On Integrating Structure and Behavior Modeling with OCL

Lars Hamann, Oliver Hofrichter, and Martin Gogolla

University of Bremen, Computer Science Department
Database Systems Group, D-28334 Bremen, Germany
{lhamann,hofrichter,gogolla}@informatik.uni-bremen.de
http://www.db.informatik.uni-bremen.de

Abstract. Precise modeling with UML and OCL traditionally focuses on structural model features like class invariants. OCL also allows the developer to handle behavioral aspects in form of operation pre- and postconditions. However, behavioral UML models like statecharts have rarely been integrated into UML and OCL modeling tools. This paper discusses an approach that combines precise structure and behavior modeling: Class diagrams together with class invariants restrict the model structure and protocol state machines constrain the model behavior. Protocol state machines can take advantage of OCL in form of OCL state invariants and OCL guards and postconditions for state transitions. Protocol state machines can cover complete object lifecycles in contrast to operation pre- and postconditions which only affect single operation calls. The paper reports on the chosen UML language features and their implementation in a UML and OCL validation and verification tool.

Keywords: Structure modeling, Behavior modeling, UML, OCL, Protocol state machine, State invariant, Guard, Transition postcondition.

1 Introduction

Executable UML [23] is designed to specify a system at a high level of abstraction, independent from specific programming languages and decisions about the implementation. Executable UML follows the ideas of the Shlaer-Mellor methodology, which separated concerns about the structure [34] and the behavior [33] of a system to be developed. It is defined as a profile of the Unified Modeling Language (UML) [26]. Executable UML models are testable, and can be compiled into less abstract programming languages to target a specific implementation. Executable UML supports model-driven development (MDD) through specification of platform-independent models. The approach proposed in this paper follows these ideas.

When using Executable UML, a system is decomposed into multiple modeling sub-languages: A class diagram defines the system structure in terms of the classes and associations; a state machine defines the states, events, and state

R.B. France et al. (Eds.): MODELS 2012, LNCS 7590, pp. 235–251, 2012.

transitions for a class instance; an action language defines the actions or operations that perform processing on model elements; the system behavior is determined by the state machines and the operations realized in the action language.

Our tool USE (UML-based Specification Environment) supports the development of class diagrams by validating OCL class invariants and operation pre- and postconditions [7,8,19]. Recently, the tool was extended with an action language [3] which is based on the Object Constraint Language (OCL) [27,36]. The present contribution explains our support for state machines in order to complete the description of behavior. Within our tool, we integrate class diagram validation with UML protocol machine validation on the basis of OCL state invariants and OCL guards and postconditions for transitions. In contrast to Executable UML, our approach extends OCL in order to express actions and operation implementations, but does not need to define a separate action language.

The need for integrating structure and behavior modeling in the OCL context arose from monitoring running Java applications in terms of UML class diagrams and OCL constraints and our state machine approach. In [12] we describe the monitoring of a non-trivial Java application with constraints. Other applications of our state machine implementation include middle-sized example models.

The rest of this paper is organized as follows. Section 2 introduces with a running example the main state machine features which we employ on the type level (at design time). Section 3 puts the state machine features which we handle in the context of UML and our implementation. In Sect. 4, model validation of state machines in connection with class diagrams is discussed on the instance level (at runtime). Section 5 connects our contribution with related work, before we conclude in Sect. 6.

2 Structure and Behavior at Design Time by Example

Our running example describes a digital support system for a library. The structural system requirements are shown in form of a UML class diagram in the top of Fig. 1. The system supports the administration of users, book copies, and books represented by respective classes and appropriate attributes. Two associations can establish object connections: the association `Borrows` between the classes `User` and `Copy` is meant to express that a `User` object has currently borrowed a `Copy` object, and the association `BelongsTo` between the classes `Copy` and `Book` expresses that a `Copy` object is an exemplar of a particular `Book` object. Further properties are specified by restricting multiplicities, role names (in the example, class names with lower first letter) and invariants (e.g., uniqueness requirements for the attributes `name`, `signature`, and `title`, as well as a range restriction for the attribute `year`). All classes possess operations for initializing objects. The association `Borrows` can be manipulated from both participating classes through the operations `borrow` and `return`. In order to support easy recognition of operation names, the first letter of the respective class has been added to these names (`borrowU`, `returnU`, `borrowC`, `returnC`). The `return` operations also modify the attribute `numReturns`.

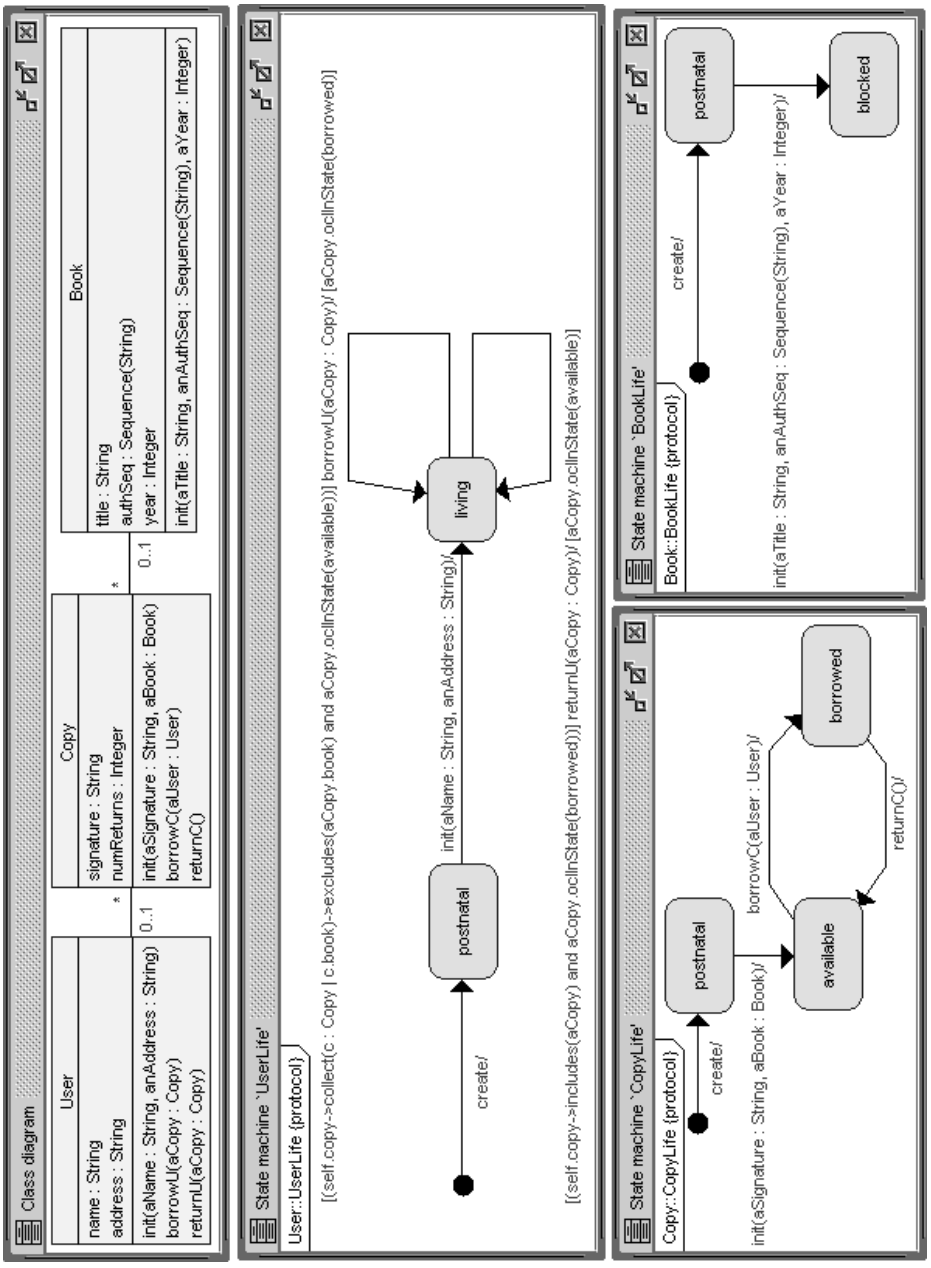

Fig. 1. Example System Requirements for Structure and Behavior (Design Time)

The behavioral system requirements are shown in the bottom of Fig. 1 as UML protocol state machines possessing states and transitions. For every class, the valid object lifecycles are depicted, which restrict the order of creation events and operation calls. As a central means to make the model precise, OCL is used in various places: States are described by state names and state invariants in form of boolean OCL expressions; transitions include (a) the triggering create or call event, (b) a guard in form of a boolean OCL expression asserting that the transition only takes places when the guard holds, and (c) a postcondition in form of a boolean OCL expression asserting that the transition only takes place in the case that after the transition the postcondition holds. Traditionally, the notion guard is used in connection with state machines; however, because of the symmetric behavior of the guard and postcondition, the guard may also be called transition precondition.

The state invariants may optionally be shown in the protocol state machine diagrams, however, we have suppressed them here. For example for the class `Book`, the two proper, non-pseudo states possess the following state invariants.

```
postnatal [title.isUndefined and authSeq->isUndefined and
           year.isUndefined and copy->isEmpty()]
blocked   [title.isDefined and authSeq->isDefined and year.isDefined]
```

In state `postnatal` (after `create`), all attributes must be undefined and the book must not be linked to any copy. In state `blocked` (after a call to the initialization operation `init`), all attributes are defined, but note that no statement about the linked copies is made, because there may or may not be copies for that book in the library (either `copy->notEmpty()` or `copy->isEmpty()` may hold).

The transitions are either labeled with the `create` event which brings the respective object into life or with an event which calls an operation of the object. The protocol state machine for the class `Book` asserts a finite lifecycle demanding that after object creation only the operation `init` may be called once. The state machine for class `Copy` guarantees that after creation and initialization, the `borrowC` and `returnC` operations switch between the states `available` and `borrowed`. The state machine for the class `User` is the only one employing OCL for transition guards and postconditions. But please be aware of the fact that all states are accompanied by OCL state invariants. Both operations, `borrowU` and `returnU` in class `User` are allowed in state `living`, however, OCL restrictions via transition guards and postconditions apply. The guard (precondition) for `borrowU` guarantees that a user cannot borrow two copies of the same book, for fairness reasons. And the guard asserts that only available, not borrowed copies can be handled with the operation `borrowU`. The postcondition of `borrowU` checks that the copy, which was available before the transition took place, is now unavailable. Conversely, the guard for `returnU` asserts that the copy to be returned belongs to the current user and is indeed a copy in state `borrowed`. The postcondition checks that the parameter copy is indeed `available` after the `returnU` call. Note that these simple example restrictions do not guarantee unproblematic behavior in all possible implementations. The state invariants, guards, and postconditions have been chosen for demonstration purposes.

An implementation on the modeling level of the operations can be realized in our language SOIL (Simple OCL-based Imperative Language) [3]. Such an implementation is indispensable for animating and validating the model. SOIL allows the developer to make system state manipulations with attribute assignments, object and link creation and destruction, and control flow using conditionals, loops, and operation calls. As an example, we show implementations for the operations of the classes `User` and `Copy`.

```
class User -- pre- and postconditions not shown
operations
  init(aName:String,anAddress:String)
    begin self.name := aName;
    self.address := anAddress; end

  borrowU(aCopy:Copy)
    begin aCopy.borrowC(self); end

  returnU(aCopy:Copy)
    begin aCopy.returnC(); end
end

class Copy
operations
  init(aSignature:String, aBook:Book)
    begin self.signature := aSignature; self.numReturns := 0;
    insert (self, aBook) into BelongsTo; end

  borrowC(aUser:User)
    begin insert(aUser, self) into Borrows; end

  returnC()
    begin delete(self.user, self) from Borrows;
    self.numReturns := self.numReturns+1; end
end
```

These operation implementations allow the developer to build up simple or complex test states and scenarios with call sequences easily. Consequently, model properties like consistency or the reachability of protocol states can be checked with scenarios constructed with SOIL statements. The SOIL command sequence in the upper right side of the forthcoming Fig. 3 is an example for such a test scenario. The validity of model properties formulated in OCL as class invariants, operation pre- and postconditions, state invariants, and transition pre- and postconditions is checked against these scenarios and by this also against the SOIL implementation given for the operations. When writing down a particular test scenario, the developer will have expectations on particular (class or state) invariants and (operation and transition) pre- and postconditions. These informal expectations are formally checked by the tool USE, and the validation results give detailed feedback to the developer about the possible discrepancy between her expectations and the actual facts: *What you write down doesn't mean exactly*

what you think it means. And when it does, it doesn't have the consequences you expected. [15, p. XIII]

3 Behavior Modeling with Protocol State Machines

3.1 Protocol State Machines in UML

The UML defines two different kinds of state machines: *Behavioral state machines* and *protocol state machines* [26, p. 535]. As the name suggests, the former can model the behavior of a model element by specifying actions which are linked to state transitions, whereas the latter focus on the specification of correct usage protocols, leaving out concrete actions associated with transitions [26, p. 547]. These protocols can be specified for any model element of type *Classifier* [26, p. 544]. The metamodel for state machines provided by the UML allows to model highly structured state machines composed of, for example, composite states, multiple regions and substate machines. At the current stage, our approach supports only a well-defined subset of these features leaving out mainly concepts to structure state machines, but allowing nearly the same expressiveness. Issues arising from the high structuring possibilities can for example be found in [21]. Next we describe the protocol state machine language as implemented in our work. Starting with the syntactical and semantical rules defined in the UML, we continue by showing the current features supported in our approach and how they are interpreted at runtime.

As other languages for (finite) state machines the core part of the state machines defined by the UML are states and transitions. The UML distinguishes between concrete and pseudo-states [26, p. 536, 549, 559]. A state machine instance cannot have a pseudo-state as its current state after a transition has been completed. Pseudo-states are only traversed during the execution of a transition. One example of such pseudo-states are choice points for a transition. Both kinds of states are derived from the metatype *Vertex* for which directed transitions are defined. Behavioral state machines consist of transitions which need a source and target vertex. In addition, transitions can specify a trigger (e.g., a call event), a guard and an effect, i. e., a behavior [26, p. 536].

As we will see, several parts of state machines can be enriched with additional boolean OCL expressions in order to add additional constraints. States can be enriched with a OCL state invariant which characterizes the state in more detail. The state invariant for a given state must be true, if a state machine is in this state. An OCL guard of a transition must be true to be able to execute this transition. For example, this allows to separate two outgoing transitions from one state with the same trigger. In protocol state machines it is also allowed to specify a boolean OCL expression which describes the system state after a protocol transition has been taken. This expression is called a postcondition of the protocol transition.

The initial pseudo-state together with a single outgoing transition marks a concrete state as the default state of the state machine. The transition from the initial state to the default state can only define a behavior and no trigger

or guard [26, p. 550]. Furthermore, the initial state, as all other pseudo-states, cannot specify a state invariant, whereas concrete states can.

Transitions inside a protocol state machine are defined by the metaclass *ProtocolTransition* [26, p. 546]. This class extends the transition class of the behavioral state machine and makes some extensions and restrictions. The main restriction for protocol transitions is that they cannot specify an effect, because they specify the usage of a protocol of a class and not its behavior. An effect of a transition is instead specified in a declarative way by means of a postcondition which cannot be specified for ordinary transitions. The trigger of a protocol transition is usually an operation call, but it can also be an event.

When a protocol state machine defines at least one transition, which refers to an operation, a call to this operation is only valid, if there exists a currently valid transition for this call event. If an operation of the owning class is not referred by a protocol state machine, a call to this operation is valid for any state of the state machine [26, p. 549]. The specification of events other than call events inside a protocol state machine defines requirements for the environment using the owning class, stating that the event can only be sent to an instance of the owning class under the current conditions specified by the protocol state machine [26, p. 549]. An additional constraint specified for a transition is usually called a guard, but for protocol transitions the naming is aligned to the area of operations, calling this constraint a precondition.

3.2 Supported Concepts for Behavior Validation

Our approach supports protocol state machines which allows to specify valid call sequences for lifecycles of an instance. A protocol state machine is defined in the context of a class. The concrete syntax of such definitions is shown below.

```
class A
 attributes
   ...
 operations
   ...
 statemachines
    psm ALife -- psm: Protocol State Machine
     states
      s_i:initial
      s_k [ state_invariant_k ]
      ...
      s_n:final
    transitions
      s_src -> { [ pre_cond ] call_event [ post_cond ] } s_trg
      ...
    end
end
```

First, more than one state machine (in the following we use the term state machine to refer to protocol state machines) can be specified for a class. Beside a name, each state machine defines two sections: `states` and `transitions`. The state section contains the definition of the pseudo- and the concrete states. A state machine must define exactly one pseudo-state of type *initial* acting as the entry point of the state machine. As already mentioned, the initial state cannot define any information except a name for the state. Concrete states are defined by their names and an optional state invariant expressed as a boolean OCL expression in the context of the owning class. State invariants will be discussed in detail during the description of the runtime behavior of state machines. Beside the concrete states and the initial pseudo-state, multiple final states can be defined.

The transition section specifies the structure of valid call sequences to the owning class. The textual syntax is aligned to the graphical representation in the state machine diagrams. For transitions, the source (`s_src`) and target state (`s_trg`) separated by an arrow (`->`) are mandatory. Except for the outgoing transition from the initial state, a `call_event` is also mandatory. These call events refer to an operation of the owning class. The call event for the outgoing transition of the initial state can either be left out or must be named `create` because a newly created object in our approach is immediately initialized with instances of all defined state machines for its class. The `call_event` can be surrounded by a pre- and postcondition given as a boolean OCL expression. Like pre- and postconditions for operations they can access the context object (the instance receiving the call event) and the parameter values of the call event. The postcondition can additionally make use of the OCL `@pre` keyword to access the values which were valid when the call event was triggered.

When a USE model containing state machines is loaded, static checks are made. These include checking the uniqueness of state names inside a single state machine and the well-formedness of transitions, i.e., checking that state names and transitions do refer to existing states and operations.

3.3 Protocol State Machines at Runtime

To validate a specified model, our approach allows the developer to instantiate it and observe its behavior. The instantiation can be done in several ways, e.g., by manually manipulating the system state using the graphical user interface or shell commands or by specifying statements in SOIL [3]. If an object of a class is created, which contains state machines[1], it is linked to the corresponding state machine instances. These state machine instances are initialized with the default state, i.e., the state reached by the outgoing transition of the initial state, as their current state.

If an operation is called on an object, all state machines, which specify a transition referring to the operation call, are checked for enabled transitions. A

[1] In the following we refer to objects of classes with defined state machines when using the word object.

transition is called *enabled*, if it is an outgoing transition leaving the current state of a considered state machine instance, if it refers to the called operation and if it has a currently valid precondition [26, p. 584]. If at least one enabled transition for each state machine under consideration exists, the operation call is valid. The transition to take is determined after the operation has been executed. This is done by evaluating for each previously enabled transition the postcondition and the state invariant of the target state. For each considered state machine instance there must be exactly one transition fulfilling both conditions. By using this mechanism, we (currently) disregard non-deterministic state machines and executions which are however generally allowed in UML. Otherwise, the operation execution is invalid. The concrete error situation is reported to the user stating that either there exists no valid transition or multiple transitions are currently valid. When a state machine instance is currently in an unstable state, i.e., it is executing a transition, all nested operation call events need to be ignored. Otherwise, a call to another operation on the same object by a called operation could for example change the current state making the previously enabled transition invalid. The modeler can turn on a notification mechanism for such situations.

The explained runtime behavior of state machines lead to valid call sequences respecting state invariants, transition pre- and postconditions, if the state of an object is only modified by operations specified by protocol state machines. However, as we described earlier, a protocol state machine can leave out operations, making them callable at any time. Because these unconsidered operations could also modify the state of an object, it is not guaranteed that a state invariant stays valid while a state machine instance remains in a certain state. Therefore, our approach is able to validate state invariants after any change to the system state, e.g., attribute assignments or link creations. A violation of state invariants is immediately reported to the user, who can then react to the error.

Another unique feature of our approach is the possibility to determine the current state of the state machines by the specified state invariants [11]. For this, the validation of transitions and state invariants can be suppressed. After a system state is constructed without the validation of state machines, the user can invoke the state determination command. The command tries to determine the current state for each state machine instance by evaluating its state invariants. If exactly one state invariant of a state machine instance evaluates to true, the state of this instance is modified. This can, for example, be used, if a given system state needs to be constructed without the execution of operations and afterwards an operation call sequence has to be validated. An application of this mechanism is the USE monitor [10,12] which allows to connect to a running Java application and to retrieve a snapshot of the current application state. When connecting to the application, not all information about previously called operations is available, and therefore the current states must be calculated to obtain the valid state machine configuration.

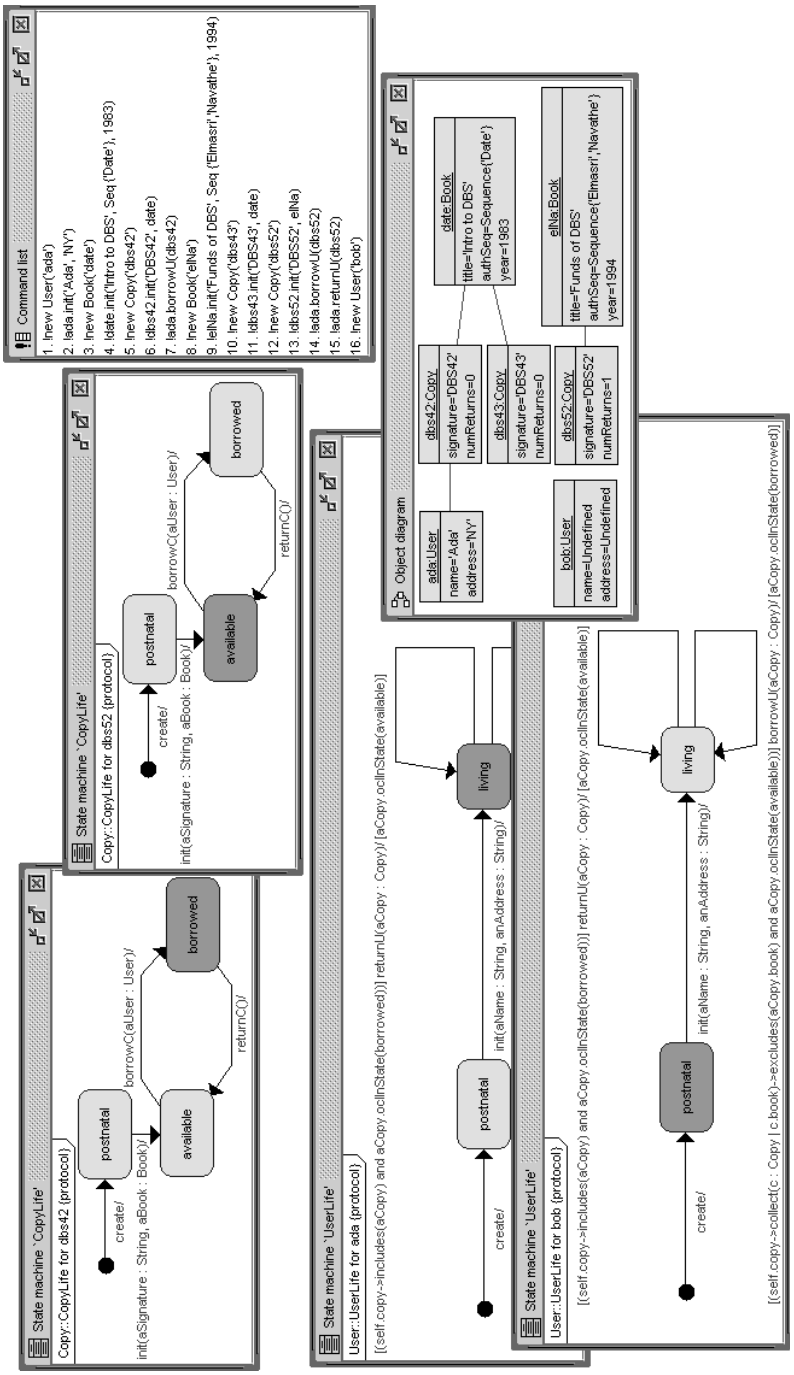

Fig. 2. Example Scenario for Structure and Behavior (Runtime)

4 Structure and Behavior at Runtime by Example

This section will explain how to apply the proposed concepts for the example. Whereas Fig. 1 pictures structure and behavior of the library system on a type level (design time), Fig. 2 displays structure and behavior of one system test scenario on the instance level (runtime). The object diagram in the lower right represents the objects, their attribute values and links after the SOIL command sequence in the upper right part of Fig. 3 has been executed. In the left of Fig. 2, the upper two state machine instances show the current protocol state for the `Copy` objects `dbs42` and `dbs52`, respectively. Also in the left, the lower two state machine instances display the current protocol state for the `User` objects `ada` and `bob` in dark grey. Please note, that the state of both `Copy` objects and the state of both `User` objects are different. The state sequence which the `Copy` object `dbs52`

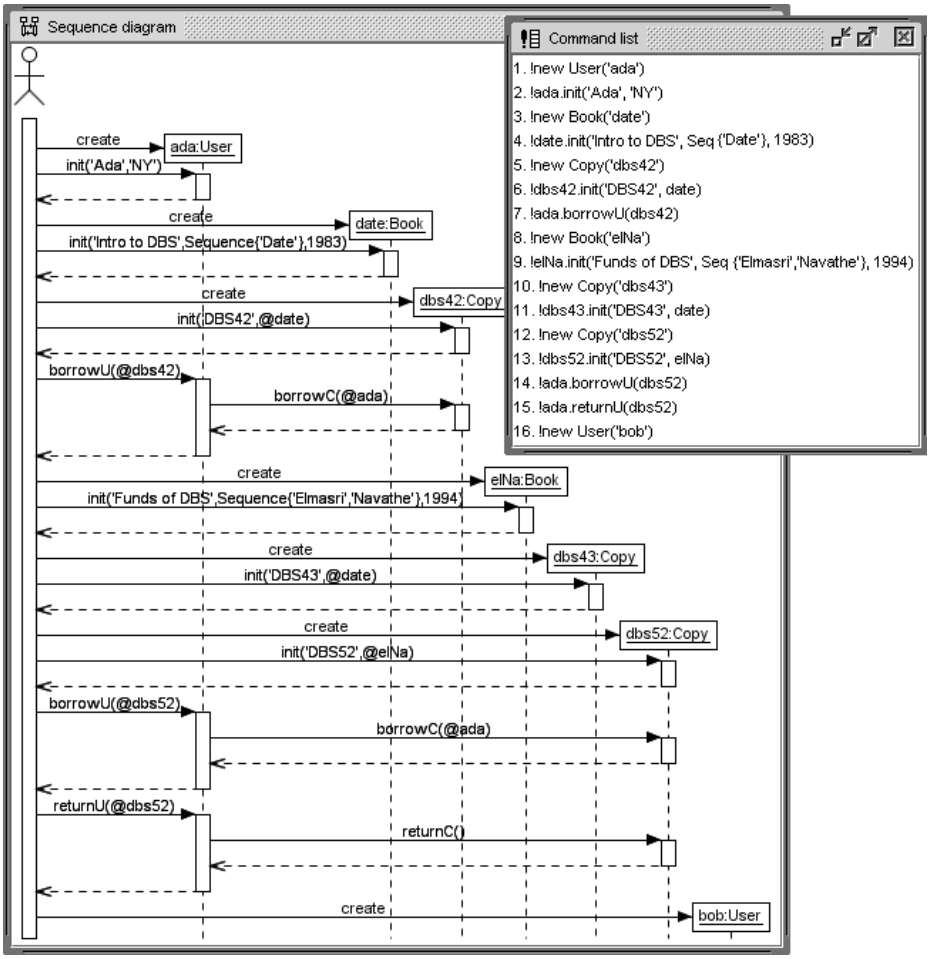

Fig. 3. Sequence Diagram and SOIL Commands for Example Scenario

went through was `postnatal`, `available`, `borrowed` and again `available`. We can conclude this from the executed operation sequence and from the attribute value `1` for attribute `numReturns`. In the shown operation sequence, all OCL restrictions have been checked and no violation occurs: all class invariants, state invariants and transition pre- and postconditions have been evaluated to `true`. Please note, that full OCL support in our approach means that we can relate OCL queries concerning structure with behavioral descriptions, for example, the OCL query in Fig. 2 checks relevant `Copy` properties and these can be compared with the current protocol state and the value of the state invariants.

This scenario can be extended by further operation calls. For example, the `User` object `ada` could try to borrow the `Copy` object `dbs43`. In this situation, the guard for the `borrowU` call on the transition from `living` to `living` would prevent the transition to take place: User `ada` has already borrowed another copy of the `Book` object `date`. On the USE shell, a message will inform about the violation and the fact that the transition should not and will not occur. The following message will be shown.

```
!ada.borrowU(dbs43)
 >> Error: No valid transition available in protocol state machine
 >> 'User::UserLife [current state: living]' for operation call
 >> User::ada.borrowU(dbs43) due to failing transition guard.
```

Analogous error messages would be displayed on the shell, if the transition postcondition or the state invariant of the next state would be violated. Summarizing we can say that taking a transition may be aborted due to four possible reasons:

- a failing transition guard (precondition),
- a failing transition postcondition,
- a failing state invariant in the resulting state, and
- non-deterministic transitions, e.g., multiple transitions for the same trigger.

In Fig. 4, another example explains the usage of state invariants and the state determination option. For a `TrafficLight` class with three boolean attributes representing the red, yellow, and green bulbs, a protocol state machine allows the traffic light to step through four phases, where each phase is represented by a single state and a state invariant in form of an OCL expression characterizing the signal in terms of the bulbs.[2] The object diagram shows four test traffic lights equipped with randomly determined attribute values for the bulbs, not all representing valid signal configurations. The attribute values have been modified not by operations, but with direct attribute assignments.

In the log window at the bottom, the result of executing the state determination command is given. This command aims to bring the state machine instances into the state corresponding to their state invariants, if possible. The command

[2] The phases are the phases used in Germany, whereas in other countries, e.g., in Italy, the phases are different.

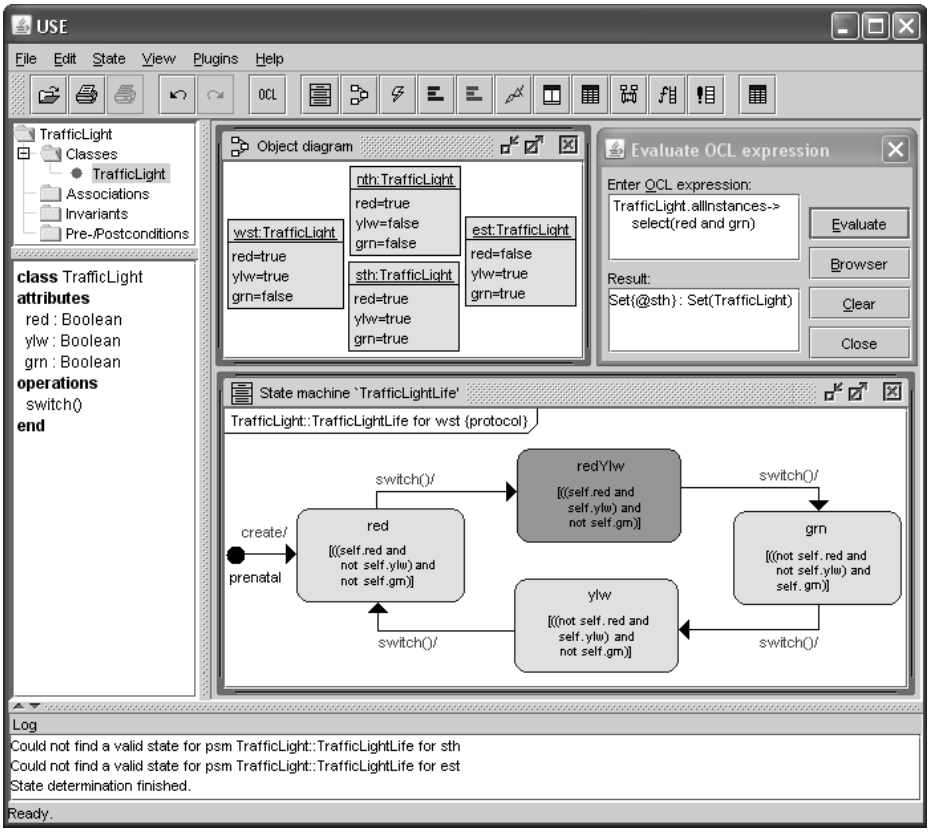

Fig. 4. Example for Usage of State Invariants and State Determination Option

can be issued through an entry in the 'State' menu. For two traffic lights (`sth` and `est`), a valid state fitting one of the four state invariants could not be found; the other state machine instances are moved into a state determined by a state invariant. The displayed state machine instance in the middle belongs to the `TrafficLight` object `wst` and shows that the attribute values (`wst.red=true` and `wst.ylw=true` and `wst.grn=false`) fit to the OCL state invariant expression (`self.red and self.ylw and not(self.grn)`) belonging to the current state `redYlw` shown in dark grey. As our approach supports OCL during all development phases, the complete system state can be inspected with OCL expressions at any point in time. The OCL query expression in the upper right retrieves all present traffic light objects which currently show both `red` and `grn`. The state determination together with OCL querying allows to check positive and negative test cases with respect to structure (objects and attributes) and behavior (operations and state machines).

5 Related Work

Specifying behavior in OCL. OCL not only allows for specifying structural model features but also constraints on the behavior of objects by means of pre- and postconditions. In order that pre- and postconditions can be interpreted unambiguously, a detailed semantics of operation specifications is needed. The approach in [14] addresses this. However, according to [16], pre- and postconditions describe static aspects of the system, as they compare states of a system, which are static entities. Therefore in [16,17] the so-called action clause is introduced to the Object Constraint Language and is provided with a semantics.

Semantics of State Machines. In our approach we use UML protocol state machines to constrain the model behavior. The structure and the semantics of protocol state machines are discussed in [28]. The authors present an approach which applies protocol state machines to produce class contracts. The semantics of behavioral state machines is discussed in [20]. The authors apply the semantics for validity proofs of refinement transformations on behavior state machines. A formal semantics for the integration of UML statecharts into OCL, which makes it possible to formulate expressions over states in UML statecharts is presented in [5]. However the authors refer to an older UML version, whereby postconditions of protocol state machine transitions are not handled. The dynamic semantics of state machines is discussed in [2].

Usage of State Machines. Different approaches for the usage of state machines in the software testing context exist. Model-based testing (MBT) tools often use UML state machines as a basis for automatic test case generation. The approach in [38] makes it possible to automatically generate state machine diagrams from use cases. This approach is also implemented in a tool and evaluated in different case studies. The approach in [31] applies behavioral state machines for modeling reactive systems and automatic generation of test cases. Based on this, the input-output conformance of the systems is tested. The presented test approach is implemented by the so-called TEAGER tool suite. In [37], the authors report on an industrial cooperation for model-based testing applying UML state machines with a German rail engineering company. Based on a given UML state machine this approach makes it possible to automatically generate unit tests. The use of UML state machines for requirements validation is described in [25]. The authors apply Formal Concept Analysis (FCA) in analyzing the association between a set of test scenarios with a set of transitions specified in a UML state machine model. The authors of [35] use protocol state machines in the field of network security. They introduce Veritas, a tool which uses applications network traces to automatically generate protocol state machines. The generated state machines are able to represent incomplete knowledge about a protocol and are labeled as probabilistic protocol state machines (P-PSM). K-statecharts are an extension of UML statecharts which allow the use of knowledge-logic formulae in the statechart transition guard and are used for runtime verification of system behavior [4].

Tools. In [32] a tool set which supports static and dynamic validation of UML models is presented. The tool mOdCL is based on Maude, an executable formal specification language and is able to validate invariants and pre- and postconditions during the execution of a system [29]. In contrast to our approach and like the tool set presented in [32], mOdCL leaves out handling and runtime validation of protocol state machines. In [29], the authors report on the experiences with the development of a tool for dynamic enforcement of OCL constraints. Applying aspect-oriented programming (AOP), ocl2j automatically instruments OCL constraints in Java programs. In [24] a prototype of a tool being able to check the conformance of components within the UML extension for real-time (UML-RT) to the respective protocol state machines, which specify the legal communication between components, is described. Rhapsody is a verification environment for UML models. The tool implements an own semantics of statecharts, as discussed in [13]. The tool TABU allows for verification of reactive systems behavior [9]. For this purpose the behavior is modeled by state machines and automatically transformed into the used formal specification SMV (Symbolic Model Verifier). Additionally a number of CASE tools suchs as [6] allow for modeling statecharts, but are not able to validate state machines at runtime. In contrast to our approach, [1] and [22] don't provide full OCL support. Epsilon [18] is a platform which allows for model validation. However handling for state machines is not integrated.

Our contribution profits from these related works. It is however the only one which combines state machine validation with full OCL support for structural modeling and validation.

6 Conclusion

We have made a proposal for integrated structure and behavior modeling and validation. Full OCL support for (class and state) invariants and (operation and transition) pre- and postconditions guarantees that the underlying graphical models become precise. We combine descriptive requirements with an OCL-like imperative language. The models are validated and verified by test scenarios.

We plan to extend the supported UML state machine features, in particular, we will care for structuring mechanism like nested states. A number of improvements on the user interface can be realized, for example, an optional indication of protocol state machine states on object lifelines in sequence diagrams. Features of the behavior models like state reachability and other dynamic properties like liveness could be supported in a (semi-)automatic way. Consistency, redundancy and other relationships between the structural and behavioral model features should be investigated. Methodological questions about the usage of (class and state) invariants, and (operation and transition) pre- and postconditions must be discussed. Last but not least, larger case studies must give further feedback about the applicability and efficiency of the approach.

References

1. Abstract Solutions Ltd: Executable UML (xUML). Internet (2012), http://www.kc.com/XUML/
2. Börger, E., Cavarra, A., Riccobene, E.: Modeling the Dynamics of UML State Machines. In: Gurevich, Y., Kutter, P.W., Odersky, M., Thiele, L. (eds.) ASM 2000. LNCS, vol. 1912, pp. 223–241. Springer, Heidelberg (2000)
3. Büttner, F., Gogolla, M.: Modular Embedding of the Object Constraint Language into a Programming Language. In: Simao, A., Morgan, C. (eds.) SBMF 2011. LNCS, vol. 7021, pp. 124–139. Springer, Heidelberg (2011)
4. Drusinsky, D.: tak Shing, M.: Using UML Statecharts with Knowledge Logic Guards. In: Schürr and Selic [30], pp. 586–590 (2009)
5. Flake, S., Müller, W.: Formal semantics of static and temporal state-oriented OCL constraints. Software and System Modeling 2(3), 164–186 (2003)
6. Geiger, L., Zündorf, A.: Statechart Modeling with Fujaba. Electr. Notes Theor. Comput. Sci. 127(1), 37–49 (2005)
7. Gogolla, M., Bohling, J., Richters, M.: Validating UML and OCL Models in USE by Automatic Snapshot Generation. Journal on Software and System Modeling 4(4), 386–398 (2005)
8. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-Based Specification Environment for Validating UML and OCL. Science of Computer Programming 69, 27–34 (2007)
9. Gutiérrez, M.E.B., Barrio-Solórzano, M., Quintero, C.E.C., de la Fuente, P.: UML Automatic Verification Tool with Formal Methods. Electr. Notes Theor. Comput. Sci. 127(4), 3–16 (2005)
10. Hamann, L., Gogolla, M., Kuhlmann, M.: OCL-Based Runtime Monitoring of JVM Hosted Applications. In: Cabot, J., Clariso, R., Gogolla, M., Wolff, B. (eds.) Proc. Workshop OCL and Textual Modelling (OCL 2011). ECEASST, Electronic Communications (2011), journal.ub.tu-berlin.de/eceasst/issue/view/56
11. Hamann, L., Hofrichter, O., Gogolla, M.: OCL-Based Runtime Monitoring of Applications with Protocol State Machines. In: Vallecillo, A., Tolvanen, J.-P., Kindler, E., Störrle, H., Kolovos, D. (eds.) ECMFA 2012. LNCS, vol. 7349, pp. 384–399. Springer, Heidelberg (2012)
12. Hamann, L., Vidács, L., Gogolla, M., Kuhlmann, M.: Abstract Runtime Monitoring with USE. In: Ferenc, R., Mens, T., Cleve, A. (eds.) Proc. CSMR 2012 (2012)
13. Harel, D., Kugler, H.: The Rhapsody Semantics of Statecharts (or, On the Executable Core of the UML) - Preliminary Version. In: Ehrig, H., Damm, W., Desel, J., Große-Rhode, M., Reif, W., Schnieder, E., Westkämper, E. (eds.) INT 2004. LNCS, vol. 3147, pp. 325–354. Springer, Heidelberg (2004)
14. Hennicker, R., Knapp, A., Baumeister, H.: Semantics of OCL Operation Specifications. Electr. Notes Theor. Comput. Sci. 102, 111–132 (2004)
15. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press (2006)
16. Kleppe, A., Warmer, J.: Extending OCL to Include Actions. In: Evans, A., Kent, S., Selic, B. (eds.) UML 2000. LNCS, vol. 1939, pp. 440–450. Springer, Heidelberg (2000)
17. Kleppe, A., Warmer, J.: The Semantics of the OCL Action Clause. In: Clark, T., Warmer, J. (eds.) Object Modeling with the OCL. LNCS, vol. 2263, pp. 213–227. Springer, Heidelberg (2002)

18. Kolovos, D., Rose, L., Paige, R.: The Epsilon Book. Internet (2012), http://www.eclipse.org/epsilon/doc/book
19. Kuhlmann, M., Hamann, L., Gogolla, M.: Extensive Validation of OCL Models by Integrating SAT Solving into USE. In: Bishop, J., Vallecillo, A. (eds.) TOOLS 2011. LNCS, vol. 6705, pp. 290–306. Springer, Heidelberg (2011)
20. Lano, K., Clark, D.: Semantics and Refinement of Behavior State Machines. In: Cordeiro, J., Filipe, J. (eds.) ICEIS, vol. (3-1), pp. 42–49 (2008)
21. Lano, K., Clark, D.: Axiomatic Semantics of State Machines, pp. 179–203. John Wiley & Sons, Inc. (2009)
22. Lano, K., Kolahdouz-Rahimi, S.: UML RSDS Model Transformation and Model-Driven Development Tools. Internet (2012), http://www.dcs.kcl.ac.uk/staff/kcl/uml2web
23. Mellor, S.J., Balcer, M.: Executable UML: A Foundation for Model-Driven Architectures. Addison-Wesley (2002)
24. Moffett, Y., Beaulieu, A., Dingel, J.: Verifying UML-RT Protocol Conformance Using Model Checking. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 410–424. Springer, Heidelberg (2011)
25. Ng, P.: A Concept Lattice Approach for Requirements Validation with UML State Machine Model. In: SERA, pp. 393–400. IEEE Computer Society (2007)
26. OMG (ed.): UML Superstructure 2.4.1. Object Management Group (OMG) (August 2011), http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF
27. OMG (ed.): Object Constraint Language 2.3.1. Object Management Group (OMG) (January 2012), http://www.omg.org/spec/OCL/2.3.1/
28. Porres, I., Rauf, I.: From Nondeterministic UML Protocol Statemachines to Class Contracts. In: ICST, pp. 107–116. IEEE Computer Society (2010)
29. Roldán, M., Durán, F.: Dynamic Validation of OCL Constraints with mOdCL. ECEASST 44 (2011)
30. Schürr, A., Selic, B. (eds.): MODELS 2009. LNCS, vol. 5795. Springer, Heidelberg (2009)
31. Seifert, D.: Conformance Testing Based on UML State Machines. In: Liu, S., Maibaum, T.S.E., Araki, K. (eds.) ICFEM 2008. LNCS, vol. 5256, pp. 45–65. Springer, Heidelberg (2008)
32. Shen, W., Compton, K.J., Huggins, J.: A UML Validation Toolset Based on Abstract State Machines. In: ASE, pp. 315–318. IEEE Computer Society (2001)
33. Shlaer, S., Mellor, S.J.: Object Lifecycles: Modeling the World in States. Yourdon Press, EngleWood Cliffs (1992)
34. Shlaer, S., Mellor, S.J.: Object-Oriented Systems Analysis: Modelling the World in Data. Yourdon Press, EngleWood Cliffs (1992)
35. Wang, Y., Zhang, Z., Yao, D.D., Qu, B., Guo, L.: Inferring Protocol State Machine from Network Traces: A Probabilistic Approach. In: Lopez, J., Tsudik, G. (eds.) ACNS 2011. LNCS, vol. 6715, pp. 1–18. Springer, Heidelberg (2011)
36. Warmer, J., Kleppe, A.: The Object Constraint Language: Getting Your Models Ready for MDA. Object Technology Series. Addison-Wesley, Reading (2003)
37. Weißleder, S.: Influencing Factors in Model-Based Testing with UML State Machines: Report on an Industrial Cooperation. In: Schürr and Selic [30], pp. 211–225 (2009)
38. Yue, T., Ali, S., Briand, L.C.: Automated Transition from Use Cases to UML State Machines to Support State-Based Testing. In: France, R.B., Küster, J.M., Bordbar, B., Paige, R.F. (eds.) ECMFA 2011. LNCS, vol. 6698, pp. 115–131. Springer, Heidelberg (2011)