Specifying UML Protocol State Machines in Alloy*

Ana Garis[1], Ana C.R. Paiva[2], Alcino Cunha[3], and Daniel Riesco[1]

[1] Universidad Nacional de San Luis, Argentina
{agaris,driesco}@unsl.edu.ar
[2] DEI-FEUP, Universidade do Porto, Portugal
apaiva@fe.up.pt
[3] HASLab, INESC TEC and Universidade do Minho, Portugal
alcino@di.uminho.pt

Abstract. A UML *Protocol State Machine* (PSM) is a behavioral diagram for the specification of the external behavior of a class, interface or component. PSMs have been used in the software development process for different purposes, such as requirements analysis and testing. However, like other UML diagrams, they are often difficult to validate and verify, specially when combined with other artifacts, such as *Object Constraint Language* (OCL) specifications. This drawback can be overcome by application of an off-the-shelf formal method, namely one supporting automatic validation and verification. Among those, we have the increasingly popular Alloy, based on a simple relational flavor of first-order logic. This paper presents a model transformation from PSMs, optionally complemented with OCL specifications, to Alloy. Not only it enables automatic verification and validation of PSMs, but also a smooth integration of Alloy in current software development practices.

Keywords: UML, OCL, Protocol State Machines, Alloy.

1 Introduction

UML state machine diagrams can be used to describe the dynamic behavior of a system or part of it. There are two variants, namely *Behavioral State Machines* and *Protocol State Machines* (PSMs) [16]. While the former is used to express behavior of various elements (e.g., class instances), the latter is a way to define the allowed behavior of classifiers; namely, classes, interfaces and components. Therefore, PSMs enable the specification of a lifecycle for objects or an order of invocation of its operations and to express usage protocols. PSMs typically omit implementation details and allow the use of the *Object Constraint Language* (OCL) [17] to specify state invariants and transitions' pre- and post-conditions. As such, PSMs are well-suited to be integrated in a *Model Driven Engineering* (MDE) context, allowing the specification of the allowed behavior of a classifier

* Work partially supported by FCT (Portugal) under contract PTDC/EIA-EIA/103103/2008.

D. Latella and H. Treharne (Eds.): IFM 2012, LNCS 7321, pp. 312–326, 2012.
© Springer-Verlag Berlin Heidelberg 2012

in a highly abstract way. PSMs have been used for the specification of dynamic views during the analysis phase, and they have been exploited for the generation of class contracts, test code and test cases [19,18,3,21].

Since UML is the industry de facto language for modeling, there exist myriad tools supporting it. In particular, UML integration in the *Model Driven Architecture* (MDA), the MDE initiative of the OMG, led to an explosion of UML based MDE tools, such as code generators and reverse engineering frameworks. Unfortunately, in part due to the fact that UML has only an informally given semantics, most of these tools do not offer adequate support for *Verification and Validation* (V&V).

Formal methods have been successfully applied in the formalization and V&V of UML state machines [22,23,12,5,21]. However, the consistency between these and other UML specification artifacts has rarely been addressed. Moreover, most of these formalizations rely on traditional formal methods, that are avoided by software developers due to the inherent complexity that makes them hard to learn and use.

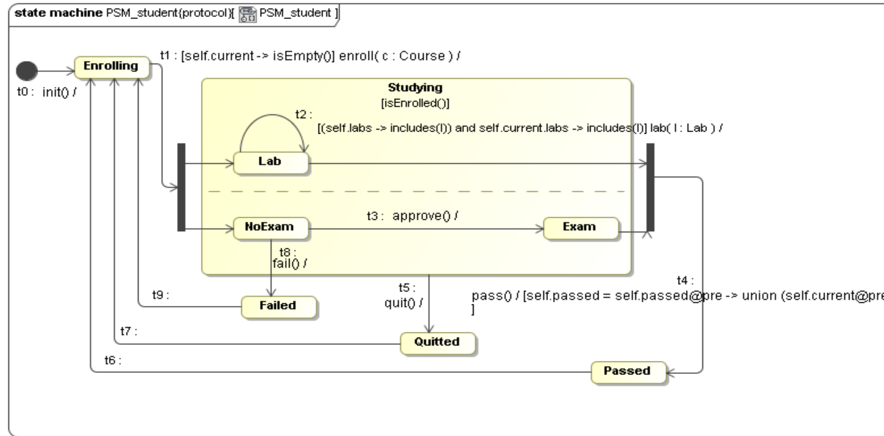
The objective of this paper is precisely to tackle both these issues: we show how both PSMs and Class Diagrams (CDs) enhanced with OCL can be formalized in Alloy [10] lightweight formal modeling language; and we present an approach to develop V&V tasks using Alloy Analyzer. This formalization allows us to simulate and verify the consistency between UML artifacts and to perform other V&V activities, such as detect unreachable states or invalid transitions. The formalization of PSMs is implemented using the model transformation language ATL [4]. For the formalization of OCL, we use the UML2Alloy tool [2] and the approach presented in [1] but adapted to support dynamic behavior.

The rest of the paper is structured as follows. Section 2 shows a case study in order to explain our proposal. Section 3 describes preliminary concepts referred to PSMs and Alloy. Section 4 presents our approach. Section 5 discusses the related work. Finally, Section 6 summarizes the contributions and exposes some ideas for future work.

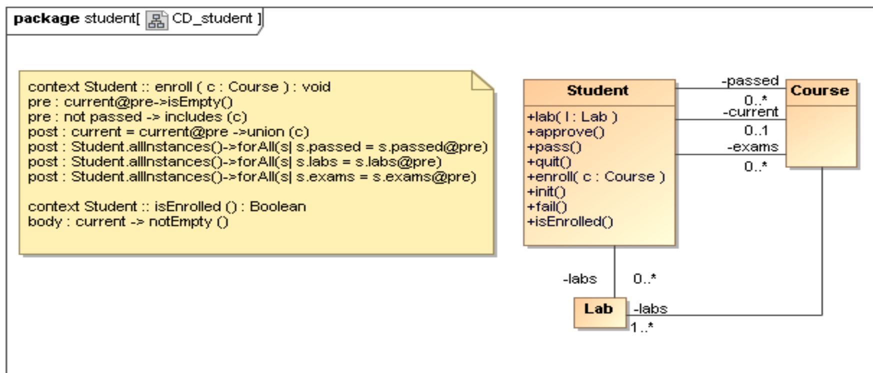
2 Case Study

Figure 1(a) shows an example of a PSM. It is a simplified model of a student coursing a career. The PSM describes the intended behavior of the class `Student`, specified also in CD enriched with OCL shown in Figure 1(b). Initially, the student is not enrolled to any course. The `enroll` operation enrolls the student in a course, and enables him to attend the course exam, while performing `lab` assignments. If he is `approve`d in the exam and completes all mandatory lab assignments he can `pass` the course. If he `fail`s the exam, he also fails the course. At any time, he can `quit` the course. After failing, quitting, or passing a course he returns to the `Enrolling` state where he can enroll in another (or the same) course.

Note that the transitions labeled with operations `enroll`, `lab`, and `pass` have attached pre- and post-conditions defined in OCL. These constrain when such



(a) Protocol State Machine



(b) Class Diagram enriched with OCL

Fig. 1. Coursing case study

operations can be invoked and their effect on the modeled student state (namely, associations `passed`, `current`, `exams`, and `labs`). Likewise, the `Studying` composite state is characterized by an invariant, forcing the student to be enrolled in order to attend the exam and the labs. The `Student` class operations are further specified in OCL. Due to space limitation we only show the specification of the operation `enroll` and the predicate `isEnrolled` in figure 1(b). Note that the OCL specification includes frame conditions, such as `Student.allInstances()->forAll(s|s.passed=s.passed@pre)`, stating which attributes should remain unchanged when executing an operation. It is not consensual whether such frame-conditions must be specified, and some authors assume an implicit invariability assumption, stating that what is not mentioned in a post-condition should remain unchanged. However, such assumption may lead to ambiguities in post-condition interpretation [11], and we require

them to be explicitly specified, however, this step will be automated to release the user of this, potential, tedious task.

When PSMs are combined with UML static models such as CDs, both annotated with OCL, the V&V task substantially increases in complexity. Namely, it is no longer trivial to manually check consistency and detect specification errors, such as unreachable states. For instance, is it possible to ensure that every student has the opportunity to pass a course, by eventually reaching the `Passed` state? And upon reaching such state, are the `Student` attributes consistent, namely, is the course part of the `exams` association that stores the exams successfully completed by the student? We will show how an Alloy formalization of PSMs, CDs, and OCL, enables automatic verification of properties such as these using the Alloy Analyzer.

3 Preliminary Concepts

We present preliminary concepts related to PSMs and Alloy. Section 3.1 explains syntactic and semantics issues of PSMs. Section 3.2 introduces Alloy, and describes how UML models enriched with OCL can be formalized using an Alloy idiom tailored for dynamic specification.

3.1 UML Protocol State Machines

The abstract syntax of a PSM is shown in Figure 2. A PSM is modeled using a directed graph where the nodes represent *states* and the arrows *transitions* between states. A transition is an expression of the form `[precondition] operation / [postcondition]`. Pre- and post-conditions can be informally defined, however UML prescribes the use of OCL for their formal specification. State invariants can be associated to each state. A state invariant should be satisfied whenever the related state is active. There exist three kind of states: *simple*, *composite* or *submachine*. The first one is a state without sub-states (neither regions nor submachines), the second one can be composed of two or more orthogonal regions, and the third one allows the specification of inner state machine (submachines). For instance, the PSM in Figure 1(a) shows a composite state, `Studying`, with two regions which we will name them as `R1` and `R2`. A region may optionally have a final state and an initial pseudostate. Other examples of pseudostates are: *join*, *fork*, *junction* and *choice*.

A transition is enabled when its source state is active, the source state invariant holds and the pre-condition associated to its operation is true. Transitions are triggered by events which represent invocation of operations. When the same operation is referred by more than one transition, if it is invoked, different transitions will be enabled resulting in a *conflict*. The UML standard [16] prioritizes firing using the state hierarchy: transitions from deeper sub-states have higher priority over the ones from including composite states.

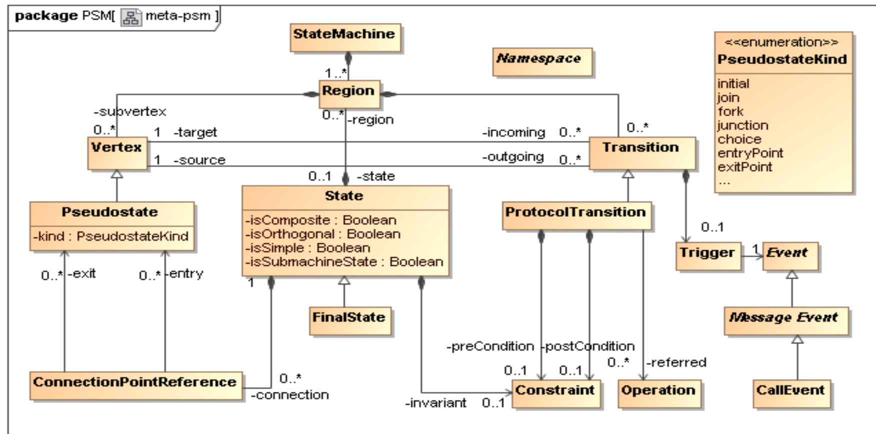


Fig. 2. Metamodel of UML Protocol State Machines

3.2 Alloy

Alloy [10] is a formal modeling language based on a relational flavor of first-order logic. Alloy is supported by the Alloy Analyzer, a SAT (satisfiability problem) based tool that enables automatic model V&V. Alloy Analyzer is inspired by model checkers, but it is implemented as a solver, performing verification within a bounded scope.

The abstract syntax of Alloy language is described in the metamodel presented in Figure 3. An Alloy module consists of a module header, a set of imports and zero or more paragraphs. The *module header* is a name of the module where paragraphs are defined. The *import* keyword specifies the inclusion of other modules. A *paragraph* can either be a signature declaration, a constraint, an assertion or a command.

A *signature declaration* denotes a set of atoms. An atom is a unity with three basic properties: it is indivisible, immutable and uninterpreted. Signature declarations can introduce *fields*. A field represents a relation among signatures. *Facts*, *predicates* and *functions* describe invariants, named constraints, and named expressions, respectively. The difference between a fact and a predicate is that the first one always holds while the second one only holds when invoked. *Assertions* allow the expression of properties that are expected to hold as consequence of specified facts. Finally, *commands* instruct the Alloy Analyzer to perform particular analysis using two possible instructions: *run* and *check*. The first checks model consistency by requesting a valid instance, and the latter verifies an assertion by searching for a counterexample. Both commands optionally define a scope, bounding the number of instances allowed for each signature.

Specifying OCL Annotated Class Diagrams in Alloy. Alloy's logic is quite generic and does not commit to a particular specification style [10]. There

Fig. 3. Metamodel of Alloy

is also no predefined way to model dynamic behavior, since instances can only be populated with immutable atoms. A standard way to circumvent this is to introduce a signature denoting the overall state of the system, and model operations as predicates that specify the relationship between pre- and post-states. Two variants of this idiom are known respectively as *global state* and *local state*. In the first one, all mutable fields are defined in the global state signature. In the second one, an extra column at the end of each mutable field is added locally to represent the state signature (usually denoted `Time`). The local state is often simpler than other competing idioms for modeling the dynamics of complex systems [24] and well-suited for modeling state machines [10]. In this idiom, operations are modeled as predicates that specify the relationship between pre- and post-states. To be more specific, an operation `op` is specified using a predicate `pred op[...,t,t':Time] {...}` with two special parameters `t` and `t'` denoting, respectively, the pre- and post-state. Predicates of the form `pred q [...,t:Time] {...}` are used to specify boolean queries. A formal characterization of this idiom can be found in [8], together with a translation to UML CDs enriched with OCL.

Figure 4 shows how the OCL annotated CD of Figure 1(b) can be specified in Alloy using the local state idiom. The `passed`, `current`, `labs` and `exams` associations are modeled as mutable relations in `Student`. Cardinality constraints at association ends yield corresponding multiplicities in field declarations. For example, the keyword `lone` in the field `current` limits the cardinality of the set `Course` to zero or one instances, when it is associated to `Time`. OCL pre- and post-conditions in operations are modeled by relational expressions evaluated at state `t` and `t'`, respectively. In Alloy *everything is a relation*. Therefore, the relational composition operator can be used to various purposes. In particular,

```
module student
sig Time {}
sig Student {
  passed  : Course     -> Time, current : Course lone -> Time,
  labs    : Laboratory -> Time, exams   : Course -> Time      }
sig Course { labs : some Laboratory }
sig Laboratory {}
fact { labs in Course lone -> Laboratory}
pred enroll [s : Student, c : Course, t,t' : Time] {
     no s.current.t
     c not in s.passed.t
     current.t' = current.t + s -> c
     passed.t' = passed.t
     labs.t' = labs.t
     exams.t' = exams.t }
pred isEnrolled [s : Student, t : Time] { some s.current.t }
```

Fig. 4. Coursing example in Alloy

when `t` is composed with a mutable field, it denotes its value at the pre-state. For example, `s.current.t` denotes the course of a student `s` prior to method invocation. In Alloy there is no implicit `self` object, and an explicit self parameter must be included in the operation signatures. This explicit parameter must then be used whenever `self` is implicitly used. For example, the OCL precondition `not passed->includes(c)` of method `enroll`, stating that a student can enroll only in courses not yet passed, can be specified in Alloy as `c not in s.passed.t`. There are many challenging issues to address when implementing an automatic translation from OCL to Alloy, such as the translation of OCL pre- and post-conditions. These have been addressed but not implemented in [1]. We will use the same approach, as in [1], to translate CDs annotated with OCL to Alloy but considering dynamic issues. In particular, we will generate a specification conforming the local state idiom; namely, to translate OCL pre- and post-conditions to predicates and to include an extra column `Time` at the end of each mutable field. Following this approach, an Alloy model equivalent to the one of Figure 4 can be generated from the UML model of Figure 1(b).

4 Specifying Protocol State Machines in Alloy

We present an approach to specify PSMs in Alloy. Firstly, we specify how CD enriched with OCL (CD+OCL) can be integrated in order to be used by a PSM. Then, we describe the transformation of a PSM to Alloy and we show how to perform V&V tasks using the Alloy Analyzer. The proposal is explained using a case study. Finally, we formalize the transformation by defining the rules in ATL.

4.1 Importing UML Class Diagram into Alloy

Two separate Alloy modules will be used: one to specify the CD+OCL, and another to specify the PSM. The latter imports the former, since the transformation from PSMs to Alloy requires the specification of classes, attributes, relations and operations, corresponding to the CD+OCL, in the local state idiom. This separation of concerns allows us to directly reuse part of the output of UML2Alloy tool, and, if the user makes changes to the Alloy model, it is possible to translate it back to a CD+OCL using Alloy2OCL, another tool previously developed for this particular effect [8].

4.2 PSM's States and Transitions

PSM simple states can be modelled directly in Alloy using singleton signatures. On the other hand, composite states and regions can be modeled using abstract signatures, to be extended by the signatures modeling its sub-states. At the top of the state hierarchy we will have the signature `State` containing all states. The pseudostate `Initial` is also modeled similarly to simple states. Following these rules, the states of our running example, in figure 1(b), can be specified as follows.

```
abstract sig State {}
abstract sig Studying extends State {}
abstract sig R1 extends Studying {}
abstract sig R2 extends Studying {}
one sig Lab extends R1 {}
one sig NoExam, Exam extends R2 {}
one sig Initial, Enrolling, Passed, Quitted, Failed extends State {}
```

The PSM itself is modeled using a singleton signature `PSM`, with a single mutable relation `state`, that, for each time instant and instance of the associated class returns the set of active states.

```
one sig PSM { state: State some -> Student -> Time }
```

Similarly to operations, transition between normal states will be modeled by a predicate that, for each instance of the class associated with the PSM, relates the pre- and post-state. The metamodel of PSMs (see Figure 2) establishes that a protocol transition can refer to zero or more operations. To simplify the presentation, we will limit this set to at most one operation per transition. The transition predicate invokes the referred operation, whose predicate is defined in the imported model. If no operation is referred, the transition predicate invokes a special `nop` predicate, with frame-conditions that constrain all mutable fields to remain unchanged. Each transition predicate also includes two constraints to model the dynamics of the machine: one checks if all the source states are active in the pre-state for the given instance, and the other changes the relation `state`, so that its target states are active in the post-state. For example, transition `t3` of figure 1(a) can be modeled as follows.

```
pred t3 [s : Student, t,t' : Time] {
  NoExam in PSM.state.t.s
  PSM.state.t' = PSM.state.t - (NoExam -> s) + (Exam -> s)
  approve[s,t,t'] }
```

The relational expression `NoExam -> s` denotes the cartesian product of `NoExam` and `s`. Since these are singletons, in this case it denotes just a tuple. As such, the second constraint ensures that relation `state` has the same pre- and post-state for all student instances, except for `s`, which changes its state from `NoExam` to `Exam`.

Some transitions are not translated as predicates. In particular, this is the case of incoming transitions of join pseudostates and outgoing transitions of fork pseudostates. Their source and target states will be handled by the respective outgoing and incoming transitions. For instance, consider the fork pseudostate whose incoming transition is `t1`, with two outgoing transitions leading to the `Studying` composite state, respectively to `Lab` and `NoExam`. These states will be activated by the predicate modeling `t1`, which is defined as follows.

```
pred t1 [s: Student, t,t' : Time] {
  Enrolling in PSM.state.t.s
  PSM.state.t' = PSM.state.t - (Enrolling -> s) + ((Lab + NoExam) -> s)
  some c : Course {  (no s.current.t) && enroll[s,c,t,t']  } }
```

Notice the usage of an existential quantifier, `some`, to introduce the parameters of operation `enroll`, and the inclusion of the Alloy translation of the specified OCL pre-condition before its invocation.

State invariants are enforced using a fact for each state that declares them. For composite states, the invariant must hold whenever any of its sub-states is active. For example, the state invariant of `Studying` is specified as follows.

```
fact Studying {
 all t:Time, s:Student | some (PSM.state.t.s & Studying)=> isEnrolled[s,t]}
```

4.3 Finite Execution Traces

To model finite execution traces, a total order will be imposed on the `Time` signature using the predefined Alloy library `ordering`. This library defines useful relations to manipulate the total order, namely `first` to denote the first atom, and `next`, a binary relation that, given an atom, returns the following atom in the order.

The `next` relation must be restricted to relate only `Time` atoms for which a transition predicate holds for one of the instances of the class associated with the PSM. Moreover, at the `first` time atom all instances must be at the `Initial` pseudostate. Both these constraints are defined in the special fact `Traces`.

```
fact Traces {
  all s : Student | PSM.state.first.s = Initial
  all t : Time, t' : t.next | some s : Student {
    t0[s,t,t'] or t1[s,t,t'] or t2[s,t,t'] or t3[s,t,t'] or t4[s,t,t'] or
    t5[s,t,t'] or t6[s,t,t'] or t7[s,t,t'] or t8[s,t,t'] or t9[s,t,t'] }}
```

Firing Priority. Our running example does not contain conflicting transitions. If two transitions `high` and `low` could potentially be in conflict (that is, the same operation is invoked in both), and `high` has higher priority then `low`, then fact `Traces` would invoke them using

```
high[s,t,t'] or  (not high[s,t,t'] and low[s,t,t']).
```

4.4 Verification and Validation of UML Diagrams

With both the PSM and the CD+OCL specified in Alloy, we can now check their consistency by asking for an execution trace. This can be done with the command `run`, that instructs the Alloy Analyzer to look for a valid instance of the model.

For instance, consider the command `run {} for 2 but 1 Student, 15 Time`. The keyword `for` can be used to define a scope bounding the number of atoms allowed for each signature. The keyword `but` establishes an exception for the boundary defined by `for`. In this case, the number of `Student` atoms is limited to 1 and the number of `Time` atoms is extended to 15.

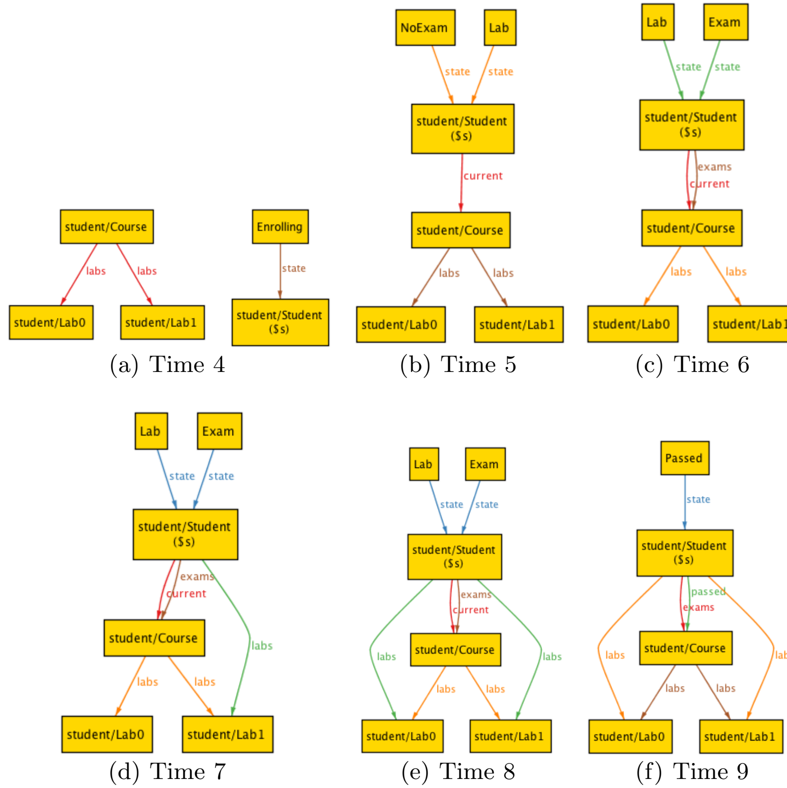
In this particular example, the `run` command returns a valid trace and thus the PSM is consistent with the OCL annotated CD. However, this notion of consistency is very basic and does not suffice in order to validate the models. A more reliable notion is to check that every state of the PSM is reachable. For example, is it possible for a student to pass at least one course? Again, using a `run` command, we can ask the analyzer to return a trace where a student reaches state `Passed`.

```
run {some t : Time, s : Student | Passed in PSM.state.t.s
} for 2 but 1 Student, 15 Time
```

In this case the analyzer cannot find a valid execution trace, meaning that state `Passed` is unreachable in 15 steps. Obviously, this means that there is some problem with one of the models. Looking back at the PSM of Figure 1(a) we can see that there is a problem with the pre-condition of transition `t2`, that requires the (to be completed) lab assignment to already been completed before. After inserting the missing `not`, changing that pre-condition to `(not self.labs->includes(l)) and self.current.labs->includes(l)` the analyzer returns a valid execution trace. Figure 5 presents 6 consecutive states of this trace moving the student from state `Enrolling` to `Passed`: the student is first approved in the exam and then completes the two mandatory lab assignments. Since reachability is a desirable property, we will define a rule transformation to generate similar runs for each simple state of the PSM.

There are other examples of V&V tasks that can be performed using the Alloy Analyzer. For example, we can check that, when a student is in the `Passed` state, the `exams` relation contains the `current` course, using the following command.

```
check {all t :Time, s : Student { Passed in PSM.state.t.s =>
           s.current.t in s.exams.t  } } for 10 but 1 Student, 30 Time
```



(a) Time 4 (b) Time 5 (c) Time 6

(d) Time 7 (e) Time 8 (f) Time 9

Fig. 5. Trace leading to a passed course

As previously mentioned, `check` verifies the assertion by searching for a counterexample. We specify a big scope in order to be confident that the assertion holds. In particular, we bound the number of atoms allowed for each signature to `10` but `30` for `Time`. Since no counterexamples are returned with such big scope, we can be more confident that this assertion holds.

4.5 Implementation

Our proposal was prototyped in a model transformation tool using the MDA approach: First, both the PSM and Alloy metamodels were specified, and then we defined a mapping from PSM elements to Alloy elements using the model transformation language ATL. Some of the ATL rules are presented in Figure 6.

Rule `Model2Module` maps a UML model of one PSM to an Alloy module, declaring the respective header and imports. Rule `PSM2Sig` creates the singleton signature `PSM` with the dynamic relation `state`. Rule `CompositeState2Sig` creates an abstract signature extending `State` for each PSM composite state. `SimpleState2Run` generate a run command for each simple state of the PSM. The ATL transformation is available for download at `http://sourceforge.net/p/psm2alloy`.

5 Related Work

Likewise other UML diagrams, the semantics of PSMs is quite ambiguous, and several attempts have been made to formalize it. For example, Bauer et al. [5] propose a model-theoretic semantics, based on labelled transition systems, for three different perspectives (namely, implementator's view, user's view, and interaction view) of the PSM. Here we will follow the user's view, that a PSM specifies the allowed call sequences on the classifiers operations.

The joint V&V of PSMs and other UML diagrams using traditional formal methods has also been proposed. For example, Lanoix et al. [12] use the B method to verify the interoperability and refinement of UML components, specified using CDs, sequence diagrams and PSMs. However, the focus is not on consistency and class methods are specified directly in B instead of the UML standard OCL. The consistency of an UML classes and the associated PSM has previously been addressed by Rash and Wehrheim [20], using a formalization to CSP. Again, classes are not specified with OCL, but using Object-Z. Lightweight formal methods have also been used for similar purposes before. In particular, Nimiya et al. [15] propose a method for verifying consistency of UML state machine diagrams and communication diagrams using Alloy, but it does not consider integration with CDs neither OCL. Ries [21] formalizes a subset of UML CDs and PSMs in Alloy, as part of a lightweight model-driven test selection process called SESAME, but it does not consider complex UML elements, such as composite states or fork and choice pseudostates, neither addresses the consistency of PSMs with CDs+OCL.

The relationship between CDs+OCL with Alloy has been extensively studied by Anastaskis et al. [2], resulting in a prototype model transformation tool named UML2Alloy that formalizes that relationship as a shallow embedding. Maoz et al [13] proposed a formalization of CDs using a deep embedding to Alloy, to support UML features not directly expressible in Alloy, such as multiple inheritance. However, both these formalizations yield Alloy specifications which are not well-suited to model dynamic behavior, namely by not making clear the distinction between pre- and post-states in method specification. Anastasakis [1] showed how UML2Alloy could be extended to solve that issue, but that extension was never incorporated into UML2Alloy. These formalizations did not consider PSMs, and thus left out some OCL features related to state machines, namely the OCL predefined operation *oclIsInState*, which evaluates whether an object is in a specific state.

UML has also been mapped to Alloy for model V&V of particular case-studies. We present three examples: the first one uses the Alloy Analyzer for formal security evaluation in a methodology called *Aspect-Oriented Risk-Driven Development* [9]; the second one describes a proposal for Alloy specification of Aspect-UML models, a UML Profile for extending UML with Aspect-oriented concepts [14]; the third one explains an approach to translate UML models, specified with OntoUML, for model validation using Alloy [6]. These examples, likewise [2] and [13], make evident Alloy potential for UML V&V, but they do not consider UML dynamic diagrams such as PSMs.

```
create OUT : MMAlloy from IN : MMUml;
rule Model2Module {
 from s  : MMUml!Model (
           MMUml!ProtocolStateMachine.allInstances()->size() =1 )
 to mId  : MMAlloy!ModuleId ( name  <- s.name    ),
    hd   : MMAlloy!Header   ( moduleId <- mId ),
    stId : MMAlloy!SigId    ( name  <- 'State'   ),
    st   : MMAlloy!Sig      ( abs   <- true, sigId <- stId ),
    m    : MMAlloy!Module   ( header<- hd,
           imports    <- MMAlloy!Import.allInstances(),
           paragraphs <- MMAlloy!Paragraph.allInstances() )
}
rule PSM2Sig {
 from s : MMUml!ProtocolStateMachine
 to sig : MMAlloy!SigId  ( name <- 'PSM' ),
    var : MMAlloy!VarId  ( name <- 'state' ),
   decl : MMAlloy!RelDecl(
          varIds <- var, mult <- #some, sigs <- getSigId('State')...),
    psm : MMAlloy!Sig (
          abs  <- false, mult <- #one, sigId <- sig, decls <- decl  )
}
rule CompositeState2Sig {
 from s  : MMUml!State  ( s.name <> '' and  s.isComposite() )
 to sigId: MMAlloy!SigId( name <- s.name       ),
   sigEx : MMAlloy!SigId( name <- s.getRegion() ),
   sig   : MMAlloy!Sig  ( abs <- true, sigId <- sigId, exts  <- sigEx )
}
rule SimpleState2Run {
 from s : MMUml!State ( s.name <> '' and  s.isSimple() )
 to
  v1  : MMAlloy!VarId ( name <- 't' ),
  v2  : MMAlloy!VarId ( name <- 's' ),
  dEx1: MMAlloy!DeclExpr (vId <- v1, sigId <- getSigId('Time')),
  dEx2: MMAlloy!DeclExpr (vId <- v2, sigId <- getSigCl() )),
  ex  : MMAlloy!BinOpExpr(op  <- #join, rEx <- ex2,lEx <- getSigId('PSM')),
  ex2 : MMAlloy!BinOpExpr(op  <- #join, rEx <- ex3,lEx <- getRel('state')),
  ex3 : MMAlloy!BinOpExpr(op  <- #join, rEx <- v1 , lEx <- v2 ),
  fm  : MMAlloy!CpExForm (op  <- #incl, lEx <- getSigId(s.name),rEx <- ex),
  qF  : MMAlloy!QuDcForm ( q  <- #some, decls<- Set{dEx1,dEx2},forms<- fm),
  tyT : MMAlloy!TypeScope( num <- '15',scopeable  <- getSigId('Time')),
  tyC : MMAlloy!TypeScope( num <- '1' ,scopeable  <- getSigIdClass()),
  sc  : MMAlloy!ButScope ( num <- '2', typeScopes <- Set{tyT,tyC}),
  run : MMAlloy!SimpleRun( form<- qF, scope <- sc ) }
```

Fig. 6. ATL rules to map a PSM to Alloy

6 Conclusions and Future Work

We have shown how both PSMs and CDs enriched with OCL can be formalized in Alloy, using the local state idiom to handle dynamics. This formalization enables us to perform automatic V&V of these UML diagrams using the Alloy Analyzer. In particular, it allows us to check they are consistent with each other, a fundamental property ignored by current UML tools. The proposed PSM formalization was prototyped using ATL. The proposed formalization of CDs+OCL could be implemented with a new version of UML2Alloy, to be (hopefully) released soon. The output (in Alloy) of this tool can be changed by the user (e.g., to correct ambiguities) and translated back into UML using the (previously developed) tool OCL2Alloy. This allows a smooth integration of Alloy in software development practices, namely allowing the use of the many available MDA tools on models which are verified and validated with Alloy.

The proposal could be scalable to other domains, such as safety-critical systems. So far, the formalization was only tested with small examples. We intent to validate it with larger case studies. Other ongoing work includes a (small) extension to Alloy to allow the specification of more complex behavioral properties in temporal logic (LTL). This will further simplify the V&V effort required by the user, by allowing him to reuse well-known temporal specification patterns [7]. In the future we also intend to use this formalization to automatically generate UML sequence diagrams, to be used in model based testing.

References

1. Anastasakis, K.: A Model Driven Approach for the Automated Analysis of UML Class Diagrams. Ph.D. thesis, University of Birmingham (2009)
2. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On challenges of model transformation from UML to Alloy. Software and Systems Modeling 9(1), 69–86 (2008)
3. de Andrade, F.R., Faria, J.P., Paiva, A.C.R.: Test generation from bounded algebraic specifications using alloy. In: ICSOFT (2), pp. 192–200 (2011)
4. ATLAS: ATLAS Transformation Language, LINA&INRIA (2009)
5. Bauer, S.S., Hennicker, R.: Views on Behaviour Protocols and Their Semantic Foundation. In: Kurz, A., Lenisa, M., Tarlecki, A. (eds.) CALCO 2009. LNCS, vol. 5728, pp. 367–382. Springer, Heidelberg (2009)
6. Braga, B.F.B., Almeida, J.P.A., Guizzardi, G., Benevides, A.B.: Transforming OntoUML into Alloy: towards conceptual model validation using a lightweight formal method. Innovations in Systems and Software Engineering 6(1-2), 55–63 (2010)
7. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of the 21st International Conference on Software Engineering, ICSE 1999, pp. 411–420. ACM (1999)
8. Garis, A., Cunha, A., Riesco, D.: Translating Alloy Specifications to UML Class Diagrams Annotated with OCL. In: Barthe, G., Pardo, A., Schneider, G. (eds.) SEFM 2011. LNCS, vol. 7041, pp. 221–236. Springer, Heidelberg (2011)
9. Georg, G., Anastasakis, K., Bordbar, B., Houmb, S.H., Toahchoodee, I.R.M.: Verification and trade-off analysis of security properties in UML system models. IEEE Transactions on Software Engineering 36(3), 338–356 (2010)

10. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press (2006)
11. Kosiuczenko, P.: Specification of Invariability in OCL. In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 676–691. Springer, Heidelberg (2006)
12. Lanoix, A., Souquières, J.: Trustworthy Assembly of Components using B Refinement. e-Informatica Software Engineering Journal (ISEJ) 2(1) (2008)
13. Maoz, S., Ringert, J.O., Rumpe, B.: CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 592–607. Springer, Heidelberg (2011)
14. Mostefaoui, F., Vachon, J.: Verification of Aspect-UML models using Alloy. In: Proceedings of the 10th International Workshop on Aspect-Oriented Modeling, pp. 41–48. ACM (2007)
15. Nimiya, A., Yokigawa, T., Miyazaki, H., Amasaki, S., Sato, Y., Hayase, M.: Model checking consistency of UML diagrams using Alloy. World Academy of Science, Engineering and Technology 71(99), 547–550 (2010)
16. OMG: UML Superstructure, Version 2.4.1 (2011)
17. OMG: Object Constraint Language, Version 2.3.1 (2012)
18. Paiva, A.C.R., Faria, J.C.P., Vidal, R.F.A.M.: Towards the integration of visual and formal models for GUI testing. Electronic Notes in Theoretical Computer Science 190(2), 99–111 (2007)
19. Porres, I., Rauf, I.: Generating class contracts from UML protocol statemachines. In: Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation, MoDeVVa 2009. pp. 8:1–8:10. ACM (2009)
20. Rasch, H., Wehrheim, H.: Checking Consistency in UML Diagrams: Classes and State Machines. In: Najm, E., Nestmann, U., Stevens, P. (eds.) FMOODS 2003. LNCS, vol. 2884, pp. 229–243. Springer, Heidelberg (2003)
21. Ries, B.: SESAME: A Model-Driven Process for the Test Selection of Small-Size Safety- Related Embebbed Software. Ph.D. thesis, Université du Luxembourg (2009)
22. Schinz, I., Toben, T., Mrugalla, C., Westphal, B.: The Rhapsody UML verification environment. In: Proceedings of the Software Engineering and Formal Methods, SEFM 2004. pp. 174–183 (2004)
23. Snook, C., Butler, M.: UML-B: Formal modeling and design aided by UML. ACM Trans. Softw. Eng. Methodol. 15, 92–122 (2006)
24. Taghdiri, M., Jackson, D.: A Lightweight Formal Analysis of a Multicast Key Management Scheme. In: König, H., Heiner, M., Wolisz, A. (eds.) FORTE 2003. LNCS, vol. 2767, pp. 240–256. Springer, Heidelberg (2003)