Translating UML State Machines to Coloured Petri Nets Using Acceleo: A Report

Étienne André, Mohamed Mahdi Benmoussa, Christine Choppy
Université Paris 13, Sorbonne Paris Cité, LIPN, CNRS, UMR 7030, F-93430, Villetaneuse, France

UML state machines are widely used to specify dynamic systems behaviours. However its semantics is described informally, thus preventing the application of model checking techniques that could guarantee the system safety. In a former work, we proposed a formalisation of non-concurrent UML state machines using coloured Petri nets, so as to allow for formal verification. In this paper, we report our experience to implement this translation in an automated manner using the model-to-text transformation tool Acceleo. Whereas Acceleo provides interesting features that facilitated our translation process, it also suffers from limitations uneasy to overcome.

Keywords: model transformation, software engineering, model checking, system safety, UML

1 Introduction

UML [10] became the *de facto* standard for modelling systems, and features a very rich syntax. UML behavioural state machine diagrams (SMDs) are transition systems used to express the behaviour of dynamic systems in response to external interactions. Although UML is widely used in the industry, its semantics is not formally expressed, hence not directly suitable for applying formal methods guaranteeing the system safety. However, a formal semantics can be given, for instance by translation to a formalism, such as coloured Petri nets (CPNs) [6]. CPNs offer a detailed view of the process with a graphical representation. One could argue that an intermediate formalism (as it is the case in this work) does not need to be clear or graphical. However, we do believe that it is important and useful to provide a readable presentation that we can check against what we expect, and that could also suggest some improvements or some properties to check. CPNs also benefit from powerful tools (such as CPN Tools [12]) to test and check the model. Translating an SMD to a CPN will allow us to formally guarantee the system safety by verifying it against properties.

In a previous work [1], we described a translation of SMDs into CPNs. These SMDs are non-concurrent, in the sense that synchronisation of events, fork and join pseudostates, are discarded. However, we do consider history pseudostates (that remember where to go back, e.g. in case of failure), do/entry/exit behaviours, hierarchy of machines with inter-level transitions, and variables appearing in guards and behaviours.

Objectives. The translation introduced in [1] was presented in an algorithmic form, but no implementation allowing an automated translation was performed, and that work did not contain precise indications to actually implement the translation. In this paper, we present an attempt to perform an automated translation of SMDs into CPNs (following the transformation rules of [1]) using model transformation techniques. The lack of formal metamodel for CPNs together with the difficulty to build one drove us to use model-to-text transformation techniques, and we chose to use the model-to-text transformation tool Acceleo[1]. Whereas Acceleo provides interesting features that eased our translation process, it also

[1] `http://www.eclipse.org/acceleo/`

J. Pang and Y. Liu (Eds.): 3rd International Workshop
on Engineering Safety and Security Systems 2014 (ESSS 2014)
EPTCS 150, 2014, pp. 1–7, doi:10.4204/EPTCS.150.1

© É. André, M. M. Benmoussa & C. Choppy
This work is licensed under the
Creative Commons Attribution License.

suffers from limitations that made our translation not entirely straightforward. We report here our experience with Acceleo, and we point out both its advantages and limitations, while suggesting some new features that would make it more powerful.

Related Work. Verification of SMDs has been often tackled (see, e.g. [3] for a survey). Some approaches directly give UML a semantics (e.g. [8]). Many approaches translate UML specifications into an intermediate model of some model checker, e.g., SMV [9] or SPIN [5]. Other approaches (e.g. [11, 7, 4]) use CPNs as an intermediate formal model, and use CPN Tools to analyse the generated CPN. The translation of [4], where a formalisation of SMDs using CPNs is proposed, is probably the closest to the technique presented in [1], although the sets of UML syntactic features taken into account are different. We are not aware of previous attempts of translations of UML state machines to (extensions of) Petri nets using model transformation techniques.

2 Preliminaries

UML State Machines. We briefly introduce some of the SMD concepts with an example of a CD player (from [13]) depicted in Figure 1a. This SMD is composed of two composite states (viz. BUSY and NONPLAYING), and each of them contains two simple states. BUSY has an entry behaviour, and PLAYING has a "do" behaviour. Recall that entry/exit/do behaviours are behaviours performed when entering/exiting/being in a state, respectively. Transitions may involve events, guards and behaviours, where (global) variables may appear (e.g. `track`). BUSY has a history pseudostate (depicted with "H").

In our setting, we design UML state machines using the Eclipse Modeling Framework (EMF).

Coloured Petri Nets. Petri nets are bipartite graphs with two kinds of nodes, viz. places and transitions. Directed arcs connect places to transitions and transitions to places. Coloured Petri nets (CPNs) [6] extend Petri nets with types. Although this extension is mainly syntactic, it greatly increases the abstraction in complex Petri nets by making their representation more compact. Figure 1b presents an example of a CPN that corresponds to the translation of the CD player of Figure 1a, following the rules of [1]. In Figure 1a state BUSY has a final state (represented in Figure 1b by Busy^F), a history state (represented by Busy^H) and an entry state (represented by transition/place FTS – that stands for "find track start"). States PLAYING, PAUSED, CLOSED and OPEN are represented by PL, PA, Closed, and Open respectively. State NONPLAYING is not represented because it has no final state, history pseudostate or behaviours.

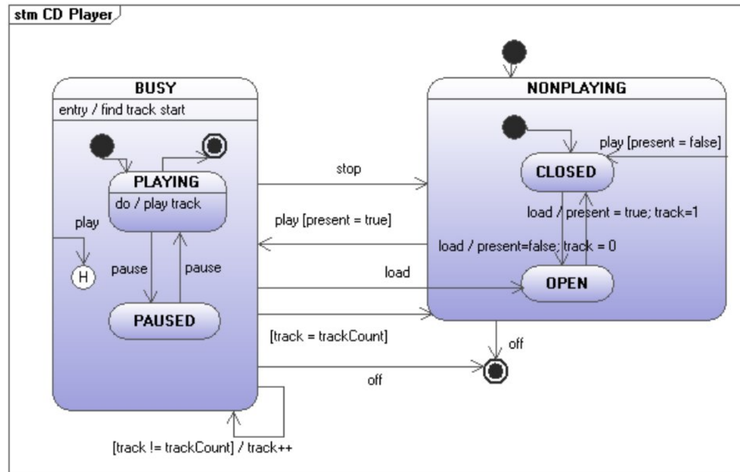
CPN Tools [12] is a powerful tool for modelling and verifying coloured Petri nets. Whereas it features a user-friendly graphical interface, one can also design CPN models in an XML-based input syntax.

Acceleo. Acceleo is a model-to-text transformation tool (its development started in 2006); its main purpose is to implement code generators. An Acceleo program requires a metamodel and a model compliant with that metamodel, from which it generates source code. Acceleo comes in the form of a user-friendly Eclipse plugin, hence easy to install, and features syntax highlighting tools, completion, etc.

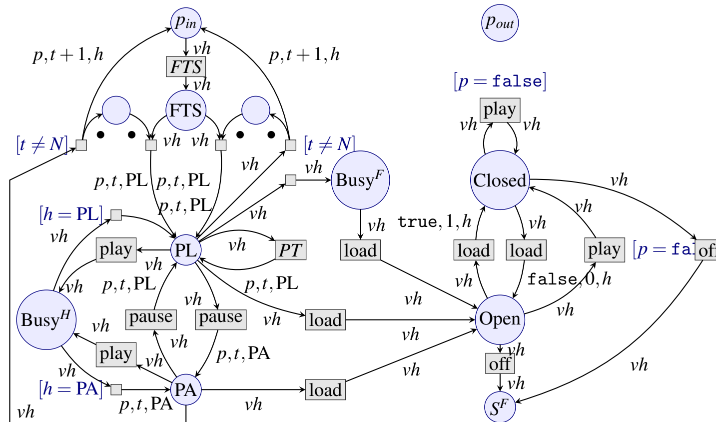
The metamodel and the model are defined using EMF, which makes Acceleo compatible with other tools based on EMF. In particular, all models created from the ATL (Atlas Transformation Language) framework [2] can be directly passed to Acceleo.

Acceleo is based on the notion of *template*, that will implement the transformation rules. Each rule in a template will map an element from the metamodel (and hence the model) to the text to be generated. An advantage is that the structure of the Acceleo templates will directly reflect the structure of the generated code; in particular, the destination model is directly generated, with no need for postprocessing.

A main feature of Acceleo is that the generated text is mixed with Acceleo syntax. Figure 2a describes function *SupEn*() (used in [1]). This function takes as parameters a state `s` and the global state machine,



(a) Specification using an SMD



(b) Translation into a CPN (partial scheme)

Figure 1: An example of a CD player [13]

and writes to the output file the entry behaviour of `s`. This function mixes control structures to find the entry behaviour with the CPN code to be written to the output file. Another example of the use of functions is given in Figure 2b. The result `s.Name` is not written to the output file, but returned to the main template (that calls the function) as text. After that, we can extract the result from the text.

Acceleo syntax offers some control structures such as `if` conditions, `for` loops, and `let` structures that allow to define variables, though with a limited scope. However, to the best of our knowledge, Acceleo syntax features neither global variables, nor more complicated user-defined data structures (hash tables, lists, etc.), nor user-defined functions.

3 Implementation of the Translation

3.1 Structure of the Translation

We first briefly recall the structure of the translation of [1]. Each state (resp. behaviour) in the SMD is translated into a place (resp. transition) of the CPN. Then, each transition in the SMD is translated to a

```
[template public SupEn1(s : State, pere :
    State, as : StateMachine)]
  [if (s.Entry().contains('true'))]
    [if (pere.Entry().contains('true'))]
      <arc id="ArcNRootSENS[s.Name/][s.Id/]"
                orientation="PtoT"
                order="1">
            <posattr x="0.000000"
                    y="0.000000"/>
      ...
[/template]
```

(a) Mixing Acceleo code with destination syntax

```
[template public substates(s : State, as :
    StateMachine)]
  [if (s.isSimple = true)]
    [s.Name/]
  [else]
    [for (x : State | as.State)]
      [if (x.Parent = s.Name)]
        [substates(x, as)/]
      [/if]
    [/for]
  [/if]
[/template]
```

(b) Simulating functions with templates

Figure 2: Example of Acceleo code

set of new places and transitions in the CPN. Various arcs are also added by the translation. The translation of [1] comes in the form of three algorithms translating the SMD states, transitions and history pseudostates, respectively. These algorithms use several loops enumerating states, transitions, etc. Furthermore, they use predefined functions (that retrieve, e.g. the substates of an SMD state, or the next exit behaviour to be executed after a given exit behaviour) and a mapping system that records in particular the translation of a given SMD state into a CPN place. Indeed, since several transitions in the SMD can have the same source and/or the same destination state, the translation process requires to be able to map (or store) easily the translation of a given SMD state into a CPN place.

3.2 Toward Model-to-Text Transformation

When implementing our translation, we had two main solutions: either implement a home-made translating tool (including lexers, parsers, and transformation functions based on an abstract syntax), or use model-to-model (M2M) transformation techniques.

M2M transformation techniques feature the following advantages: standard concepts and technologies, tools integrated in user-friendly frameworks (often Eclipse), and no need to write error-prone and time-consuming lexers, parsers, and printers. The main drawback of M2M transformation techniques is that they usually require a metamodel for both the source model (UML state machines in our case) and the destination model (coloured Petri nets in our case). Whereas the metamodel of UML state machines is provided by the OMG [10], there is unfortunately no widely recognised metamodel for coloured Petri nets. A solution could have been to translate UML state machines to non-coloured, classical Petri nets ("place/transition"), for which metamodels exist, but this solution is not satisfactory, since the translation would be much more cumbersome, and the resulting model much larger and less easy to understand.

3.3 Model-to-Text Transformation Using Acceleo

The main advantage of model-to-text transformation techniques is that they do not require a destination metamodel. As a consequence, Acceleo is adapted to our situation of a destination formalism without a widely recognised metamodel. Hence, we considered our destination CPN Tools model (described in an XML-like syntax) as a *text* document, hence suitable to be generated automatically using Acceleo.

Our first step has been to set up a metamodel for UML state machines, given in Figure 3. We mostly reused the metamodel specified by the OMG [10], with minor modifications: indeed, we simplified this metamodel to the non-concurrent case, and performed a few minor changes (addition of some attributes

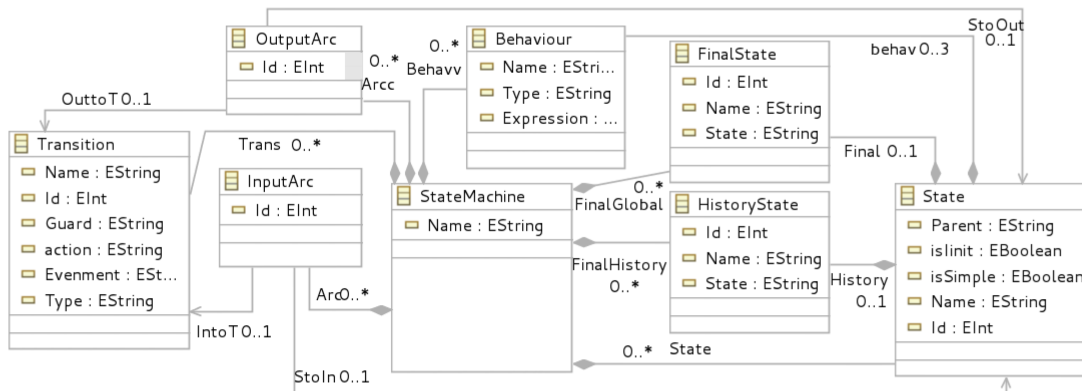


Figure 3: Metamodel of UML state machines used in our translation

that can be automatically derived from other attributes, and aim at easing the transformation). In short, the system is represented using a global state machine (class `StateMachine`). Then, each state machine is composed of states (classes `State` and `FinalState`), transitions (class `Transition`), behaviours (class `Behaviour`), pseudostates (class `HistoryState`) and arcs between states and transitions (classes `InputArc` and `OutputArc`).

3.4 Advantages and Limitations of Acceleo

Our implementation of the translation mechanism of [1] using Acceleo has been greatly facilitated by the user-friendly features of Acceleo: integration into the powerful Eclipse environment, and the use of the metamodel, avoiding us to deal with complex parsing mechanisms. However, the translation also suffered from several drawbacks coming from the features (or lack of features) of Acceleo.

Functions. First, the lack of user-defined functions has been a major problem while implementing our translation. Note that, even when changing the form of the algorithms of [1], the notion of function, in the sense of an operation depending on some input, would still be required by the translation. However, the notion of template offered by Acceleo partially compensates the lack of function, at the price of a less clear code and lower performances. Above all, Acceleo templates can only output text, which constrains us to rewrite our functions so that they output text, which also requires us to then parse their result.

Figure 2b presents our translation of the $substate(s)$ function of [1], that returns the list of all substates (in a recursive manner) of an SMD state s, with a recursive template `substate`. This solution is correct (in an algorithmic point of view) but certainly not optimal. First, since there is no possibility to store the information computed previously (see below), one needs to iterate not only on the substates of a given state (since this information cannot be stored), but on all the SMD states. This iteration is performed several times, leading to a cost exponential in the number of SMD states. Second, this template returns a list of states in the form of a text; hence, checking that a state indeed belongs to this “list” will require us to use the `substring` operation, hence again costing time.

Global variables and data structures. A second (and stronger) limitation of Acceleo is the impossibility to store information in the form of (global) variables or, more generally, of user-defined data structures such as arrays, lists, hash tables, etc. This is a strong limitation, at least on an efficiency point of view: we were able to overcome this limitation, but at the cost of repeated recomputation of useful information, since it cannot be stored.

3.5 Summary of our Experience

We managed to implement using Acceleo the whole translation of SMDs into CPNs following the rules defined in [1] (except the third algorithm of [1], i.e. the translation of history states, that was not implemented due to lack of time[2]). Using the resulting tool UML2CPN, we could successfully translate several simple examples of SMDs, thus allowing for verification using CPN Tools. The Acceleo code of our translation is available on a dedicated Web page[3]. Nevertheless, we do not consider our overall experience of model transformation using Acceleo as entirely successful, for several reasons.

First, due to some exponential loops in our Acceleo code, we think that our tool may be very inefficient in the case of large SMD examples. This said, for the (small) examples we have translated using UML2CPN, the translation time was always below 1 second.

Second, the lack of features (in particular global variables, functions and data structures) have been limitations during our implementation of the translation, may be not in a theoretical point of view, but in a practical point of view (leading to a larger and less clear code).

Third, Acceleo, as a code generator, is intrinsically dependent on the destination syntax. Hence, a future change in the syntax of CPN Tools would lead us to considerably change our Acceleo templates. One could argue that we could generate using Acceleo an abstract destination syntax instead of the CPN Tools concrete syntax; but one of the advantages of model-to-text transformation is actually to directly generate the destination syntax without the need for an additional parsing tool. Similarly, we could have separated the destination syntax from the rest of the Acceleo code, by using templates that would either generate destination concrete code, or encode functions without any concrete code. Whereas this would have been technically possible, that way of implementing the translation would have been in contradiction with a main feature of Acceleo which is to mix the templates with the destination code.

For these reasons, we are not entirely convinced that our tool will be easy to maintain; in particular, a change in the CPN Tools syntax, or an improvement of the translation of [1] (e.g. to add concurrency), may result in very large changes in our Acceleo code.

4 Final Remarks and Perspectives

We presented here a report on an automated translation of UML state machines to coloured Petri nets using Acceleo. The resulting tool UML2CPN allows to automatically generate a CPN that can then be analysed by CPN Tools and hence formally prove or disprove the original system safety. Due to the problems we encountered when using Acceleo, we think that our tool may not be easily maintained.

Whereas it is understandable that Acceleo was limited to only some features to keep it simple, some extensions would greatly increase its expressive power, and ease its usability and dissemination. In particular, adding global variables, functions and user-defined data structures would certainly make easier the use of Acceleo even beyond its primary purpose (code generation). Another problem we met is the lack of extensive documentation and examples online, maybe due to a still relatively small community.

Perspectives. We were not entirely convinced by our experience of model-to-text translation using Acceleo, and we will likely rewrite our translation mechanism using a home-made compiler-like tool, so as to be able to easily adapt it to future improvements of the translation (e.g. extension of the translation of [1] to concurrent and/or timed state machines).

[2] This work has been carried during Mohamed Mahdi Benmoussa's Master thesis at LIPN.

[3] `http://lipn.univ-paris13.fr/~benmoussa/UML2CPN/`

Also note that the resulting CPN may be simplified in some cases. For instance, some places and transitions added by the translation may sometimes be unnecessary. These simplifications, that are beyond the scope of this paper, could help to speed up the automated verification of the resulting CPN.

Finally, although it goes beyond of the scope of this paper, a challenging future work is to formally prove the equivalence between the original SMD and the resulting CPN. Of course, the problem is that the OMG does not formally define a formal semantics for SMDs. However, we could reuse the operational semantics that we recently proposed for SMDs [8], and define a trace equivalence taking into account active states, behaviours and events.

Acknowledgement. This work also benefited from Taieb Ben Niha's experience regarding a similar translation from UML activity diagrams to coloured Petri nets, during his internship at LIPN. We would also like to thank Alessandro Tiso for his suggestions regarding Acceleo.

References

[1] Étienne André, Christine Choppy & Kais Klai (2012): *Formalizing non-concurrent UML state machines using colored Petri nets*. ACM SIGSOFT Software Engineering Notes 37(4), pp. 1–8. Available at http://doi.acm.org/10.1145/2237796.2237819. UML&FM 2012.

[2] ATLAS, LINA & INRIA (2006): *ATL : Atlas Transformation Language user manual, version 0.7*. Report.

[3] Purandar Bhaduri & Sethu Ramesh (2004): *Model Checking of Statechart Models: Survey and Research Directions*. CoRR cs.SE/0407038. Available at http://arxiv.org/abs/cs.SE/0407038.

[4] Christine Choppy, Kais Klai & Hacene Zidani (2011): *Formal verification of UML state diagrams: A Petri net based approach*. SIGSOFT Software Engineering Notes 36, pp. 1–8. Available at http://doi.acm.org/10.1145/1921532.1921561.

[5] Gerard J. Holzmann (2003): *The SPIN Model Checker: Primer and Reference Manual*. Addison Wesley.

[6] Kurt Jensen & Lars Michael Kristensen (2009): *Coloured Petri Nets – Modelling and Validation of Concurrent Systems*. Springer. Available at http://dx.doi.org/10.1007/b95112.

[7] Jiexin Lian, Zhaoxia Hu & Sol M. Shatz (2008): *Simulation-based analysis of UML statechart diagrams: methods and case studies*. Software Quality Journal 16(1), pp. 45–78. Available at http://dx.doi.org/10.1007/s11219-007-9020-9.

[8] Shuang Liu, Yang Liu, Étienne André, Christine Choppy, Jun Sun, Bimlesh Wadhwa & Jin Song Dong (2013): *A Formal Semantics for the Complete Syntax of UML State Machines with Communications*. In: iFM, Lecture Notes in Computer Science 7940, Springer, pp. 331–346. Available at http://dx.doi.org/10.1007/978-3-642-38613-8_23.

[9] Kenneth L. McMillan (1992): *Symbolic model checking: An approach to the state explosion problem*. Ph.D. thesis, Pittsburgh, PA, USA.

[10] OMG (2011): *OMG Unified Modeling Language (OMG UML) Superstructure. Version 2.4.1, 2011-08-06*. Available at http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/.

[11] Robert G. Pettit IV & Hassan Gomaa (2006): *Modeling Behavioral Patterns of Concurrent Objects Using Petri Nets*. In: ISORC, IEEE Computer Society, pp. 303–312. Available at http://doi.ieeecomputersociety.org/10.1109/ISORC.2006.55.

[12] Michael Westergaard (2013): *CPN Tools 4: Multi-formalism and Extensibility*. In: Petri Nets, Lecture Notes in Computer Science 7927, Springer, pp. 400–409. Available at http://dx.doi.org/10.1007/978-3-642-38697-8_22.

[13] Shaojie Zhang & Yang Liu (2010): *An Automatic Approach to Model Checking UML State Machines*. In: SSIRI (Companion), IEEE Computer Society, pp. 1–6. Available at http://doi.ieeecomputersociety.org/10.1109/SSIRI-C.2010.11.