i<params.size(); i++)
            if (params[i]!=this.actionParameters[i])
                parsCorrect = false;
    } else{
        parsCorrect = false;
    }
    return parsCorrect;
}

/*FILE "StatechartChecker_"+name+"_"+(isMale?"Male":"Female") + "_NewConfiguration.txt"*/
//NOTE:
//This file is part of StatechartChecker_NewConfiguration.cpp
//Since Xpand can only generate code for one model at a time,
//the results from different peers and interfaces are later combined into a single file.

//BEGIN_STATE_CONFIGURATION_INITIALIZERS:"name":""+(isMale?"Male":"Female")
} else if (root->name == ""+name+""+this.name _Configuration &"+this.name+";
    "EXPAND generateAncestorConfigurationDeclarationDeclarationList ("+name+"+this.name _Configuration &"+this.name+";
FOR this.parent;
"ENDIF"
"EXPAND generateAncestorConfigurationDefinitionDeclarationList ("+this.name+"+this.name _Configuration &"+this.name+";
FOR this.parent;
"ENDIF"
"ENDDEFINE"
"EXPAND generateAncestorConfigurationDefinitionDeclarationList ("+this.name+"+this.name _Configuration &"+this.name+";
FOR this.parent;
"ENDIF"
"ENDDEFINE"

Listing 19: Template for transition configurations
Listing 20: Template for the creation of new configurations
C++ trace checker

```cpp
#ifndef __STATECHARTCHECKER_H
#define __STATECHARTCHECKER_H

#include <vector>
#include <string>

// Forward declarations
class StateConfiguration;
class TransitionConfiguration;
class State;
class Transition;

// The checker
class StatechartChecker {

private:
    // The scope is the vector of all state instances that are available in a state
    std::vector<StateConfiguration*> scope;
    // The stateChart root state
    State* rootNode;
    // The type of peer
    bool isMalePeer;
    // The current half-way transition
    // (If the last accepted transition had an action that still has to be executed,
    // this variable points to that transition. Otherwise it is NULL.)
    TransitionConfiguration* halfWayTransition;

    // Enter a given state
    void EnterState(State* s);
    // Leaves the top state of the scope
    void LeaveState();
    // Goes to the target descendant state
    void DescendTo(State* target);
    // Enters history states (initial if history is NULL)
    void DescendHistory();
    // Recursively enters the initial child state until there is no initialSubState
    void DescendInitial();
    // Goes to the target ancestor state
    void AscendTo(State* s);

public:
    // Constructor
    StatechartChecker(State* scRoot, bool isMale);
    // Creates a new configuration for a state or transition
    StateConfiguration* newStateConfiguration(State* root, bool isMalePeer, State* s, std::vector<StateConfiguration*> &scope);
    TransitionConfiguration* newTransitionConfiguration(State* root, bool isMalePeer, Transition* t, std::vector<StateConfiguration*> &scope, std::vector<void*> &params);

    // Initialize the statechart
    void Initialize();

    // Returns a transition activated by a Reflection message
    TransitionConfiguration* FindTransition(std::string msg, std::vector<void*> &params, bool isEvent);
    // Accepts an incoming Reflection message and executes a transition
    std::string DoTransition(std::string msg, std::vector<void*> &params, bool isEvent);

    // Returns a list of possible transitions leaving from the current scope
    std::vector<std::string> ListTransitions();
};
#endif
```
Listing 21: The StatechartChecker.h

```c++
#include "StatechartChecker.h"
#include "StatechartExceptions.h"
#include "State.h"
#include "Transition.h"
#include "StateConfiguration.h"
#include "TransitionConfiguration.h"

// Function definitions for the checker
StatechartChecker::StatechartChecker(State* scRoot, bool isMale)
{
    this->isMalePeer = isMale;
    this->rootNode = scRoot;
    this->halfWayTransition = NULL;
}

// Initialize the statechart
void StatechartChecker::Initialize()
{
    // Start the execution.
    this->EnterState(this->rootNode);
    // Enter the initial state and go down as long as there are initial sub-states
    this->DescendInitial();
}

// Returns a possible transition depending on the incoming message
// A transition is:
// – available if its source state is in the current scope
// – condition-enabled if the condition is satisfied
// – event-enabled if the event (and number of parameters) is satisfied
// – action-enabled if the action (and parameter expressions) is satisfied
// – enabled if it is (condition-enabled ∧ (event-enabled ∨ action-enabled))
// – unique if no other transition is enabled
TransitionConfiguration* StatechartChecker::FindTransition(std::string msg, std::vector<void*> &params, bool isEvent)
{
    if(this->halfWayTransition)
    {
        // The input message must be an action
        if(!isEvent)
        {
            // Check if the action message matches
            if(this->halfWayTransition->transition->_action == msg)
            {
                // Check if the parameters match
                if(this->halfWayTransition->CheckActionParameters(params))
                {
                    return this->halfWayTransition;
                }
            } else
            {
                throw InvalidActionParameters();
            }
        } else
        {
            throw InvalidActionMessage(this->halfWayTransition->transition->_action);
        }
    } else
    {
        throw ExpectedAction();
    }

    // Else, iterate over all transitions in the current scope.
    // (This is more efficient than iterating over all transitions.
    // If the source of a transition is in the scope, its LCA is as well.)
    // Find all enabled transitions
```
// Running time $O(|T| \times D)$, where $T=$ transitions and $D=$ maximal depth of the state tree
TransitionConfiguration* result = NULL;
for(unsigned int i=0; i<scope.size(); i++){
    for(unsigned int j=0; j<scope[i]->state->transitions.size(); j++){
        Transition* t = scope[i]->state->transitions[j];
        // Check if the source of this transition is in the current scope
        bool sourceInScope = false;
        for(unsigned int k=i; k<scope.size(); k++){
            if(t->source == scope[k]->state){
                sourceInScope = true;
                break;
            }
        }
        if(sourceInScope){
            // If the transition is event-enabled
            if(t->_event == msg && isEvent){
                // Initialize the transition variables
                TransitionConfiguration* tc = newTransitionConfiguration(this->rootNode, this->isMalePeer, t, scope, params);
                // If the transition is enabled
                if(tc->CheckTransitionEnabled()){  
                    if(!result){
                        result = tc;
                    } else{
                        delete tc;
                        throw MultipleAcceptingTransitions();
                    }
                } else{
                    // This transition was not enabled, so delete the configuration
                    delete tc;
                }
            } else{
                // If the transition is action-enabled (in case the event is empty)
                TransitionConfiguration* tc = newTransitionConfiguration(this->rootNode, this->isMalePeer, t, scope, params);
            }
        } else{
            delete tc;
        }
    }
}
// Return the configuration of a uniquely enabled transition, otherwise NULL
return result;
}

// Enters a given state
void StatechartChecker::EnterState(State* s)
{
  // Make a new instance to enter this state
  StateConfiguration* sconf = this->newStateConfiguration(this->rootNode, this->isMalePeer, s, scope);
  // Add the instance to the scope
  this->scope.push_back(sconf);
  // Adjust the last visited state in the parent ('save history')
  if(s->parent)
    s->parent->historySubState = s;
}

// Leaves a state
void StatechartChecker::LeaveState()
{
  delete scope.back();
  scope.pop_back();
}

// Goes to the target descendant state
void StatechartChecker::DescendTo(State* target)
{
  // Follow the path from the LCA to the target state, entering each successive level
  std::vector<int> path = scope.back()->state->GetPath(target);
  // Loop over the path in reverse and enter states from the LCA to the target
  for(size_t i=0; i<path.size(); i++)
  {
    size_t rev_i = path.size()-i-1;
    State* nextOnPath = scope.back()->state->children[path[rev_i]];
    this->EnterState(nextOnPath);
  }

  // Enters history states
  void StatechartChecker::DescendHistory()
  {
    State* s = scope.back()->state;
    // Verify that the state has a history state
    if(s->hasHistory)
      State* nextState = s->historySubState? s->historySubState : s->initialSubState;

      // If it has recursive history, follow history/initial pointers while possible
      if(s->hasRecursiveHistory)
      {
        do{
          this->EnterState(nextState);
          nextState = nextState->historySubState? nextState->historySubState : nextState->initialSubState;
        }while(nextState);
      }
      // If this is a normal history state, just enter one level deep and then follow initial states
      else
      {
        this->EnterState(nextState);
        this->DescendInitial();
      }
  }
  // Recursively enters the initial child state until there is no initialSubState
  void StatechartChecker::DescendInitial()
  {
    State* initialState = scope.back()->state->initialSubState;
    // if the initialState pointer is not null, enter the state it points to
    while(initialState)
```cpp
this->EnterState(initialState);
  // Get the next initial sub-state
  initialState = initialState->initialSubState;
}

void StatechartChecker::AscendTo(State *s) {
  while (scope.back()->state != s) {
    LeaveState();
  }
}

// Accepts an incoming Reflection message and executes a transition
std::string StatechartChecker::DoTransition(std::string msg, std::vector<void*> &params, bool isEvent) {
  TransitionConfiguration* tc = NULL;
  // Select an available, activated and enabled transition from all transitions in the scope
  tc = FindTransition(msg, params, isEvent);
  if (!tc) {
    throw NoAcceptingTransitions();
  }
  // From the scope, exit states until we reach the ancestor of t
  this->AscendTo(tc->transition->GetLCA());
  // First, check whether we are in a half-way state, that is,
  // if we are waiting for a transition event or a transition action
  if (this->halfWayTransition) {
    // Execute the transition code
    tc->ExecuteTransitionCode();
    // Enter the target state
    this->DescendTo(tc->transition->target);
    // Enter initial substates
    if (tc->transition->targetsHistory) {
      this->DescendHistory();
    } else {
      this->DescendInitial();
    }
    // Clear the halfWayTransition pointer
    this->halfWayTransition = NULL;
  } else {
    // If there is no action, execute the transition code and enter the target state
    if (tc->transition->_action == "") {
      tc->ExecuteTransitionCode();
      this->DescendTo(tc->transition->target);
      if (tc->transition->targetsHistory) {
        this->DescendHistory();
      } else {
        this->DescendInitial();
      }
    } else {
      this->halfWayTransition = tc;
      return "Executed event. Waiting for action: "+ tc->transition->_action + "...";
    }
  }
}
```
```cpp
class StatechartChecker {
    // returns a list of possible transitions leaving from the current scope
    std::vector<std::string> ListTransitions() {
        std::vector<std::string> translist;
        // If we are currently waiting for an action message, return the halfway transition
        if (this->halfWayTransition)
            translist.push_back(this->halfWayTransition->transition->_action + this->halfWayTransition->transition->_actionparams);
        else {
            // Else, check all transitions whose ancestor is in the current scope
            for (unsigned int i = 0; i < scope.size(); i++)
                for (unsigned int j = 0; j < scope[i]->state->transitions.size(); j++)
                    // Check if the source of this transition is in the current scope (must be inside ancestor i)
                    bool sourceInScope = false;
                    for (unsigned int k = i; k < scope.size(); k++)
                        if (t->source == scope[k]->state)
                            sourceInScope = true;
                            break;
            if (sourceInScope)
                if (t->event == "")
                    translist.push_back(t->event + t->eventparams + " | " + t->condition + " | SystemOutput " + t->action + t->actionparams);
                else if (t->event == "" && t->action != "")
                    translist.push_back(" " + t->condition + " | " + t->action + t->actionparams);
            return translist;
        }
    }
private:
    // Return the new state
    std::string returnState() {
        std::string new_state = " Done. New state: " + scope.back()->state->name;
        return new_state;
    }
};
```

Listing 22: StatechartChecker.cpp
References


