Verifying access control in statecharts

Levi Lúcio[†], Qin Zhang[‡], Vasco Sousa[†], Tejeddine Mouelhi[†]

Laboratory for Advanced Software Systems (LASSY), University of Luxembourg[†] & Software Modeling and Verification Group (SMV), University of Geneva[‡]

Introduction

Access control is one of the main security mechanisms in software applications. Because of this it is important that any issue regarding access control is detected early in the development cycle. In a Model Driven Development [10] approach, this implies detecting the issues in the early modeling stages. A way of specifying access control rules is with Organization Based Access Control (OrBAC) [8] policies, that are implemented over a Statechart expressing the behavior of a system. The implementation of the OrBAC policies is achieved by reinforcing Statechart transitions with access control conditions. The goal of our work is to verify that a set of OrBAC policies has been well enforced in the system's Statechart.

In order to achieve this goal we transform Statecharts into Algebraic Petri Nets (APNs) and the original OrBAC policies into APN properties. We can then use a model checker to verify if the policies are well implemented in the Statechart, by using the translated artifacts. The model checking results at the APN level are then remapped onto the statechart so that faulty statechart transitions regarding OrBAC policies can be identified and corrected.

The **Library Management System** can be used by: a Secretary to order and archive books; Borrowers (Professors or Students) to borrow, return, reserve books, or cancel reservations on books. Library activities are conducted only during Workdays

OrBAC policies:

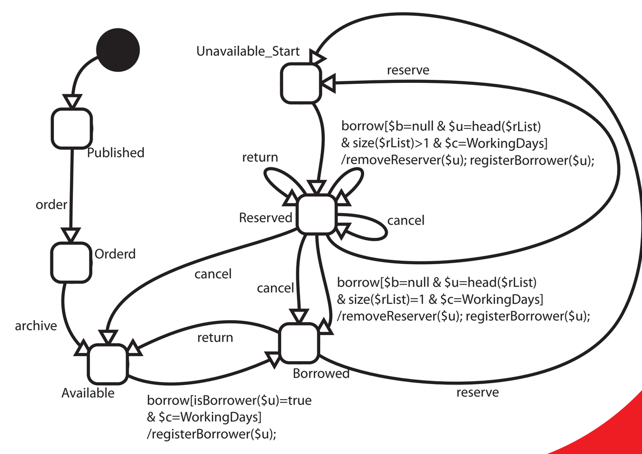
Permission(Secretary, Order, Book , WorkingDays)
Permission(Secretary, Archive, Book , WorkingDays)
Permission(Borrower, Borrow, Book , WorkingDays)
Permission(Borrower, Return, Book , WorkingDays)
Permission(Borrower, Reserve, Book , WorkingDays)
Permission(Borrower, Cancel, Book , WorkingDays)

[11]

APN properties:

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

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



```
import "user.adt"
import "context.adt"
import "eventname.adt"

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;
```

```
import "boolean.adt"

Adt user

Sorts user;

Generators
  Null: user;
   // for removeBorrower() method
  Secretary: user;
  Teacher: user;
  Student: user;

Operations
  isBorrower: user -> bool;
   // implementing "u \in Borrower"

Axioms
  isBorrower(Null)=false;
  isBorrower(Secretary)=false;
  isBorrower(Teacher)=true;
  isBorrower(Student)=true;
```

[12]

DSLTrans [1]

ALPINA [3]

Verification Process

● From the Statechart events we produce a set of Algebraic Data Types (ADT) type to algebraically define generators for those events.

● The Statechart is flattened by placing all Nested States at the same hierarchical level. To achieve this, all transitions that previously pointed to a nested state are redirected to its inner Start State. The Start State is renamed by appending the name of the nested state that contains it. The Transitions that originated from a Nested State are replicated to all States previously belonging to that Nested State.

● ● We combine the generated APNs with the flattened Statechart to produce an Algebraic Petri Net where we have: (1) a place for each State, typed by an ADT for logging purposes; (2) a place for each Event with an ADT token of the event type; and (3) for each Statechart transition an APN transition, where the input arcs connect to the corresponding State and Event Places and the output arcs connect back to the Event Place and the target State Place. We use the following naming convention: Places are named after their respective States or Events in the Statechart; Transitions are named after their corresponding transition in the Statechart appended with the source and target State names. This naming convention allows us keep a mapping of elements from the Statechart into elements of the APN.
The Start Place is initialized with an Indicator token. The Indicator type dynamically collects all the information of the last transition triggered. This allows us to retrieve from a model checker counterexample the event that caused the access control violation and link it back to the corresponding transition in the original Statechart.

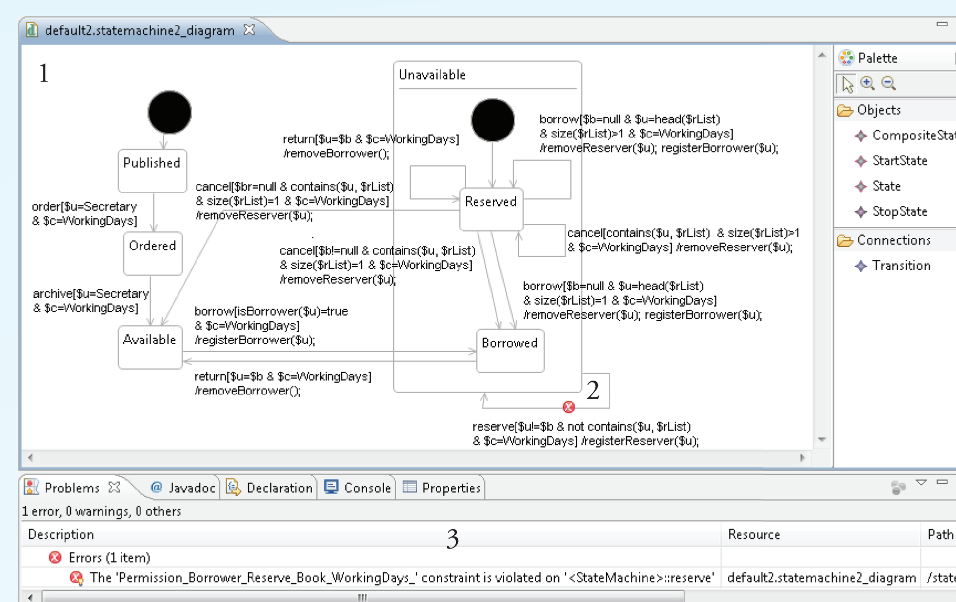
● The OrBAC policies are transformed into temporal formulas to allow the verification of the APN generated from the statechart.

● The generated properties are verified on the APN generated from the statechart. In case of failure a counter example is produced allowing us to observe the statechart access control issue that triggered the failure.

Suporting Tool

Having the main steps of the approach fully automated or algorithmically described, we are now focusing on the development of a push button tool that abstracts the inner workings of the verification mechanism from the modelers and presents the verification results in a seamless manner.

In this tool the user will be able to model the Statecharts in the implemented editor (1), check the results of the verification on the Statechart (2), and view the information of the counter example to help the modeler understanding which particular policies failed and with which parameters (3).



Conclusion

We have shown that we can verify access control policies in Statecharts, by transforming them to APNs and retrieving the verification information back to the Statechart abstraction. This is done in a automated and transparent way. The goal of this work is to improve the state of the art of the tooling for modeling system behavior including access control concerns.

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.

[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.

[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.

[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/bla.pdf, 2011.

[12] Q. Zhang. 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.

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

levi.lucio@cs.mcgill.ca
qin.zhang@unige.ch
vasco.dasilva@uni.lu
tejeddine.mouelhi@uni.lu