Electronic Communications of the EASST
No \volume defined!

No \volumetitle defined!

Verifying Access Control in Statecharts

Levi Lúcio, Qin Zhang, Vasco Sousa and Tejeddine Mouelhi

12 pages

No *ed(s) defined!
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 Tejeddine Mouelhi

(levi.lucio,vasco.dasilva,tejeddine.mouelhi)@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 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 [10], 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 a UML model, more precisely on statecharts, which represents the dynamic aspect the system and is key for the understanding of how the system will be executed. In our approach we consider statecharts embedding the logic of access control policies specified as a set of OrBAC rules [8], , which defines an access control policy as a tuple of five elements: *Organization*, *Role*, *Activity*, *View* and *Context*. The embedding is modular in the sense that, in order to introduce access control logic in a statechart, the statechart's guards are reinforced in a predefined fashion. 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

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

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, we relocate it in the statechart the transition or the state that is causing that problem using the information 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 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 build a UML statechart for the business logic of the *Book*, which representing the whole LMS, see Figure 1. Regarding security, we have a set of access control policies defined by OrBAC [8]. Since there is only one organization *Library* in our example, we omit the *Organization* element in each access control policy. To secure the system, we embed these access control policies in the system business logic by adding *Role* and *Context* conditions in the state transition guards, since in the statechart the events are *Activities* 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, embedded as the statechart's transition guards, secure these business functions. For example, 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

The technique we propose for verifying access control policies is depicted in Figure 2. We wish to show that a statechart (top box on the right) *implements* a set of access control policies (top box on the left) correctly. In order perform this check automatically, we will translate access control policies into temporal properties (bottom box on the left) of a specific APN (bottom box on the right). This APN will be obtained by transformation and will have an operationally exploitable semantics of the statechart we are interested in. The purpose of this translation is to automatically check if the obtained APN *satisfies* the obtained temporal formulas. In order to tackle the proposal presented in Figure 2 two main issues need to be addressed: 1) a model transformation that is semantic preserving needs to be built such that an APN representation of a statechart implementing access control can be derived and its model checking counterexamples 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 con-

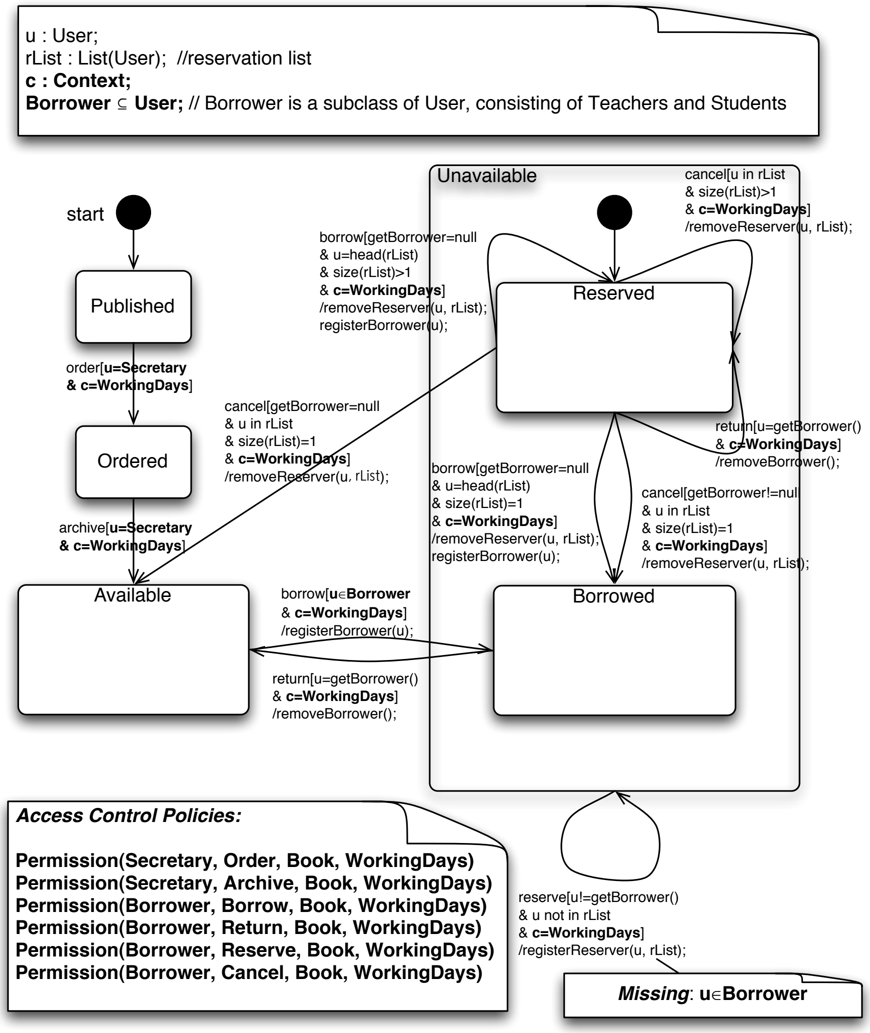

u : User;
rList : List(User); //reservation list
c : Context;
Borrower ⊆ User; // Borrower is a subclass of User, consisting of Teachers and Students
start
Published
order[u=Secretary & c=WorkingDays]
Ordered
archive[u=Secretary & c=WorkingDays]
Available
Unavailable
Reserved
Borrowed
borrow[getBorrower=null & u=head(rList) & size(rList)>1 & c=WorkingDays] /removeReserver(u, rList); registerBorrower(u);
cancel[u in rList & size(rList)>1 & c=WorkingDays] /removeReserver(u, rList);
cancel[getBorrower=null & u in rList & size(rList)=1 & c=WorkingDays] /removeReserver(u, rList);
return[u=getBorrower() & c=WorkingDays] /removeBorrower();
borrow[getBorrower=null & u=head(rList) & size(rList)=1 & c=WorkingDays] /removeReserver(u, rList); registerBorrower(u);
cancel[getBorrower!=null & u in rList & size(rList)=1 & c=WorkingDays] /removeReserver(u, rList);
borrow[u∈Borrower & c=WorkingDays] /registerBorrower(u);
return[u=getBorrower() & c=WorkingDays] /removeBorrower();
Access Control Policies:
Permission(Secretary, Order, Book, WorkingDays)
Permission(Secretary, Archive, Book, WorkingDays)
Permission(Borrower, Borrow, Book, WorkingDays)
Permission(Borrower, Return, Book, WorkingDays)
Permission(Borrower, Reserve, Book, WorkingDays)
Permission(Borrower, Cancel, Book, WorkingDays)
reserve[u!=getBorrower() & u not in rList & c=WorkingDays] /registerReserver(u, rList);
Missing: u∈Borrower

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

trol policies are expressed over the statechart, this last transformation needs to be parameterized by the mapping between the statechart and the APN obtained using the transformation described in 2). The following sections will address these issues.

3.1 Background Formalisms and Tools

We have already informally introduced the notion of statechart in this paper in Figure 1 – a statechart representing the behavior of a *book* object. It is possible to observe in Figure 1 that a statechart is composed of basic named *states* and *transitions*. States may include *entry* and *exit*

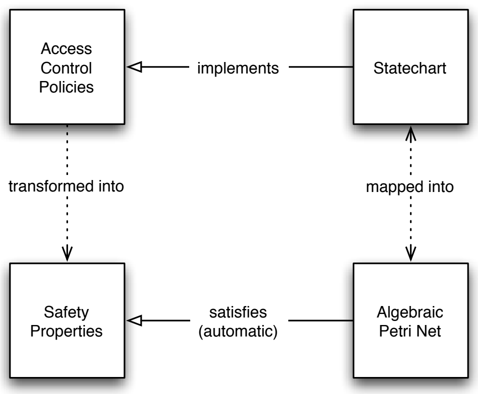


Figure 2: Commutative Verification of Access Control Policies in Statecharts

actions, which can change the value of *extended state variables* – these hold additional data enriching a statechart *state*. For example, in the statechart in Figure 1 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. For example in the transitions of statechart in Figure 1 any variable except for *rList* is an input variable. Transitions may also include actions which should be executed if a guard holds and there is a state change. As an example in Figure 1 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, where a state can be composed of substates – e.g. in Figure 1 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 common to all the nested states.

On the other hand, APNs are 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. Normally 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 include 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

We have developed a set of rules for model transformation from UML statechart into APN. These rules cover all the basic syntax of UML statechart and are built to preserve the statechart semantics. In this paper, we discuss some significant points extracted from these model transformation rules. The complete text describing the transformation rules in detail can be found in [12].

Decomposition of nested states : Since the AlPiNA model checker does not directly support hierarchical Petri net, we need first to decompose composite states in the UML statechart before model transformation. The decomposition rule consists of transiting each incoming transition pointing to the composite state to the inner start state and split each outgoing transition starting from the composite state into transitions each of them starting from one sub state. We then relocate the entry actions of the composite state to each incoming transitions, either pointing to the composite state or to a substate as their event actions and relocate the exit actions of the composite state to transitions starting from the composite state or from sub states but crossed the boundary of the composite state. At last we rename the inner start state by adding the original composite state name, which makes it distinguishable from the start state of the whole statechart.

Transformation of data types : After the decomposition of 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 relevant transitions in the APN may be tested for all possible behaviors. These values of the ADTs may be found in other sub packages of the object-oriented specification, e.g. the class diagram. We then create in the net places to hold the exhaustive initial data (tokens), see the elliptic places in Figure 4. Regarding the methods, in event or state entry/exit actions, operating on the event parameters or extended state variables in the statechart, we implement them by respectively extending the corresponding ADTs with operations and corresponding axioms and creating corresponding places with initialized multi-sets of the variable types which simulate machine memory occupied by these methods – for instance see the places "Borrower" and "ReserverList" in Figure 4.

Transformation of model structure : When we have the complete set of data definitions as ADTs, the next step is to transform UML statechart into APN structure, by mapping corresponding components. For example, regarding each state in the decomposed statechart, including "Start" and "End" states, we build a place with the same name in APN. Similarly for each event (state transition in the statechart) we create a corresponding transition with necessary input/output arcs. If there are entry/exit actions in any original state, we extract these actions from the statechart and transform them into APN transitions inserted before/after the place resulting from transforming the state.

After having built the "backbone" APN structure, we supplement the transitions with necessary input/output arcs from the places created to hold data. If there are some actions in the mapped event of this transition we need to extend the corresponding ADTs to implement the logic of actions as ADT operations, and locate these operations on the output arcs of this transition to produce a token with correct data – e.g. the operation “removeReserver($u, $rList)” on the output arc from transition “Reserved_borrow_Reserved” to place “ReserverList” in Figure 4. Finally we complete the business logic of original events by adding the statechart guard conditions on the transformed transitions.

In UML statecharts there is a special kind of states, called Choice Pseudo States. As this kind of states is used to reason about conditions related to the extended state variables we transform them into a group of parallel transitions with mutually exclusive guards.

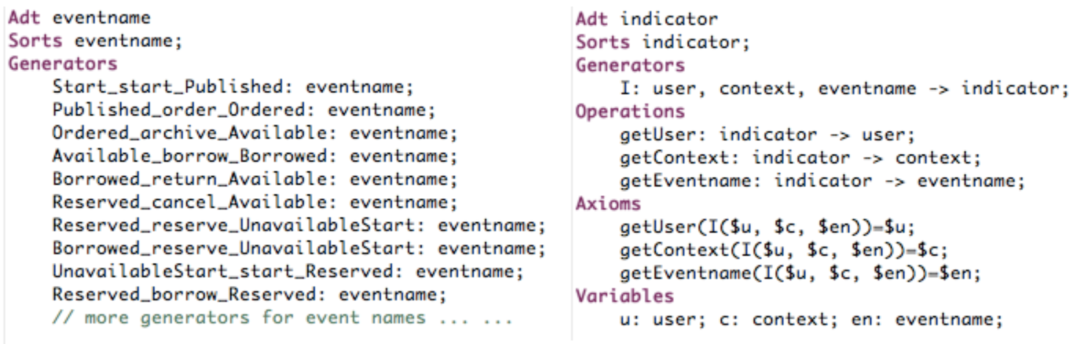
Concurrency, Security and Traceability Issues : PN is a modeling language well suited for describing the concurrent behavior of distributed systems. However, the basic UML statecharts we are using do not deal with concurrency. At any time only an adjacent set of states, pointed by the state transitions from current active state, is available to be reached, depending on the nondeterminism of events. Therefore, we do not allow concurrency in the transformed APN model. Furthermore, to enable checking the security properties transformed from the access control policies, we also need to record necessary the – bounded – input of the fired transition, e.g. user and context, as a “log”.

Finally, after model checking on APN, if there is any error found, we need to trace it back to statechart and understand where this error occurs. In order to teckle this issue we take advantage of the fact in a statechart every state is uniquely identified. Furthermore, the events in a statechart may have duplicated names, but there are no redundant events between the same *ordered*-pair of states (here *ordered* means the first state is the source while the second one is the destination). Thus we are able to add source and destination state names to each event, which makes every event distinguishable – see Figure 3 a).

To solve these issues, we define a special ADT called *indicator*, which consists of three fields that are able to record *actor* and *context* information, according to the relevant element fields defined in access control policies, as well as the renamed event names. The *indicator* is used to insure the simulation of transitions among states in UML statechart: at any time, there is one and only one indicator token in the APN model which makes it that only the transitions following the place holding this indicator token may fire. The places transformed from original states in UML statechart are of ADT type indicator. The input/output arcs of the transitions between these places consume/produce the indicator. This simulation mechanism restricts the concurrency of APN model and insures its concurrency logic is accordance with that of the source UML statechart. Meanwhile, when a secure transition fires, the indicator is responsible for recording information of actor, context and renamed transition name (working as a “log”) so that we can model check security properties and trace the counterexample back to that statechart. The ADT definition of *indicator* 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



```
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

produce the required temporal formulas we reuse and adapt the technique presented in [12], built to enable model checking access control policies in systems modeled as APN. The technique is based on the premisses that, **a)** each system activity requiring access control is modeled as an APN transition, and that **b)** 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 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.

The justification that verifying either points 1 or 2 is enough for our purposes is given in [11]. From here on we will continue our reasoning by relying solely on permission verification.

In terms of temporal logic, point 1 can be formally written for a given activity *act* as:

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

where t is a token containing log information which is stored in place *act_log*, recording log information about activity *act*. 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 one access permission is respected by the collected log tokens for *act*. Note that formulas $Perm_1^{act}(t) \ldots Perm_n^{act}(t)$ are disjunct, which allows for accesses to the activity *act* that respect *any* of the permissions for *act*. Finally, the **AG** temporal operator makes sure the disjunction is checked for all accesses to *act* during any run of the system. Notice that in order to check all declared permissions for a model one formula such as formula (1) needs to be built per activity.

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 so that; **a)** the temporal formulas to verify an activity need to be generated for multiple log places, rather than for only one, because, an APN place resulting from a transformation of a statechart state which acts as an arrival state for a transition of a given activity will act as a log place for that activity in the APN, and given that a statechart activity may correspond to several transitions in the APN (e.g. activities *borrow, return, cancel* in the statechart in Figure 1) we need to allow several log places for the same activity; and that, **b)** the temporal logic formulas are produced such that log tokens in a log place are verified for a permission only if they are logs for that particular activity, as 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 the *borrow* activity in the LMS example are the following:

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

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

Note that 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*. Note also that, 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 the 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). If there would be more permissions for activity *borrow* they would be disjunct with the existing conditions as presented in formula 1.

3.4 Mapping the Verification Result Back

When model checking on the obtained APN a temporal formula obtained from the access control permissions using the technique described in section 3.2, two results can be arise: 1) the formula is satisfied and the permission is respected; 2) the formula is not satisfied and a counterexample is found, meaning the permission was violated. For each transition in the APN transformed from a statechart permission an exhaustive set of input values exercising access control variables in the transition guard is provided in places adjacent to the transition (see the elliptic places in the APN in Figure 4). This allows simulating access control failures during model checking.

In the case a permission is violated it is possible to extract additional information from the counterexample provided by the model checker to pinpoint in the statechart where the problem is coming from. Given that each transition in the APN produced by transformation records in the log token the name of the original statechart transition and the names of both the original source and destination states (see section 3.2), we know which statechart transition led to the permission violation and can thus identify the faulty guard. Notice that because several formulas

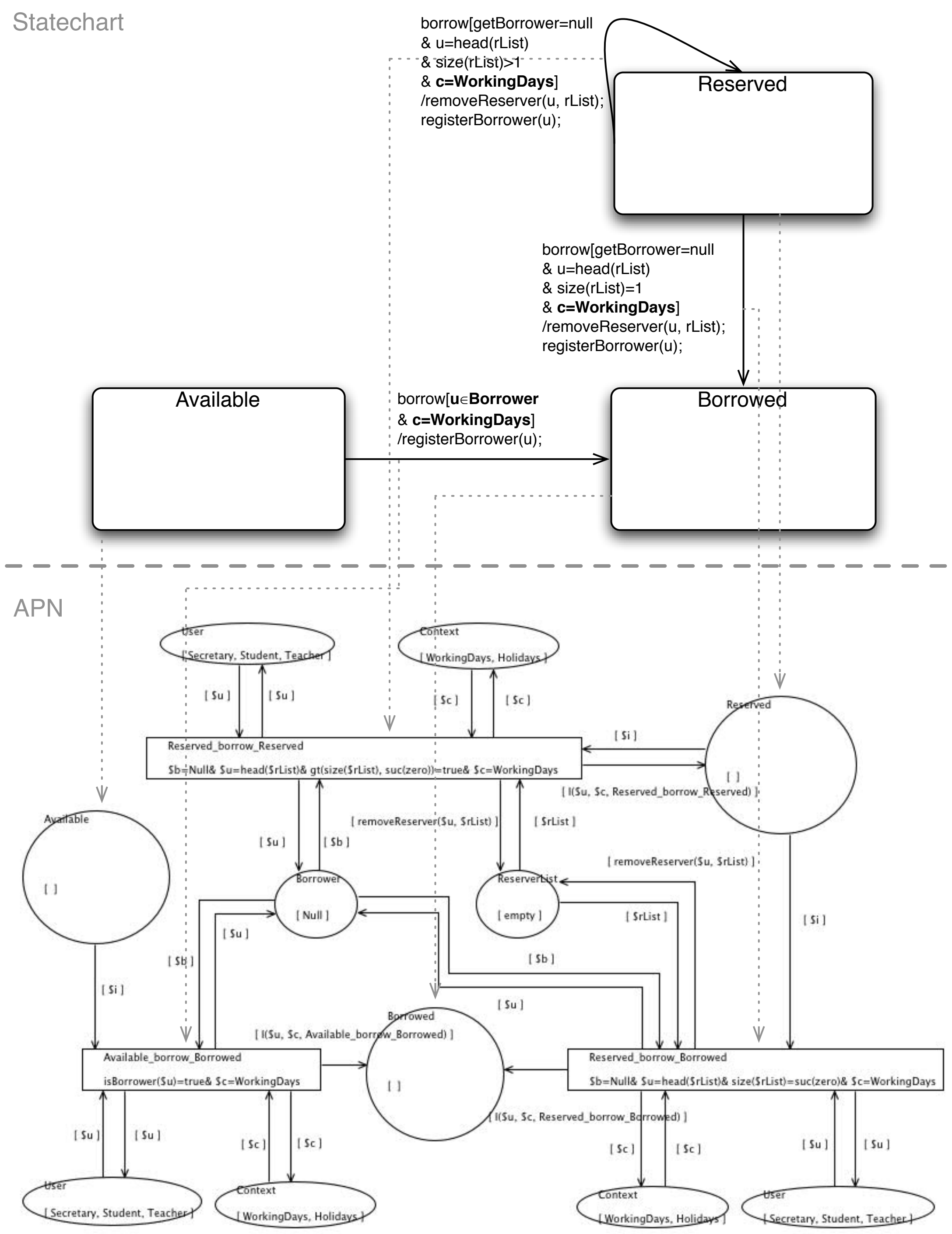
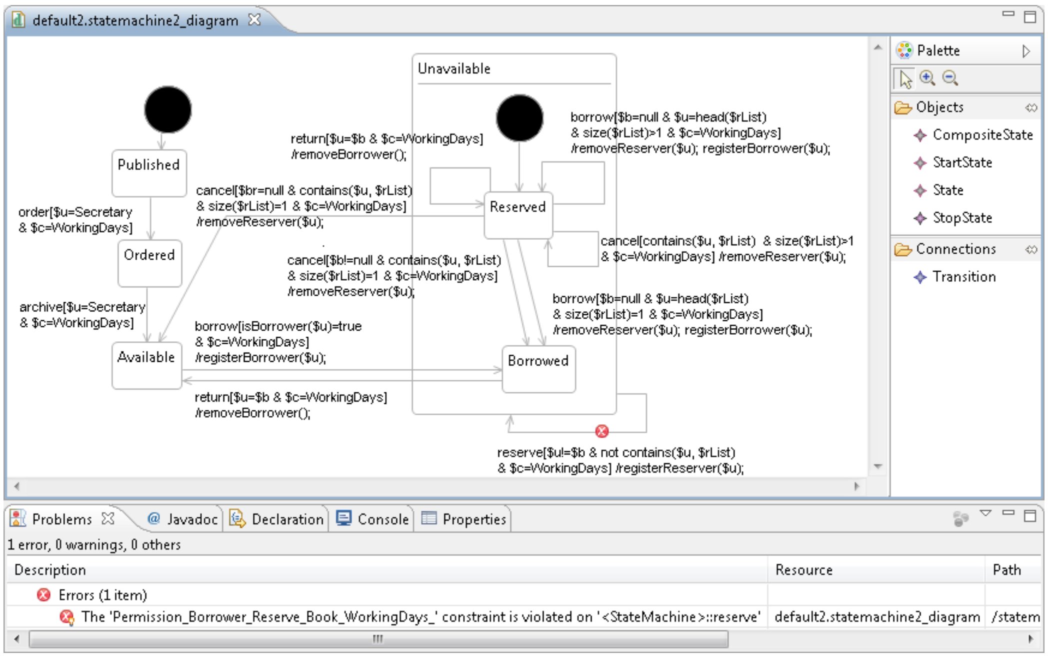

Statechart
borrow[getBorrower=null
& u=head(rList)
& size(rList)>1
& c=WorkingDays]
/removeReserver(u, rList);
registerBorrower(u);
Reserved
borrow[getBorrower=null
& u=head(rList)
& size(rList)=1
& c=WorkingDays]
/removeReserver(u, rList);
registerBorrower(u);
Available
borrow[u∈Borrower
& c=WorkingDays]
/registerBorrower(u);
Borrowed
APN
User
[Secretary, Student, Teacher]
Context
[WorkingDays, Holidays]
[$u]
[$u]
[$c]
[$c]
Reserved
[ ]
[$i]
Reserved_borrow_Reserved
$b=Null& $u=head($rList)& gt(size($rList), suc(zero))=true& $c=WorkingDays
[I($u, $c, Reserved_borrow_Reserved)]
[removeReserver($u, $rList)]
[$rList]
[$u]
[$b]
Available
[ ]
Borrower
[Null]
ReserverList
[empty]
[removeReserver($u, $rList)]
[$rList]
[$i]
[$u]
[$b]
[$b]
[$i]
[$u]
Borrowed
[I($u, $c, Available_borrow_Borrowed)]
[ ]
Available_borrow_Borrowed
isBorrower($u)=true& $c=WorkingDays
Reserved_borrow_Borrowed
$b=Null& $u=head($rList)& size($rList)=suc(zero)& $c=WorkingDays
[I($u, $c, Reserved_borrow_Borrowed)]
[$u]
[$u]
[$c]
[$c]
[$c]
[$c]
[$u]
[$u]
User
[Secretary, Student, Teacher]
Context
[WorkingDays, Holidays]
Context
[WorkingDays, Holidays]
User
[Secretary, Student, Teacher]

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

are checked in the APN, all the permission violating transitions can be simultaneously identified in the the statechart.

4 Supporting Tool


default2.statemachine2_diagram
Unavailable
Published
Ordered
Available
Reserved
Borrowed
order[$u=Secretary & $c=WorkingDays]
archive[$u=Secretary & $c=WorkingDays]
return[$u=$b & $c=WorkingDays] /removeBorrower();
cancel[$br=null & contains($u, $rList) & size($rList)=1 & $c=WorkingDays] /removeReserver($u);
cancel[$b!=null & contains($u, $rList) & size($rList)=1 & $c=WorkingDays] /removeReserver($u);
borrow[$b=null & $u=head($rList) & size($rList)>1 & $c=WorkingDays] /removeReserver($u); registerBorrower($u);
cancel[contains($u, $rList) & size($rList)>1 & $c=WorkingDays] /removeReserver($u);
borrow[$b=null & $u=head($rList) & size($rList)=1 & $c=WorkingDays] /removeReserver($u); registerBorrower($u);
borrow[isBorrower($u)=true & $c=WorkingDays] /registerBorrower($u);
return[$u=$b & $c=WorkingDays] /removeBorrower();
reserve[$u!=$b & not contains($u, $rList) & $c=WorkingDays] /registerReserver($u);
Palette
Objects
CompositeState
StartState
State
StopState
Connections
Transition
Problems
Javadoc
Declaration
Console
Properties

```
1 error, 0 warnings, 0 others
Description                                                                                          Resource                          Path
  Errors (1 item)
    The 'Permission_Borrower_Reserve_Book_WorkingDays_' constraint is violated on '<StateMachine>::reserve'   default2.statemachine2_diagram   /statem
```

Figure 5: Mockup interface of the access policy verification tool

In section 3 we have abstractly described a procedure for verifying access control policies in statecharts. In order to build a tool for this procedure such as sketched in Figure 2, we need to automate three main processes. The transformation of a statechart into an APN process (vertical right arrow of Figure 2), wich has been fully implemented as a model tranformation using the DSLTrans [1] transformation language. The complete model transformation description implemented in the DSLTrans language can be observed in [12]. The transformation of access control policies into temporal logic formulas (vertical left arrow of Figure 2) process, has not been tackled operationally yet, but can also be seen as a model transformation, parameterized by the relation between the elements of the statechart and the corresponding elements in the resulting APN. Finally, the verification of temporal logic formulas (bottom horizontal arrow of Figure 2) process, that is already automatic as it is implemented by the AlPiNA tool [3].

Given the implementation of these three processes and the interaction between them and a suitable graphical editor for statecharts and access control policies is mastered, it is possible to build a 'push-button' tool such that: given a set of access control policies and a statechart, the tool will highlight in the statechart which transition guards violate which access control policies. During this process, all the verification machinery can be hidden from the user. We have built a mockup of such a tool and a screenshot of its interface can be seen in Figure 5.

5 Discussion and Related Work

Several approaches have been developed to extend UML with security concerns [9, 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 already (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 pf Ray et al. [7], 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 behaviour. Such an integrated model is first required for verification of the system' security. Indeed, even if the task of integrating access control into a statechart is straightforward and systematic, its manual nature is error-prone and requires verification checks to be performed, as presented in this paper. An integrated model may also become productive in two possible exclusive ways: 1) security-oriented MBT [2, 6] for generating test artifacts to execute on the system implementation; 2) generation of the implementation code from the integrated statechart.

Concerning MBT, the objective of an integrated model is to offer a formal representation that allows covering behaviours 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. The second use of the integrated model would consist of developing a code generator from the integrated statechart. In that case the traditional separation of PDP/PEPs [5] with dedicated components would not exist anymore at design level.

In this paper, we synchronize UML statecharts with the APN underlying formalization. The idea is to align the UML and the APN powerful capabilities to enable access control verification and safe evolution. This implies some assumptions on the way access control is expressed into statecharts, which is important to make explicit: policies are formally implemented by adding user and context conditions on the state transition guards; all accesses are either explicitely allowed or denied; data types for the statecharts are implemented as ADT for consistency; in order to allow automated analysis and we explicitly bound all data types to a finite amount of values (e.g. in our LMS example lists of reservers are bounded to size 3). Since we are verifying access control the data type bounds does not in principle affect the completeness of the approach.

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

I this paper have presented a technique for verifying the implementation of access control policies in statecharts. We assume the access control policy implementation is modular is the sense that existing statechart transitions are strengthened in a predefined fashion. The approach is based on translating statecharts into APNs and 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. We have shown that our technique can be fully automated. 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, while the procedure and required instrumentation for checking 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, F. Fleurey, B. Baudry, and Y. Traon. A model-based framework for security policy specification, deployment and testing. In *Proceedings of the 11th MoDELS*, pages 537–552. Springer-Verlag.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

[12] 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.