Correctness of an ATL Model Transformation from SysML State Machine Diagrams to Promela

Georgiana Caltais, Stefan Leue and Hargurbir Singh
University of Konstanz, Germany

Keywords: Model Transformation, ATL, SysML, Promela, Correctness, Observational Equivalence.

Abstract: In this paper we discuss the correctness of an ATL-based model transformation from the systems engineering modelling language SysML into Promela, the input language of the SPIN model checker. More precisely, we reduce showing the correctness of the transformation to showing a notion of what we refer to as observational equivalence of the SysML and the generated Promela models, respectively. This paves the way to a proof technique that could be further exploited in order to argue the correctness of model transformations from SysML to various model checkers, based on the observable actions generated by the systems under analysis.

1 INTRODUCTION

The implementation of a comprehensive Model-Based Systems and Software Engineering (MBSSE) methodology faces the challenge that, at least at the time of writing, a plethora of disconnected, incongruent and somewhat orthogonal modeling and analysis methods and tools are available in academia and industrial practice. This becomes particularly evident when considering modeling environments for safety-critical system architectures, such as SysML (Object Management Group, 2017a), on the one hand side, and formal analysis and verification tools, such as SPIN (Holzmann, 2004), NuSMV 2 (Cimatti et al., 2002), UPPAAL (Bengtsson et al., 1995) or PRISM (Kwiatkowska et al., 2004), just to mention some of them, on the other hand. Modeling languages supporting MBSSE, such as SysML, possess an, at best, semi-formally defined semantics. With few exceptions, the semantics that the tools supporting these languages offer are undocumented. Some of the tools supporting UML-style languages offer a limited form of commonly understood syntactic model representation, for instance using the XMI standard (Object Management Group, 2015). On the other hand, formal verification tools such as the ones cited above possess a somewhat rigorous semantics, which is necessary for the formal analysis to return meaningful and reproducible results. These semantics are sometimes openly defined, at least in parts, but sometimes deeply buried in the code of the analysis tool. Formal verification tools typically possess tool-specific, sometimes rather idiosyncratic input languages.

However, linking the worlds of modeling and formal analysis both syntactically and semantically is highly desirable. This is in particular true in the realm of MBSSE of safety-critical systems, where formal assurance cases for the safety of a system are of prime interest to industry, and society as a whole. We claim that to enable an automated translation and transformation process from MBSSE languages to models understood by formal analysis and verification tools is pivotal in order to obtain both syntactically and semantically dependable mappings between these model domains. In particular, we maintain that an excellent way to reach this goal is to use rule-based transformations, such as it is for instance supported by the Atlas Transformation Language (ATL) (Jouault et al., 2008). While ATL based model transformation rules define the syntactic correctness of these transformations, it is the objective of this paper to consider a framework for proving the semantic correctness of ATL based model transformations. In doing so, we will focus on the semantics of an SysML model as defined by its state machine (stm) diagrams. Stm diagrams are derived from the Statechart (Harel, 1987) notation and are widely used in UML and SysML based MBSSE.

Contributions of this Paper. Related work documented in (Kölbl et al., 2018) has considered an ATL-based translation from SysML into the input language of various model checking tools, including Promela (Holzmann, 2004), the input language of the widely popular SPIN explicit state model checker. The approach taken there defines ATL rules to translate SysML into an intermediate model, and then uses a hand-coded code generator for the translation from the intermediate model to Promela.

Caltais, G., Leue, S. and Singh, H.
Correctness of an ATL Model Transformation from SysML State Machine Diagrams to Promela.
DOI: 10.5220/0008968303600372
In *Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2020)*, pages 360-372
ISBN: 978-989-758-400-8; ISSN: 2184-4348

We take this idea further and define an ATL translation from SysML to Promela as follows. We first consider a suitable SysML source meta-model and a Promela target meta-model. Then, we define an ATL translation from SysML models conforming to the aforementioned source meta-model into Promela models conforming to the target meta-model. Note that both the meta-models and the models are in XMI format, as required by the ATL framework. Finally, we provide a simple and straightforward hand-written translation from the Promela models in XMI format into Promela code.

The key contribution of this paper is then to show that the ATL-based transformation above is correct. The novelty of our approach consists in reducing the correctness proof to showing that the SysML source models and their corresponding Promela target models are observationally equivalent. The sets of observations are derived in accordance with the operational semantics of SysML and Promela as in (Liu et al., 2013; Weise, 1997), for instance.

In our case, observational equivalence coincides with trace equivalence between the SysML models and the Promela models. Consequently, safety properties proven during state space exploration of the Promela models, including the presence of faults, also hold of the SysML model from which the Promela code was derived. Recall that safety properties can be proven or disproven using reachability analysis.

More broadly speaking, the approach presented here can be adapted to other ATL-based model to model transformations as long as the involved models can be given an observational semantics similarly to the one sketched in this paper.

Related Work. There is a wide plethora of work dedicated to the topic of model transformation.

The results in (Mikk et al., 1998) tackle the translation of statecharts into Promela sequential or parallel code. The main difference with our work is that the approach is not based on ATL-like rule based transformations, and no behavioural equivalence between the two models is shown. Performance analysis of the resulted code is performed.

(Anastasakis et al., 2007) proposes a declarative model transformations in a formalism called Alloy (Jackson, 2019). The main difference with our work is that Alloy does not have any built in notion of state machine, hence the approach can only be used to reason about static properties of the transformation.

Graph transformation based approaches to showing correctness of model transformations in terms of behavioural equivalence were proposed in (Dyck et al., 2015; Engels et al., 2008), for instance. ATL was not exploited in the aforementioned works. In (Dyck et al., 2015), correctness is addressed in the context of a more simplistic case study on transforming lifelines into automata. In (Engels et al., 2008), the authors show trace equivalence between activity diagrams and the corresponding translated TAAL (Rensink, 2006) programs.

The work in (Lano et al., 2015) introduces a systematic language-independent framework and techniques for model transformation verification. As a verification technique for statemachines transformations, the authors suggest showing preservation of traces by proving preservation of a *suitable* invariant formula.

For a more comprehensive study on formal verification techniques for model transformations we refer to the survey in (Amrani et al., 2015), for instance.

Closer related to our approach in this paper, we address the following contributions. The work in (Latella et al., 1999) proposes a transformation from a behavioural subset of UML Statechart diagrams into Promela. The correctness of the approach relies on a translation solely based on the operational semantics of hierarchical automata. The benefits of using a modular, syntax-based framework such as ATL are, nevertheless, not exploited.

The verification of ATL transformations was also addressed in (Büttner et al., 2012). The main difference with our work is that the correctness is established based on an additional encoding of ATL into OCL. The latter is further used to analyze different properties of the transformations, but not a semantic equivalence between the source and target models.

In (Troya and Vallecillo, 2011) the authors introduce a formal semantics of the ATL language itself, using rewriting logic semantics implemented in Maude (Clavel et al., 2003). The Maude toolkit is then exploited to reason about the application of the ATL rules, and syntactic properties of the target models. In our approach, the focus is on reasoning about the semantic equivalence of the source and its translation into the target model.

Structure of the Paper. In Section 2 we provide an overview of SysML, Promela and ATL. In Section 3 we discuss the ATL transformation from SysML into Promela. Section 4 sets the basis for the observational semantics of SysML and Promela. The correctness of the ATL transformation is shown in Section 5. Conclusions and pointers to future work are provided in Section 6.

2 PRELIMINARIES

In this section we provide a brief overview of the SysML, Promela and ATLAS Transformation Language aspects relevant for our work.

2.1 SysML

The Systems modeling language (SysML) (Object Management Group, 2017a), is a standardized, semi-formal language for defining static and dynamic aspects of systems as well as their communication behavior. SysML uses a subset of the UML (Object Management Group, 2017b). It encompasses, amongst others, state-machine (stm) diagrams and, at the same time, extends UML with block definition and internal block definition diagrams.

In this paper, we focus on stm diagrams like the one in Fig. 1. A stm diagram consists of states and labelled transitions defining the behaviour of a system component such as StateMachine *C* in Fig. 1. States play the role of control locations for the system. In Fig. 1, the current control location, or the active state is *B* (note that *B* is coloured darker). Moreover, some states are called "initial" and, intuitively, serve as entry points within the behaviour of a stm. See, for example, states *Q*, *A* or *R* in Fig. 1. Each transition is labelled with a guard that needs to be satisfied, and an action that is performed whenever transiting between the states. A guard in SysML can be a trigger waiting for the occurrence of an event, or a check on a variable value. An action can be the initialization of an event or a variable value change. For an example, the generic label *a–b* in Fig. 1 could be of shape $\texttt{true}/[x \leftarrow v]$, for the trivial guard `true` and the action associating to variable x value v. Each component in SysML can have multiple behaviors, but we restrict each component to one behavior for our work. Hence, we have one stm for each component. A system modelled by means of several SysML components is understood as a set of associated stms operating concurrently. Note that stms can be hierarchical, i.e., they consist of subsequent stms (such as *Q* in Fig. 1). We refer to hierarchical stms as complex, and single-state stms such as *D* in Fig. 1 as single, or simple. In this paper, we limit ourselves to at most two-level stms.

The stms semantics is "run-to-completion". The semantics is defined with respect to an active state, which is the current control location of the model. If a complex state is active, it means that one of the sub-states is active along with the complex state. We restrict the behavioral information to transitions and do not use the SysML semantics of state entry and exit behaviors. Moreover, the model can only have

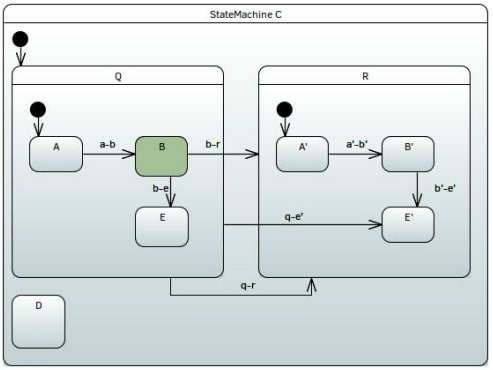

Figure 1: SysML Component.

one action per transition. SysML execution semantics defines an event pool that stores events which are available for "consumption". The basic semantics defines the execution environment for the selection of an event from the event pool, but the precise mechanism is not specified. The run-to-completion semantics states that whenever the action of the current transition is completed, SysML will select an event from the event pool to trigger a transition. The communication among processes in SysML is defined based on First-In-First-Out (FIFO) queues, with the ability to lose messages if the queue is full. In our work, we consider message-based (a)synchronous communication, with simple messages lacking parameters and return values.

We support activity diagrams as a way to input the communication information. The usage of activity diagrams is limited to *sendSignalAction* and *callOperationAction* with only one action per diagram.

2.2 Promela

Promela has similar syntax to the C programming language, with guarded commands and communication primitives. Processes in Promela are defined as `proctypes` and are concurrent in nature. Intuitively, such processes correspond to SysML components. Promela syntax uses variables that can be defined with both local and global scope. Additionally, channels are constructs used according to the intended operational semantics to send and receive messages between processes. These channels are FIFO queues and have user-defined size. A channel of size 0 cannot save the message and is called synchronous: both the sender and receiver processes need to be ready to perform the communication. Otherwise, the sender has to wait. An asynchronous channel can become full and the message is discarded, similarly to the SysML behavior for full channels.

Listing (1) illustrates a toy Promela process en-

coding the SysML component in Fig. 1. The correspondence is easy to follow: `proctype C` defines a Promela process corresponding to the aforementioned component *C*. Each state within *C* is associated with a flattened state in Promela. For instance, state *A* in *C* can be interpreted as state `Q_A_ini` in Promela. The naming convention is made in order to preserve the static information on the "ancestry" and "initial nature" of *A*. Namely, *A* is within the hierarchical stm *Q* and is also the initial state of *Q*. `Q_A_ini` has transitions without any guard. State label changes are described with the `goto` statement. `break` defines states with no behaviour.

```
1  proctype C{
2   Q_A_ini:
3      do
4       :: a-b -> goto Q_B;
5       :: q-r -> goto R_A'_ini;
6       :: q-e' -> goto R_E';
7      od
8   Q_B: [...]
9   Q_E: [...]
10  R_A'_ini:
11   do
12      :: a'-b' -> R_B';
13   od
14  R_B': [...]
15  R_E':
16   do
17    :: break;
18   od
19  D:
20   do
21    :: break;
22   od
23 }
```

Listing 1: Promela Code of the SysML Component in Fig. 1.

2.3 ATL

ATL (Jouault et al., 2008) is a domain specific language which was designed to provide a framework for rule-based model-to-model transformation of XMI models. ATL has two benefits over conventional programming languages: due to its rule-based nature it provides syntactic correctness of the source and target models, and it supports uniqueness of the transformation rules which means that each type of source element will be mapped only once per target element. ATL uses the concept of meta-models for representing the structure of the models. A meta-model is a special model that contains the meta-definitions of all model elements used in the transformation. Intuitively, a meta-model can be seen as a grammar, whereas the corresponding model is given according to the rules

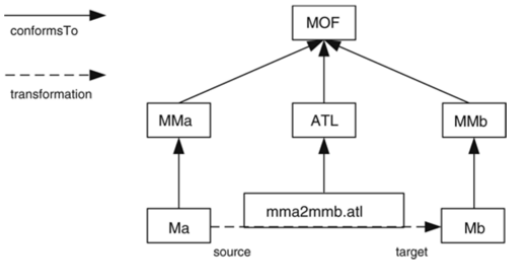

Figure 2: Overview of the ATL Transformation.

of the grammar. ATL requires meta-models for both the source and the target models. The transformation rules can refer to elements from the source and target models.

The idea behind an ATL transformation is sketched in Figure (2). ATL reads a source model Ma which has to conform to a source meta-model MMa. If the source model does not conform to the source meta-model, ATL throws an error advising that the source model is ill-defined. The ATL transformation in `mma2mmb.atl` describes how a source model can be converted to a target model Mb which conforms to a target meta-model MMb. All meta-models must conform to the meta-meta-model *Meta Object Facility* (MOF). ATL defines its own syntax but also borrows some functions from Java and OCL. All transformation rules in ATL are a set of mappings between the source model elements and the target model elements with imperative operations performed on the source elements.

```
1 rule exampleATL{
2  from s: MMSysML!State(
3   s.name.startsWith('A')
4  )
5  to t: MMPromela!State(
6   name <- s.name, ID <- s.ID
7  )
8 }
```

Listing 2: ATL-code.

To illustrate the ATL translation mechanism, in Listing 2 we define a sample rule which accepts `State` elements from a SysML source model where the name of the element starts with `A`. This set of elements are then transformed to `State` elements of the Promela target model where the `name` and `ID` attributes are transferred.

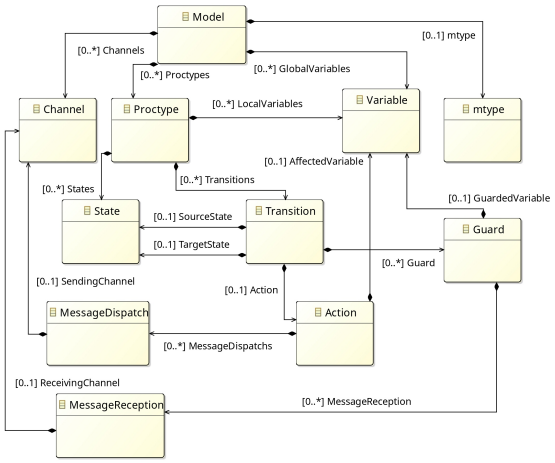

Figure 3: Class Diagram of the Promela Target Meta-Model.

3 ATL TRANSFORMATION FROM SysML TO Promela

As previously mentioned, the ATL transformation relies on the meta-models of the source and target, respectively. In short, ATL parses the SysML model and checks whether it conforms to the UML 2.5.0 meta-model of Eclipse, encoded in XMI format. On the Promela side, the equivalent class-diagram representation of the XMI target meta-model is given in Fig. 3. The referred class diagram encodes models formed from several `Proctype` elements, which consist of states and transitions labelled with actions and guards, in the spirit of the Promela description in Section 2.2.

The ATL transformation performs a syntactic conversion of the SysML source model into the Promela target model. The rules specify how the source model elements must be processed and translated to the target model. It is important to observe that the conversion from SysML into Promela is based on a "flattening" procedure, which maps hierarchical state-machines into corresponding `proctypes`. Intuitively, the mapping transfers states and transitions as follows.

States:

(1) A simple state from SysML is translated into a Promela state with the same name.

(2) All simple states within hierarchical SysML stms are translated into Promela states respecting the naming convention: the Promela names should encode the "ancestry" and possibly the "initial nature" information.

Transitions:

(3) If the source state of a SysML transition is simple, then (a) if the SysML target is simple, then the transition is translated into Promela directly or (b) if the SysML target is complex, then the transition is translated with the Promela target as the initial state of the target hierarchical state-machine preserving the naming convention.

(4) If the source state of the SysML transition is complex, then (a) if the SysML target is simple, then corresponding transitions originating from each state of the hierarchical source state-machine are translated into Promela (again, the naming convention is fulfilled) or (b) if the SysML target is complex, transitions as above are created with target as the initial state of the hierarchical target state-machine (naming convention preserved).

The flattening procedure can be easily identified based on the SysML source model in Listing 3 and the derived Promela target model in Listing 4. Note that both models are given in XMI format as required by ATL, and provide corresponding encodings of the component in Fig. 1.

The XMI format provides entries for the `Components` (line `1` in Listing 3 and, line `1` in Listing 4) and entries for the `InitialState` of the possibly hierarchical stms (e.g., lines `2, 6` in Listing 3, and line `2` in Listing 4). Each `State` entry has a name and an `ID` that coincides with the name, in our examples. States have outgoing transitions specified via their `ID`'s within the `outTrans` field. Additionally, each transition is defined in terms of its label, `Source` and `Target`, as expected.

Simple states such as `D` are translated directly from SysML to Promela, as mentioned in (1) above. We refer to line `3` in both Listing 3 and Listing 4 for the corresponding entries.

Complex states are flattened as described in (2) above. For instance, the complex state `Q` starting at line `4` in Listing 3 is represented via the three enclosed simple states `Q_A_ini`, `Q_B` and `Q_E` at lines `4, 8` and `13` in Listing 4. Observe that their `ID`'s are in accordance with the naming convention.

For an example of a "simple to complex" transition as in $(3)(b)$ above, we refer to the `b-r` labelled transition at line `15` in Listing 3. Its Promela correspondent is encoded at line `25` in Listing 3. Note how in Promela, the target becomes the initial state of the state-machine `R`.

An example of "complex to simple" transition as in $(4)(a)$ is the `q-e'` labelled transition at line `32` in Listing 3. On the Promela side, it is captured via the transition at line `37` Listing 4. Observe how, the source in Promela becomes the initial state of the hierarchical stm `Q`.

A "complex to complex" transition as in $(4)(b)$ is given in Listing 3 at line 30. It corresponds to the q-r labelled transition between the hierarchical stms Q and R. The translation to Promela is realized in terms of the q-r labelled transitions at lines 31, 33, 35 in Listing 4. Observe that, all targets are the initial state of R, whereas the sources are the stms enclosed within Q.

```
 1  <Component C ID = C>
 2     <InitialState Q/>
 3     <State D ID = D/>
 4     <State Q ID = Q
 5            outTrans = Qq-rR, Qq-e'E
                '>
 6      <InitialState A/>
 7      <State A ID = A outTrans = Aa-
          bB/>
 8      <State B ID = B
 9             outTrans = Bb-eE, Bb-rR
                />
10      <State E ID = E/>
11      <Trans a-b ID = Aa-bB
12             Source = A Target = B/>
13      <Trans b-e ID = Bb-eE
14             Source = B Target = E/>
15      <Trans b-r ID = Bb-rR
16             Source = B Target = R/>
17     <State/>
18     <State R ID = R>
19      <InitialState A'/>
20      <State A' ID = A'
21               outTrans = A'a'-b'B
                    '/>
22      <State B' ID = B'
23               outTrans = B'b'-e'E
                    '/>
24      <State E' ID = E'/>
25      <Trans a'-b' ID = A'a'-b'B'
26               Source = A' Target
                    = B'/>
27      <Trans b'-e' ID = B'b'-e'E'
28               Source = B' Target
                    = E'/>
29     <State/>
30     <Trans q-r ID = Qq-rR
31               Source = Q Target =
                    R/>
32     <Trans q-e' ID = Qq-e'E'
33               Source = Q Target =
                    E'/>
34  </Component>
```

Listing 3: The XMI Source Model for the Component in Fig. 1.

We continue by providing some details about the actual ATL transformation rules encoding the mapping described above. Each of the discussed rules is invoked with respect to the name-space of the elements. Moreover, each rule works on a type of

```
 1  <Component C ID = C>
 2   <InitialState Q_A_ini/>
 3   <State D ID = D/>
 4   <State Q_A_ini ID = Q_A_ini
 5     outTrans = Q_A_inia-bQ_B,
 6                Q_A_iniq-rR_A'_ini,
 7                Q_A_iniq-e'R_E'/>
 8   <State Q_B ID = Q_B
 9     outTrans = Q_Bb-eQ_E,
10                Q_Bb-rR_A'_ini,
11                Q_Bq-rR_A'_ini,
12                Q_Bq-e'R_E'/>
13   <State Q_E ID = Q_E
14     outTrans = Q_Eq-rR_A'_ini,
15                Q_Eq-e'R_E'/>
16   <State R_A'_ini ID = R_A'_ini
17     outTrans = R_A'_inia'-b'R_B'/>
18   <State R_B' ID = R_B'
19     outTrans = R_Ba'-b'R_E'/>
20   <State R_E' ID = R_E'/>
21   <Trans a-b ID = Q_A_inia-bQ_B
22      Source = Q_A_ini Target = Q_B
          />
23   <Trans b-e ID = Q_Bb-eQ_E
24     Source = Q_B Target = Q_E/>
25   <Trans b-r ID = Q_Bb-rR_A'_ini
26     Source = Q_B Target = R_A'_ini
          />
27   <Trans a'-b' ID = R_A'_inia'-b'
        R_B'
28     Source = R_A'_ini Target = R_B
          '/>
29   <Trans b'-e' ID = R_Bb'-e'R_E'
30     Source = R_B' Target = R_E'/>
31   <Trans q-r ID = Q_A_iniq-rR_A'
        _ini
32     Source = Q_A_ini Target = R_A'
          _ini/>
33   <Trans q-r ID = Q_Bq-rR_A'_ini
34     Source = Q_B Target = R_A'_ini
          />
35   <Trans q-r ID = Q_Eq-rR_A'_ini
36     Source = Q_E Target = R_A'_ini
          />
37   <Trans q-e' ID = Q_A_iniq-e'R_E'
38     Source = Q_A_ini Target = R_E
          '/>
39   <Trans q-e' ID = Q_Bq-e'R_E'
40     Source = Q_B Target = R_E'/>
41   <Trans q-e' ID = Q_Eq-e'R_E'
42     Source = Q_E Target = R_E'/>
43  </Component>
```

Listing 4: The ATL-generated XMI Target Model.

source element and can be used to generate one or more target elements. The transformation reads a SysML model and extracts information which is targeted through the rules. We navigate the model using subsequent calls to (lazy) sub-rules: the model rule will call the component rule for each component in the source model. The transformation completes

```
1  lazy rule C2P{
2   from s: MMSysML!Component
3   to t: MMPromela!Proctype(
4   \\collect component attributes
5    ID <- s.id, [...]
6   )
7   do{
8    \\iterate over component states
9      for(p in s.normalBehaviour.
           states){
10      if(p.subMachines.
            oclIsUndefined()){
11      \\call sub-rule for simple
            states
12        t.States <- thisModule.S2S
              (p);
13      }
14      else{
15       for(q in p.subMachines.
             states){
16       \\call sub-rule for complex
             states
17         t.States <- thisModule.
             SMS2S(q);
18        }
19     }
20  } [...]
21  \\iterate over component states
22  for(p in s.normalBehaviour.
      transitions.union(thisModule.
      TransitionList)){
23   \\call sub-rule for transitions
24   if(p.source)
25     t.Transitions <- thisModule.
         T2T(p);
26  } [...] }
```

Listing 5: Component to Proctype Transformation.

when each relevant element is processed. The ATL code in the listings of this section has been simplified for presentation. The rule in Listing 5, for instance, is used to transfer information about a component of the SysML model and create an equivalent `proctype` in the Promela model. The rule extracts information about the simple states, complex states, and transitions. This information is further sent to sub-rules for processing.

Listing 6 defines a rule which is used to transfer states which are of simple type and mark the initial state, according to the naming convention. This rule is a sub-rule of the component rule and operates on elements received from its parent rule as parameters (see line 12 in Listing 5).

In a similar fashion, we defined a rule which for transferring SysML complex states into Promela. This rule receives the children of the complex state as parameters and names them according to the naming convention. We also defined a rule which is used to

```
1  lazy rule S2S{
2   from s: MMSysML!State
3   to t: MMPromela!State(
4   \\collect state attributes
5    ID <- s.id
6   )
7   do{
8    \\check if state is initial
9    if(s.refImmediateComposite().
       initialState.name = s.name){
10    \\annotate name with 'initial'
        info
11    t.Name <- s.name.concat('_init
        ');
12    t.IsInitial <- true;
13   } } }
```

Listing 6: State to State Transformation.

transfer transitions along with the source and target state information. This rule changes the source and target states based on their type. So, if source is complex, it will generate multiple transitions with source state's children as source. If target is complex then target is changed to the initial state of the target state.

Our automated ATL-based tool for generating Promela models from SysML specifications can be found at: https://github.com/SysMLATL/SysMLPromelaTransformation

4 ENRICHING XMI MODELS WITH OBSERVATIONAL SEMANTICS

In this section we provide an observational semantics of SysML and Promela models, on top of their XMI encodings discussed in Section 3. Intuitively, this semantics "collects" the sequences of actions that can be executed by the SysML and Promela models, according to associated operational semantics as in (Liu et al., 2013; Weise, 1997), for instance.

At this point we would like to emphasize on the uniformity of the proposed observational semantics. The latter enables computing the possible observable executions of both SysML and Promela models by using the same sets of semantic rules. This, despite the fact that SysML allows the specification of hierarchical state-machines, whereas Promela does not.

The observational semantics will be further exploited in order to show the correctness of the ATL transformation in terms of a notion of observational equivalence of the SysML and the associated Promela models.

We write *obs* to denote the set of traces w that can be observed by executing a certain SysML/Promela

model. Executions w are words over SysML/Promela actions. As usual, we write ε to denote the empty word, and $w_1.w_2$ to represent the concatenation of two words w_1 and w_2.

In order to mimic variable and channel updates, we introduce a notion of memory (*mem*) that associates values v to variables and, respectively, channels x. Such an assignment is denoted by $mem[x \leftarrow v]$. A lookup function is denoted by $lkp(mem,x)$. For simplicity, and without loss of generality, we assume that all variables x within a model are "allocated" in the memory and initialised with the default value $\perp$.

The observational semantics will be given via rules:

$$\frac{p_1 \quad \ldots \quad p_n}{c_1 \quad \ldots \quad c_m}$$

consisting of premises $p_1,\ldots,p_n$ and conclusions $c_1,\ldots,c_n$ which capture sequences of observations that simulate model executions and, whenever appropriate, memory updates. Due to typesetting constraints, we sometimes syntactically split the semantic rules into "shorter" rules such as, for instance:

$$\frac{p_1 \quad \ldots \quad p_i}{c_1 \quad \ldots \quad c_j}$$

$$\frac{p_{i+1} \quad \ldots \quad p_n}{c_{j+1} \quad \ldots \quad c_m}$$

Furthermore, in the subsequent sections we define a series of notational conventions enabling the uniform handling of the aforementioned XMI representations and their observational semantics. More precisely, we will provide equivalent, but more compact and modular representations of concurrent components and their state-machines.

4.1 An Observational Semantics of SysML

As seen in Section 3, SysML source models are given as a "configuration" encoding a set of concurrent components C_i, for $i \in \{1,\ldots,N\}$. We write $C_1 \mid \ldots \mid C_N$, or $|_{i\in\{1,\ldots,N\}}\ C_i$ for these components. Naturally, each such component is defined in terms of a set of stms, a current state $curr_C$ and an initial state ini_C. When proving correctness of our transformation, we let $curr_C$ "evolve" according to the observational semantics of SysML provided later on in this paper. Initially, the current state $curr_C$ is set to ini_C.

A stm of C is given by: its *ID*, its *type* (that is either complex and denotes a SysML hierarchical stm, or simple and denotes a state), a set of sub-machines (possibly empty, if *type* is simple), a set of transitions and an initial state *ini*. A transition is defined over the *ID* of its source (that corresponds to the *ID* of the stm) and the *ID* of its target. More formally, a state-machine SM_i of a SysML component C can be given as:

$$\langle ID_i, type_i, \bigcup_{j\in\{1,\ldots,m\}} \{SM_{i,j}\}, \bigcup_{l\in\{1,\ldots,r\}} \{ID_i \xrightarrow{\alpha_l} ID_l^{i,t}\}, ini_i\rangle \tag{1}$$

If $type_i$ is complex, then each $SM_{i,j}$ stands for a simple sub-machine of SM_i. Otherwise, $\bigcup_{j\in\{1,\ldots,m\}}\{SM_{i,j}\} = \emptyset$, as expected. We define $ID(SM_i) = ID_i$, $type(SM_i) = type(ID_i) = type_i$ and $ini(SM_i) = ini(ID_i) = ini_i$ for a stm as in (1). Recall that, in accordance with the SysML semantics, *ini* is the initial state of the sub-machine for hierarchical stms. Naturally, for simple sub-machines (or states), *ini* is defined as the associated *ID*.

In short, we write $SM_i \in C$ in order to refer to a (hierarchical) stm of C as in (1). Subsequently, we write $SM_{i,j} \in SM_i$ in order to refer to a simple stm of SM_i.

For an example, the complex stm Q in Fig. 1 is equivalently represented as:

$$SM_Q = \langle Q, complex,\ \{SM_A, SM_B, SM_E\},\ \{Q \xrightarrow{q-e'} E', Q \xrightarrow{q-r} R\}, A\rangle \tag{2}$$

The simple sub-machine SM_B, for instance, is given as:

$$SM_B = \langle B, simple, \emptyset, \{B \xrightarrow{b-r} R,\ B \xrightarrow{b-e} E\}, B\rangle \tag{3}$$

A component C is captured as:

$$C = [\bigcup_{i\in\{1,\ldots,n\}} SM_i, curr_C, ini_C] \tag{4}$$

with SM_i as in (1), for $i \in \{1,\ldots,n\}$.

For an example, component C in Fig. 1 is:

$$C = [\{SM_Q, SM_R, SM_D\}, B, Q]. \tag{5}$$

Given C as in (4) and SM_i as in (1), we write $tr_C^1(ID_i)$ for the set of "first-level" transitions:

$$\bigcup_{l\ \in\ \{1,\ldots,r\}} \{ID_l^{i,s} \xrightarrow{\alpha_l} ID_l^{i,t}\}.$$

Intuitively, these transitions originate in hierarchical stms, or simple state machines at the top-most level in a component C. Symmetrically, we write $tr_C^2(ID_i)$, for the "second-level" transitions:

$$\bigcup_{j\ \in\ \{1,\ldots,m\}} tr_C^1(ID(SM_{i,j})).$$

These are transitions that originate in simple states of the hierarchical stms of C. For the simplicity of notation, we write tr^i_C to denote the set of all first, respectively, second-level transitions of a component C, for $i = 1$, respectively $i = 2$.

For an example, the set of first-level transitions tr^1_C in Fig. 1 is:

$$\{Q \xrightarrow{q-r} R, Q \xrightarrow{q-e'} E'\}.$$

The set of second-level transitions is:

$$\{B \xrightarrow{b-r} R, A \xrightarrow{a-b} B, B \xrightarrow{b-e} E, A' \xrightarrow{a'-b'} B', B' \xrightarrow{b'-e'} E'\}.$$

We also define a "children" function that extracts the *ID*'s of the sub-machines of a given stm:

$$ch(ID_i, C) = \begin{cases} \bigcup_{SM_{i,j} \in Set_{SM}} \{ID(SM_{i,j})\} & \text{if } Set_{SM} \neq \emptyset \\ \{ID_i\} & \text{[owise]} \end{cases}$$

$$\text{if } \exists SM_i \in C \text{ s.t } ID(SM_i) = ID_i \text{ and } Set_{SM} = \cup_{j=\{1,\ldots,m\}} \{SM_{i,j} \mid SM_{i,j} \in SM_i\} \tag{6}$$

With these ingredients at hand, we are ready to provide the semantics of the SysML stms relevant to our purpose. Let C, C_i, C_j range over components of a SysML "configuration" $C_1 \mid \ldots \mid C_N$. Let *obs* refer to the current set of executions, and *mem* stand for the content of the memory.

We proceed by first formalising asynchronous communication. Intuitively, a transition $ID^s \xrightarrow{g/\alpha} ID^t$, where α is an action, can be triggered whenever (one of the children of) its source matches the current state of the enclosing component C, and the guard g is satisfied. As a consequence, the observation set is enriched with the corresponding action α and the current state becomes the initial state of the transition's target as in (7).

$$\frac{ID^s \xrightarrow{g/\alpha} ID^t \in tr^i_C \quad curr_C \in ch(ID^s) \quad lkp(mem, g) = true}{curr_C := ini(ID^t) \quad obs := \cup_{w \in obs}\{w.\alpha\}} \quad i \in \{1,2\} \tag{7}$$

Variable or channel updates in the context of asynchronous communication are handled in a similar fashion, with the additional observation that the corresponding update is stored in the memory as shown in (8).

$$\frac{ID^s \xrightarrow{g/x \leftarrow v} ID^t \in tr^i_C \quad curr_C \in ch(ID^s) \quad lkp(mem, g) = true}{curr_C := ini(ID^t) \quad obs := \cup_{w \in obs}\{w.x \leftarrow v\} \quad mem[x \leftarrow v]} \quad i \in \{1,2\} \tag{8}$$

Synchronous communication is formalised as in (9). The transition is triggered when its guard g is satisfied and the synchronous action α can see its synchronous reception $g(\alpha)$. In other words, we write $g(\alpha)$ as a syntactic sugar for a guard g "waiting" to synchronise with its counterpart α (here, by α we denote both an action, and a variable/channel update). Note that the synchronising transitions can be at different levels in the corresponding (hierarchical) stms.

$$\frac{ID^s_i \xrightarrow{g/\alpha} ID^t_i \in tr^i_C \; curr_C \in ch(ID^s_i)}{curr_C := ini(ID^t_i)}$$

$$\frac{ID^s_j \xrightarrow{g(\alpha)/\beta} ID^t_j \in tr^j_{C'} \;\; curr_{C'} \in ch(ID^s_j)}{curr_{C'} := ini(ID^t_j)}$$

$$\frac{lkp(mem, g) = true}{obs := \cup_{w \in obs}\{w.\alpha.\beta\}} \quad i, j \in \{1,2\} \tag{9}$$

Synchronous variable or channel updates is handled in a similar fashion, with an additional memory update, as formalised in (10).

$$\frac{ID^s_i \xrightarrow{g/x \leftarrow v} ID^t_i \in tr^i_C \; curr_C \in ch(ID^s_i)}{curr_C := ini(ID^t_i)}$$

$$\frac{ID^s_j \xrightarrow{g(x \leftarrow v)/\beta} ID^t_j \in tr^j_{C'} \;\; curr_{C'} \in ch(ID^s_j)}{curr_{C'} := ini(ID^t_j)}$$

$$\frac{lkp(mem, g) = true}{obs := \cup_{w \in obs}\{w.(x \leftarrow v).\beta\} \quad mem[x \leftarrow v]} \quad i, j \in \{1,2\} \tag{10}$$

4.2 An Observational Semantics of Promela

Similarly to the approach in Section 4.1, we base the semantic rules of Promela on configurations $\overline{C}_1 \mid \ldots \mid \overline{C}_N$ encoding the components $\overline{C}_i$ within the XMI target model of the ATL transformation. The main difference with the SysML counterpart is that each such component encloses a set of simple stms. The latter are "flattened" versions of the (hierarchical) stms specified in SysML, derived as described in Section 3.

Recall that the flattening procedure entails simple stms with *ID*'s built according to the naming convention encoding their "ancestry" and "initial nature", whenever the case. We formalize the naming convention by means of the function in (11). Assume a SysML component C (in essence, encoded as in (4)), and let ID_k refer to a stm of C (s.a, Q, A or D in Fig. 1). The naming convention maps *ID* from the SysML XMI to $nmc(ID_k, C)$ in the Promela XMI, where:

$$nmc(ID_k, C) = \begin{cases} ID(SM_i)\_ID_k\_ini \\ \quad \text{if } \exists SM_i \in C, \exists SM_{i,j} \in SM_i \text{ s.t.} \\ \quad ID(SM_{i,j}) = ID_k \text{ and} \\ \quad ID(SM_{i,j}) = ini(SM_i) \\ \\ ID(SM_i)\_ID_k \\ \quad \text{if } \exists SM_i \in C, \exists SM_{i,j} \in SM_i \text{ s.t.} \\ \quad ID(SM_{i,j}) = ID_k \text{ and} \\ \quad ID(SM_{i,j}) \neq ini(SM_i) \\ \\ ID_k \\ \quad \text{[owise]} \end{cases} \tag{11}$$

In words:

- If the stm ID_k refers to a simple (and initial) state within a hierarchical stm, then the ancestry-based flattening keeps track of the enclosing parent (and the initially information) as well. This is the case of $ID_k = B$ and $ID_k = A$.
- If the stm is a stm not enclosed within another machine, then there is no flattening. This is the case of $ID_k = D$.

Assume a SysML component C, consisting of n (hierarchical) state-machines SM_i, for $i = \{1, \ldots, n\}$, each of each consisting of a (possibly empty) set of simple state-machines $SM_{i,j}$, for $j = \{1, \ldots, m\}$, naturally captured as:

$$\begin{array}{rcl} C & = & [\bigcup_{i\in\{1,\ldots,n\}} SM_i, curr_C, ini_C] \\ SM_i & = & \langle ID_i, type_i, \ \bigcup_{j\in\{1,\ldots,m\}}\{SM_{i,j}\}, \\ & & \quad \bigcup_{l\in\{1,\ldots,r\}}\{ID_i \xrightarrow{\alpha_i} ID_l^{i,t}\}, ini_i\rangle \\ SM_{i,j} & = & \langle ID_{i,j}, simple, \emptyset, \bigcup_{k\in\{1,\ldots,o\}}\{ID_{i,j} \xrightarrow{\alpha_k} ID_k^t\}, ID_{i,j}\rangle \end{array} \tag{12}$$

The corresponding "flattened" component $\overline{C}$ within the XMI target model is:

$$\overline{C} = [\bigcup_{\substack{i\in\{1,\ldots,n\} \\ j\in\{1,\ldots,m\} \\ SM_{i,j}\in SM_i}} (\{\langle \overline{ID}, simple, \emptyset, T, \overline{ID}\rangle\}), curr_{\overline{C}}, ini_{\overline{C}}] \tag{13}$$

where $\overline{ID} = nmc(ini(SM_{i,j}), C)$ and T stands for the set of transitions

$$\bigcup_{\substack{k=\{1,\ldots,o\} \\ ID_{i,j} \xrightarrow{\alpha_k} ID_k^t \in tr^1(ID_{i,j})}} \{\overline{ID} \xrightarrow{\alpha_k} nmc(ini(ID_k^t), C)\} \tag{14}$$

$$\bigcup_{\substack{l=\{1,\ldots,r\} \\ ID_i \xrightarrow{\alpha_l} ID_l^t \in tr^1(ID_i) \\ ID_{ch} \in ch(ID_i, C) \\ \overline{ID} = nmc(ID_{ch}, C)}} \{\overline{ID} \xrightarrow{\alpha_l} nmc(ini(ID_l^t), C)\} \tag{15}$$

$curr_{\overline{C}} = nmc(curr_C, C)$ and $ini_{\overline{C}} = nmc(ini(ini_C), C)$.

For an example, the target XMI in Listing 4 is equivalently represented as the component:

$$\begin{array}{rcl} \overline{C} & = & [\{ \\ & & \langle Q\_A\_ini, simple, \emptyset, T_{Q\_A\_ini}, Q\_A\_ini\rangle, \\ & & \langle Q\_B, simple, \emptyset, T_{Q\_B}, Q\_B\rangle, \\ & & \langle Q\_E, simple, \emptyset, T_{Q\_E}, Q\_E\rangle, \\ & & \langle R\_A'\_ini, simple, \emptyset, T_{R\_A'}, R\_A'\rangle, \\ & & \langle R\_B', simple, \emptyset, T_{R\_B'}, R\_B'\rangle, \\ & & \langle R\_E', simple, \emptyset, T_{R\_E'}, R\_E'\rangle, \\ & & \langle D, simple, \emptyset, T_D, D\rangle \\ & & \} \\ & & Q\_B, Q\_A\_ini] \end{array} \tag{16}$$

where, for instance, the transitions of the simple stm Q_A_ini are

$$\begin{array}{rcl} T_{Q\_A\_ini} & = & \{Q\_A\_ini \xrightarrow{a-b} Q\_B, \\ & & Q\_A\_ini \xrightarrow{q-r} R\_A'\_ini, \\ & & Q\_A\_ini \xrightarrow{q-e'} R\_E'\}. \end{array}$$

The "observational" semantics of Promela can be now defined as in (7)–(10), based on a configuration

$$\overline{C}_1 \mid \ldots \mid \overline{C}_N,$$

a set *obs* of executions, and the memory *mem*.

5 A PROOF OF CORRECTNESS

In this section we formalize the correctness of the ATL transformation in Section 3, in terms of the so-called observational equivalence of the associated SysML and Promela XMI's as in Sections 4.1 and 4.2.

Let *MMS*, *MS*, *MMT* and *MT* refer to the source meta-model, source model, target meta-model and target model, respectively, as in Section 3. We write

$$ATL(MMS, MS, MMT) = MT$$

as a shorthand for the aforementioned transformation.

Assume *mem* a memory-like structure whose variables could be initially set to the default value $\perp$. Furthermore, consider *obs* a set of observations. We write

$$\begin{array}{l} |_{i\in\{1,\ldots,N\}} \tilde{C}_i \ \ mem \ \ obs \xrightarrow{((7)-(10))^*} \\ |_{i\in\{1,\ldots,N\}} \tilde{C'}_i \ \ mem' \ \ obs' \end{array} \tag{17}$$

to denote the evolution of components $\tilde{C}_i$, for $i \in \{1, \ldots, N\}$, into $\tilde{C'}_i$, via the repeated application of the semantic rules (7)–(10), given the memory *mem* and a set of observations *obs*.

Intuitively, we say that a set of concurrent components $|_{i\in\{1,\ldots,m\}} C_i$ are *observational equivalent* with the concurrent components $|_{i\in\{1,\ldots,n\}} \overline{C}_i$ whenever,

given the same initial memory configuration, they produce the same observations under the repeated application of derivations $\xrightarrow{((7)-(10))^*}$.

Definition 1 (Observational Equivalence)**.** *Consider a memory configuration mem and a set of observations obs. The concurrent components* $|_{i\in\{1,\ldots,m\}} C_i$ *are* observational equivalent *with the concurrent components* $|_{i\in\{1,\ldots,n\}} \overline{C}_i$ *whenever the following holds:*

(a) If

$$|_{i\in\{1,\ldots,m\}} C_i \;\; mem \;\; obs \xrightarrow{((7)-(10))^*} |_{i\in\{1,\ldots,m\}} C'_i \;\; mem' \;\; obs'$$

then there exists a derivation

$$|_{i\in\{1,\ldots,n\}} \overline{C}_i \;\; mem \;\; obs \xrightarrow{((7)-(10))^*} |_{i\in\{1,\ldots,n\}} \overline{C}'_i \;\; mem' \;\; obs'$$

(b) Symmetrically.

We write $|_{i\in\{1,\ldots,m\}} C_i \approx |_{i\in\{1,\ldots,n\}} \overline{C}_i$ to denote the observational equivalence above.

Theorem 1 (Correctness)**.** *Assume*

$$ATL(MMS, MS, MMT) = MT.$$

Let $|_{i\in\{1,\ldots,N\}} C_i$ *be the concurrent components associated to MS, and* $|_{i\in\{1,\ldots,N\}} \overline{C}_i$ *be the components of MT. Then:*

$$|_{i\in\{1,\ldots,m\}} C_i \approx |_{i\in\{1,\ldots,n\}} \overline{C}_i.$$

Theorem 1 follows as a direct consequence of Lemma 1.

Lemma 1. *Assume*

$$ATL(MMS, MS, MMT) = MT.$$

Let $|_{i\in\{1,\ldots,N\}} C_i$ *denote the concurrent components corresponding to MS, and* $|_{i\in\{1,\ldots,N\}} \overline{C}_i$ *denote the components of MT. Consider the initial memory configuration mem and the initial set of observations obs. Then:*

(a) If

$$|_{i\in\{1,\ldots,m\}} C_i \;\; mem \;\; obs \xrightarrow{((7)-(10))^*} |_{i\in\{1,\ldots,m\}} C'_i \;\; mem' \;\; obs'$$

then there exists a derivation

$$|_{i\in\{1,\ldots,n\}} \overline{C}_i \;\; mem \;\; obs \xrightarrow{((7)-(10))^*} |_{i\in\{1,\ldots,n\}} \overline{C}'_i \;\; mem' \;\; obs'$$

(b) Symmetrically.

And, moreover,

$$nmc(curr_{C'_i}, C'_i) = curr_{\overline{C}'_i}, \forall i \in \{1,\ldots,N\}.$$

for (a) *and* (b) *above.*

Observe that, apart from observational equivalence, Lemma 1 guarantees equality of memory contents and a one-to-one correspondence between the current states of the SysML and Promela components.

Lemma 1 – Proof Sketch. The proof of both cases "(a)" and "(b)" is by induction on the length of the semantic rules derivation $((7)-(10))^*$. □

Remark 1. *A direct consequence of Lemma 1 is that observational equivalence coincides with trace equivalence between the SysML models and the Promela models. Hint: assume derivations* $\xrightarrow{((7)-(10))^*}$ *of length* 1.

6 DISCUSSION

In this paper we presented an ATL transformation from SysML hierarchical state-machines into Promela, the input language of the SPIN model-checker. The transformation is implemented in an automated tool that can be found at https://github.com/SysMLATL/SysMLPromelaTransformation

The ATL transformation is syntactic in nature, whereas state-machines are inherently semantic. Thus, the main challenge of our work consisted in filling the gap between the two worlds of syntax and semantics, and proving the proposed transformation correct. We chose to formulate the correctness result in terms of a notion of observational equivalence between the SysML source models and the Promela target models. The interplay between syntax and semantics can be better observed in the context of Lemma 1. The proof of Lemma 1 shows that both models display the same observational behaviour by exploiting the fact that the associated current states are in a one-to-one correspondence which preserves the naming convention characteristic to the flattening procedure encoded within the ATL transformation.

We believe that our approach to proving correctness can be easily adapted for similar ATL transformations into the input languages of other model-checkers. As future work we consider tackling more such transformations.

It is beyond the scope of this paper to cover the entire SysML. In this work, we mainly focused on devising a methodology for proving semantic correctness of the ATL syntactic transformation. Additional aspects such as entry/exit behaviours, for instance, could be handled in a rather straightforward fashion by simply inserting additional states and transitions

labelled accordingly within the Promela model. Further features of the SysML state-machines, such as multiple-layer hierarchies, or time events, are left as future work.

ACKNOWLEDGEMENTS

The work of Georgiana Caltais and Hargurbir Singh was supported by the DFG project "CRENKAT", proj. no. 398056821.

REFERENCES

Amrani, M., Combemale, B., Lucio, L., Selim, G. M. K., Dingel, J., Traon, Y. L., Vangheluwe, H., and Cordy, J. R. (2015). Formal verification techniques for model transformations: A tridimensional classification. *Journal of Object Technology*, 14(3):1:1–43.

Anastasakis, K., Bordbar, B., and Küster, J. (2007). Analysis of model transformations via alloy. In Baudry, B., Faivre, A., Ghosh, S., and Pretschner, A., editors, *Proceedings of the workshop on Model-Driven Engineering, Verification and Validation (MoDeVVA 2007), Nashville, TN (USA)*, volume 5002, pages 47–56. Springer.

Bengtsson, J., Larsen, K. G., Larsson, F., Pettersson, P., and Yi, W. (1995). UPPAAL - a tool suite for automatic verification of real-time systems. In *Hybrid Systems*, volume 1066 of *Lecture Notes in Computer Science*, pages 232–243. Springer.

Büttner, F., Egea, M., Cabot, J., and Gogolla, M. (2012). Verification of ATL transformations using transformation models and model finders. In Aoki, T. and Taguchi, K., editors, *Formal Methods and Software Engineering - 14th International Conference on Formal Engineering Methods, ICFEM 2012, Kyoto, Japan, November 12-16, 2012. Proceedings*, volume 7635 of *Lecture Notes in Computer Science*, pages 198–213. Springer.

Cimatti, A., Clarke, E. M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., and Tacchella, A. (2002). Nusmv 2: An opensource tool for symbolic model checking. In *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer.

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., and Talcott, C. L. (2003). The Maude 2.0 system. In Nieuwenhuis, R., editor, *Rewriting Techniques and Applications, 14th International Conference, RTA 2003, Valencia, Spain, June 9-11, 2003, Proceedings*, volume 2706 of *Lecture Notes in Computer Science*, pages 76–87. Springer.

Dyck, J., Giese, H., Lambers, L., Schlesinger, S., and Glesner, S. (2015). Towards the automatic verification of behavior preservation at the transformation level for operational model transformations. In Dingel, J., Kokaly, S., Lucio, L., Salay, R., and Vangheluwe, H., editors, *Proceedings of the 4th Workshop on the Analysis of Model Transformations co-located with the 18th International Conference on Model Driven Engineering Languages and Systems (MODELS 2015), Ottawa, Canada, September 28, 2015.*, volume 1500 of *CEUR Workshop Proceedings*, pages 36–45. CEUR-WS.org.

Engels, G., Kleppe, A., Rensink, A., Semenyak, M., Soltenborn, C., and Wehrheim, H. (2008). From UML activities to TAAL - towards behaviour-preserving model transformations. In Schieferdecker, I. and Hartman, A., editors, *Model Driven Architecture - Foundations and Applications, 4th European Conference, ECMDA-FA 2008, Berlin, Germany, June 9-13, 2008. Proceedings*, volume 5095 of *Lecture Notes in Computer Science*, pages 94–109. Springer.

Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274.

Holzmann, G. J. (2004). *The SPIN Model Checker - primer and reference manual*. Addison-Wesley.

Jackson, D. (2019). Alloy: a language and tool for exploring software designs. *Commun. ACM*, 62(9):66–76.

Jouault, F., Allilaire, F., Bézivin, J., and Kurtev, I. (2008). ATL: A model transformation tool. *Sci. Comput. Program.*, 72(1-2):31–39.

Kölbl, M., Leue, S., and Singh, H. (2018). From SysML to Model Checkers via Model Transformation. In Gallardo, M. and Merino, P., editors, *Model Checking Software - 25th International Symposium, SPIN 2018, Malaga, Spain, June 20-22, 2018, Proceedings*, volume 10869 of *Lecture Notes in Computer Science*, pages 255–274. Springer.

Kwiatkowska, M. Z., Norman, G., and Parker, D. (2004). Probabilistic symbolic model checking with PRISM: a hybrid approach. *STTT*, 6(2):128–142.

Lano, K., Clark, T., and Rahimi, S. K. (2015). A framework for model transformation verification. *Formal Asp. Comput.*, 27(1):193–235.

Latella, D., Majzik, I., and Massink, M. (1999). Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. *Formal Asp. Comput.*, 11(6):637–664.

Liu, S., Liu, Y., André, É., Choppy, C., Sun, J., Wadhwa, B., and Dong, J. S. (2013). A formal semantics for complete UML state machines with communications. In Johnsen, E. B. and Petre, L., editors, *Integrated Formal Methods, 10th International Conference, IFM 2013, Turku, Finland, June 10-14, 2013. Proceedings*, volume 7940 of *Lecture Notes in Computer Science*, pages 331–346. Springer.

Mikk, E., Lakhnech, Y., Siegel, M., and Holzmann, G. J. (1998). Implementing statecharts in PROMELA/SPIN. In *2nd Workshop on Industrial-Strength Formal Specification Techniques (WIFT '98), October 20-23, 1998, Boca Raton, FL, USA*, pages 90–101. IEEE Computer Society.

Object Management Group (2015). XML metadata interchange, specification 2.5.1. http://www.omg.org/spec/XMI/.

Object Management Group (2017a). OMG Systems Modeling Language, Specification 1.5. http://www.omg.org/spec/SysML.

Object Management Group (2017b). Unified Modelling Language, Specification 2.5.1. http://www.omg.org/spec/UML.

Rensink, A. (2006). Model checking quantified computation tree logic. In Baier, C. and Hermanns, H., editors, *CONCUR 2006 - Concurrency Theory, 17th International Conference, CONCUR 2006, Bonn, Germany, August 27-30, 2006, Proceedings*, volume 4137 of *Lecture Notes in Computer Science*, pages 110–125. Springer.

Troya, J. and Vallecillo, A. (2011). A rewriting logic semantics for ATL. *Journal of Object Technology*, 10:5: 1–29.

Weise, C. (1997). An incremental formal semantics for PROMELA. In *In Proceedings of the Third SPIN Workshop, SPIN97*.