SysML to NuSMV Model Transformation via Object-Orientation

Georgiana Caltais[1(✉)], Florian Leitner-Fischer[2], Stefan Leue[1], and Jannis Weiser[1]

[1] Department for Computer and Information Science, University of Konstanz, Konstanz, Germany
{Georgiana.Caltais,Stefan.Leue,Jannis.Weiser}@uni-konstanz.de
[2] ZF Friedrichshafen AG, Active and Passive Safety Technology, Friedrichshafen, Germany
florian.leitner-fischer@zf.com

Abstract. This paper proposes a transformation of SysML models into the NuSMV input language. The transformation is performed automatically using SysMV-Ja and relies on a notion of intermediate model structuring the relevant SysML components in an object-oriented fashion.

1 Introduction

The complexity and size of safety-critical systems is steadily growing as technology advances. Hence, (semi-) formal approaches to the design, modelling and reasoning on the correctness of such systems plays a very important rôle. Nevertheless, introducing "friendly" formal frameworks into industrial settings is not at all a trivial task.

The OMG System Modelling Language (SysML) [14,17,25] is a graphical modelling language fairly intuitive and easy to learn by software engineers. SysML has been successfully used in practice. Nevertheless, the application of rigorous verification techniques such as model-checking on SysML-based inputs is usually not something that engineers are keen or trained to do.

In this paper, we propose a model transformation from SysML block definition diagrams and state machines to the input language of the NuSMV model-checker [8], implemented in the automated tool SysMV-Ja. Our approach exploits a SysML intermediate model. The intermediate model provides an object-oriented view of the SysML modelling concepts relevant for the work in this paper. This object-oriented approach could be exploited, in the future, to transform SysML into the languages of other model-checkers, in a structured way.

The intermediate representation is then exploited to guide a 2-step transformation from SysML to NuSMV input, in a structured way. Advantages of considering such an intermediate model include: the familiarity of developers with the Object-Oriented Programming-paradigm, the modularity of the approach, and the possibility of tracing back into the model potential sources of unwanted behaviour, as reported by the model-checker.

C. Berger et al. (Eds.): CyPhy 2016, LNCS 10107, pp. 31–45, 2017.
DOI: 10.1007/978-3-319-51738-4_3

Related work. There is a considerable amount of literature on providing (formal) semantics of SysML/UML, or on automatically translating associated models into inputs for different analysis tools.

The work in [9], for instance, presents a systematic, but direct translation of statecharts to SMV. As the approach is strictly tailored for the input language of SMV, it cannot be easily adapted for other model checkers or verification tools.

Hugo/RT [1] is a tool that translates UML into corresponding input for the Spin [18] model checker, via the so-called UTE intermediate format. UTE is a textual format that most of the engineers and programmers have to become acquainted with, in contrast to the more familiar Object-Oriented Programming-paradigm exploited in our paper. Another approach for verifying UML models using Spin is given in [23]. Even though the translation from UML to Promela– the input language of Spin–is straightforward and thus, little reusable, the automated tool vUML provides intuitive feedback to the user in case an error was found during verification.

In [21], SysML specifications are automatically translated into equivalent behavioural UML models. The latter are further used to derive test cases and executable test scripts, in the context of a model-based testing tool. The main difference with the work in [21] is that we use the Object-Oriented Programming-paradigm in order to model both the static and the dynamic structure of systems. The approach in [21] uses UML Class Diagrams to represent the static structure of systems and UML State Machines to represent their dynamics. Moreover, the unifying framework of object orientation enables us to define stereotypes and facilitates extensions of the standard SysML/UML semantics, if so desired. Nevertheless, our work does not tackle the issue of combining multiple profiles and avoiding specification conflicts. For a contribution along this research direction we refer, for instance, to [13] where some of the challenges of combining SysML and the OMG MARTE profile [15] are addressed.

More theoretical approaches, usually less appealing for engineers and software developers, propose formalisations of SysML/UML as Process Algebras [4,16] and Petri Nets [10,12]. Model checking of hierarchical state machines has been addressed in [3], for instance, where Kripke structures were employed as their formalisation.

A formal intermediate model of UML behavioural diagrams was also proposed in [11], in terms of the so-called Configuration Transition Systems (CTS's). Similarly to our approach, the results in [11] provide a systematic way of generating inputs for the NuSMV model checker based on intermediate models. In [11], the authors also emphasise on the importance of exploiting intermediate models in order to provide useful feedback to the designer. In accordance, the CTS's can be graphically visualised.

Labelled Transition Systems and Structural Operational Semantics [26] were exploited in [27] in order to provide a modular semantics of UML-RT –a dialect of UML that supports the development of hierarchical systems following a component-oriented approach. As for the case of UML-RT, rigorous formalisations are easier to define over textual terms. Such representations, however,

are difficult to use and follow in practice. For an attempt to overcome this type of issues, we refer to the results in [19] where the authors present a graphical user interface-based tool that supports a visual language called v-Promela. This language is the graphical extension of Promela, and the v-Promela notation inherits largely from the aforementioned UML-RT notation. Additionally, a semantics of UML-RT in AsmL–an object-oriented software specification language based on the theory of Abstract State Machines–was proposed in [22]. In connection with our current work, the idea of employing a meta-model defining the syntactic structure of the UML-RT modelling concepts was exploited as well. One the one hand, in our context, following the AsmL approach is not necessary as the syntactic structure of SysML/UML models can be expressed by means of Block Definition Diagrams. On the other hand, AsmL is a language that most of the engineers and developers would have to acquire.

The work in [24] is a classical reference on how to implement statecharts in Promela/SPIN using hierarchical automata defined based on operational semantics as intermediate format. A denotational meta-modelling of the semantics of a part of UML suitable for describing and constraining object structures was proposed in [20]. The results in [7] pave the way to a formalisation of UML in terms of the so-called System Models consisting of elements that describe the structure, behaviour and interaction of systems.

These more formal approaches are orthogonal works that go beyond the scope of providing a recipe for translating SysML/UML in terms of intuitive (intermediate) models, for the practical-minded. For a more detailed survey on model checking statecharts we refer to [6].

Structure of paper. In Sect. 2 we provide a brief overview of SysML modelling and NuSMV, by emphasising on the corresponding concepts relevant for our work. In Sect. 3 we introduce the intermediate model used for the transformation of SysML models into NuSMV-compatible inputs. In Sect. 4 we illustrate how the intermediate model can be exploited for the aforementioned transformation into NuSMV. Section 5 introduces SysMV-Ja, a Java-based tool for the automated model transformation. Two case studies, a railway and an airbag system are also discussed. In Sect. 6 we draw the conclusions and provide pointers to future work.

2 Preliminaries

In this section we proceed by first introducing a railway example, used throughout the paper in order to explain our approach.

Example 1 (Running example). The scenario considers a railroad track that is crossed by a street. On the crossing there is a gate, that can close when a train approaches, thus blocking cars from entering the crossing. A car or a train can be in one of four states: approaching, entering, being in the crossing or leaving the crossing. The gate can be in one of the two states: opened or closed. The situation that one does not want in this example is a train and a car in the crossing at the same time, as this would determine a crash.

In what follows, we provide a brief overview of SysML, the modelling language used by practitioners for designing systems such as the one in Example 1. Afterwards, we succinctly introduce the NuSMV model checker–a tool that can automatically detect hazardous situations such as a car-train crash.

The OMG System Modelling Language (SysML). SysML [14,17] is an industry standard for specifying and designing a broad range of systems. SysML was created as a general purpose modelling language for systems that may include anything from hardware and software to staff and facilities.

On the one hand, SysML can be used for the intuitive modelling of systems. We refer to Fig. 1 for a representation of the components of the railway in Example 1, and to Fig. 2 for a modelling of their behaviours. On the other hand, SysML can be employed similarly to a meta-modelling language defining the syntactic composition of the SysML modelling concepts considered by our approach. For instance, **iBDD** and **iStateMachine** in Fig. 3 define the parts (that are relevant for our approach) that constitute SysML Block Definition Diagrams and State Machine Diagrams, respectively.

Intuitively, SysML Block Definition Diagrams (BDD's) and State Machine Diagrams (STM's) are used in order to define the static aspects of systems, and to capture the dynamics of systems, respectively. BDD's are built on top of the so-called SysML *blocks*, and enable the modelling of systems in a modular fashion. Blocks correspond to units of a system description. See, for instance the block **Gate** in Fig. 1, that corresponds to the UML representation of the gate system in Example 1. A block can include properties of certain types and references to other blocks. For instance, the gate being open/closed corresponds to the boolean property "open" in Fig. 1 being set to *true*/*false*. Moreover, BDD's can capture relationships between blocks such as associations, and dependencies. For an example, we refer to Fig. 3. An aggregation stating that *one* **iModel** (intuitively, the railway system) consists of *one or more* **iBDD**'s (intuitively, the car, train and gate in the railway example) is illustrated via the connector $\diamond\overset{1}{\_\_\_\_}\overset{1..^*}{}$ with multiplicities *one*: 1 and *one or more*: 1..*.

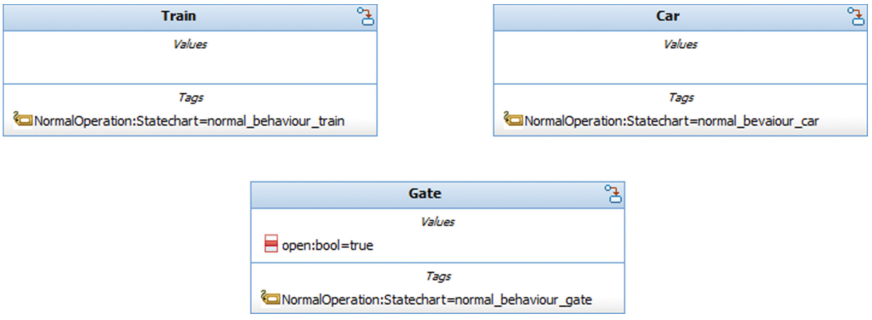


Fig. 1. The BDD's for the railway in Example 1.

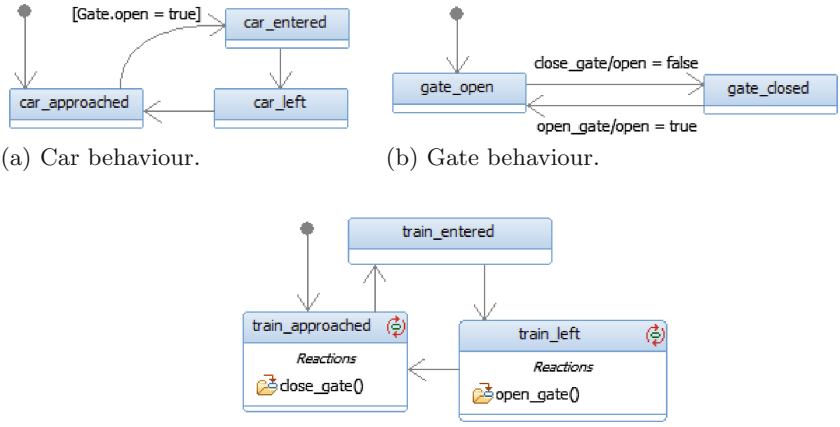
Behaviours can be associated to BDD's via properties of type StateChart. In Fig. 1, for instance, the train is associated a behaviour via the "operation" property. At this point it is important to mention that, in our approach, concurrent behaviour is modelled by synchronising multiple BDD's via *events*. Events occur in the context of *triggers* that specify points in the definition of a behaviour at which some effect can be observed.

STM's, or statecharts, are a form of finite state automata used in order to model the behaviour of systems. *States* in an STM can express different statuses in a behaviour of a system. For instance, the gate being either open or closed is captured by two simple sates "gate_open" and "gate_closed", respectively, in Fig. 2(b).

States can enclose so-called *regions* denoting behaviour fragments that may execute concurrently. Each region contains the nested disjoint states and corresponding transitions. Consequently, there exist the following kinds of *composite* states: *simple composite*–whenever the state contains exactly one region, and *orthogonal*–whenever it contains multiple regions. In this paper we only consider *simple composite* states. A *submachine state* refers to an entire STM nested within the state.

Either simple, composite or submachines, states can specify "entry", "exit" or "doActivity" behaviours. In short, entry (respectively, exit) behaviours are executed when the state is entered (respectively, exited) via an external transition. "doActivity" executes concurrently with any other behaviour associated with the state, as soon as the state entry behaviour has completed. An instance of a "doActivity" is the operation "close_gate" in Fig. 2(c).

Another special kind of states are the so-called *pseudostates*. Pseudostates are states with special behaviour. For instance, the *initial* pseudostate is the state in which an STM is initialised (see, for an example, the three bullet-like initial states in Fig. 2), or *exit* pseudostates. Additionally, the system cannot be in a



(a) Car behaviour.

(b) Gate behaviour.

(c) Train behaviour.

Fig. 2. The STM's for the railway in Example 1.

pseudostate. As soon as a pseudostate is entered, it is left again in a single atomic step. In this paper we only handle *initial* pseudo states.

Transitions can be seen as valid fragments of behaviour illustrating how the system evolves from one "source" state to a "target" state. A "guard" enables a transition whenever it is evaluated to *true*. We refer, for an example, to the guard "[Gate.open = true]" in Fig. 2(a) that enables the car to enter the crossing whenever the gate is open. The "effect" behaviour is enabled when the transition is executed. The effect "open = false" in Fig. 2(b) sets the value of the gate property "open" in Fig. 1 to *false*. A "trigger" specifies an event whose occurrence determines the execution of a transition. For instance, the event "close_gate" in Fig. 2(b) determines the gate to close. Recall that "close_gate" is also a "doActivity" in the state corresponding to train approaching in Fig. 2(c). Hence, its purpose is to simulate the synchronised communication between the train and the gate.

NuSMV. NuSMV [8] is a symbolic model checker successfully used for the verification of synchronous and asynchronous finite state systems. In short, NuSMV analyses specifications expressed in Computation Tree Logic (CTL) and Linear Temporal Logic (LTL) [5], using BDD-based and SAT-based model checking techniques.

In this section, we focus on the parts of the NuSMV input language relevant for our work. For a thorough description of NuSMV inputs, we refer the interested reader to the user manual in the distribution package[1] of the NUSMV model checker.

Intuitively, a NuSMV *program* consists of a list of *modules* further instantiated to so-called *processes* that model interleaving. A "process" has a special boolean variable associated with it, called "running", whose value is *true* if and only if the corresponding process instance is currently selected for execution. Each module is associated an *identifier* and a series of *parameters*. The *body* of a module consists of *elements* that can denote *variable declarations*, *variable initialisations/assignments*, *LTL specifications* or, for instance, behaviours defined based on *transitions*. Transitions are introduced by the "TRANS" keyword, followed by a boolean expression expressing whether or not two states belong to the transition relation. Therefore, the aforementioned boolean expression can include the "next" operator in order to relate the current and the next state variables, and express transitions in the state-machine corresponding to the behaviour of the module.

3 The Intermediate Model

In this section we provide an object-oriented representation of the relevant SysML components we consider for modelling the static and dynamic aspects of concurrent safety-critical systems. This representation serves as an intermediate step in the model transformation from SysML to NuSMV.

[1] http://nusmv.fbk.eu.

The advantages of using the object-orientation paradigm include software developers' familiarity with the concept and enables a structured, modular model transformation flexible to further extensions, and appropriate for automation.

The translation of the SysML relevant components into the intermediate model follows naturally. The **iModel** comprises all the elements of the system. All information that is obtained during the transformation from SysML to this intermediate model is either directly, as an attribute, or indirectly, as an attribute of one of its attributes, contained in the **iModel**. Directly contained as attributes in the **iModel** are all components, events, global variables which do not belong to any component, and the properties of the model captured by **iStateConfigurations**.

Each instance of **iStateConfiguration** stands for a safety or reachability property. These properties are expressed by the *configuration states* that shall "never be reached" or "eventually be reached", connected via "AND"/"OR" *configuration operations*.

Another element is the **iAttribute**, representing variables of the system. It can have a *default value*, saved as a string. If the attribute is an integer then it has a *lower* and *upper bound* and a *type* given by strings such as "integer" or "boolean", for instance. An **iAttribute** can be either a *global variable*, in which case it is saved in the **iModel**, or part of a system component, saved as an *attribute* in the corresponding **iBDD**.

An **iBDD** corresponds to a BDD and is characterised by the associated *attributes*. The connection with the STM's defining its *normal* and *failure* behaviours is established via class attributes of type **iStateMachine**.

The **iStateMachine** contains all the important information from an STM: all its *states*, including the *initial* one, and all its *transitions*. A *type* is associated in order to mark the behaviour of the **iStateMachine** as being *normal* or a *failure* one. As expected, an **iState**, corresponds to the concept of SysML *state*. An **iState**, encapsulates the *entry*, *exit* and *during* ("doActivity") behaviours a SysML state can display. **iStates** also include a list of incoming and outgoing *transitions*. If the state has submachines, then they are given by the *submachines* attribute. Note that only the *initial pseudostate* has a translation into the intermediate model as the "intialState" attribute of the **iStateMachine** class. SysML transitions are represented in this model via **iTransition**. The *source* and *target* states are the states from which the transition originates and to which it leads. The *guard* is a boolean formula that enables the transition whenever is evaluated to *true*. Intuitively, *action* collects all changes to attributes that happen when the transition is executed and it encodes the triggers and the behaviour of the transition. Finally, a transition can have a corresponding *event*. If that is the case, then the transition is only enabled if the event was triggered. SysML events are captured by the **iEvent** class which contains the *transitions that are triggered* by the event.

Moreover, note that all the blocks in Fig. 3 have a "name" and an "ID", as they inherit from **iElement**. We omit explicitly depicting the inheritance relationships, for readability reasons.

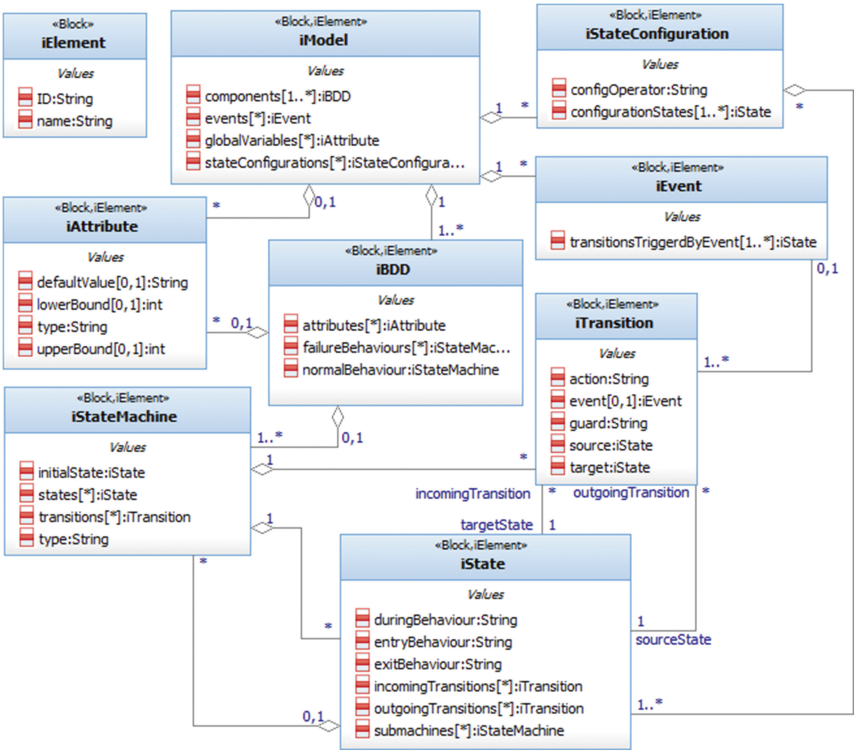


Fig. 3. The SysML intermediate model.

4 Transformations to NuSMV Input

In this section we provide an overview of the translation from SysML constructs into NuSMV input. We emphasise on the usefulness of the intermediate model in Fig. 3, as it enables a top-down, structured approach.

First, the *main* NuSMV module, corresponding to the **iModel** in Fig. 3 is implemented to contain the declaration of a series of modules, as given by its **iBDD** components. Each module in NuSMV is created as a "process". This enables the use of the "running" variable. NuSMV always chooses exactly one "process" for which "running" has the value *true*, and for all others the value *false*. This is useful to guarantee that only one module changes its state at a certain time. Then, all variables (attributes) are declared within the *main* module. The attributes are further initialised with the initial value from the associated element in the intermediate model, or if they do not have one, with the default values. The assignments are performed in the corresponding module of each variable. Relevant fragments of the NuSMV modules and variables declarations corresponding to the railway scenario in Example 1 are as follows:

```
Module main
[...]
  VAR gate: process Gate(self);
  VAR car: process Car(self);
  VAR train: process Train(self);

  VAR Gate_open: boolean;
  VAR open_gate_active: boolean;
```

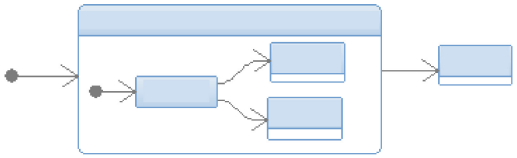
```
Module Gate(g)
[...]
 ASSIGN init(g.Gate_open) := TRUE

Module Car(g) [...]

Module Train(g) [...]
```

Translations of STM's, or **iStateMachine**s, is less straightforward as states and transitions are strongly interrelated. In the NuSMV code, the state, or the **iState** itself is integrated into the transition system. As illustrated later, state behaviours are translated into variable changes handled in the context of transition executions. Note that we combine the *during* behaviour of a state ("doAction") with its "exit" behaviour, as changes can not be modelled as happening over time.

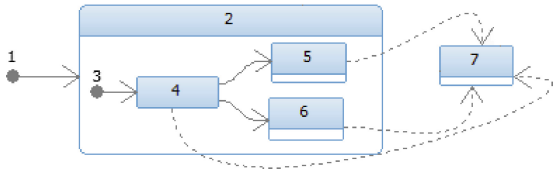
Moreover, in order to be identified within the NuSMV code, states are numbered in an ascending order. For the case of the gate, for instance, we can declare `VAR Gate_states: 0..10` in the *main* module. Additionally, recall that states in an STM can have a hierarchical structure. In our context, they can be *simple composite*. Assume an STM with three states, out of which one is an STM with four states, as in Fig. 4(a). By recursively apply the numbering procedure we assign, for example, values 1, 2 and 7 to the states of the STM as in Fig. 4(b).

Regarding the modelling of transitions out of submachines: in short, *initial* pseudostates and normal states in a submachine can exit the submachine behaviour at any time. Hence, we translate a transition (with target s) out of a submachine, to one transition (with target s) enabled in each state of the submachine. The original transition out of the submachine is then removed. This transition distribution procedure is represented via the dashed transitions in Fig. 4(b). The soundness of this approach is guaranteed by the fact that each newly added dashed transition inherits the "exit" behaviour and the "doAction" of the enclosing state (numbered 2 in our example). Moreover, each dashed transition has to execute the action corresponding to the transition out of submachine.

Recall that the transition structure in NuSMV is introduced via the "TRANS" keyword, followed by a boolean statement. This statement can be divided into three parts: (a) the transitions which can be executed when the module is running, together with statements regarding changed/unchanged variables, (b) the statement about what happens when the module is not running and (c) a statement to define when the module cannot perform any transitions and therefore has to stop running. In the context of (b), we assert that the variables do not change while the module is not running. Nevertheless, there is one exception to this: if there is a trigger to an event where a variable can change if the event is consumed by another module. Because of the way NuSMV parses a model, all variables that are not changed must be specified as such. This has to be done only for the variables of that module. A sketch-example of a transition system is as follows:



(a) 3-state STM with transition out of submachine.



(b) STM with numbered states and distributed transitions.

Fig. 4. Handling *simple composite* STM's.

```
TRANS
  -- (a) When the module is running
  (running &
   (next(g.event) = iTransitionID1 &
   (g.BlockName_states = currentState) &
   (g.BlockName_AttributeName = TRUE) & -- guard for the transition
   -- changed variables
    next(g.BlockName_states) = nextState &
    next(g.BlockName_AttributeName) = FALSE
   -- unchanged variables
    next(g.BlockName_AttributeName2) = g.BlockName_AttributeName2)

  -- (b) When the module is not running
  | !running &
    next(g.BlockName_states) = g.BlockName_states &
    next(g.BlockName_AttributeName) = g.BlockName_AttributeName
    next(g.BlockName_AttributeName2) = g.BlockName_AttributeName2)

  -- (c) When the module has to stop running
  & !(next(g.event) = iTransitionID1 & g.BlockName_states = currentState)
     -> !running)
```

In the listing above `g` stands for the constructor of the current module `BlockName`. In the railway example these can be represented, for instance, by `self` and `Gate`, respectively. `Block_Name_states` and `Block_AttributeName`/ `Block_AttributeName2` stand for the states and some attributes of the current module. These can be `Gate_states` and `Gate_open`, for instance. `currentState` is the number associated to the current state. `iTransitionID1` is the "ID" of a

transition. Recall from Sect. 3 that **iTransition** has an "ID" field, as it inherits from **iElement**. As expected, `g.event` denotes an *event*.

In NuSMV, events are translated as boolean variables. See, for instance, the variable declaration `VAR open_gate_active: boolean;` in the *main* module. Its value is set to *true* when a state or transition includes a trigger for the event in its behaviour, or to *false* after the execution of a transition that requires the event to be enabled.

An important aspect is that, in order to ensure module synchronisation via triggers, we have to enrich the NuSMV model. In case the module associated with the trigger is not running, the trigger variable has to be handled differently from normal variables because it has to be synchronised with the other modules that consume the trigger. This is done by specifying that the value of the trigger variable stays the same except when the next transition is the event transition:

```
TRANS [...]
  & ( ! (next(g.event) = triggeredEventName)
  -> next(g.triggeredEventName_active) = g. triggeredEventName_active )
```

Regarding the properties of the model captured by **iStateConfigurations** in Fig. 3: note that we are currently handling only *safety*, or *reachability*, specifications. Intuitively, these are of form "never the case to be in all of these states at once" or "never the case to be in at least one of these states". As expected, the former case is modelled via the logical "AND" operator, whereas the latter case is modelled using "OR". Consider, for a generic example, the following:

```
-- if the operator is AND
LTLSPEC G! ((Comp1_states = a) & (Comp2_states = b) & (Comp3_states = c))

-- if the operator is OR
LTLSPEC G! ((Comp1_states = a) | (Comp2_states = b) | (Comp3_states = c))
```

Above, `Comp1_states` can be, for instance, `Train_states`, whereas `a`, `b` and `c` denote state numbers.

5 SysMV-Ja at Work

Given a SysML model, the transformation to the corresponding NuSMV input via the intermediate model as described in Sects. 3 and 4, can be performed automatically using the SysMV-Ja tool. SysMV-Ja is a Java application with a simple graphical user interface that enables specifying the path to the XMI file of the SysML model, and the path of the output folder where the NuSMV-compatible input will be generated. The repository[2] containing the tool, instructions on how to use it, and the SysML models for the two case studies discussed in this paper can be accessed with the username "anon".

[2] https://svn.uni-konstanz.de/soft/SysMV-Ja/release.

5.1 Case Study: A Railway System

The first case study we consider is the railway system in Example 1, introduced for illustrative purposes. After generating the corresponding NuSMV code, we used the model checker to find a counterexample for the safety property "never car and train in the crossing at the same time". NuSMV successfully identified a counterexample. Even though the generated state space consists of approximately 700 000 states (including those associated to some extra bounded integers from `BlockName_states` definitions), the reachable states are approximately 300–in the order of what we expected:

```
NuSMV > print_reachable_states
###############################################################################
system diameter: 17
reachable states: 314 (2^8.29462) out of 684288 (2^19.3842)
###############################################################################
```

5.2 Case Study: An Airbag System

We further consider the transformation of an industrial size model of an airbag system taken from [2]. The architecture of this system was provided by TRW Automotive GmbH, and is schematically shown in Fig. 5. The airbag system can be divided into three major parts: sensors, crash evaluation and actuators. The system consists of two acceleration sensors (*main* and *safety*) for detecting front or rear crashes, one microcontroller to perform the crash evaluation, and an actuator that controls the deployment of the airbag. The deployment of the airbag is also secured by two redundant protection mechanisms. The Field Effect Transistor (FET) controls the power supply for the airbag squibs that ignite the airbag. If the Field Effect Transistor is not armed, which means that the FET-pin is not high, the airbag squib does not have enough electrical power to ignite the airbag. The second protection mechanism is the Firing Application Specific Integrated Circuit (FASIC) which controls the airbag squib. Only if it receives first an "arm" command and then a "fire" command from the microcontroller it will ignite the airbag squib which leads to the pyrotechnical detonation inflating the airbag.

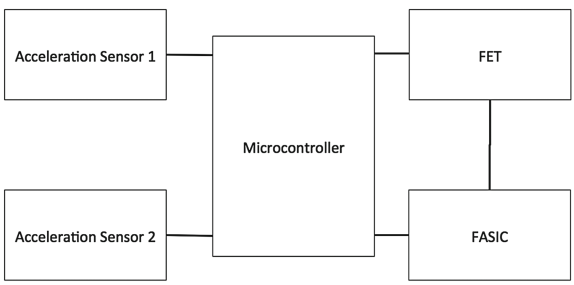


Fig. 5. Architecture of the airbag system.

Although airbags are meant to save lives in crash situations, they may cause fatal accidents if they are inadvertently deployed. This is because the driver may lose control of the car when an inadvertent deployment of the airbag occurs. It is a pivotal safety requirement that an airbag is never deployed if there is no crash situation. Intuitively, the corresponding safety property can be stated as "never no-crash and airbag deployed".

In short, the SysML model (also included in the repository of SysMV-Ja) consists of five BDD's and five STM's, each one associated to one component of the airbag system. The largest STM consists of twelve states, out of which two states with submachines. The remaining STM's enclose at most five states. When running the NuSMV model checker on the input generated via SysMV-Ja from the corresponding SysML modelling, we obtain a state space of size approximately 2^{10}, with about 1 000 reachable states that can be analysed for inadvertent deployment almost instantaneously.

6 Conclusions

In this paper we provide a model transformation from SysML block definition diagrams and state machines to NuSMV input, implemented in the automated tool SysMV-Ja. The procedure takes a file in XMI format, encoding the SysML model, and returns the corresponding NuSMV model provided in an `.smv` file. The proposed translation relies on an object-oriented intermediate model of SysML, thus making the whole approach more structured and easy to follow, possibly serving as a recipe for other model-transformations. The semantics of SysML exploited in this paper corresponds to the OMG specification [25]. We also discussed the results of model-checking models corresponding to a railway and an airbag system, generated with SysMV-Ja. The reachable state space did not suffer from exponential blowups.

Ideas for future work include the integration of an LTL property editor within SysMV-JA. At the moment, LTL specifications are added by hand at the end of the NuSMV input file. Apart from safety, we would also like to handle liveness properties as well.

We plan to investigate to what extent the translation procedure can be adapted to include other types of SysML diagrams such as activity charts, for instance.

Another interesting extension would be the integration of orthogonal submachines. For the time being, we only consider simple composite ones. Nevertheless, this kind of limitation can be overcome by providing an equivalent modelling of orthogonal submachines via multiple simple composite ones, synchronised via events.

Furthermore, the transformation of pseudostates can be enhanced in some ways. For optimisation purposes, the initial state can be replaced by its descendant, as initial states have at most one outgoing edge and can not have a behaviour. It is a minor enhancement, though, since it only decreases the state space minimally. Nevertheless, such an approach might make the generated NuSMV code smaller and therefore, easier to read and maintain. We would also like to allow

"exit" pseudo states. However, we foresee that this would change the handling of transitions out of submachines, as in Fig. 4.

We consider integrating a backward translation allowing to replay counterexamples found by NuSMV in a SysML tool. The formal correctness of the model would be another thing that is interesting to look into. For this, a formal semantics of the intermediate model might have to be created and the transformation rewritten as a set of functions/rules.

Last, but not least, we want to analyse the proposed approach for more case studies, and we want to perform efficiency studies as well. Moreover, we want to perform comparisons with other similar model transformation tools, regarding modularity, adaptability to different model-checkers, and portability.

Acknowledgements. We are grateful to the anonymous reviewers of CyPhy 2016, for their useful comments and observations.

References

1. Hugo/RT. https://www.informatik.uni-augsburg.de/en/chairs/swt/sse/hugort
2. Aljazzar, H., Fischer, M., Grunske, L., Kuntz, M., Leitner-Fischer, F., Leue, S.: Safety analysis of an airbag system using probabilistic FMEA and probabilistic counterexamples. In: QEST 2009, pp. 299–308. IEEE Computer Society (2009)
3. Alur, R., Yannakakis, M.: Model checking of hierarchical state machines. ACM Trans. Program. Lang. Syst. **23**(3), 273–303 (2001)
4. Ando, T., Yatsu, H., Kong, W., Hisazumi, K., Fukuda, A.: Translation rules of SysML state machine diagrams into CSP# toward formal model checking. IJWIS **10**(2), 151–169 (2014)
5. Baier, C., Katoen, J.: Principles of Model Checking. MIT Press, New York (2008)
6. Bhaduri, P., Ramesh, S.: Model checking of statechart models: Survey and research directions. CoRR, cs.SE/0407038 (2004)
7. Breu, R., Hinkel, U., Hofmann, C., Klein, C., Paech, B., Rumpe, B., Thurner, V.: Towards a formalization of the unified modeling language. In: Akşit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 344–366. Springer, Heidelberg (1997). doi:10.1007/BFb0053386
8. Cimatti, A., Clarke, E.M., Giunchiglia, F., Roveri, M.: NUSMV: a new symbolic model checker. STTT **2**(4), 410–425 (2000)
9. Clarke, E.M., Heinle, W.: Modular Translation of Statecharts to SMV. Technical report (2000)
10. de Andrade, E.C., Maciel, P.R.M., de Almeida Callou, G.R., e Silva Nogueira, B.C.: A methodology for mapping SysML activity diagram to time Petri Net for requirement validation of embedded real-time systems with energy constraints. In: ICDS 2009, pp. 266–271. IEEE Computer Society (2009)
11. Debbabi, M., Hassaïne, F., Jarraya, Y., Soeanu, A., Alawneh, L.: Verification and Validation in Systems Engineering - Assessing UML / SysML Design Models. Springer, Heidelberg (2010)
12. Ermel, C.: Visual modelling and analysis of model transformations based on graph transformation. Bull. EATCS **99**, 135–152 (2009)

13. Espinoza, H., Cancila, D., Selic, B., Gérard, S.: Challenges in combining SysML and MARTE for model-based design of embedded systems. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) ECMDA-FA 2009. LNCS, vol. 5562, pp. 98–113. Springer, Heidelberg (2009). doi:10.1007/978-3-642-02674-4_8
14. Friedenthal, S., Moore, A., Steiner, R.: A Practical Guide to SysML: Systems Modeling Language. Morgan Kaufmann Publishers Inc., San Francisco (2008)
15. Graf, S., Gérard, S., Haugen, Ø., Ober, I., Selic, B.: Modelling and analysis of real time and embedded systems – using UML. In: Kühne, T. (ed.) MODELS 2006. LNCS, vol. 4364, pp. 126–130. Springer, Heidelberg (2007). doi:10.1007/978-3-540-69489-2_16
16. Hansen, H.H., Ketema, J., Luttik, B., Mousavi, M.R., van de Pol, J.: Towards model checking executable UML specifications in mCRL2. ISSE **6**(1–2), 83–90 (2010)
17. Hause, M.: The SysML modelling language. In: Fifteenth European Systems Engineering Conference (2006)
18. Holzmann, G.J.: The SPIN Model Checker - Primer and Reference Manual. Addison-Wesley, Reading (2004)
19. Kamel, M., Leue, S.: VIP: a visual editor and compiler for v-Promela. In: Graf, S., Schwartzbach, M. (eds.) TACAS 2000. LNCS, vol. 1785, pp. 471–486. Springer, Heidelberg (2000). doi:10.1007/3-540-46419-0_32
20. Kent, S., Gaito, S., Ross, N.: A meta-model semantics for structural constraints in UML. In: Kilov, H., Rumpe, B., Simmonds, I. (eds.) Behavioral Specifications of Businesses and Systems. The Springer International Series in Engineering and Computer Science, vol. 523, pp. 123–139. Springer, New York (1999)
21. Lasalle, J., Bouquet, F., Legeard, B., Peureux, F.: SysML to UML model transformation for test generation purpose. ACM SIGSOFT Softw. Eng. Notes **36**(1), 1–8 (2011)
22. Leue, S., Ştefănescu, A., Wei, W.: An AsmL semantics for dynamic structures and run time schedulability in UML-RT. In: Paige, R.F., Meyer, B. (eds.) TOOLS EUROPE 2008. LNBIP, vol. 11, pp. 238–257. Springer, Heidelberg (2008). doi:10.1007/978-3-540-69824-1_14
23. Lilius, J., Paltor, I.: vUML: a tool for verifying UML models. In: ASE 1999, pp. 255–258. IEEE Computer Society (1999)
24. Mikk, E., Lakhnech, Y., Siegel, M., Holzmann, G.J.: Implementing statecharts in PROMELA/SPIN. In: WIFT 1998, pp. 90–101. IEEE Computer Society (1998)
25. OMG: OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1, August 2011
26. Plotkin, G.D.: A structural approach to operational semantics. J. Log. Algebr. Program. **60–61**, 17–139 (2004)
27. Beeck, M.: A formal semantics of UML-RT. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MODELS 2006. LNCS, vol. 4199, pp. 768–782. Springer, Heidelberg (2006). doi:10.1007/11880240_53