Electronic Communications of the EASST
Volume 50 (2011)

Recent Advances in Multi-paradigm Modeling (MPM 2011)

Verifying Access Control in Statecharts

Levi Lúcio, Qin Zhang, Vasco Sousa and Yves Le Traon

12 pages

Guest Editors: Vasco Amaral, Cécile Hardebolle, Hans Vangheluwe, László Lengyel, Peter Bunus
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Verifying Access Control in Statecharts

Levi Lúcio*, **Qin Zhang, Vasco Sousa and Yves Le Traon**

(levi.lucio,vasco.dasilva,yves.letraon)@uni.lu,
qin.zhang@unige.ch
Laboratory for Advanced Software Systems (LASSY), University of Luxembourg
Software Modeling and Verification Group (SMV), University of Geneva

Abstract: Access control is one of the main security mechanisms for software applications. It ensures that all accesses conform to a predefined access control policy. It is important to check that the access control policy is well implemented in the system. When following an MDD methodology it may be necessary to check this early during the development lifecycle, namely when modeling the application. This paper tackles the issue of verifying access control policies in statecharts. The approach is based on the transformation of a statechart into an Algebraic Petri net to enable checking access control policies and identifying potential inconsistencies with an OrBAC set of access control policies. Our method allows locating the part of the statechart that is causing the problem. The approach has been successfully applied to a Library Management System. Based on our proposal a tool for performing the transformation and localization of errors in the statechart has been implemented.

Keywords: Access Control Policies, UML Statecharts, Verification, Model Transformation, Model Checking

1 Introduction

Access control is one the most important security mechanisms in existence. It guarantees that system resources are accessed according to a previously specified policy. It is therefore of the utmost importance to check that an access control policy is well implemented.

In a Model Driven Development (MDD) context [9], models can be exploited in order to enable locating errors early during the development lifecycle. As UML models are the de facto standard for modeling, this paper will focus on UML statecharts, which represent the dynamic aspect the system and are the key for the understanding how the system will execute. In our approach we consider statecharts embedding the logic of access control policies specified as a set of OrBAC rules [7]. OrBAC rules define an access control policies as a tuples of five elements: *Organization*, *Role*, *Activity*, *View* and *Context*. In order to verify such a statechart respects the access control policies defined in the OrBAC rules, we propose a new approach that is based on the usage of Algebraic Petri Nets (APNs) as a means to perform the verification of access control rules. The process is straightforward: first, the statechart is automatically transformed into an APN; then the access control policies are model checked as properties of the APN to find any inconsistency or violation of the access control policies in APN; finally, if an issue is found,

* Levi Lúcio and Qin Zhang were sponsored by the FNR CORE project MOVERE, ref. C09/IS/02.

we locate in the statechart the transition that is causing that problem by using the information present in the model checker's counterexample.

The remainder of this paper is organized as follows: section 2 introduces the Library Management System we use in the paper to illustrate our approach; section 3 presents the technicalities of our approach; section 4 introduces the tool we have developed for our proposal; in section 5 we contextualize our results; finally section 6 concludes.

2 The Library Management System running Example

In this paper we will employ a running example of simple Library Management System (LMS) to illustrate our proposal. As the object *Book* is the core component of the system, we will build a UML statechart for the business logic of the *Book* which represents the whole LMS (see Figure 1). Regarding security, the LMS has attached a set of access control policies defined using OrBAC [7] (see Figure 1, bottom left corner). To secure the system, we embed these access control policies[1] in the system business logic by adding *Role* and *Context* conditions in the state transition guards – since in the statechart the events are *Activities* (e.g. *Order* or *Borrow* in Figure 1) while the *View* (resource) is always the object *Book*.

As shown in Figure 1, when a book is published, it can be ordered and archived by the secretary of the library and then can be borrowed and returned by borrowers, including teachers and students. When a book is already borrowed, other users may reserve it by being registered in a reservation list. Reservers may also cancel their reservations. Access control policies are embedded as the statechart's transition guards in order to secure these business functions. For example, the guard in the transition from state *Published* to state *Ordered* means the function "order a book" can be only executed when the role is *Secretary* while the context is *WorkingDays* – which satisfies the access control policy *Permission(Secretary, Order, Book, WorkingDays)*.

3 Verifying Access Control Policies

In figure 2 we depict the technique we propose for verifying access control policies have been well implemented in a statechart. We assume security experts have expressed a set of access control policies (top box on the left) and that the functional behavior of the system was described using statecharts (top box on the right). We also assume the access control policies have been *implemented* in the statechart model by following the directives we have introduced in section 2. The verification technique we propose is based on transforming the original statechart into an APN and the access control policies into temporal properties (bottom box on the left) of that APN (bottom box on the right). The purpose of these two transformations is to use an existing model checker to automatically decide if no access control policy is violated by the dynamic behavior of the secured statechart.

Since we are performing exogenous model transformations from access control policies into temporal logic expressions and from statecharts to APN, we have to solve the classical problem

[1] Since there is only one organization Library in our example, we omit the Organization element in each access control policy.

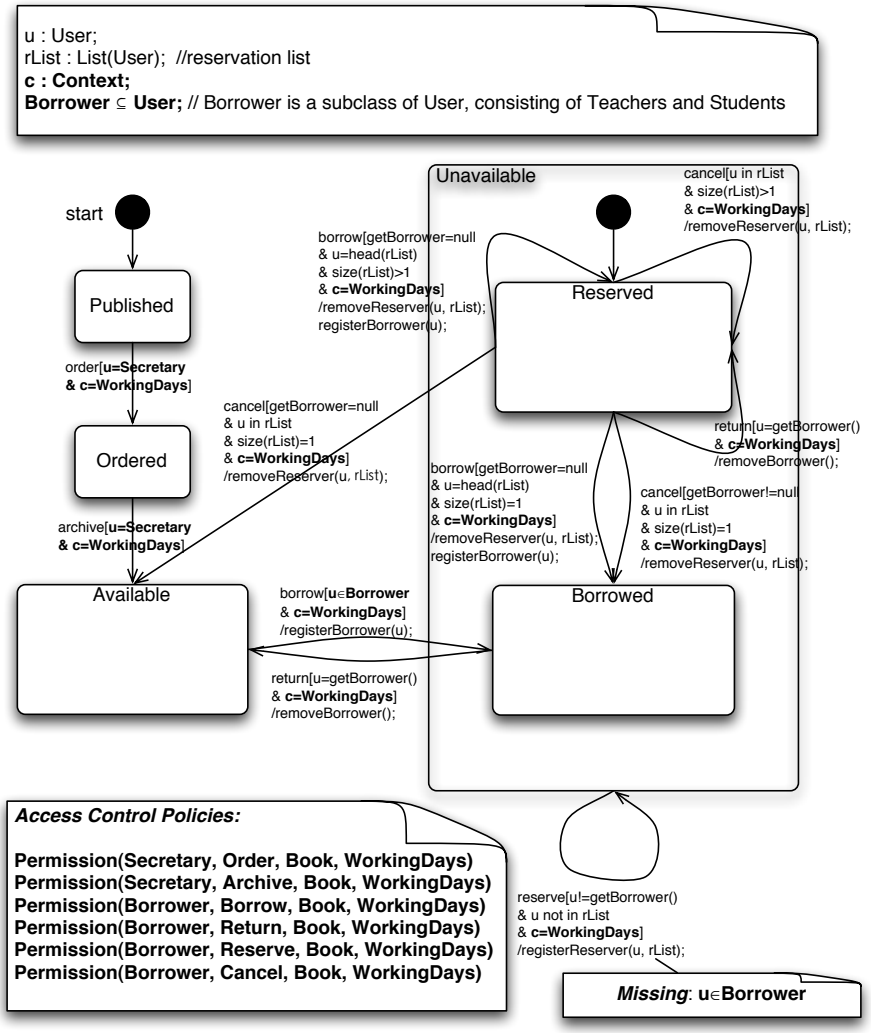

Figure 1: Secured UML statechart for the Book in the Library Management System

of semantic preservation for these model transformation. In particular, two main issues arise: 1) a model transformation that is semantics preserving needs to be built. This ensures the safe derivation, without semantic gap, of an APN representation from a statechart. Model checking counterexamples can then be interpreted in the statechart (arrow *mapped into*); 2) we need to build a model transformation for converting access control policies into temporal properties of the statechart obtained by the transformation described in 1) (arrow *transformed into*). In particular, given that the access control policies are implemented over the structure of the statechart, this last transformation needs to take into consideration the mapping between the elements of the statechart and the elements of the APN obtained using the transformation described in 2). The following sections will address these issues.

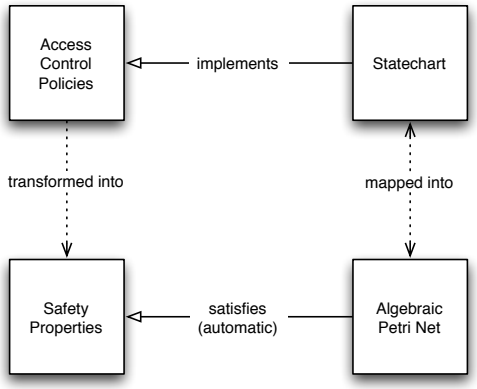

Figure 2: Commutative Verification of Access Control Policies in Statecharts

3.1 Background Formalisms and Tools

UML offers a state-based formalism to graphically represent object or system behaviors. As shown in Figure 1, which describes the behavior of an *book* object, a statechart is composed of basic named *states* and *transitions*. States may include *entry* and *exit* actions, which can change the value of *extended state variables* – these hold additional data enriching a statechart *state*. For example, the variable *rList* maintaining a list of users having a reservation on the books is one such extended state variables. *Transitions* model state changes in a system and are guarded by conditions on event inputs and extended state variables. In the example, except for *rList*, all the variables of the transitions are input variables. Transitions may also include actions which should be executed if a guard holds and there is a state change. For instance, the *registerBorrower*, *removeBorrower*, *registerReserver* or *removeReserver* operations on the user list *rList* are executed as actions in several transitions. Statecharts also include the notion of hierarchical nested states (composite states), where a state can be composed of substates – e.g. the *reserved* and *borrowed* states are nested inside the *unavailable* state. This notion facilitates grouping states that are part of a more abstract behavior and dealing with transitions that are pertinent to that abstract behavior and thus shared by all the nested states.

On the other hand, APN is a formalism used for modeling, simulating and studying the properties of concurrent systems. They are based on the well known Place/Transition (P/T) Petri Nets (PN) formalism where *places* hold resources – also known as tokens – and *transitions* are linked to places by input and output *arcs*, which are weighted. A PN has a graphical concrete syntax consisting of circles for *places*, boxes for *transitions* and arrows to connect the two. The semantics of a P/T PN involves the sequential non-deterministic firing of enabled transitions in the net – where firing a transition means consuming tokens from the set of places linked to the input arcs of the transition and producing tokens into the set of places linked to the output arcs of the transition. The algebraic data type (ADT) extension allows defining tokens as elements of

sets (with associated operations) which are models of algebraic specifications. The arcs of APNs can be annotated with weights defined by terms of the algebraic specification and the transitions can be guarded by algebraic equations.

For automating access control policy verification we will use the AlPiNA model checker [3] for APN. AlPiNA is able to decide on the satisfaction of invariant (also called *safety*) properties of APNs. The invariants are expressed as conditions on the tokens contained by places in the net at any state of the net's semantics. Invariants are built using first order logic, the operations defined in the algebraic specification and additional functions and predicates on the number of tokens contained by places.

3.2 Transforming Secure Statechart into APN

In order to transform UML statechart models into APN models we have developed a set of transformation rules. These rules cover the basic syntax of UML statecharts and are built to both preserve the original statechart semantics in the APN resulting from the transformation and add instrumentation artifacts for verification purposes. In the next points we will discuss the main highlights of these model transformation rules. The complete text describing these rules in detail can be found in [11].

Decomposition of nested states : Since the AlPiNA model checker does not directly support hierarchical Petri nets, we need to decompose composite states in the UML statechart before applying the model transformation. The decomposition rules consists of: 1) redirecting each incoming transition pointing to a composite state or to one of that composite state's inner states to the composite state's initial state and 2) replicating each outgoing transition starting from the composite state as a transition starting from each of the inner states. We then add the entry actions of the composite state to each newly created incoming transition as its event action. We also replicate the exit actions of the composite state to the newly created outgoing transitions. At last we rename the inner initial state by adding the original composite state name, which makes it distinguishable from the initial state of the whole statechart.

Transformation of data types : After the decomposition of the nested states, we define an ADT for each type of event parameters or extended state variables in the UML statechart. In these ADTs there should be sufficient generators to provide exhaustive input values such that the concerned transitions in the APN may be model checked for all possible behaviors. These interesting input values may be found in other sub packages of the object-oriented specification, e.g. class diagrams or sequence diagrams. For each transition in the APN transformed from a statechart transition, an exhaustive set of input values exercising access control variables in the transition guard is provided in places created adjacently to the transition (see the elliptic places in the APN in Figure 4). This allows exhaustively exercising access control conditions when activities are simulated in the APN. Regarding method calls in event or state entry/exit actions operating on the event parameters or extended state variables, we implement them by extending the corresponding ADTs with operations having the necessary behavior. We also create in the APN places with ADT tokens representing extended state variable values in order to simulate the state machine's memory (e.g. see the places "Borrower" and "ReserverList" in Figure 4).

Transformation of the model's structure : After having a complete set of data definitions as

ADTs, the next step is to transform the UML statechart into an APN structure. This is achieved as follows: 1) for each state in the decomposed statechart, including "Start" and "End" states, we build an APN place with the same name; 2) for each event (state transition in the statechart) we create a corresponding transition with the necessary input/output arcs. If there are entry (resp. exit) actions an original state, we extract these actions from the statechart and transform them into APN transitions inserted before (resp. after) the place resulting from transforming the state.

After having built the backbone of the APN structure, we enrich the created transitions with the input (resp. output) arcs from (resp. to) the places created to hold extended state variable data. If there are actions attached to the original event of one of such transitions, we need to call those operations on the output arcs of this transition to produce a token with the correct data. An example of such a call in Figure 4 is the operation "removeReserver($u, $rList)" on the output arc from transition "Reserved_borrow_Reserved". We complete the logic of the original events by adding the original statechart guard conditions to generated APN transitions.

Traceability and instrumentation issues : If a counterexample is found during the model checking of the produced APN we need to be able trace its origin in the statechart. In order to allow traceability it is necessary that the mapping between the the original statechart structure and the structure of the obtained APN is known. In what concerns statechart states, there is a surjection between states and their transformed counterparts on the transformed model. The same happens for statechart transitions. Note that given that events with the same name may occur in different statechart transitions, both the transition name and the source and target states are required to univocally identify a transition in a statechart.

Note that because AlPiNA implements reachability (the **AG** CTL temporal operator), instrumentation of the APN model resulting from the transformation becomes necessary in order to simulate checking more complex temporal operators. For this purpose we use a special ADT called *indicator*, which consists of three fields that are able to record *actor*, *context* and place *origin/destination* information for each fired transition in the APN. The indicator token circulates among the places of the APN that result from transforming the original statechart states. When a transition corresponding to statechart activity fires, an indicator token is placed in the output place of the transition. This serves two purposes: on the one hand the statechart state transition semantics is simulated by updating the marking of the APN in a way state reflects statechart state change; on the other hand information about under which access control context the activity was executed is logged for verification purposes. The ADT definition for the *indicator* type is shown in Figure 3 b).

3.3 Transforming Access Control Policies into APN Properties

In order to verify the access control policies on the statechart we will transform those policies into temporal properties of the APN obtained by the transformation described in section 3.2. To produce the required temporal formulas we reuse and adapt the technique presented in [11], built to enable model checking access control policies in systems modeled as APN. The technique is based on the premisses that 1) each system activity requiring access control is modeled as an APN transition, and that 2) when fired, every transition modeling an activity outputs log information including data about under which context and by whom the activity was executed.

Because in such an APN model all the accesses to activities are explicitly recorded in places

```
Adt eventname
Sorts eventname;
Generators
    Start_start_Published: eventname;
    Published_order_Ordered: eventname;
    Ordered_archive_Available: eventname;
    Available_borrow_Borrowed: eventname;
    Borrowed_return_Available: eventname;
    Reserved_cancel_Available: eventname;
    Reserved_reserve_UnavailableStart: eventname;
    Borrowed_reserve_UnavailableStart: eventname;
    UnavailableStart_start_Reserved: eventname;
    Reserved_borrow_Reserved: eventname;
    // more generators for event names ... ...
```

a) ADT definition of re-defined event names

```
Adt indicator
Sorts indicator;
Generators
    I: user, context, eventname -> indicator;
Operations
    getUser: indicator -> user;
    getContext: indicator -> context;
    getEventname: indicator -> eventname;
Axioms
    getUser(I($u, $c, $en))=$u;
    getContext(I($u, $c, $en))=$c;
    getEventname(I($u, $c, $en))=$en;
Variables
    u: user; c: context; en: eventname;
```

b) ADT definition of indicator

Figure 3: ADT Definitions of Renewed Transition Names and Indicator

connected to the transitions that model them, we can verify the access control policies by checking that every marking of model's state space is such that it does not violate any access control policy. In particular we are interested in checking either of the following:

1. for all recorded accesses to an activity, at least one of the activity's permissions is met;
2. for all recorded accesses to an activity, none of the prohibitions for that activity is met.

In [10] the justification that verifying either points 1 or 2 is enough for our purposes is given. In the following, we continue our reasoning by relying solely on permission verification. In terms of temporal logic, point 1 can be written for a given activity *act* as:

$$\mathbf{AG}\left(\forall t \in act\_log : Perm_1^{act}(t) \vee \ldots \vee Perm_n^{act}(t)\right) \tag{1}$$

where t is a token in place act_log containing log information about the firing conditions of the incoming transitions. Formulas $Perm_1^{act}(t) \ldots Perm_n^{act}(t)$ are predicate logic formulas, one for each permission for activity *act*. Each of those formulas checks the collected log tokens in act_log for a possible violations of an access control policy. Note that formulas $Perm_1^{act}(t) \ldots Perm_n^{act}(t)$ are disjunct. This is so because several permissions may exist for the same activity. Finally, the **AG** temporal operator makes sure the formula holds for all reachable states of the APN. Notice that in order to check all declared permissions for a statechart a formula such as formula (1) needs to be built for any activity for which access control policies are declared.

In order to verify if a statechart implements the access control policies correctly, we will reuse the technique presented above to produce temporal logic formulas for the APN obtained from the transformation presented in section 3.2. Given the additional level of indirection between the model we want to analyze (a statechart) and the artifact we have at our disposal (the APN resulting from the transformation), we will adapt the technique such that:

1. the temporal formulas to verify an activity are generated for multiple log places, rather than for only one. Indeed, an APN place will act as a log place for an activity when it is the result of transforming an arrival state for a transition implementing this activity in

the statechart. Thus, when a statechart activity corresponds to several transitions in the statechart – and thus indirectly several transitions in the resulting APN – we need to allow several log places for the same activity;

2. the temporal logic formulas are produced such that an indicator token in a log place is checked for a permission only if contains log information about the activity the permission refers to. This is to cope with the fact that a place of the APN can serve as log place for multiple activities (e.g. state *available* serves as log place for activities *archive, return* and *cancel*).

As an example, the temporal formulas necessary to check there are no access control violations to the *borrow* activity in the LMS example are the following:

$$\mathbf{AG}\big(\forall t \in borrowed : getActivity(t) = borrow => (isBorrower(getUser(t)) = true \ \& \ getContext(t) = WorkingDays)\big) \quad (2)$$

$$\mathbf{AG}\big(\forall t \in reserved : getActivity(t) = borrow => (isBorrower(getUser(t)) = true \ \& \ getContext(t) = WorkingDays)\big) \quad (3)$$

Note that, according to point 1) above both formulas (2) and (3) need to be generated to check the *borrow* activity because both the *borrowed* and the *reserved* places resulting from the transformation (see Figure 4) are log places for *borrow*. According to point 2) above, given that both places *borrowed* and *reserved* are log places for other activities, the $getActivity(t) = borrow$ condition for the implication in both formulas (2) and (3) guarantees only indicator tokens regarding the *borrow* activity are checked for the *borrow* permissions. Finally, given there is only one permission for activity *borrow*, only one formula (a conjunction of access conditions) exists after the implication for formulas (2) and (3). Other eventual permissions for activity *borrow* would be disjunct with the existing conditions as explained by formula 1.

3.4 Mapping the Verification Result Back

When model checking on the obtained APN a set of temporal formulas obtained using the technique described in section 3.2, two results can arise: 1) the formulas are satisfied and all permission is respected; 2) a formula is not satisfied and a counterexample is found, meaning the corresponding permission was violated.

In case a permission is violated, it is possible to extract from the offending log token in the counterexample the firing conditions of the APN transition that led to its production. Moreover the log token tracks the name of the original statechart transition and the names of both the original source and destination states (see section 3.2). Thanks to the traceability mechanism explained in section 3.2 it is then straightforward to interpret the APN counterexample information in the original statechart.

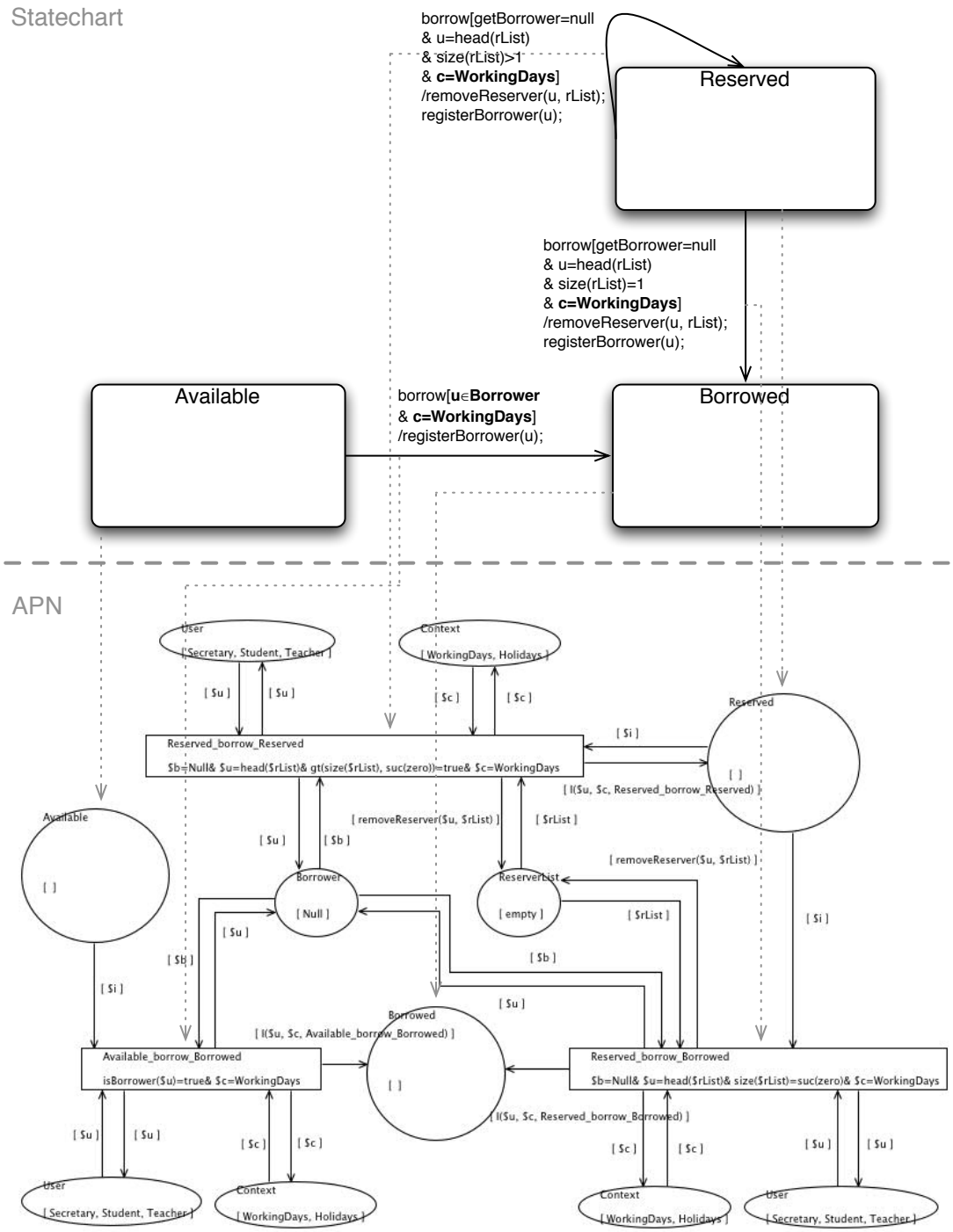

Figure 4: Excerpt of the full transformation of the LMS from Statechart to APN

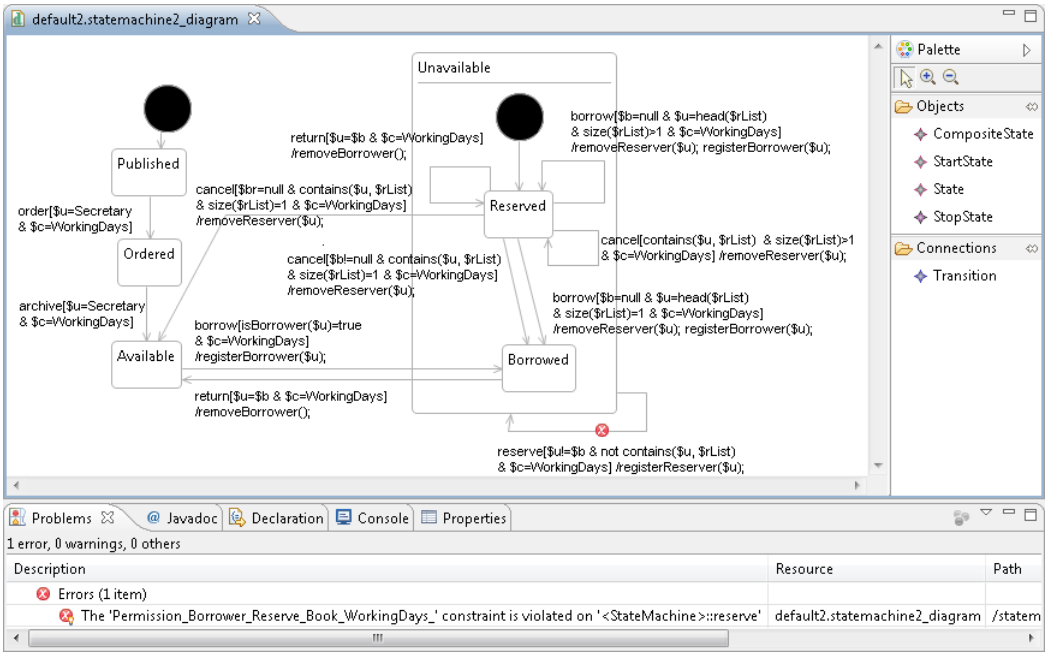

Figure 5: Mockup interface of the access policy verification tool

4 Supporting Tool

In section 3 we have described a process for verifying that access control policies are enforced by a statechart. In order to build a tool for this procedure such as sketched in Figure 2, we need to automate three main steps: 1) the transformation of a statechart into an APN process (vertical right arrow of Figure 2), 2) the transformation of access control policies into temporal logic formulas (vertical left arrow of Figure 2), and 3) the verification of temporal logic formulas (bottom horizontal arrow of Figure 2) process.

Step 1 has been fully implemented as a model transformation using the DSLTrans [1] transformation language. The complete model transformation description implemented in the DSLTrans language can be observed in [11]. Step 2 is not yet fully automated. In particular, the transformation of the access control policies into temporal logic formulas transformation needs to be aware of the names of the APN places that are created from the statechart states – such that the permissions are checked on the correct log places. Step 3 is implemented by the AlPiNA [3] tool and is thus fully automated.

The implementation of these three steps opens the possibility to provide a 'push-button' tool. Such a tool will, given a set of access control policies and a statechart, highlight in the statechart the transition guard(s) that violate a particular access control rule. During this process, all the verification machinery is transparent for the user, who only manipulates the statechart model. We have built a mockup of such a tool and a screenshot of its interface can be seen in Figure 5. Notice that the access control fault detected in figure 5 is due to the fact that someone other than a *borrower* was able to accomplish a *reserve* activity. The fault is due to the missing guard

'$u \in Borrower$' error described as an annotation in figure 1. From the fault description in the tool the modeler is able to understand that a *secretary* (not visible in the figure due to lack of space) was able to *reserve* a book, which is disallowed by the access control policy specification in figure 1.

5 Discussion and Related Work

Several approaches have been developed to extend UML with security concerns [8, 4]. We target a narrower part of UML by adapting a part of UML statecharts to embed access control mechanisms. Our work considers that the access control policy is manually woven into the functional model of the system, thus assuming that functional and security concerns are explicitly linked at design time. Our vision is similar to the one of Ray et al. [6], but with a special focus on dynamic behavioral models (statecharts) instead of static ones. This does not mean that the underlying separation of PDP and PEP is necessarily broken at architectural level, but we promote the creation of an integrated model of the system combining access control with behavior. Such an integrated model may become productive in two possible exclusive ways: 1) security-oriented MBT [2, 5] for generating test cases to execute on the system implementation; 2) generation of deployment code from the integrated statechart.

Concerning point 1), the objective of an integrated model is to offer a formal representation that allows covering behaviors with the intent of testing access control. The test cases generated from the integrated statechart describe the action/event sequences for exercising access control mechanisms. Concerning point 2), the use of the integrated model would consist of developing a code generator from the integrated statechart, allowing the generation of a secure implementation of the functionalities described in the statechart.

In this paper we translate UML statecharts into APN models for formal verification. This implies some assumptions on the type of statecharts used such that our method may be applied. These assumptions are: policies are implemented in the statechart by adding user and context conditions on the state transition guards; all accesses to a secured activity are either allowed or denied for a given user type under a given context; data types for the statecharts are implemented as Algebraic Data Types for ease of translation back and forth into the APN world[2]; in order to allow automatic analysis all data types used in the statechart are bound to a finite amount of values (e.g. in our LMS example lists of reservers are bounded to size 3).

Finally, given the closeness between the semantics of UML Statecharts and APNs, we do not provide a formal proof of the preservation of the semantics of statecharts by the transformation. Such a proof could be obtained by providing a formal semantics to UML Statecharts and proving its equivalence with the semantics of the APNs obtained using the transformation in section 3.3.

6 Conclusion

In this paper we propose a technique for verifying a set of activities described in a statechart are correctly secured regarding a set of access control policies. We assume the access control

[2] Note that there is no loss of generality when, instead of using UML types, we use corresponding Algebraic Data Types.

enforcement is achieved by reinforcing statechart transitions with access control conditions. The approach is based on translating statecharts into APNs and OrBAC access control policies into temporal logic properties. We then use a model checker for automating the verification procedure. The results of the verification can be directly highlighted in the statechart. The strength of our approach lies in the fact that the modeler is only required to provide the access control policies and the statechart in their original format. The procedure and required instrumentation for verifying access control can be automatically generated.

Bibliography

[1] B. Barroca, L. Lúcio, V. Amaral, R. Félix, and V. Sousa. Dsltrans: a turing incomplete transformation language. In *Proceedings of the 3rd SLE*, SLE'10, pages 296–305. Springer-Verlag, 2011.

[2] M. Blackburn, R. Busser, and A. Nauman. Model-based approach to security test automation. In *International Software Quality Week*, 2002.

[3] D. Buchs, S. Hostettler, A. Marechal, and M. Risoldi. Alpina: A symbolic model checker. In *Petri Nets*, volume 6128 of *Lecture Notes in Computer Science*. Springer, 2010.

[4] J. Jürjens. Umlsec: Extending uml for secure systems development. In *Proceedings of the 5th International Conference on The Unified Modeling Language*, UML '02, pages 412–425. Springer-Verlag, 2002.

[5] T. Mouelhi, Y. L. Traon, and B. Baudry. Transforming and selecting functional test cases for security policy testing. In *Proceedings of the 2nd international conference on Software Testing, Verification, and Validation (ICST)*, pages 171–180. IEEE Computer Society, 2009.

[6] I. Ray, R. France, N. Li, and G. Georg. An aspect-based approach to modeling access control concerns. *Information and Software Technology*, 46:575–587, 2004.

[7] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-Based Access Control Models. *IEEE Computer*, 29(2):38–47, 1996.

[8] D. B. Torsten Lodderstedt and J. Doser. Secureuml: A uml-based modeling language for model-driven security. In *Proceedings of the 5th International Conference on The Unified Modeling Language*, pages 426–441. Springer, 2002.

[9] A. Uhl. Model-driven development in the enterprise. *IEEE Software*, 25:46–49, 2008.

[10] Q. Zhang. Analysis of integrity of access control policies and security coverage of transformed properties in apn. Technical Report TR-LASSY-11-09, http://hera.uni.lu/~levi.lucio/verifying_access_control_statecharts/integrity_coverage.pdf, 2011.

[11] Q. Zhang and V. Sousa. Practical model transformation from secured uml statechart into algebraic petri net. Technical Report TR-LASSY-11-08, http://hera.uni.lu/~levi.lucio/verifying_access_control_statecharts/transformation_rules.pdf, 2011.