PRISM Code Generation for Verification of Mediator Models

Weidi Sun and Meng Sun
LMAM & DI, School of Mathematical Sciences, Peking University, Beijing, China
{weidisun, sunm}@pku.edu.cn

***Abstract*—Component-Based Software Engineering (CBSE) has played an important role in software industry for several decades. The Mediator language is proposed to formally model complex hierarchical component-based systems, which provides a proper automata-based formalism for specifying both high-level system layouts and low-level behavior units. In this paper, we develop a framework for translating Mediator models into the model checker PRISM, and build such a "translator" which can generate PRISM codes from Mediator models automatically and cooperates with PRISM to verify properties of Mediator models.**
Keywords: Mediator, PRISM, Code generation, Model checking, Verification

I. Introduction

Component-based software engineering has played an important role in software industry for several decades. Implementation details inside components are encapsulated and different components are composed together to construct value-added systems through interfaces. Mediator [10] is a hierarchical modeling language that provides proper formalism for both high-level system layouts and low-level automata-based behavior units of component-based software systems, together with a full-featured type system and powerful coordination mechanisms. However, having powerful modeling languages does not mean correctness of the system models. In practice, errors can still be introduced by the modeling activities. Therefore, we need to investigate the formal analysis and verification techniques for Mediator to guarantee the correctness and reliability of Mediator models.

Model checking [5] is a widely adopted technique for verification of both hardware and software systems. PRISM [9] is a probabilistic model checker that provides a specification language based on the Reactive Modules formalism [1] and a powerful tool support for model checking properties specified in different temporal logics such as LTL, CSL, PCTL, PCTL*, etc. [14]. It has been successfully applied in many areas, including network protocols [7], security protocols [8], coordination languages [4], and so on.

In this paper, we present a framework for translating Mediator models into PRISM such that we can make analysis and verification of Mediator model behavior by using the probabilistic model checker PRISM. This is a further extension to our previous work on the Mediator language [10] and its Arduino C code generation [11].

DOI reference number: 10.18293/SEKE2019-069.

The following of this paper is organized as follows: After this general introduction, we briefly review some main concepts of the Mediator modeling language in Section II. In Section III, we show how different elements in Mediator can be translated into PRISM. Finally, Section IV concludes the paper and discusses some future work.

II. A Mediator Primer

In this section, we briefly review the primary concepts in the Mediator language which concentrates on both high-level system layouts and low-level automata-based behavior units. More details about the language and its semantics can be found in [10], [12].

The syntax tree of a Mediator program is defined as follows:

program ::= (*typedef* | *function* | *automaton* | *system*)*

Typedefs. Mediator provides various data types that are widely used in different formal modeling languages and programming languages. Basic types such as *Integer*, *Bounded Integer*, *Boolean* and *Enumeration* can be easily used to define new data types in Mediator.

Functions. Functions can be either *common functions* which have both interfaces describing inputs and return types of the functions, and function bodies specifying the behavior of functions, or *native functions* that have no function bodies but only interfaces. More discussions about functions can be found in [10].

Automata. *Automata* and *system* are the core modeling elements in Mediator. They are also called *entities* or *components* in a Mediator program. The syntax of *automata* is shown as follows.

automaton	::=	**automaton** *template*$^?$ *identifier* (*port**) { (**variables** {*varDecl**})$^?$ **transitions** {*transition**}}
port	::=	*identifier* : (**in** | ***out***) *type*
transition	::=	*guardedStmt* | **group** {*guardedStmt**}
guardedStmt	::=	*term* –> (*stmt* | {*stmt**})
stmt	::=	*assignStmt* | *iteStmt* | **sync** *identifier*$^+$
assignStmt	::=	*term* := *term*
iteStmt	::=	**if** (*term*) *stmt*$^+$ (**else** *stmt*$^+$)$^?$
varDecl	::=	*identifier : type* (**init** *term*$^?$)

An *automaton* consists of four parts: *templates*, *interfaces*, *local variables* and *transitions*, which are interpreted as follows:

1) *Templates.* Templates include a set of parameter declarations. A parameter can be either a type or a value.
2) *Interfaces.* Interfaces consist of directed ports and describe how automata interact with their contexts. Ports can be regarded as structures with three fields: *value*, *reqRead*, and *reqWrite*, which correspondingly denote the values of ports, the status of reading requests and the status of writing requests.
3) *Local Variables.* Each automaton contains a set of local variables.
4) *Transitions.* The behavior of an automaton is defined by guarded transitions. Each transition consists of a boolean term guard and a sequence of statements. Transitions encapsulated in a group are not ruled by priority in Mediator. In other words, when the guards of two transitions are both satisfied we cannot decide which transition occurs. However the *stmts* in a *guardedStmt* are ruled by priority, they will occur according to the order in the sequence of statements.

The following three types of statements are supported by our translation framework:

1) Assignment statement, including an expression and an assignment target, that evaluates the expression and assigns the result to its target if possible,
2) Ite (if-then-else) statement that acts as a conditional choice statement in other programming languages,
3) Synchronizing statement, labeled with *sync*, that are the flags requiring synchronized communication with other entities.

Systems. A *system* organizes its sub-entities which can be *automata* or *systems*. The syntax of *system* is as follows:

system	::=	**system** *template*$^{?}$ *identifier* (*port**) { (**internals** identifier^{+})$^{?}$ (**components** { *componentDecl** }$^{?}$) **connections** { *connectionDecl** }}
componentDecl	::=	*identifier*$^{+}$: *systemType*
connectionDecl	::=	*systemType params* (*portName*$^{+}$)

Besides the *templates* and the *interface*, a *system* contains the following parts:

1) *Components.* Entities can be placed and instantiated in systems as components. Each component is considered as a unique instance and executed in parallel with other components and connections.
2) *Connections.* Connections are used to connect the ports of the system itself, the ports of components and the internal nodes. Inspired by the coordination language Reo [6], [3], [2], complex connection behavior can also be determined by other entities.
3) *Internals.* Sometimes we need to combine multiple connections to perform more complex coordination behavior. Internal nodes declared in **internals** segments are untyped identifiers which are capable to weld two ports with consistent data-flow direction.

III. From Mediator to PRISM

In this section we introduce our framework for translation from Mediator to PRISM. The aim of this work is to make analysis of Mediator model behavior by using the probabilistic model checker PRISM.

A Mediator entity will be translated into a module in PRISM. First of all, we consider the "Flat" algorithm which was proposed in [10] and can be used to flatten a hierarchical system into a canonical automaton. The syntax of canonical automata is as follows:

automaton	::=	**automaton** *identifier ()* { (**variables** { *varDecl** }) **transitions** { *transition* }}
transition	::=	**group** {*guardedStmt**}
guardedStmt	::=	*term* –> *(stmt* | {*stmt**} *)*
stmt	::=	*assignStmt* | *iteStmt*
assignStmt	::=	*term* := *term*
iteStmt	::=	**if** *(term) stmt*$^{+}$*(* **else** *stmt* $^{+}$*)*$^{?}$
varDecl	::=	*identifier : type (* **init** *term)*$^{?}$

Such a flattening of Mediator system model has been proven to be valid, and the syntax of the resulting canonical automaton is similar to the corresponding PRISM model defined by a module as follows:

model ::= **module** identifier
declaration^{+};
(transition;)*
endmodule

Both the variables declarations and guarded statements for the transitions in an automaton can be easily mapped to the corresponding PRISM model as well.

In our framework, we have six components that work together to generate PRISM code from Mediator models: *entity generator*, *typedef generator*, *term generator*, *virtual term generator*, *transition generator* and *automaton generator*. We will show details about these generators, especially the last two, in this section.

A. Generators for Entity, Typedef, Term and Virtual Term

The *entity generator* is designed for calling the algorithm for flattening and fed the returning canonical automaton to the *automaton generator*. The canonical automaton has no parameters and ports which indicates that it does not communicate with the environment.

With the help of *typedef*, we can give an alias to an existing definition to simplify the expression in Mediator. Although we cannot use the *typedef* and alias in PRISM, the absence of similar syntax can be overcomed by using the original definitions directly. The *typedef generator* returns a map which maps aliases to original definitions and helps us find the original definition when meeting an alias.

There are some slight differences between terms in Mediator and PRISM. For example, in Mediator we use "==" to denote the equality operator, while in PRISM "=" is used instead.

Term generator and *virtual term generator* are designed for dealing with such subtle distinctions and the only difference between them is that the latter, as the name suggested, is for virtual terms. More details of these two generators can be found in [13].

B. Transition Generator

The transition generator is designed for generating transitions. There are two main differences between Mediator transitions and PRISM transitions:

- In Mediator, the *stmt*s in a *guardedStmt* are ruled by priority. However, in PRISM we do not have priority because of its assignment method. For example, if we have `a = 0, b = 0` and then make the assignment `a,b: ‘‘a = 1, b = a’’`. In Mediator the result is `a=1, b=1`, because we first assign the value 1 to `a` and then assign the value of `a`, which has been changed to 1, to `b`. In PRISM the result is `a=1, b=0` which is completely different. Because the assignment statement is `‘‘(a’ = 1) & (b’ = a ) ’’`. We assign `a’,b’` at the same time and then assign values of `a’,b’` to `a,b`.
- The *ite* statements can be nested in Mediator, but this is not permitted in PRISM.

Transition generator cannot work while ignoring these two problems. To solve the first problem we take the transition apart and execute the transitions sequentially. Though the execution of transitions in PRISM does not have priority, we can create the priority by adding a “pedometer” to guards. A new variable *tranmark* is provided, which is a counter to record the number of executed *stmt*s. For every transition with at least two *stmt*s, we separate it into several transitions. In each new transition, the new *stmt* contains one original transition’s *stmt* and an assignment statement: “`tranmark_i = tranmark_i + 1`”, the new guard consists of the original transition’s guard and a condition: “`tranmark_i=n`”. Fig. 1 shows such an example of transition separation.

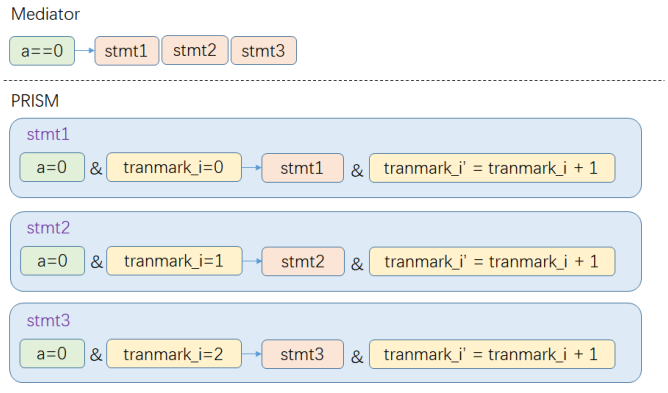


Fig. 1. Separation of transitions

In Fig. 1, we give every new transition a `tranmark_i` and initialize it to 0. When we want to execute a transition, the first new transition’s guard "`a = 0 & tranmark_i = 0`" must be satisfied. If it is satisfied we can execute `stmt1` and add 1 to `tranmark_i`. After that, the second new transition’s guard will be satisfied, and `stmt2` will be executed. These steps will be repeated until "`tranmark_i = n`". Following these steps we can execute every statement once and only once in order and the priority will be guaranteed.

However only separating the transitions is not enough for our framework. When we finish executing `stmt1` (for example `stmt1` is"`a’ = a + 1`"), the value of variable `a` may change, and the next new transition’s guard "`a = 0`" may not be satisfied. Here we need a new concept *virtual variable* to replace "`a`" in calculation. For example, "`a’ = a + 1`" in the transitions will be substituted with "`v_a’ = v_a + 1`" so that the original variables in guards will not change. Once all the executions of the original transition are finished, we assign the value of the *virtual variable*s to the original variables. An example of such virtual variables for transition separation is shown in Fig. 2.

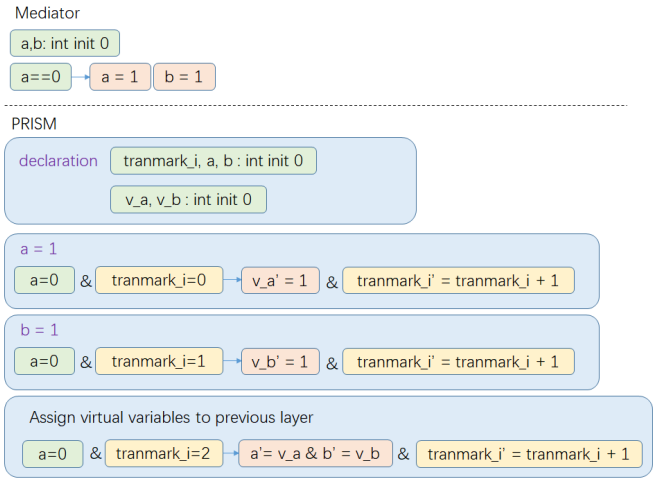
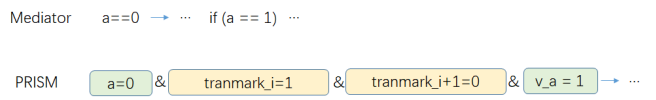


Fig. 2. Virtual variables for transition separation

The nesting problem for *ite* statements is entangled with the priority problem and thus more complicated. In other words, in the recursive generation, the nested *iteStmt* being treated as a new transition (regard the condition of *iteStmt* as a guard) shares all the troubles of the transition, the priority problem is no exception. We introduce a new variable: *layer* to denote the number of the current *iteStmt*’s nesting layers.

The new transition’s guard generated from the nested *iteStmt* will be the combination of the old guard and the *iteStmt*’s condition. Furthermore, we will give every *iteStmt* a new `tranmark` as well, for example:



Once a new transition for *iteStmt* is created, the generating process will enter a new layer, and the value of *layer* will be increased by 1. A set of new *virtual variables* corresponding to the *virtual variables* in the previous layer is also needed so that the change of variable values does not affect the

satisfiability of the new guard which contains the *iteStmt*'s condition. The generating process will exit the current layer when the execution of the *iteStmt* is completed, and the value of *layer* will be decrease by 1. We also introduce a variable *maxlayer* to record the largest *layer* that appeared. Combining with the above solution the example is shown in Fig. 3.

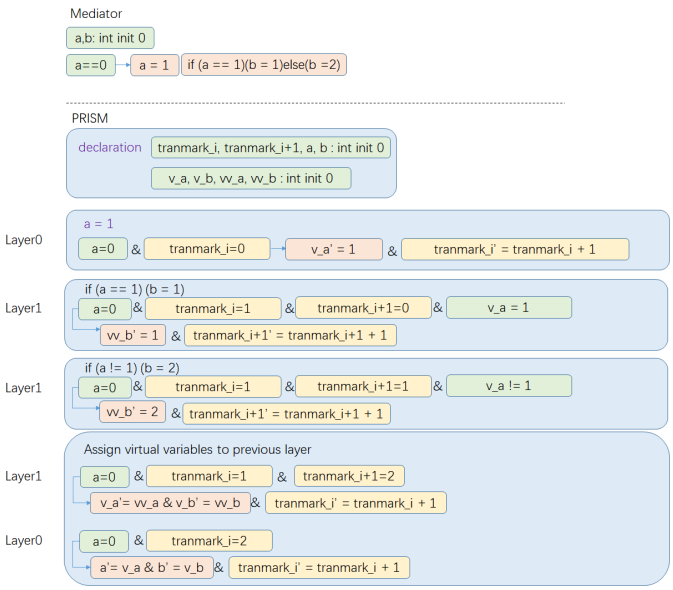


Fig. 3. Example of layer solution

The transition generator is designed as a recursive function TGF, which is invoked when we generate a transition and terminates when the generation process finishes. If we meet an *iteStmt* on the sidelines of a TGF execution, a new TGF will be invoked inside the old one.

To put it in a nutshell, the transition generator treats each statement as an atomic operation , i.e., an operation which cannot be interrupted and executes them in order.

C. Automaton Generator

The *Automaton Generator*'s goals are two-fold: adding the *global declarations* and combining different parts of the generated PRISM model.

Most types in *global declarations* which are supported by the PRISM language are easy to define. For these types, the only work we need to do is changing some grammar formats in Mediator's definitions. However, it does not work for `EnumType` and `ListType`. The solving of `EnumType` and `ListType` are similar, for `EnumType`, we define `IntType` variables for every identifier in it and initialize them to 0,1,2... in order. For `ListType`, we define `IntType` variables which is named as "`ListNamei`" for every element in the list. It needs to be pointed out that the list we defined is a fixed-length list and all the elements in it are initialized to 0. Besides, before defining such variables we need to change the user-defined types to base types, and the changing approach is mentioned in the *typedefGenerator*.

Then we have the model type, the model name, the global declarations and the transitions which can be generated by *transition Generator*; the final step is to combine these parts to build the PRISM model. After finishing all these works, the *automaton Generator* returns the result module in PRISM.

IV. Conclusion and Future Work

In this paper, we presented a code generator that converts Mediator models to PRISM models. Mediator provides a component-based modeling language in which components and systems can be defined in a hierarchical way. With the help of this code generator, the Mediator models for complex systems can be transformed into PRISM automatically such that properties of the Mediator models can be verified by using the PRISM model checker.

In the future we plan to extend the Mediator language and investigate more quantitative aspects of system models such as reliability, security, etc., in Mediator. Providing support for more hardware platforms and programming languages is in our scope for future work as well.

Acknowledgements

The work is partially supported by NSFC under grant no. 61772038, 61532019, 61202069 and 61272160, and the Guangdong Science and Technology Department (Grant no. 2018B010107004).

References

[1] Rajeev Alur and Thomas A. Henzinger. Reactive modules. *Formal Methods in System Design*, 15(1):7–48, 1999.
[2] Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical structures in computer science*, 14:329–366, 2004.
[3] Farhad Arbab, Christel Baier, Frank de Boer, and Jan Rutten. Models and temporal logical specifications for timed component connectors. *Software & Systems Modeling*, 6:59–82, 2007.
[4] Farhad Arbab, Sun Meng, Young-Joo Moon, Marta Z. Kwiatkowska, and Hongyang Qu. Reo2mc: a tool chain for performance analysis of coordination models. In *Proceedings of ESEC/FSE'09*, pages 287–288. ACM, 2009.
[5] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.
[6] Christel Baier, Marjan Sirjani, Farhad Arbab, and Jan Rutten. Modeling component connectors in reo by constraint automata. *Science of computer programming*, 61:75–113, 2006.
[7] Marie Duflot, Laurent Fribourg, Thomas Hérault, Richard Lassaigne, Frédéric Magniette, Stéphane Messika, Sylvain Peyronnet, and Claudine Picaronny. Probabilistic model checking of the CSMA/CD protocol using PRISM and APMC. *ENTCS*, 128(6):195–214, 2005.
[8] Salekul Islam and Mohammad Abu Zaid. Probabilistic analysis and verification of the ASW protocol using PRISM. *International Journal of Network Security*, 7(3):388–396, 2008.
[9] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of Probabilistic Real-Time Systems. In *Proceedings of CAV 2011*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.
[10] Yi Li and Meng Sun. Component-based modeling in mediator. In *Proceedings of FACS 2017*, volume 10487 of *LNCS*, pages 1–19. Springer, 2017.
[11] Yi Li and Meng Sun. Generating Arduino C Codes from Mediator. In *It's All About Coordination*, volume 10865 of *LNCS*, pages 174–188. Springer, 2018.
[12] Mediator github repository. https://github.com/mediator-team.
[13] Mediator to PRISM translator. https://github.com/Weidi-Sun/mediator-master.
[14] PRISM. http://www.prismmodelchecker.org.