From SysML to Model Checkers via Model Transformation

Martin Kölbl, Stefan Leue, and Hargurbir Singh

University of Konstanz, Germany

Abstract. In this paper we present an automated translation from the systems engineering modeling language SysML into the input languages of the NuSMV, Prism and Spin model checkers. A special focus of this work is the semantics of the communication mechanisms used in a syntactic fragment of SysML, in particular synchronous and asynchronous, broadcast and buffered communication. In order to achieve generality of our approach, which supports establishing the consistency of the translation as well as enabling easy adaption between different source and target languages, we use a model based transformation approach. In particular, we use the ATLAS Transformation Language (ATL) framework that is nicely integrated in the Eclipse Modeling Framework (EMF) and in the Meta-Object Facility. We illustrate the application of this model transformation approach using an airbag system as a case study.

1 Introduction

The use of model-based software and systems engineering in the design of critical systems implies the need to prove high dependability properties, including correctness, of these designs since human life or substantial damage to the environment is at stake. While a vast array of formal analysis techniques, such as model and causality checking [1, 2], have been developed to analyze model based designs for compliance with these properties, there is a substantial gap between the syntax and the semantics of model-based engineering languages and the input formats that the formal analysis tools accept. In particular, each model checking tool typically provides its own input language, designed to provide optimal abstractions enabling efficient model checking. Even if some model checking tools aim to provide open interfaces, e.g., LTSmin [3], there is no commonly accepted input format for models that would be processed by a large number of model checking tools. Also, none of the available model checkers can directly process the OMG System Modelling Language (SysML) [4, 5] which is widely used in industrial practice to model system architectures. SysML is supported by a large number of commercial and open source modeling tools (e.g., Rhapsody [6], Enterprise Architect [7], Papyrus [8].) SysML allows for modeling the structure and behavior of systems, including inter-object communication. If used in a design process, the architecture models need to be manually transformed into the input language of the chosen model checker. This is human intensive, and therefore extremely expensive, and furthermore an error prone activity. With an automated

transformation from SysML towards different model checkers, a system can easily be checked for faults. When a model checking algorithm detects a fault in a model, it returns a trace to the faulty state in the form of a counterexample. Each model checking tool, however, returns a counterexample using a different syntax. Interpreting the counterexample in order to correct the fault in the model requires intimate knowledge of this syntax.

We propose an automated, rule-based approach to transforming SysML models to models specified in the input languages of various model checking tools that we consider. In particular, we will define transformations for the Spin [9], Prism [10] and NuSMV2 [11] model checkers. Automated model transformation facilitates maintaining consistency by following defined transformation mappings. Notice that since the input languages of model checkers typically have a very specific semantics, only subsets of the SysML language will be transformed in our approach. We base our translation on the following precursory research.

- The safety analysis of SysML models implemented in the QuantUM [12] approach and tool includes a transformation of SysML models to the input languages of the Prism and Spin model checkers. Using causality checking, QuantUM computes causes for faults of a system and presents the computed causes by fault trees to the user.
- In [13], an approach is described which transforms SysML to NuSMV2. The transformation is based on an object-oriented view and first transforms the SysML model to a general model checker model using the support of JDOM [14]. Afterwards, the general model is translated to NuSMV2 code using a non-automated approach.

We exploit, extend and generalize these ideas as follows.

- First, our approach reuses the idea of an object-oriented meta-model and generalizes it by using model-to-model transformation (MMT) technology. The previous translation approaches implemented model transformation by using JDOM or by direct Java programming. A concern with these non rule based translation approaches is that it is difficult to ensure consistency of the translation. They also lack flexibility and concept reuse when changing from one target modeling language to another. We propose to use an MMT approach, which ensures consistency of the input, output and transformation model against meta-models during the model transformation, and allows for flexible re-definition of translation rules when considering a different target modeling language. As an MMT framework we use the Atlas Transformation Language (ATL) [15], and the Xpand framework [16] for the model-to-code transformation, both integrated into the Eclipse Modeling Framework (EMF) [17].
- Second, the works cited above transform states, transitions and nested states along with guards, actions and asynchronous communication. Our approach extends the forms of communication that the previous approaches support by also considering SysML synchronous and asynchronous point-to-point communication, asynchronous broadcast, and buffered communication. Although asynchronous communication can be used to emulate synchronous communication and broadcasting, we want to take advantage of the expressiveness of

different communication paradigms provided by SysML and enable the user to model the system using the envisioned logical syntax. The usage of different communication mechanisms also helps us produce more efficient code in the target model checker input languages, contributing to addressing the state space explosion problem [1].
- Finally, our approach enables a user to understand a given counterexample by a transformation from the counterexample syntax of the used model checker to a SysML sequence diagram.

We illustrate the application of our approach by a case study that applies the proposed transformation to an airbag model used in previous transformations [13].

Related Work. A number of publications related to the transformation of SysML to different analysis tools are available in the literature. For instance, in [18] a transformation is presented which transforms SysML to a hardware description language. Similar to our work, that work uses an object-oriented approach, but the target model is fundamentally different from our target model. We are not aware of any work that translates SysML using automated model transformation technology into any of the input languages of the model checkers that we consider.

2 Preliminaries

SysML Model Elements. The purpose of the SysML standard is to define a general modeling language for system engineering [4]. SysML uses a subset of the Unified Modeling Language (UML) [19], but also adds diagrams to the UML. In the remainder of the paper we will only refer to SysML in the understanding that many of the syntax and semantics definitions can be found in the UML specification. In the current paper, we transform only a syntactic subset of the full SysML language. In SysML, a system model is given by block definition diagrams (bdd), depicting the structure of the architecture, internal block definition diagrams (ibd), representing the internal structure of blocks, and state chart diagrams (stm), specifying the behavior of the respective block they are associated with. A bdd describes the overall structure of a system. It comprises blocks which represent classes, with associations between blocks depicted by straight lines. An ibd is a refinement of a bdd and encompasses the same structural elements as a bdd. A block that possesses a behavior description is called an active block. Using an stm, the behavior of a block is described by states and transitions. States represent a location of control, and transitions between states representing state changing activities in the system. In particular, a state transition may be labeled by a trigger, indicating a wait condition for an event to occur, a guard controlling the activation of the transition, and effects which are being executed when the transition is taken. In the context of this paper, we only consider activities and opaque behavior as transition effects. An opaque behavior is defined as an arbitrary text that is not specified in terms of its syntax and semantics in the SysML standard. In SysML, an activity is defined as being represented by a complex

activity diagram. In this paper we only consider activities that consist of a single send action. The stms of all blocks in the system are executed concurrently. Stms of one block can interact with stms of other blocks using shared variable as well as message passing based communication. Stms follow the hierarchical state machine idea and can contain substates that represent other hierarchical state machines. We call such states composite states.

Execution of SysML Models. When interpreting the behavior of an SysML model operationally, for instance by a model checker, we need to comply with the `run-to-completion` principle in the semantics of executing stms. As we now illustrate, this principle leads to ambiguous interpretations of SysML model behavior. In an SysML model, a state can be active, which means that the current location of control is in this state. If a composite state is active, this means that control rests in one of its sub states as well as in the composite state itself, which means that a set of states can be active at any given point in time. The set of active states is called a state configuration. A state may be labeled by entry and exit behaviors which are executed when the state is entered or exited, respectively, as well as by behavior executed while the system is in that state. The stm maintains an event pool that contains events that are available to trigger transitions. A state configuration is called stable if all entry behaviors of the current state configuration are completed and no more transitions are enabled. The SysML specification defines an execution environment that selects an event to be processed from the event pool. The precise mechanism how this selection is performed is not specified. The `run-to-completion` principle means that when an stm is in a stable state configuration, the execution environment of the SysML model selects an event to trigger a transition, the effects of the selected transition will be executed and the system returns to a stable state. Applying this principle to a loop in an stm that consists exclusively of transitions that have enabled guards but no triggers implies that the stm enters a livelock and never reaches a stable state configuration again. This is not consistent with a further specification in the SysML standard which states that once the execution of an stm reaches a state, the stm remains in the state until a transition is triggered by an event from the event pool, or an external asynchronous message terminates the execution. In order to resolve this inconsistency, we assume that in the context of this paper each transition without an explicitly specified trigger will find an implicitly defined trigger event in the event pool that allows it to perform the next transition. Under this assumption a cycle of transition reaches a stable configuration in each state of the cycle.

Communication in SysML. The SysML standard defines a large number of types of communication, including messages. In this paper we only consider communication between the stms by messages without parameters and return values. We use the syntax options of SysML to express different forms of communication. A message event in SysML can be instantiated as a call or a signal event. The behavior implied by a message event depends on this instantiation. If the message event is of type signal event, then the communication is asynchronous and the stm continues after sending a message without blocking. If the message event is

of the type of a synchronous call event, then the communication is synchronous and the stm waits until the called operation has finished. Notice that in accordance with the SysML specification, a transition with a synchronous call only completes the current execution step when the called operation has completed. In this paper we do not consider asynchronous call events for which the invoking stm only waits until the operation is called. The type of a send action depends on the kind of message event that is to be sent. A send signal action sends a signal event. A call operation action sends a call event. A send action sends a message to a reception if both refer to the same message event. In the models we consider in this paper the sending stm is assumed always to be different from the receiving stm. Signal events and call events are received when executing a trigger in a transition. If several triggers can receive an event, then the SysML standard suggests that the trigger from an active substate has priority in execution over the trigger of the composite state. If several transitions have the same priority, then the transition to be executed is selected non-deterministically.

Special forms of asynchronous communication are broadcast and buffered communication:

- SysML provides *broadcast communication* using a send action of type broadcast signal action. SysML defines the receivers of a broadcast as all potentially available targets and mentions that the exact set of targets is not defined. To represent broadcast in this paper we assume that the broadcast is directed to every block that has an stm with a matching receiving trigger on one of its transitions.
- SysML supports the modeling of *buffered communication* by adding First-In-First-Out (FIFO) queues to the sending actions. Since the target of our work is the use of finite state verification technology to analyze SysML models we restrict the capacity of the queues to be finite. If no queue length is specified in the SysML model we assume a default queue size of one message. According to SysML specification, in case of a full queue the message sent by a send signal action will be lost. This is a consequence of the fact that a send signal action just sends a message but does not consider any reception of the signal event.

Properties in SysML. Since the objective of this paper is to use model checking technology to verify properties against SysML models we need to consider how to specify the desired type of properties in SysML. We express a specification in SysML by using invariants. An invariant is an expression that before and after each execution step has to evaluate to true and consists of a composition of Boolean condition about states and variable values inside of a block. Invariants can be specified using the Object Constraint Language (OCL) [20] which is a higher order logic formalism defined to specify logical constraints on the SysML. As a property specification we add an OCL invariant to a model by adding an OCL formula expressing the property to the topmost element of the model, called the root. An invariant can refer to other invariants in the other blocks of the model and use those in order to check a combination of states or variable values belonging to different blocks. We restrict the OCL formula representing the

desired property to be a propositional formula ϕ, where the propositions in ϕ refer to variable values and states being active or not. The model checkers NuSMV2, Prism and Spin are capable of verifying such invariant properties, for instance by translating them to the Linear Time Temporal Logic (LTL) [21] formula $\Box\phi$ and performing LTL model checking using this property on the model.

SysML Sequence Diagrams. A SysML sequence diagram [4] is an interaction diagram and depicts the message flow between actors and blocks of the system. The diagram consists of several `Lifelines` which model concurrent processes. Behaviors like activities and send actions of a process can be added to a lifeline by a behavior execution specification. A behavior execution specification is depicted by a rectangle on the lifeline of the executing process. `Lifeline`s are arranged next to each other and along each line, the order of the events in each of the depicted actors and blocks is from top to bottom. A message is depicted by arrows from a sender to a receiver process. Asynchronous message are depicted with an arrow and synchronous messages are depicted with a filled arrow.

The NuSMV2 Modeling Language. NuSMV2 is a symbolic model checker which is used for the verification of synchronous and asynchronous finite state systems [11]. NuSMV2 can perform finite state model checking for LTL specifications. In this section, we introduce the parts of the NuSMV2 input language which are relevant for our approach. A short example of NuSMV2 code is given in Listing 1.1.

```
MODULE main(events)
  VAR
       state: {run, undetected};
  ASSIGN
    init(state) :=      run;
    next(state) :=
      case
        state = run & events != ECU_error: undetected;
        TRUE: state;
      esac;
```

Listing 1.1. NuSMV2 Example

An asynchronous model consists of several concurrent processes. In NuSMV2, each concurrent process is defined by the keyword `MODULE`. Each MODULE consists of two sections. In the section `VAR` the variables are declared. Section `ASSIGN` contains variable assignments. The keyword `init` declares the initial value of a variable. The NuSMV2 model specifies a transition system. Variable values can be changed in the `next` clauses which describe how the value of a variable changes in the course of a state transition. In the example, the statement `next(state)` changes to the value of the variable `undetected` if the guards `state = run` and `events != ECU_error` evaluate to true. The `TRUE:state` clause is executed if the guards are false, which means that the state remains unchanged. All possible variable changes of a single process are done at once, but only of one process at a time. NuSMV2 allows the declaration of variables as an enumeration of non reserved strings as value range. The MODULE called `main` can define global

variables and other MODULES. NuSMV2 has no special syntax or semantic definition for the declaration of communication channels.

The Promela Modeling Language. Promela is the input language of the explicit state model checker Spin. It is combining a fragment of the syntax of the C programming language with guarded commands and specific communication primitives [9].

```
int process1_states = 0;
chan channel1 = [0] of { bool };
chan channel2 = [1] of { bool };
proctype process1 {
 do
 :: process1_states == 0 -> process1_states = 1; channel1!true;
 :: process1_states == 1 -> channel2?true; process1_states = 0;
 od;
```

Listing 1.2. Promela example

Concurrent processes in Promela are defined as `proctype`s, as illustrated in Listing 1.2. Variables are declared with a type and can either be defined locally inside a proctype, or globally. Computation steps from different proctypes are arbitrarily interleaved. Sequences of Promela statements included in an `atomic` statement will not be interleaved by statements in concurrent proctypes. Promela provides keywords for the definition of channels as well as the sending and receiving of messages. As shown in the example, if the `process1` is in state 0 then it can send a message to `channel1` and if the process is in state 1 it can receive a message from `channel 2`. Channels are defined with the keyword `chan`, indicating a type and a capacity of the channel. A channel is defined as synchronous if the size is zero as in the example with `channel1`, or asynchronous with a non-zero positive size as with `channel2`. A synchronous message can only be sent by a sender if another process is ready to receive the message. The synchronization statement of a channel breaks the atomicity of an `atomic` statement at the point of sending the message and goes on with the receiving statement. For asynchronous communication in SysML we use a asynchronous First-In-First-Out (FIFO) channel of Spin. If the channel is full we instruct Spin to lose the message and proceed.

The Prism Modeling Language. The model checker Prism allows for the model checking of a probabilistic timed variant of CTL relative to discrete or continuous time finite state Markov chain models. SysML has language elements that allow to add probabilities and stochastic rates to the model. We currently do not interpret these SysML elements an assume all probabilities to have a rate of 1. A Prism code sample is presented in Listing 1.3. Concurrent processes in Prism are defined by the keyword `module`. Variables can be defined locally by a name, type, initial value and range. Transitions can have a synchronization event name inside of the brackets `[...]`, a guard, a probability and several actions. If different processes have a transition labeled with the same synchronization event, then these transitions can only be executed synchronously. All other transitions are taken sequentially in an arbitrary sequence. The update of a variable is indicated using the frequently encountered "prime" notation. For instance, in the course of the

`transition1` transition in Listing 1.3 the variable `done` is assigned the new value `true`. There is no explicit syntax in Prism to define a communication channel.

```
module process1
  states : [0..1] init 0;
  done : bool init false;
  [transition1] (done = false) -> 0.01 : (done'=true);
endmodule
```

Listing 1.3. Prism example

The Atlas Transformation Language (ATL). ATL is a domain specific transformation language and provides a framework for the rule-based model-to-model transformation of XMI [22] based models. ATL addresses two important issues in model transformation. First, it checks the syntactic correctness of the input model and hence avoids an ill formed input model to lead to an ill formed output model. Second, it supports the assurance of correctness and unambiguity of the transformation rules for complex model transformations. ATL addresses these problems by exploiting the idea of meta-models for the purpose of model transformation. A meta-model is a special model representing the model elements of the modeling language used [23]. For instance, it specifies rules that determine what the correct structure of a SysML model is, and what its admissible elements are. ATL allows a declarative description of transformation rules by the use of meta-models. Transformation rules can refer to the elements in the source and target model. This leads to a more dependable model transformation since the complexities entailed by the selection of source elements and the application of rules is handled automatically by ATL [15].

The structure of an ATL transformation process is depicted in Figure 1. ATL parses a source model `MA` in accordance with a source meta-model `MMA` and ensures that a source model conforms to its source meta-model or recognizes the source model as ill defined input model. Then the ATL code `mma2mmb.atl` describes how a source model is converted to a target model which conforms to a target meta-model `MMB`. The ATL code itself has to conform to the ATL meta-model `ATL`. To ensure correctness of the transformation, all meta-models have to conform to the standard meta-meta-model *Meta Object Facility* (MOF) [15] proposed by the OMG. ATL has its own syntax, but also inherits a subset of Java and OCL syntax. OCL operations provide a common way to work with collections, for instance the forall operator that can be used to iterate over a collection. Transformation rules are described in the ATL code as a set of mappings between source and target patterns with imperative operations performed on the source elements. Each element of the source model matches at most to one transformation rule.

An example for an ATL rule `E2E` is given in Figure 1.4. The rule refers to elements of the meta-models `MMa` and `MMb`. In the rule, we describe the transformation from an source element of type `EnumerationLiteral` to a target element of type `StringEnumeration` where the rule transfers the information from the attributes name and id. The source pattern contains a conditional statement which restricts the matching to source elements whose name starts with an `A`.

```
rule E2E{from s: MMa!EnumerationLiteral (s.name.startsWith("A"))
        to t: MMb!StringEnumeration (name <- s.name, ID <- s.ID)}
```

Listing 1.4. ATL example

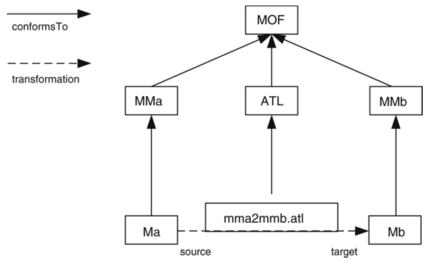

Fig. 1. Overview of ATL Transformation [15]

There are three types of ATL rules:

1. *Matched rules.* These are the basic rules which will be matched against the source elements. The rule name must be unique and contain an source and an target pattern. The example in Figure 1.4 is a matched rule.
2. *Lazy rules.* This type of rule is only called through other rules. It is usually used to create child elements and helps in traversing the XMI model.
3. *Called rules.* These rules behaves similar to lazy rules but don't contain any source pattern. They can be called at the entry and exit of the transformation execution. Called rules are used to create new elements in the output model for which no source elements exist.

Xpand is a framework supporting the model-to-text transformation for the generation of text files for domain specific languages (DSLs). We use Xpand here to generate code of the model checker input languages based on the XMI produced by the ATL model transformation. To do so, we need a source model, a source meta-model and the Xpand code which is executed to create the output text file, representing the model checker input code. For the source meta-model in Xpand, we reuse the target meta-model of ATL. The Xpand code for a transformation is called a template, for an example see Listing 1.5. In a template, we define the structure of the target text file. Xpand specific code is written between the signs « and ». Any text outside of these characters is considered part of the target output file syntax and is written directly to the target file.

The Xpand code example in Listing 1.5 first checks whether the input model conforms to a meta-model `MMb`. The Xpand code starts inside the `«DEFINE»` statement `main` by creating a file and writing the text "states =" to the file. Afterwards the `«EXPAND»` statement `substates` iterates over a comma separated list `states` of states. For each state of the list the `«DEFINE»` statement with the same name as the `EXPAND` statement is called with a element of type `State`. The called `«DEFINE»` statement writes the name of the current state into the file.

```
«IMPORT MMb»
«DEFINE main FOR Model-»
«FILE filename+".txt"-»
states = {«EXPAND substate FOREACH this.states SEPARATOR ',' -»}
«DEFINE substate FOR State-»«this.name-»«ENDDEFINE»
«ENDFILE»
«ENDDEFINE»
```

Listing 1.5. XPAND example

3 Model Transformation

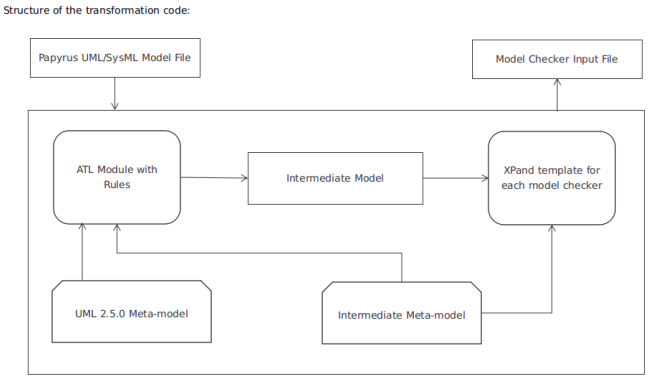

Fig. 2. Structure of the transformation

Model Transformation Approach. Our approach consists of two transformations. First, we transform a SysML model into the input language of the considered model checkers. Second, if a model checker identifies a counterexample to the property that we are interested in, we translate this counterexample into a SysML sequence diagram. The SysML models that we wish to analyze are edited using the Papyrus editor and saved in XMI format. We then use ATL to parse the XMI model of Papyrus and transform the model to an `Intermediate Model`, and finally transform this model to the different model checking languages using XPand. An overview of the model transformation approach is depicted in Figure 2. ATL parses the input `Papyrus UML/SysML Model File` and checks if the contained model conforms to the UML 2.5.0 meta-model of Eclipse. Afterwards, the `ATL Rules` transforms the model into an `Intermediate Model`. As a target meta-model we use the one defined as `intermediate meta-model` in [13]. We use the intermediate meta-model in order to ensure that the intermediate model contains all necessary information, for example stms, states, transitions, guard, etc. After the generation of the intermediate model and using the specific templates of the model checkers, Xpand translates the elements of the intermediate

model into the code of the model checker. The support for further model checkers can be added by using additional Xpand templates.

State and Transition Transformation Rules. We shortly sum the basic transformation rules from SysML to a model checker model that we adopt from our precursory work, [13] and [12]. We only support nested states in stm diagrams and not referenced stms. Each stm defines a concurrent process in the respective model checking input language. We flatten nested states by encoding each state in the model by a unique name. The unique name is the combination of the names of all nesting states of the current state and the name of the state itself. Spin directly uses state names and can change the current state by a `goto` statement to the next state. In NuSMV2 and Prism, the current state of an stm is stored by a variable with a name of the stm extended with "`_states`" for Prism and in the variable named `state` in NuSMV2. In NuSMV, the variable is of type enumeration and stores the current state name. In Prism, the state variable stores a unique number for each state of a block. A model transition can change the current state by changing the value of the state variable. A model transition is only enabled if the source state is the current state of the stm. We add an extra guard, which enables a transition if the current state of the stm is the source state of the transition, to each transition. If the current state is a substate and has a composite state, then the transitions of the composite state are also enabled. Our transformation flattens the state hierarchy and hence, a transition that was enabled in the composite state is no longer enabled when entering a substate. As a consequence we enable all those disabled transitions in the composite state again by copying them to the substate. The translation of guards and effects of transitions is straightforward. Entry and exit behavior of a model state are added during the model transformation as effects to the corresponding transition in the intermediate model. An effect of a transition can be an activity or opaque behavior. We use an activity with a single action to represent the sending of a message event, and an opaque behavior in OCL format to characterize the update of a variable using a logical formula. In the context of this paper, we only allow a single effect for each transition. This ensures consistency of the generated model checking input code among the different model checkers that we support since it avoids problems ensued by the different execution order semantics that these model checkers assume. In particular, just like SysML, Spin executes effects of a transition one after another in a sequential sequence while the Prism and NuSMV models that we generate execute all effects of taken transitions in one atomic step. With only one effect on a transition the model checkers do not deviate from the restricted SysML behavior.

Messages in Promela. Synchronous and asynchronous point-to-point communication messages are first class citizens in the Promela language. They are introduced in the language by declaring synchronous or asynchronous channels, and listing certain message names as being valid message names along these channels. We use the language constructs that Promela provides in order to define synchronous and asynchronous buffered communication. A transition with a synchronous call is executed at once or not at all since we clasp the guard and effects of a transition

in Promela by an `atomic` statement. Sequences of Promela statements inside an `atomic` statement are not interleaved by other concurrent statements. However, with a synchronous message, the `atomic` statement of the sender will break and execute the atomic statement of the receiver. Since in our modeling the sending transition can have only one action as an effect, all effects in the synchronization are executed in the correct sequence. In order to emulate asynchronous broadcast, we create an asynchronous channel for each receiver and send a message to each one of them. Since we need to send all messages at once, we enclose the sending actions to these channels by an `atomic` statement.

Messages in NuSMV2. There is no native syntax for messages in NuSMV2. In order to emulate *synchronous communication* we add the guard conditions for the receive transition in the receiving process to the guard condition of the send transition in the sender, thus ensuring that the sending can only be executed when the receiver is ready to receive. Furthermore, in order to ensure that in the course of one transition in the NuSMV2 model only one synchronous communication occurs the sending process sets a global variable to a value that is unique for this communication. This variable deactivates all transitions except the receiving transition. When executing the receiving of the message, the receiver resets the global variable. In order to emulate an *asynchronous broadcast* communication we create an array with one element corresponding to one receiving stm, for every receiving stm. When broadcasting the message, each entry of the array is set to true at once. Since the semantics of this broadcast is asynchronous in nature, there is no requirement to suspend the execution of any of the other modules. To emulate *buffered communication*, we use an array of variables that behaves like a FIFO queue. The length of the array is equal to the capacity of the sender queue as defined in the SySML model. Each entry of the array represents a position in the queue. As a consequence, multiple sending of messages are stored in the queue until the queue is full and then the extra messages are discarded. This persists until the receiver processes previously sent messages.

Messages in Prism. There is no native syntax for messages in the Prism input language either. Similar to the case of NuSMV2, we emulate the channels by using variables, but with slight differences. In order to emulate *synchronous communication* we take advantage of the implicit synchronization that Prism performs for several transitions synchronization event names, for which we use the name of the call event and the sender process. With this name encoding, we distinguish between several possible senders sending the same call event to the same receiver. In order to emulate *asynchronous communication*, like for NuSMV2 we create a variable. Prism does not permit the use of global variables together with label based transition synchronization. We therefore use a local variable inside the sending stm in order to coordinate the asynchronous sending and receiving. When sending the message, this variable is set to true. Since the variable is local, it can only be reset in the sending process. We generate an auxiliary transition in the sender which is independet of the remaining behavior of the sender. It is synchronized with the receive transition in the receiving process via a transition label and resets the local variable. *Broadcasting* and *buffered communication* are

emulated similarly as in the case of NuSMV2. A minor difference is that Prism has no syntax for arrays. We therefore create an individual variable for each entry of the array. In buffered communication, when the first element of the fifo queue is read, the value of each entry of the queue is copied to the entry ahead of it.

Counterexample to SysML Sequence Diagram Transformation. We propose the following steps to transform the counterexample that one of the considered model checkers produces into a Sequence Diagram. We interpret the counterexample, for which every one of the considered model checkers uses a different syntax, as a model and apply ATL based model transformation to this syntax in order to obtain a sequence diagram representation in XMI format.

1. *Parsing the Counterexample.* For an ATL transformation, we first need to parse the counterexamples which are stored in different textual formats depending on the model checker that was used. In order to accomplish this we need extra information from the original model, for instance regarding the names of the blocks that exchange messages, the events occurring along a state sequence, etc., since not all of this information is included in the state name sequences that the model checkers generate as counterexamples. We obtain this information by parsing the original SysML model.
2. *Transformation into a Sequence Diagram.* We next transform the counterexample in the source target model to a sequence diagram using an ATL transformation. For each process in the input model of the model checker we create a `Lifeline` in the sequence diagram, representing a concurrent thread of execution. A counterexample consists of a sequence of state and transition whereas a sequence diagram depicts the control flow of the system by a sequence of message send and receive events. Transitions of a stm without messages are depicted by a behavior execution specification. The behavior execution specification has as a name the current state name and contains the name and values of any changed variable. When a message is sent to another stm, then the message points from the sending to the receiving stm and the message name is set to the name of the sender, the current state name of the sender and the message event name corresponding to the message sending. A possible variable change of the reiving transition is added as an invisible attribute to the message arrow.
3. *Generation of Graphical View.* Using another ATL transformation step we generate a graphical view for the Papyrus IDE from the sequence diagram.

4 Case Study

The Airbag Model. We illustrate our approach by applying it to a real world model of an airbag system adopted from [24]. The SysML model of this system was edited using the Papyrus tool and is an extension of on the SysML model of the same system used in [13]. In particular, we added different inter process communication mechanisms in order to be able to experiment with the model translation rules for these mechanisms that we defined above. An overview of the

airbag SysML model is given as a bdd in Figure 3. The most important block in the airbag model is the `MicroController`. It continuously evaluates whether the two sensors represented by the block `MainSensor` and the block `SafetySensor` detect a critical accident of the vehicle, represented by the block `Car`. When this is the case, the deployment procedure for the airbag will be activated.

There are two meaningful properties to check for an airbag system. The first is to ensure proper functioning, i.e., to ensure that the airbag can be deployed. The second property is the absence of an inadvertent depoyment of the airbag when no accident has occurred. From a system safety point of view this is the more significant property to check, and we have focussed on it in some of our previous work on this model, e.g., in [24]. However, the counterexamples to inadvertent deployment are relatively short. We focus on the proper functioning property in this paper since it returns longer counterexamples and is hence better suited to illustrate the application of our approach.

In order to understand the behavior of the Airbag model, an understanding of the two safety mechansims designed to avoid an inadvertent deployment is essential. First, the Field Effect Transistor (`FET`) controls the power supply of the airbag squib. Only if the `MicroController` enables the `FET`, the airbag squib has enough power to deploy the airbag by igniting the explosive. Second, the Firing Application Specific Integrated Circuit (`FASIC`) only ignites the airbag squib if it first receives an `armFASIC` message and then a `fireFASIC` message from the `MicroController`.

Communication in the Airbag System. We use different forms of communication to forward information in the airbag system. In case of an accident the block `Car` broadcasts a message `crashHappened` to both sensors. The two sensors receive the broadcast message and start to repeatedly forward the information regarding the accident by buffered communication. The block `MainSensor` sends a message `mainSensorCrashDetection` and the block `SafetySensor` sends a message `safetySensorCrashDetection`. The repetition of the message ensures, that not a single wrong message can deploy the airbag. The microcontroller receives the messages of both sensors and starts the airbag deployment process after receiving the accident notification of each sensor two times. In order to start the deployment process the block `MicroController` sends the following three asynchronous messages. A message `armFASIC` is sent to the block `FASIC` and causes the fasic to go into state `arm`. A message `FETPoweredOn` is sent to the block `FET` which enables the power supply when received. A message `fireFASIC` is sent to the block `FASIC` which causes the fasic to transit from state `arm` into state `fire`. It is now important to ensure that the squib will only explode and deploy the airbag if the power supply is enabled at the time of firing the squib. We model the coordination regarding the deployment of the airbag between the `FET` and the `FASIC` by two synchronous messages that are exchanged between these two processes. In case the `FET` is enabled, it can send a message `FETPoweredOn`, otherwise it can send a message `FETPoweredOff` in order to communicate its state to the `FASIC`. If the `FASIC` accepts the synchronization via the `FETPoweredOn` message this means that the airbag will actually be deployed by the `FET` applying an ignition voltage to the squib. The `FASIC` then transits into the state `fired`. If, however, the `FASIC`

and the `FET` synchronize via the `FETPoweredOff` message, this means that the `FET` is not prepared to deploy the airbag and the `FASIC` transits into its initial state.

Property Specification. We specify the proper functioning property of the airbag system using an invariant. The invariant expresses that it is always not the case that the airbag is deployed and the car has an accident. If a model checker finds a counterexample for this invariant then the counterexample contains a sequence of states and transitions that starts with a car accident and terminates with the deployment of the airbag.

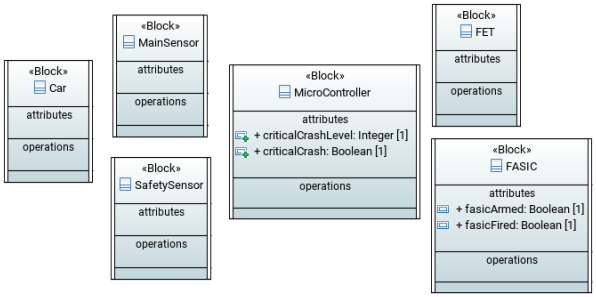

Fig. 3. bdd of the Airbag model

Analysis of the Airbag Model. We model check the models that we obtain from the above described translation into the target languages of the NuSMV2, Spin and Prism model checkers for the proper functioning property. Each model checker results in a counterexample to the property. We automatically transform the resulting counterexample of NuSMV2 to a SysML sequence diagram, depicted in Appendix A, Figure 4. A similar translation of the counterexamples for the other two model checkers is easily possible, but currently not implemented. Note that the change of the variable values is not visible in the figure, but it is viewable when browsing the diagram in the Papyrus IDE. The model transformations were performed on a computer with an i7-4820K CPU (3.7GHz), 32GB of RAM and a 64 bit Linux operation system. In Table 1 we show the memory usage and time necessary to transform SysML models to the different model checker input models, to verify the airbag model in the different model checkers, and for the NuSMV2 case to transform a counterexample to a sequence diagram. Additionally, we depict the count of states searched by each model checker.

Result Interpretation. The sequence diagram representing the counterexample produced by NuSMV2 consists of 6 lifelines, one for each module in the NuSMV2 code of the airbag model. The sequence diagram depicts ordered sequences of local module events, corresponding to local steps in the Airbag SysML model, as well as synchronous and asynchronous message passing events that lead up to the firing of the airbag, indicated by the `FASIC` entering the `fired` state.

	Model Transformation		Model Checking			Sequence Diagram Generation	
	Memory	Time	Memory	Time	States	Memory	Time
NuSMV2	9.6 MB	37 ms	10.0MB	0.092s	3279	9.6 MB	75 ms
Spin	9.6 MB	35 ms	128.3 MB	<1s	1432		
Prism	9.6 MB	34 ms	8.5 MB	0.023s	984		

Table 1. Computational Effort

The counterexamples produced by all three model checkers are similar. Each counterexample contains the necessary transitions to get from a car accident to a deployment of the airbag. The counterexamples mainly only differ in the order in which the transitions in the model are executed, but all contain the same set of transitions. NuSMV2 performs a short loop that the other model checkers do not include in the counterexample. For example with the deployment of the airbag, in NuSMV2 the messages `armFasic`, `fireFasic` and `enableFet` are all send before any of the messages is received, but in Prism and Promela each message is received before the next one is triggered.

5 Conclusion

We have presented an approach to automatically translating SysML models to the input languages of the model checkers NuSMV2, Spin and Prism, using the ATL framework for model to model transformation. We also propose to use ATL in order to translate the counterexamples for reachability properties to SysML sequence diagrams, thus facilitating error interpretation and debugging. We have illustrated the application of this approach using an industrially relevant case study.

In spite of the fact that at the time of writing only the SysML to NuSMV2 model transformation is fully automated we anticipate that the proposed automated model transformation approach is a lot more flexible in adapting to the target languages of other model checkers, compared to a manual encoding approach. We also foresee that the implicit consistency of the generated target models with meta models of the used modeling and target languages will support syntactic and semantic correctness of the generated target models. This will greatly help to bridge the syntactic and semantic gaps between domain specific modeling languages, such as SysML, and the somewhat idiosyncratic input languages of various model checking and other verification tools.

Currently, the flexibility of the approach is somewhat limited by requiring substantial specific, manual coding effort in the Xpand framework when generating the target models. We plan to devise meta models for each of the considered model checker input languages and transform to them from the general meta model. This will allow to greatly reduce the Xpand related coding effort. We also plan to establish semantic correctness properties of the model to model transformation using this more refined model transformation approach.

References

1. C. Baier and J. Katoen, *Principles of model checking*. MIT Press, 2008.
2. F. Leitner-Fischer and S. Leue, "Causality checking for complex system models," in *VMCAI*, vol. 7737 of *Lecture Notes in Computer Science*, pp. 248–267, Springer, 2013.
3. S. Blom, J. van de Pol, and M. Weber, "Ltsmin: Distributed and symbolic reachability," in *CAV*, vol. 6174 of *Lecture Notes in Computer Science*, pp. 354–359, Springer, 2010.
4. Object Management Group, "OMG Systems Modeling Language, Specification 1.5," 2017. `http://www.omg.org/spec/SysML`.
5. S. Friedenthal, A. Moore, and R. Steiner, *A Practical Guide to SysML 3rd Edition*. Morgan Kaufmann, 2014.
6. IBM Corporation, "Rational Rhapsody," 2017. `https://www.ibm.com/us-en/marketplace/rational-rhapsody`.
7. Sparx Systems, "Enterprise Architect," 2017. `http://www.sparxsystems.com/products/ea/`.
8. Eclipse Foundation, "Papyrus IDE," 2015. `https://www.eclipse.org/papyrus/index.php`.
9. G. J. Holzmann, *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.
10. M. Z. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of probabilistic real-time systems," in *CAV*, vol. 6806 of *Lecture Notes in Computer Science*, pp. 585–591, Springer, 2011.
11. R. Cavada, A. Cimatti, C. A. Jochim, G. Keighren, E. Olivetti, M. Pistore, M. Roveri, and A. Tchaltsev, "Nusmv 2.6 user manual," 1998. `http://nusmv.fbk.eu/NuSMV/userman/v26/nusmv.pdf`.
12. F. Leitner-Fischer and S. Leue, "Quantum: Quantitative safety analysis of UML models," in *QAPL*, vol. 57 of *EPTCS*, pp. 16–30, 2011.
13. G. Caltais, F. Leitner-Fischer, S. Leue, and J. Weiser, "Sysml to nusmv model transformation via object-orientation," in *CyPhy*, vol. 10107 of *Lecture Notes in Computer Science*, pp. 31–45, Springer, 2016.
14. J. Hunter and R. Lear, "Java Data Object Model," 2015. `http://www.jdom.org/index.html`.
15. F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "ATL: A model transformation tool," *Sci. Comput. Program.*, vol. 72, no. 1-2, pp. 31–39, 2008.
16. Eclipse Foundation, "Xpand," 2007. `https://www.eclipse.org/modeling/m2t/?project=xpand`.
17. Eclipse Foundation, "Eclipse Modeling Framework," 2017. `https://www.eclipse.org/modeling/emf/`.
18. J. Gauthier, F. Bouquet, A. Hammad, and F. Peureux, "Verification and validation of meta-model based transformation from sysml to VHDL-AMS," in *MODELSWARD*, pp. 123–128, SciTePress, 2013.
19. Object Management Group, "Unified Modelling Language, Specification 2.5.1," 2017. `http://www.omg.org/spec/UML`.
20. Object Management Group, "OMG Object Constraint Language, specification 2.4," 2014. `http://www.omg.org/spec/OCL`.
21. Z. Manna and A. Pnueli, *The temporal logic of reactive and concurrent systems - specification*. Springer, 1992.
22. Object Management Group, "Xml metadata interchange, specification 2.5.1," 2015. `http://www.omg.org/spec/XMI/`.

23. Object Management Group, "OMG Meta Object Facility (MOF) Core Specification, specification 2.0," 2016. http://www.omg.org/spec/MOF.
24. H. Aljazzar, M. Fischer, L. Grunske, M. Kuntz, F. Leitner-Fischer, and S. Leue, "Safety analysis of an airbag system using probabilistic FMEA and probabilistic counterexamples," in *QEST*, pp. 299–308, IEEE Computer Society, 2009.

Appendix A

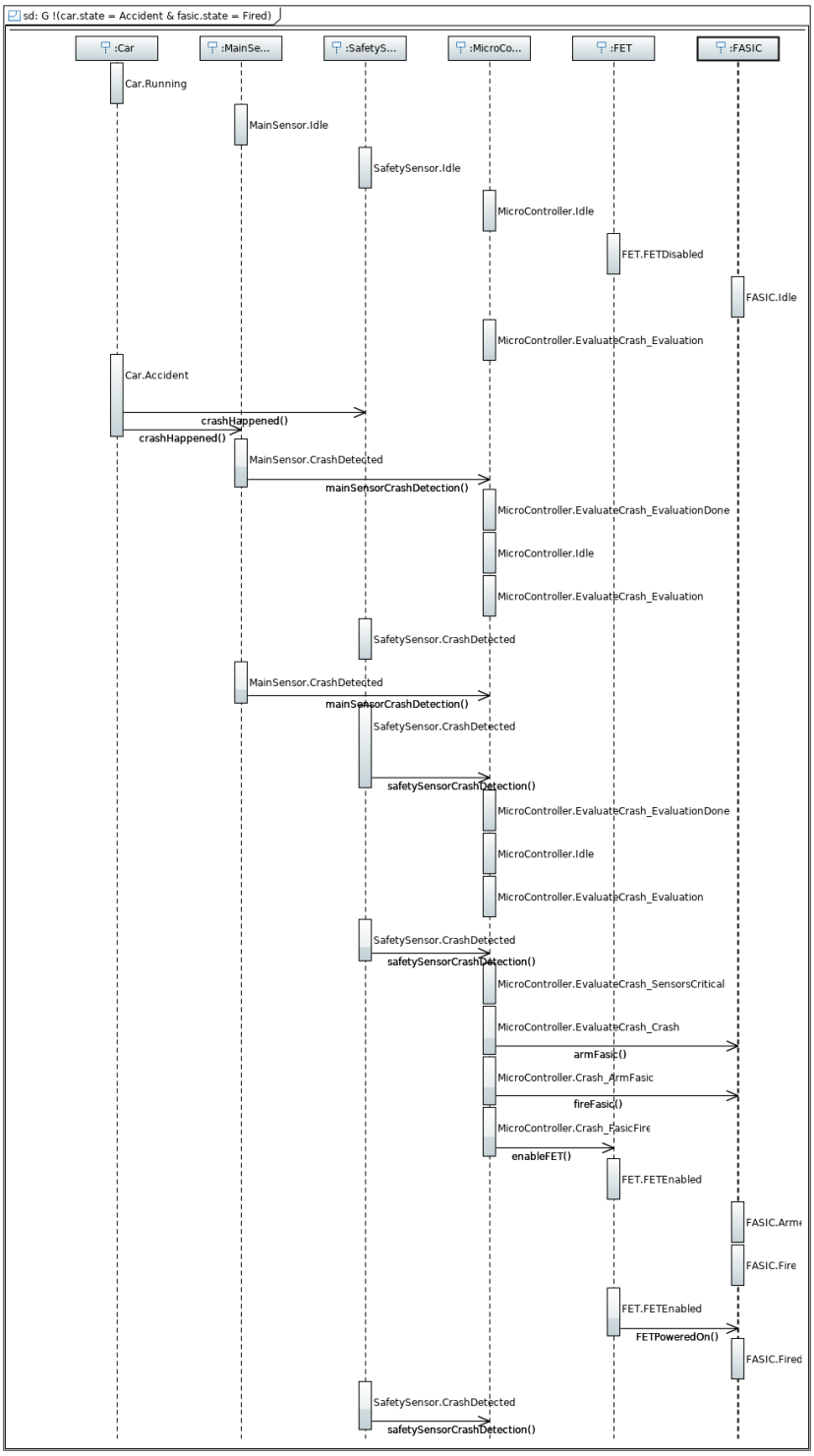

Fig. 4. Sequence diagram depicting the NuSMV2 counterexample, rendered by Papyrus