Software & System Modeling (2005) 4(4): 399–423
DOI 10.1007/s10270-005-0090-5

SPECIAL ISSUE PAPER

L. C. Briand · Y. Labiche · J. Cui

Automated support for deriving test requirements from UML statecharts

Received: 31 May 2004 / Revised version: 17 January 2005 / Published online: 16 September 2005

Abstract Many statechart-based testing strategies result in specifying a set of paths to be executed through a (flattened) statechart. These techniques can usually be easily automated so that the tester does not have to go through the tedious procedure of deriving paths manually to comply with a coverage criterion. The next step is then to take each test path individually and derive test requirements leading to fully specified test cases. This requires that we determine the system state required for each event/transition that is part of the path to be tested and the input parameter values for all events and actions associated with the transitions. We propose here a methodology towards the automation of this procedure, which is based on a careful normalization and analysis of operation contracts and transition guards written with the Object Constraint Language (OCL). It is illustrated by one case study that exemplifies the steps of our methodology and provides a first evaluation of its applicability.

1 Introduction

In modeling Object-Oriented software systems, the state-dependent behavior of objects, object clusters or subsystems is modeled by a state machine. In the context of the Unified Modeling Language (UML) [1], the model to be used to specify such state-dependent behavior is a UML statechart (simply called statechart in the rest of the document). Such a model specifies state changes under the form of transitions, i.e., an event enabling the transition, a guard condition specifying under which condition the transition fires, and actions performed as a result of the firing of the transition. Both events and actions can be further specified by parameters. Additionally, other behavior can be specified by statecharts, such as concurrent behavior.

Statecharts are used during the Analysis to better specify object behavior (most methodologies promote its use at that stage, e.g. [2]), as well as during testing,[1] where the implementation is tested (verified) against its specification (i.e., its statechart) [3]. Because statecharts are—according to most analysis and design methodologies—used to model classes or small clusters of classes, statechart-based testing is often used for class or component testing.

When using a statechart for testing, two main problems have to be solved. First a decision has to be made as to what transitions or sequences of transitions should be tested since exhaustive testing (i.e. testing the entire behavior specified by the statechart) is usually impossible. A number of authors [3, 4] have proposed test strategies for UML statechart, based on several coverage criteria (e.g., all transition pairs). These criteria all assume a test case to be in the form of a feasible sequence of transitions. The generation of a set of transition sequences for a given coverage criterion can usually be automated, under specific assumptions, so that the tester does not have to go through the tedious procedure of deriving paths manually to comply with the coverage criterion (e.g., [4]). The second problem to be solved when deriving test cases from a statechart is to determine test data that allow the execution of those transition sequences. Test data include the identification of the state in which the object (object cluster) under test should be before the execution of a transition sequence, as well as possible arguments (i.e., parameter values) for events and actions in transitions. Deriving test data is by no means a trivial task. This problem is similar to the path sensitization problem, where one tries to find inputs that will drive a routine along a specific control flow path [5]. Solving the problem amounts to finding a solution to a set of Boolean predicates extracted from the control flow path. In our context, statechart-based

L. C. Briand (✉) · Y. Labiche · J. Cui
Software Quality Engineering Laboratory Carleton University–Department SCE, 1125 Colonel By Drive Ottawa, ON K1S5B6, Canada
E-mail: {briand, labiche, jfcui}@sce.carleton.ca

[1] The scope of the testing activity depends on what is modeled by the statechart. If the statechart models the behavior of a single class, then it can be used to support unit testing. If the behavior of a class-cluster, a subsystem or a component is modeled, then we are concerned with integration testing. If the whole system is modeled, then the focus of statechart-based testing is system testing.

testing, those predicates are guard conditions, preconditions, postconditions, and class invariants. This problem is further complicated because of the possibly very complex relations between objects, resulting in object behavior affecting and being affected by other objects' state.

The problem of deriving test data for transition sequences can be further decomposed into two subproblems: extracting constraints on test data (e.g., the value of an event's parameter should be smaller than an attribute's value), and solving these constraints to obtain actual test data values. We refer hereafter to those constraints as *test constraints*. This article focuses on the former subproblem and provides guidance on possible solutions to the latter. The derivation of constraints is the most important and difficult of the two problems as numerous techniques exist for constraint solving. Though they will need to be adapted for the Object Constraint Language (OCL) [6, 7], we do not foresee any major obstacle.

As further discussed in Sect. 2, existing testing techniques based on UML statecharts only support a limited subset of the UML notation, and thus only address the derivation of test data in a restrictive context, thus limiting their application.

Our objective in this article can then be defined as follows. Given (1) a class[2] to be tested, (2) its associated classes, (3) its statechart, (4) a set of interacting statecharts belonging to some of the associated classes, and (5) a specific sequence of transitions to be tested, we want to determine test constraints (i) on the state of the system at different points in the transition test sequence and (ii) on specific arguments for events and actions, so that when the sequence of events for this transition sequence is received by a specific instance of the class under test, the transition sequence can be properly executed. This entails, in particular, that guard conditions in the statechart(s) be specified with the Object Constraint Language [6, 7], and that actions (in transitions and states) and call events be specified using Contracts [8, 9] (i.e., with pre and post-conditions). Based on such information, it is then expected that a test tool can derive test requirements and then fully specified test cases automatically. Our main motivation in this article is to explore automated support for test engineers to extract useful information from statecharts and contracts to derive the specification of a test plan under the form of test constraints.

Section 2 provides some information on related works. Section 3 discusses the testability of UML statecharts. Section 4 describes the methodology we propose to derive test constraints, and illustrate it on a case study. Section 5 then concludes by summarizing the results and outlining future work.

[2] It is of course conceivable to have a statechart modeling the behaviour of a cluster of classes (e.g., a composite class and its component classes). But that does not affect what is presented in this article and we will assume in the reminder of the text that a class is being tested.

2 Related work

A number of authors [3, 4] have proposed test strategies for UML statechart, based on several coverage criteria such as the all transitions, all transition pairs, full predicate (specifically targeting guarded transitions) and all round-trip paths (i.e., all transition sequences that begin and end with the same state). These criteria all assume a test case to be in the form of a feasible sequence of transitions. (Note however that the problem of checking whether a transition sequence is feasible is not addressed by these authors.) Techniques and algorithms to automatically generate test cases according to these criteria have also been proposed (e.g., [4, 10]). However, they only support a limited part of the UML statechart notation [1]. For instance, they only account for change events (an event that occurs when a Boolean expression becomes true) or signal events (an event that occurs when a specific signal is received), guard conditions are expressed through Boolean or primitive type class attributes, and states do not hold actions. However, the UML notation also allows the specification of call events (the event corresponds to the invocation of an operation), guard conditions can be much more complicated when expressed with the Object Constraint Language (OCL), and states can have entry and exit actions.

Other strategies consist in adapting Chow's method [11, 12] in a UML context [13, 14], accounting for the hierarchical nature of UML statecharts for instance. However, again, a subset of the UML notation is supported: only signal events, and no guard condition (except in [13] where they only involve time, i.e., this corresponds to UML time and change events).

Last, some authors suggest transforming the statechart into a flow graph and applying conventional control and data flow testing techniques [15, 16]. But their techniques also have similar limitations in terms of guard conditions and the kinds of events supported.

None of these works adopt Design by Contract [8] in defining operations, and modifications to the object's state due to events and actions are simply modeled as assignments to attributes [15]. Furthermore these techniques consider statecharts in isolation without accounting for interactions among object statecharts. However, even if our focus is on testing classes or class clusters, we need to account for such interactions with associated classes which are not under test as associated objects may affect the state or behaviour of the objects under test at run-time.

3 Testability

In this section, after an initial set of definitions, we discuss a number of issues related to the testability of statecharts and contracts. In other words, we clearly specify our assumptions and the trade-offs that are involved in applying our approach.

3.1 Definitions

We refer to the *owning class* as the class that owns the statechart based on which the test case (transition test sequence) is specified. An *owning object* is an instance of the owning class. The *associated classes* are classes to which the owning class navigates through associations (e.g., as specified in operation contracts or guard conditions in OCL), and the *class cluster* consists of the owning class and all its associated classes. The *object cluster* is a set of instances for the class cluster.

We can view an object state according to two levels of abstractions. The *concrete state* is defined as the combination of the object's attribute values plus its links to other objects. We refer to the *collective state* as a set of the concrete states of all the objects in the object cluster. The concrete view of a state, which is sometimes referred to as the *primitive state* [3], is often too granular for statechart modeling (i.e., it would lead to a large, possibly infinite number of states). At a more abstract level, concrete states with some common properties can be grouped together into so called *abstract states* [3]. Typically, state invariants associated with states in a statechart describe abstract states, whereas guard conditions and event/action postconditions use and model changes to the collective state.

We identify two types of state-dependent objects under test as they require different processing (and thus different algorithms) with respect to test constraints derivation (Sect. 4.5). An *associated state-dependent object* is an object whose behavior depends on the state of other objects, whereas an *orphan state-dependent object* is an object whose behavior does not depend on the state of other objects. In the former case for instance, an object `a`'s statechart (i.e., the statechart for `a`'s class) has a guard condition involving object `b`'s state.

According to [1], an object property is one of the following: An attribute (whose type can be any OCL type); An association-end (whose type is either a class or a collection type); A query operation, that is an operation with no side effect (it returns a value but does not modify the system state); A non-query operation. A postcondition expression can refer to the value of a property at the start (resp. upon completion) of the operation, defined as the *pre-value* (resp. *after-value*) using the `@pre` prefix. We then define the *pre-state* (resp. *after-state*) of an operation to be the collective state of an object cluster before (resp. after) executing the operation.

Each OCL constraint is written in the context of a specific class and the reserved word `self` is used to refer to the contextual instance, referred to as the *context object* [7]. When used in a precondition, postcondition, or guard, a property has a *navigation path* as its prefix (if not, the property has an implicit navigation path, which is "`self.`"). The navigation path starts from a context object, possibly followed by a sequence of navigations [7] until the class that owns the required property is reached. Note that in transitions, call actions can also have navigation paths, starting from a context object, possibly followed by a sequence of navigations until the class that owns the operation is reached.

3.2 UML statecharts

This section discusses the specifics of the UML statechart notation[3] [1] with respect to testability. Statechart diagrams can be used to describe the behavior of individual entities (e.g., a class) as well as a collection of entities (e.g., class cluster, subsystem, system etc.). For the sake of simplicity, we assume in the rest of this article that a class is being tested, though testing several classes in a cluster modeled by a statechart does not affect what is being presented.

3.2.1 Statechart flattening

The use of composite and concurrent substates in statecharts helps to cope with complexity but makes it hard to generate test cases. In order to apply the state-based criteria mentioned in the introduction, it is necessary to remove all hierarchy and concurrency in the statecharts under study. The results are *flat statecharts*, in which every distinct state is represented by a node and all possible transitions are shown explicitly [3]. Though flattened statecharts might be highly complex, the process can be automated and such statecharts are not meant to be visualized by software engineers as they are only used as internal representations for our algorithms. It should be noted that flattening a statechart may impact OCL constraints that refer to composite states since these states are removed in the resulting statechart. In this case, the OCL constraints may need to be transformed so that the composite states are replaced by their substates. Existing flattening algorithms (e.g., [3, 15]) do not address this issue and would thus have to be adapted, though this does not present any serious obstacle.[4] Such transformations are not addressed by this article and we assume statecharts are already flattened.

3.2.2 Other assumptions about UML statecharts

The semantics of UML statecharts allow for the possibility of non-determinism in state transitions: Multiple transitions, triggered by the same event, may be enabled for firing from the same source state at the same time. This research does not handle such cases as we assume statecharts to be deterministic. In the same vein, non-determinism may also result from concurrent state machines interacting with one another (e.g., race conditions where computation results and state changes depend on the unknown and unpredictable execution order of concurrent actions [19]). Identifying such problematic behaviour is however not the purpose of state-based testing, or any other form of functional testing, and should anyway be carefully avoided through design reviews and synchronization mechanisms [20] or detected

[3] In this work our tools and algorithms assume UML 1.4 [6]. However, they can easily be adapted to UML 2.0 [17, 18] since the feasibility of our approach does not depend on constructs which definition and semantics have changed from UML 1.4 to UML 2.0.

[4] A composite state invariant involved in a constraint can simply be replaced by the disjunction of all its substates' invariants.

at runtime [21]. We therefore assume that, if there is concurrency (modeled as asynchronous signals in statecharts), it does not entail indeterminism in terms of state changes and computational results.

In a UML statechart, a transition has five parts [1]: a source and a target state, an event trigger (the event triggering the transition), a guard condition (Boolean expression possibly written in OCL), and an action (or list of actions). According to the UML specification, there are five kinds of events and eight kinds of actions. One kind of event, namely `call` event, is of particular interest in this research, as this is the only event associated with an operation. Its effect on the collective state can thus be specified through a postcondition (possibly written in OCL). Among the eight kinds of actions, this research only considers `call`, `send`, `assignment`, `create` and `destroy`, since the other actions do not affect the collective state. Although `assignment` (an attribute is given a value), `create` and `destroy` actions do not have an explicit postcondition, their effect on the collective state can be considered as their implicit "postcondition", e.g., creating an object adds one element to the collection of all the instances of the corresponding class.

According to [22, 23], events and actions are *atomic* so their effect is deterministic. To clarify, what is meant is that their execution cannot be interrupted by external events. Because we assume there is no race condition, then the result is deterministic. In contrast, an activity can be interrupted by an event. As a result, the effect of an activity on the collective state might, in principle, be non-deterministic if it changes the system state. In practice, however, this is not an appropriate modeling strategy as one cannot guarantee the system's state integrity. We, therefore, assume that either an activity does not change the system state or, if it does, we assume it completes without interruption and triggers a completion transition.

An action in a transition indicates that an atomic computation (as defined above) is performed when the transition fires. Actions can be attached to states as entry/exit actions or associated with transitions. Note that entry and exit actions can be substituted with transition actions, without changing the semantics of a statechart [22]. During the firing of a transition, the execution of events and actions are in the following order: event, exit action of the source state, action sequence attached to the transition, entry action of the target state [23]. Actions in the transition action sequence are usually assumed to be independent of one another, and as a result their order of execution does not matter. In cases when the order is relevant, it is assumed that modellers specify transition actions in the right sequence. We conform to this assumption here and analyze transition actions according to their written order. When dynamic choice points are present [20], the actions to be executed before checking the guard are treated as being executed just after the exit actions of the source state.

According to [23], an action has a target object set: A copy of the action with its arguments list is sent concurrently to every object in the set, and each object receives and handles this action copy independently, thus potentially constituting a complex concurrent system. This does not matter in our context as long as the triggered concurrent actions do not result, as above for asynchronous signals, in state change indeterminism.

3.3 Operation postconditions

In this research we assume that contracts [8] are defined to specify operations, and we discuss here requirements on postconditions so that they offer testable operation specifications.

In the remainder of this article, we use the term *model constraints* to refer to preconditions, postconditions or guard conditions [7]—which are used in defining UML models—in order to clearly differentiate them from test constraints. When a discussion applies to both types of constraints, we simply use the generic term "constraints".

A postcondition specifies the result of the service that is provided by an operation, given that the precondition is satisfied [8]. In practice, a postcondition usually contains different predicates that are satisfied all together when the operation terminates, e.g., an attribute's value is updated *and* a link between two objects is changed. In addition, it is common that the result of a predicate depends on the inputs of the operation (e.g., operation's arguments, object's state), e.g., if the first argument's value is 1 then the new object's state is A; otherwise it is state B. Such a situation is easily expressed in OCL with Boolean operations `implies` and `if-then-else`. For these reasons and to facilitate subsequent analyses of the statechart, we assume that postconditions are in Conjunctive Normal Form[5] (CNF) [24] whose conjuncts are either clauses or `if-then-else` or `implies` constructs. If this is not the case, postconditions can be automatically transformed to meet our assumptions using algorithms in [24, 25].

We further assume that postconditions are complete, i.e., all changes to the collective state directly resulting from the execution of the operation are modeled. These include changes to the attribute values, creation and destruction of objects, and addition or deletion of links to existing objects. Postconditions must furthermore be normalized using algorithms presented in Sect. 4.4.4. We also provide guidelines (under the form of patterns) to help developers achieve concise, consistent, and complete descriptions of postconditions.

Another assumption relates to actions in transitions. Where their computation is actually performed depends on the implementation of the statechart. When using the state design pattern [26], for instance, the actions are actually

[5] An expression is said to be in CNF if it consists of the conjunction of clauses, and each clause consists of a disjunction of atomic formulae or the negation of an atomic formula. In our context (UML and OCL), examples of atomic formulae are attributes and OCL collection operations. An expression is said to be in Disjunctive Normal Form (DNF) if it consists of a disjunction of conjunctions of atomic formulae (or negations of atomic formulae).

invoked by the *event handler* operations.[6] The result is that the event postcondition could be defined so that it includes the actions' postconditions. But in order to reduce the complexity of postconditions, and remove redundancy between postconditions (of call events and actions belonging to the same transitions), we assume that when a transition is triggered by a call event, and has actions, the actions' postconditions are not part of the call event's postcondition. This assumption does not affect the modelling of the overall effect of a transition. As we will see below, it does not affect our strategy either: If this principle is not followed, the constraints generated will simply be more complicated.

In general, postconditions model changes to object properties. According to [9] such postconditions are *change specifications*, which define what is changed by an operation. However, postconditions can also specify what does not change. Such postconditions are *frame rules* [9], and are important to a full understanding of the meaning of an operation. For example, if we have an operation that adds an object at the end of a sequence, the change specification can state that the size of the sequence increases by one and the last element is the added object while the frame rule can state that a subsequence (containing elements from the first to the second last) of the new sequence equals the old sequence. Therefore, the implementer knows that the operation does nothing else except appending the object to the sequence.

The use of frame rule has drawbacks: It makes the postcondition much more complex and adds to the burden when postconditions are checked. For instance, a class has two `Integer` attributes `a` and `b`, and an operation `incrementA()` that increments attribute `a`, and leaves the other attribute unchanged. A common way to write the operation's postcondition is to model changes to attribute `a` only: `a=a@pre+1`. However, if we follow the frame rule strictly, the postcondition would be: `(a=a@pre+1)and(b=b@pre)`. A tradeoff is to specify frame rules only where ambiguities would result from not defining them and to adopt the following convention [9]: all operations come with an implicit frame rule which states that, a property does not change unless explicitly specified.

3.4 Practical implications

The assumptions above require the pre-processing of statecharts (flattening) and postconditions (normalization, completeness) before test constraints can be derived. However, this process can be automated as long as postconditions are expressed in a precise and complete manner, an objective that we support by providing precise guidelines.

Though the rigorous use of OCL for describing model constraints imposes a certain discipline on the designers, there are many benefits to it [27], including an easier transition to implementation. We face, however, the usual, expected trade-off where potential users have to choose between rigor and precision to achieve (test) automation or the informal use of statecharts for communication purposes only.

We assume that the result of a sequence of events on a class (cluster) under test should be deterministic in terms of state changes. Therefore, since concurrency can bring indeterminism (e.g., race conditions) [20], we can only handle asynchronous signals between the owning object and associated objects in the object cluster as long as they do not entail indeterminism. We assume that, in principle, a situation in which computational results and state changes depend on an execution order, which is inherently unknowable, should be avoided through the careful use of synchronization mechanisms. Our approach should therefore be applicable to a significant proportion of statecharts in practice.

4 Methodology

This section describes our methodology to derive test constraints for a given transition test sequence (TTS). Section 4.2 precisely defines the form of a TTS. Section 4.3 introduces a data structure, the invocation sequence tree (IST), that captures the interactions among state-dependent objects, and provides a procedure to build an IST from a TTS and class-cluster statecharts. Section 4.4 provides normalization rules for the OCL expressions that we use to ease the generation of test constraints. We then describe algorithms to derive test constraints on the collective state and event/action arguments from the IST (Sect. 4.5). Finally, Sect. 4.6 briefly describes a prototype tool implementing the methodology. In order to facilitate the reading of the article, we present a selected case study along with the definitions of concepts. This case study is first presented in Sect. 4.1.

4.1 Case study

We applied our methodology on two case studies using our prototype tool: a Video Store System (VSS) that contains a class cluster whose behavior is described by a statechart, and a container class implementing a data structure that has a state-based behavior. These two case studies were selected for the following reasons: (i) a variety of OCL operations (including collection operations) are involved in both the guards and operation contracts, (ii) associated and orphan state dependent classes are involved. By studying these two different examples, we wish to provide evidence of the generality of our methodology. Due to space limitations, we only report the result of the VSS case study in this article (as it exhibits a larger variety of constructs) and refer the reader to [28] for results on the second case study.

The VSS is a UML specification for a video store management system adapted from [29]. We choose to study a cluster of three classes, namely, `Copy`, `Reservation`, and `Title` (see an excerpt of the class diagram in Fig. 1).

[6] Event handlers are operations that process events received by instances of the owning class.

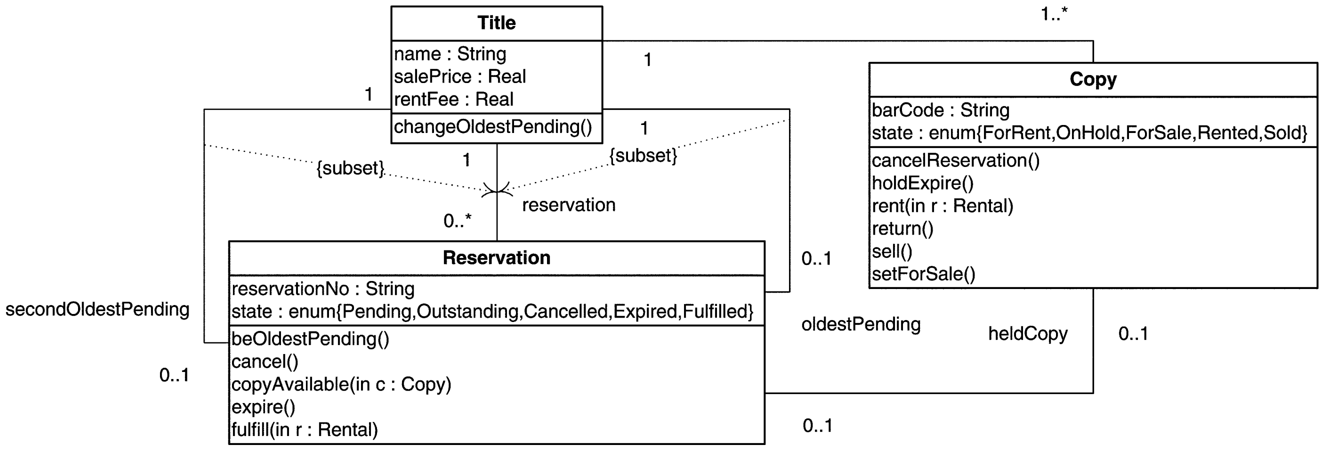


Fig. 1 Class diagram of entity classes in the VSS case study (excerpt)

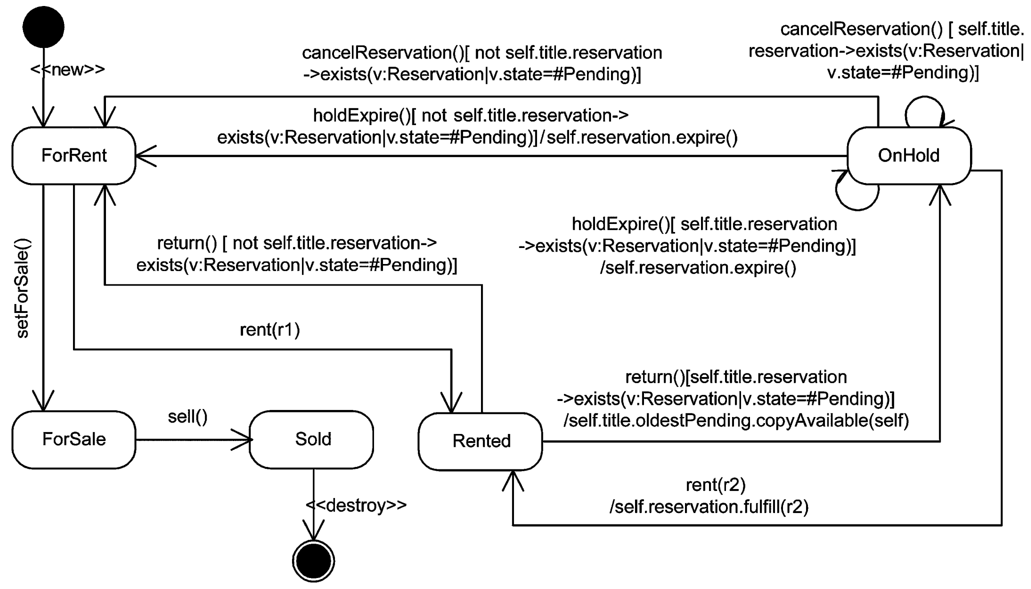

<<new>>
ForRent
OnHold
ForSale
Sold
Rented
cancelReservation()[not self.title.reservation ->exists(v:Reservation|v.state=#Pending)]
cancelReservation() [self.title. reservation->exists(v:Reservation| v.state=#Pending)]
holdExpire()[not self.title.reservation-> exists(v:Reservation|v.state=#Pending)]/self.reservation.expire()
return() [not self.title.reservation-> exists(v:Reservation|v.state=#Pending)]
holdExpire()[self.title.reservation ->exists(v:Reservation|v.state=#Pending)] /self.reservation.expire()
setForSale()
rent(r1)
return()[self.title.reservation ->exists(v:Reservation|v.state=#Pending)] /self.title.oldestPending.copyAvailable(self)
sell()
<<destroy>>
rent(r2) /self.reservation.fulfill(r2)

Fig. 2 Statechart of a `Copy` object in the VSS case study

Instances of `Copy` in the VSS have a state-dependent behavior and Fig. 2 depicts the corresponding statechart: State `OnHold` specifies that the `Copy` object is put on hold for a client who made a reservation for the corresponding `Title`, and a `Copy` is for rent (state `ForRent`) only when there is no `Reservation` for the corresponding `Title`. The behavior of a `Copy` object may thus affect, and be affected by `Reservation` and `Title` objects. A `Reservation` object has a state dependent behaviour as well, as depicted in Fig. 3: A `Reservation` object becomes `Outstanding` when a `Copy` is available. `Title` objects do not have state-dependent behavior.

4.2 Transition test sequence (TTS)

As mentioned in Sect. 2, all the proposed criteria based on statecharts assume a test case specification to be in the form of a feasible sequence of transitions [3, 4], or *transition test sequence* (TTS). We denote such a sequence as: `@state0@event0@state1@event1@...` where symbol `@` is used as a delimiter between states and event expressions in TTSs. Note that in our testing context, we assume that every TTS starts from the initial state of the owning object. So there is no need to obtain the *test prefix* [30], which is a sequence of transitions that puts the system into the initial state required for the test case to execute correctly.

When the statechart contains guards, the TTS also specifies a predicate (i.e., Boolean expression that may contain logical operators) for each event, and the way those predicates are derived from the corresponding guard conditions depends on the coverage criterion. Using a simple example, when one chooses to cover all transitions in a statechart, in the case of guard conditions, one has to choose one disjunct in the guard condition when expressed in a Disjunctive Normal Form (DNF, see Sect. 4.4.1). A number of more thorough criteria exist to test guards [30] but a sequence of transitions is always denoted as follows: `@state0@event0[pred0]/actions0@state1@`

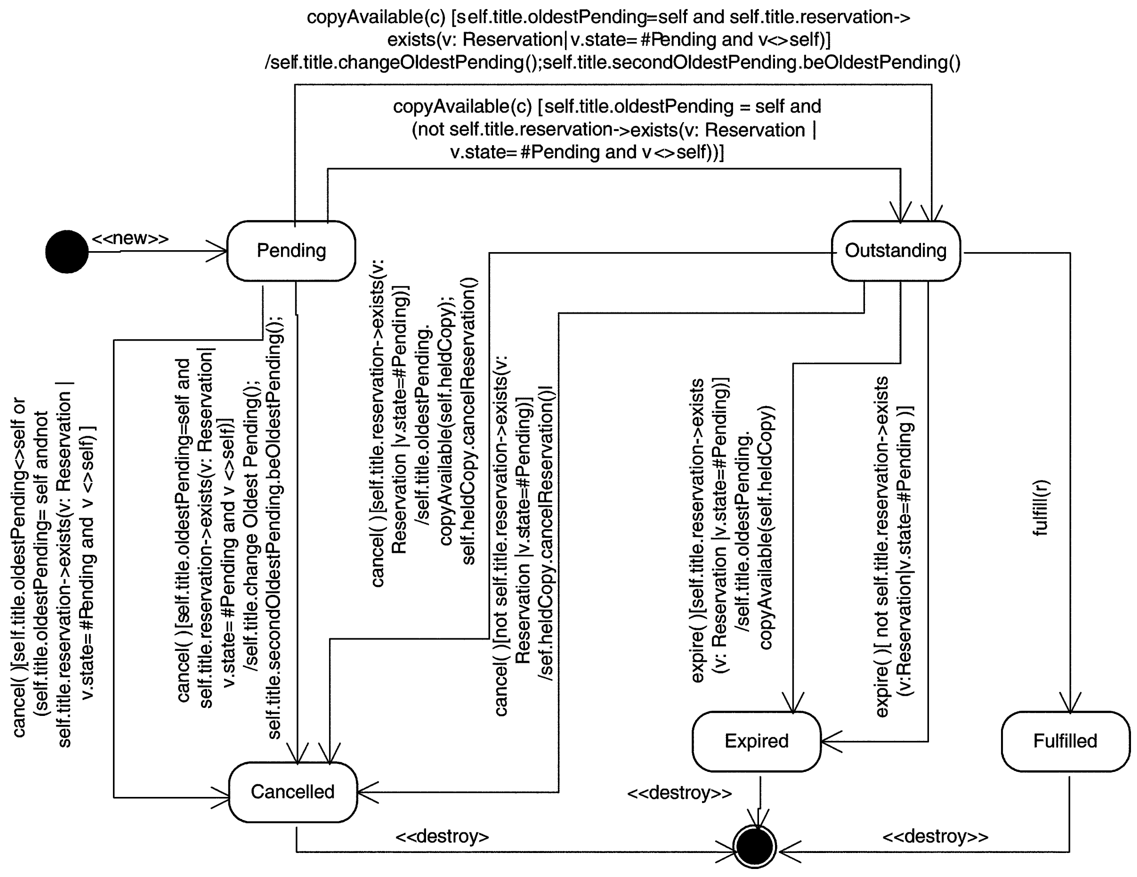

copyAvailable(c) [self.title.oldestPending=self and self.title.reservation->exists(v: Reservation| v.state= #Pending and v<>self)] /self.title.changeOldestPending();self.title.secondOldestPending.beOldestPending()
copyAvailable(c) [self.title.oldestPending = self and (not self.title.reservation->exists(v: Reservation | v.state= #Pending and v<>self))]
<<new>>
Pending
Outstanding
cancel( )[self.title.oldestPending<>self or (self.title.oldestPending= self andnot self.title.reservation->exists(v: Reservation | v.state= #Pending and v <>self)]
cancel( )[self.title.oldestPending=self and self.title.reservation->exists(v: Reservation| v.state= #Pending and v <>self)] /self.title.change Oldest Pending(); self.title.secondOldestPending.beOldestPending();
cancel( )[self.title.reservation->exists(v: Reservation |v.state=#Pending)] /self.title.oldestPending. copyAvailable(self.heldCopy); self.heldCopy.cancelReservation()
cancel( )[not self.title.reservation->exists(v: Reservation |v.state=#Pending)] /sef.heldCopy.cancelReservation()
expire( )[self.title.reservation->exists (v: Reservation |v.state=#Pending)] /self.title.oldestPending. copyAvailable(self.heldCopy)
expire( )[not self.title.reservation->exists (v:Reservation|v.state=#Pending)]
fulfill(r)
Expired
Fulfilled
Cancelled
<<destroy>>
<<destroy>
<<destroy>>

Fig. 3 Statechart of a `Reservation` object in the VSS case study

`event1[pred1]/actions1@...` where `pred0`, `pred1`, ... are the predicates derived from the corresponding guard conditions according to the criterion used, and where `actions0`, `actions1`, ... are the actions executed (there may be more than one action for a transition) when transitions fire. Those predicates are always in the form of a conjunction of atomic formulae.

Coming back to our case study, we assume in this article that we want to test the following Transition Test Sequence (TTS) on an instance of `Copy`:

```
@ForRent@rent(r1)[true]
@Rented@return()[self.title.reservation->
  exists(v:Reservation| v.state=#Pending)]/
  self.title.oldestPending.copyAvailable(c)
@OnHold@holdExpire()[self.title.reservation->
  exists(v:Reservation|v.state=#Pending)]/
  self.reservation.expire()
@OnHold@rent(r2)/self.reservation.fulfill(r2)
@Rented
```

This sequence corresponds to the following scenario: A copy is rented, i.e., there is no reservation for the title (call event `rent()` in state `ForRent`), and a reservation for the corresponding title is made before the copy is returned (state `OnHold` follows state `Rented` on call event `return()` in the sequence). Then, the reservation expires, but before it expires, another reservation is made for the title (the object stays in state `OnHold` on call event `holdExpire()`). The second reservation does not expire and the copy is rented.

4.3 Invocation sequence tree (IST)

As a result of firing a transition, the owning object may trigger changes on other objects (through actions or signals). If the target objects have state-dependent behaviours, this may also result in transitions being fired and actions triggered on yet other objects. In a TTS, each transition can then be associated with one or more invocation sequences of events and actions, depending on the number of actions in the transition (and actions in states). An *invocation sequence tree* (IST) is the data structure we define to represent these invocation sequences for a TTS, and then facilitate the derivation of constraints for that TTS (Sect. 4.5).

In this article, we assume that, if an invocation sequence forms a cycle,[7] then the sequence is infinite. We consider an infinite invocation sequence to be a sign of an ill-designed system. As a result, invocation sequences terminate when an event is sent to an object that does not have a state-dependent behavior, when the event is not recognized as a trigger event by the target object, when the target of the actions is not

[7] As a result of triggering a transition in an object statechart, an event is sent (possibly indirectly) to that same object.

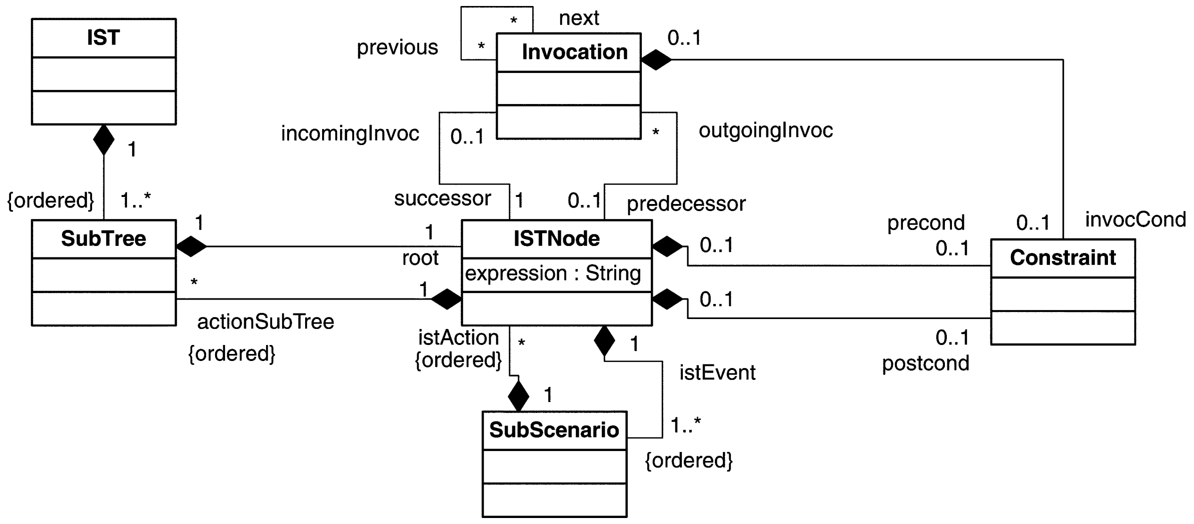


Fig. 4 IST metamodel

another object, or when the triggered transition does not have actions.

We model the structure of invocation sequence trees by means of the class diagram in Fig. 4 (metamodel). This metamodel helps us to not only precisely define what an invocation sequence tree is but also allows us to define algorithms to derive test constraints in a precise but abstract manner (Sect. 4.3.2). Such a metamodel is also a good starting point to design a tool (Sect. 4.6) for the derivation of such constraints.

4.3.1 Definitions

An IST is an acyclic digraph that shows all possible *invocation scenarios* that may occur during the execution of a TTS. However, only one scenario needs to be chosen to execute the selected test sequence. In this digraph, nodes denote events or actions and edges denote *invocation orders*. An edge is directed and connects two nodes: the *predecessor* and *successor*, the latter being invoked after the former in the invocation sequence. Nodes and edges in the IST are represented by classes `ISTNode` and `Invocation` in the IST metamodel (Fig. 4). Nodes in the IST (instances of class `ISTNode`) are labelled (attribute `expression` in class `ISTNode`) with the fully qualified name of the event or action, that is, the name of the event/action is prefixed by a path starting from the owing object. For instance a transition with an event and a sequence of two actions results into three nodes for the event and actions, and two invocations between the event node and the action nodes. An invocation scenario consists of a sequence of *invocation subscenarios* (abbr. subscenario). A subscenario corresponds to an event in the TTS and is the sequence of all the invocations involved in an invocation sequence. Figure 5 shows an example IST and the corresponding two (simple) statecharts, corresponding to classes X and Y, where X is the class under test and Y is an associated class. We see that `event1` has two subscenarios and `event2` has one

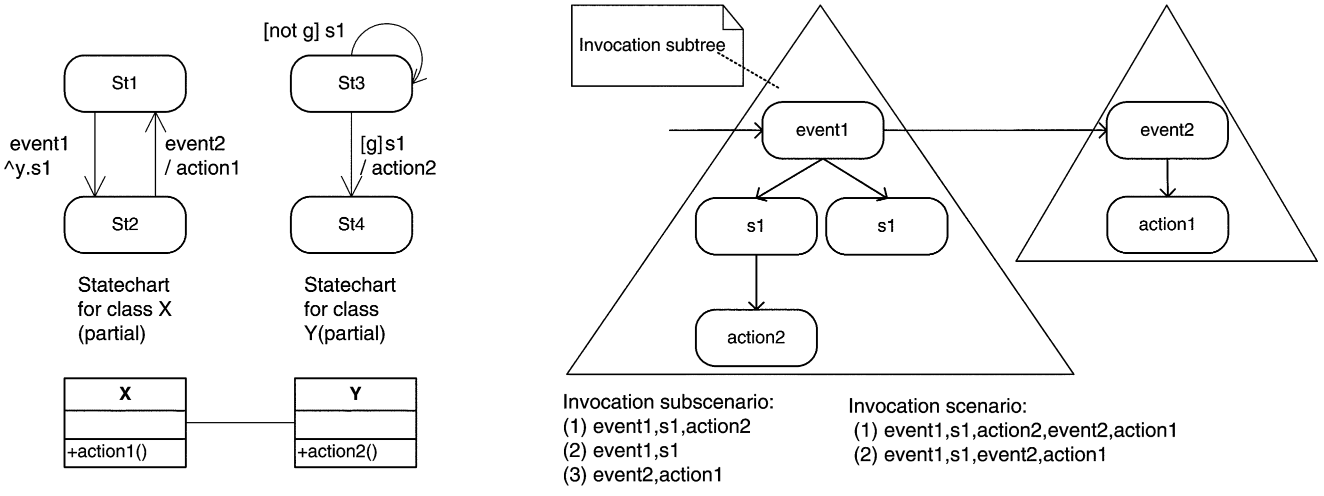


Fig. 5 An example IST

subscenario, resulting in two possible invocation scenarios for the TTS. A node corresponding to an event in the TTS and all the nodes that can be invoked directly or indirectly after it form an *invocation subtree* (abbr. subtree): class `SubTree` in Fig. 4. The subtree models all the invocation subscenarios that can be invoked when receiving an event. The IST for a TTS is then a sequence of `SubTrees`.

Invocations are labelled with conditions, referred to as *invocation conditions*, stating the condition that must be fulfilled to execute the event/action associated with the successor node. An invocation condition is a conjunction of a maximum of three parts, namely a precondition, a state condition and a guard, depending on the following three situations:

1. The successor node represents an event in the TTS (`event2` is a successor of `event1` in Fig. 5). The corresponding invocation condition is the conjunction of the event precondition and the guard of the transition.
2. The successor node represents an action that does not trigger any transition in another statechart (`action1` in Fig. 5). The corresponding invocation condition is simply the precondition of the action.
3. The successor node represents an action or signal (send clause) that triggers a transition in another statechart. The corresponding invocation condition is the conjunction of the precondition of the action (it is not relevant for signals), the guard of the triggered transition (in the other statechart), and the state invariant of the source state of that transition. In Fig. 5, `signal1` is sent to a `Y`'s instance and triggers a transition to state `St3` and the execution of `action2`.

When a transition triggered by event `ev` has several actions (or signals), say `act1,... actn`, this results into an `ISTNode` instance for `ev`, an `ISTNode` instance for each of the actions (`n` instances), and `n Invocation` instances linking the `ISTNode` for `ev` and each of the action's `ISTNode` instances. These invocation instances have to be considered in sequence as the corresponding actions are executed in sequence. Recall that actions are analyzed according to their written order (Sect. 3.2.2).

If any of those actions is invoked on a different object than the one receiving `ev`, and enables more than one transition, say `p` transitions, this also results in several `ISTNode` and `Invocation` instances: `p ISTNode` instances corresponding to `p` possible executions of the action (the `expression` of those `p ISTNode` is the same) and `p Invocation` instances between the `ISTNode` for `ev` and those `p ISTNode` instances. Here the invocation conditions for all those `p Invocation` instances are mutually exclusive, since we consider only deterministic statecharts. In other words, only one of the `Invocation` instances can be considered at a time (i.e., in a given scenario).

We now have to distinguish between sequences of `Invocation` instances and alternatives `Invocation` instances that have the same predecessor `ISTNode`. This is achieved by means of self association `previous-next` on `Invocation` (Fig. 4). If there is a `previous-next` link between two `Invocation` instances, then one has to

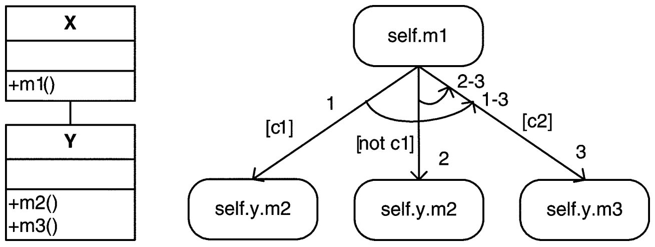


Fig. 6 Alternative relation vs. sequential relation

be considered before the other. This relation is transitive. Alternatively, when there is no such `previous-next` link between two `Invocation` instances, then only one of the two `Invocation` instances can be considered at a time: this denotes alternative invocations.

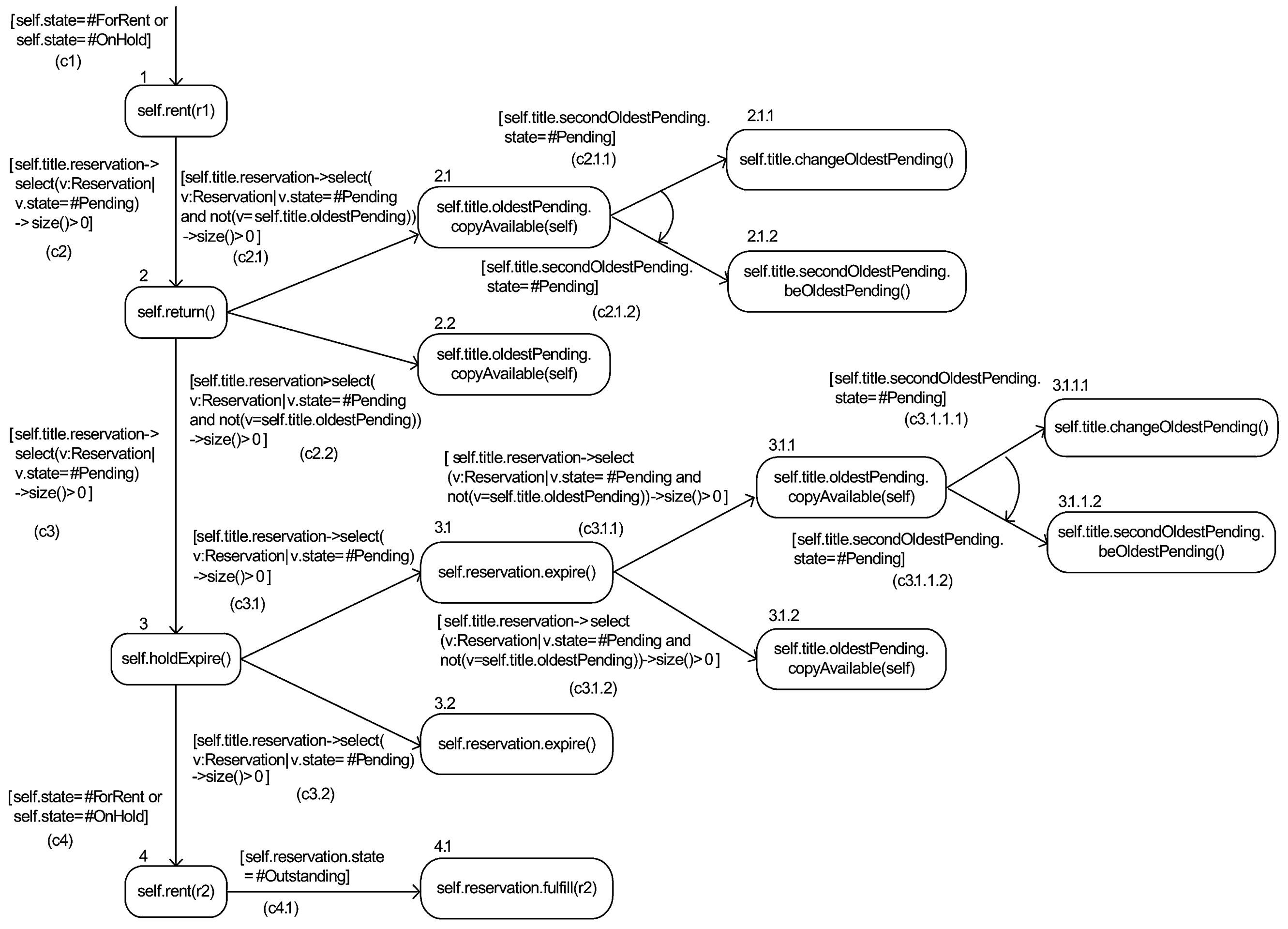
Graphically, when drawing the IST, invocation sequences are denoted with an arc connecting the `Invocation` instances involved in the same sequence, and there is no specific notation for alternative invocations.

Figure 6 illustrates these notions of invocation sequences and alternative sequences, assuming class `X`'s statechart has a transition `self.m1()[g]/self.y.m2(); self.y.m3()`, and operation `m2()` enables two different transitions in class `Y`'s statechart. As a result of firing the transition in `X`'s statechart, actions `m2()` and `m3()` are executed. First, since `m2()` enables two transitions in `Y`'s statechart, two alternative `Invocation` instances are created: both have `ISTNode` instance for `self.m1()` as a predecessor, and have different, mutually exclusive,[8] invocation conditions (respectively, `c1` and `not c1`). Since the transition in `X`'s statechart triggers a sequence of actions (i.e., `m2()` and `m3()`), those two alternative invocations are followed by another invocation instance, resulting in two sequences of invocations (represented by two arcs in Fig. 6).

In the case where the multiplicity of the association end on Y's side is higher than 1, m2 is possibly invoked on more than one instance of Y. If those are executed concurrently, it has no effect on our methodology as long as there are no race conditions involved (Sect. 3.2.2). There are two possible situations.

– Invocation condition `[c1]` is defined in terms of the state of `Y` instances and different scenarios may be executed on the different instances of `Y` depending on their concrete state.
– Invocation condition `[c1]` is defined in terms of the number of associated `Y` instances or the collective state of associated `Y` instances, and the same scenario will be executed on all associated `Y`'s instances.

[8] If the two enabled transitions originate from the same state in Y's statechart, then their guard conditions must be mutually exclusive (we consider deterministic statecharts) and the invocation conditions are thus mutually exclusive. Otherwise, the two guard conditions may be identical, however, the state invariants (also part of the invocation conditions) are then mutually exclusive, making the two invocation conditions mutually exclusive too.



[self.state=#ForRent or self.state=#OnHold] (c1)
1 self.rent(r1)
[self.title.reservation-> select(v:Reservation| v.state=#Pending) -> size()>0] (c2)
2 self.return()
[self.title.reservation->select(v:Reservation| v.state=#Pending and not(v=self.title.oldestPending)) ->size()>0] (c2.1)
2.1 self.title.oldestPending. copyAvailable(self)
[self.title.secondOldestPending. state=#Pending] (c2.1.1)
2.1.1 self.title.changeOldestPending()
[self.title.secondOldestPending. state=#Pending] (c2.1.2)
2.1.2 self.title.secondOldestPending. beOldestPending()
2.2 self.title.oldestPending. copyAvailable(self)
[self.title.reservation>select(v:Reservation| v.state=#Pending and not(v=self.title.oldestPending)) ->size()>0] (c2.2)
[self.title.reservation-> select(v:Reservation| v.state=#Pending) ->size()>0] (c3)
3 self.holdExpire()
[self.title.reservation->select(v:Reservation| v.state=#Pending) ->size()>0] (c3.1)
3.1 self.reservation.expire()
[self.title.reservation->select (v:Reservation| v.state= #Pending and not(v=self.title.oldestPending))->size()>0] (c3.1.1)
3.1.1 self.title.oldestPending. copyAvailable(self)
[self.title.secondOldestPending. state=#Pending] (c3.1.1.1)
3.1.1.1 self.title.changeOldestPending()
[self.title.secondOldestPending. state=#Pending] (c3.1.1.2)
3.1.1.2 self.title.secondOldestPending. beOldestPending()
[self.title.reservation->select (v:Reservation| v.state=#Pending and not(v=self.title.oldestPending))->size()>0] (c3.1.2)
3.1.2 self.title.oldestPending. copyAvailable(self)
[self.title.reservation->select(v:Reservation| v.state= #Pending) ->size()>0] (c3.2)
3.2 self.reservation.expire()
[self.state=#ForRent or self.state=#OnHold] (c4)
4 self.rent(r2)
[self.reservation.state =#Outstanding] (c4.1)
4.1 self.reservation.fulfill(r2)

Fig. 7 The IST of VSS

Whether in the first or second situation, the multiplicity of the association does not affect ISTs as our goal is to derive test constraints for each possible invocation scenario for a given TTS to be tested.

The Invocation Sequence Tree (IST) corresponding to the example TTS presented at the end of Sect. 4.2 is shown in Fig. 7. Note that in this figure, though this is not part of the notation, nodes and edges are numbered to facilitate the discussion. Nodes 1 to 4 represent the events in the TTS. Transition `rent()` from `ForRent` to `Rented` (Fig. 2) does not have any action and is not guarded. The corresponding node in the IST (node 1) does not have any subtree and the condition of its incoming invocation only involves the precondition of `rent()` (among the three parts involved in invocation conditions – recall Sect. 4.3.1). Figure 7 shows that call event `return()` has two subscenarios (the first one involves nodes 2, 2.1, 2.1.1 and 2.1.2 and the second one only 2 and 2.2), call event `holdExpire()` has three subscenarios, and the last `rent()` node has one subscenario.

Consider node 2 in Fig. 7, corresponding to guarded transition `return()` from `Rented` to `OnHold`. The condition `(c2)` of its incoming invocation (i.e., the edge between node 1 and node 2) involves the guard of the transition, the precondition of `return()` and the state invariant of the origin state (`Rented`). The guard condition[9] is `self.title.reservation->select(state=#Pending)->size()>0`, which is referred to as `(expr1)`. Operation `return()` precondition is `self.state=#Rented (expr2)`, which specifies that the object must be in state `Rented`. State `Rented` invariant, referred to as `(expr3)` is: `self.state=#Rented`[10] and `self.rental->notEmpty()` and `self.reservation->isEmpty()`. The invocation condition is thus the conjunction of `(exp1)`, `(exp2)` and `(exp3)`. Note that in Fig. 7, the state invariant part of invocation conditions is omitted because it is straightforward and would only clutter the diagram.

[9] Note that in the following, when there is no ambiguity, the iterator variable used in OCL collection operations is omitted.

[10] In this example, for the sake of simplicity, we assume there is a state attribute and, therefore, a simple invariant is defined. In practice, no state attribute may be used and state invariants tend to be more complicated. We decided, however, that the level of complexity of our current case study was adequate.

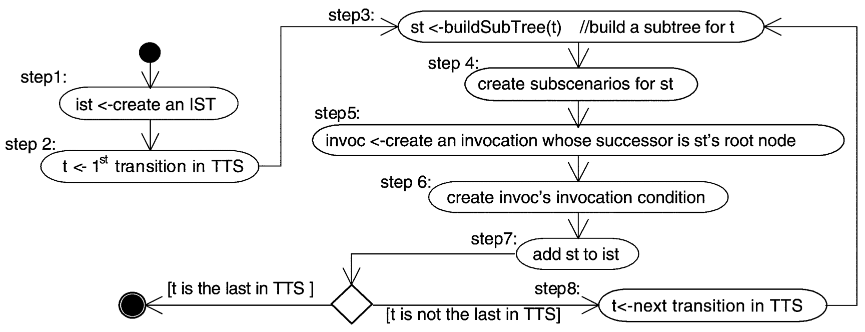

step1:
ist <-create an IST
step 2:
t <- 1st transition in TTS
step3:
st <-buildSubTree(t) //build a subtree for t
step 4:
create subscenarios for st
step5:
invoc <-create an invocation whose successor is st's root node
step 6:
create invoc's invocation condition
step7:
add st to ist
[t is the last in TTS]
step8:
t<-next transition in TTS
[t is not the last in TTS]

Fig. 8 Building IST (`buildIST()`), overall abstract algorithm

If we now turn our attention to condition c2.1, transition `return()` has one action that can trigger two different transitions on the oldest pending reservation, i.e., the two transitions between states `Pending` and `Outstanding` in Fig. 3, resulting in alternative invocations in the IST (leading to nodes 2.1 and 2.2, both labelled `self.title.oldestPending.copyAvailable (self)`). Let us consider the invocation leading to node 2.1. It corresponds to the firing of the top most transition between states `Pending` and `Outstanding` in Fig. 3. Its invocation condition is thus the conjunction of the following three elements:

– The guard conditions:
  `self.title.oldestPending=self and self. title.reservation-> exists (v:Reservation|v.state=#Pending and v<>self)`
– Operation `copyAvailable()` precondition:
  `self.state=#Pending and self.title. oldestPending=self`
– The state invariant of the origin state of the transition, namely, `Pending`.

4.3.2 Building the invocation sequence tree

To build an IST (i.e., an instance of the IST metamodel) for a given TTS, we of course need the TTS but also the UML statechart of the class under test. If the statechart contains actions that are invoked on instances of other classes, we also need the UML class diagram (to show the associations among classes) and the statecharts (if any) of those classes. We also need preconditions and postconditions for operations involved in those statecharts, and state invariants, all written in OCL.

Building an IST for a TTS proceeds in a systematic way, following a sequence of steps. In this section, we describe those steps at a high level of abstraction, by means of two functions. More detailed algorithms and examples can be found in [31].

The main function, namely `buildIST()`, whose algorithm is shown in Fig. 8 by means of an activity diagram, consists of a loop that visits all the transitions in the TTS. For each transition, function `buildSubTree()` is used to build a tree of `ISTNodes` and `Invocations`, i.e., events/actions triggered when the transition fires (step 3). Subscenarios, i.e., the actual alternative sequences of events/actions in the subtree, are then identified (step 4). The root nodes of the subtrees, i.e., `ISTNodes` corresponding to the events of the transitions in TTS, are then linked together by `Invocations` whose invocation condition is the conjunction of the guard and the precondition for each of the subtree's events (steps 5 and 6).

Figure 9 is an activity diagram describing function `buildSubTree(t)`, which recursively builds a subtree for transition `t` (recursive call at step 9). First, a subtree is created and its root node (an `ISTNode` instance) is set using `t`'s event (steps 1 to 3). Then, steps 4 to 19 sequentially consider each action in transition `t`. When the end of the action sequence is reached (or the action sequence is empty), `buildSubTree(t)` terminates (step 20). For each action in the action sequence, two different series of steps have to be followed, depending on whether the action is an event handler, i.e., depending on whether the action enables transitions in other objects. If the action is an event handler, steps 7 to 13 are involved, otherwise, steps 14 to 18 are followed. In the former situation, all the possible transitions enabled by the action are identified (step 7), and a subtree for each of them is created (steps 9 to 13). These steps are very similar to what has been described for `buildIST()` (a subtree is created, its root node is linked in the current IST with `Invocations`, ...). What is new is that sequential relations between invocations may have to be set (step 12): for instance, if the current action (which is an event handler) is the second action in the action sequence of `t`, `previous-next` links have to be set between the invocation created for the first action in the sequence and the invocations created for the event handler action. In the latter situation, i.e., when the action is not an event handler, steps are simpler: an `ISTNode` instance is created for the action, an `Invocation` instance is created to link the action `ISTNode` to the `ISTNode` for `t`'s event, and sequential invocations are accounted for (steps 14 to 18).

4.4 Normalization of OCL expressions

As further described in Sect. 4.5, deriving constraints from an IST requires that model constraints (operation contracts,

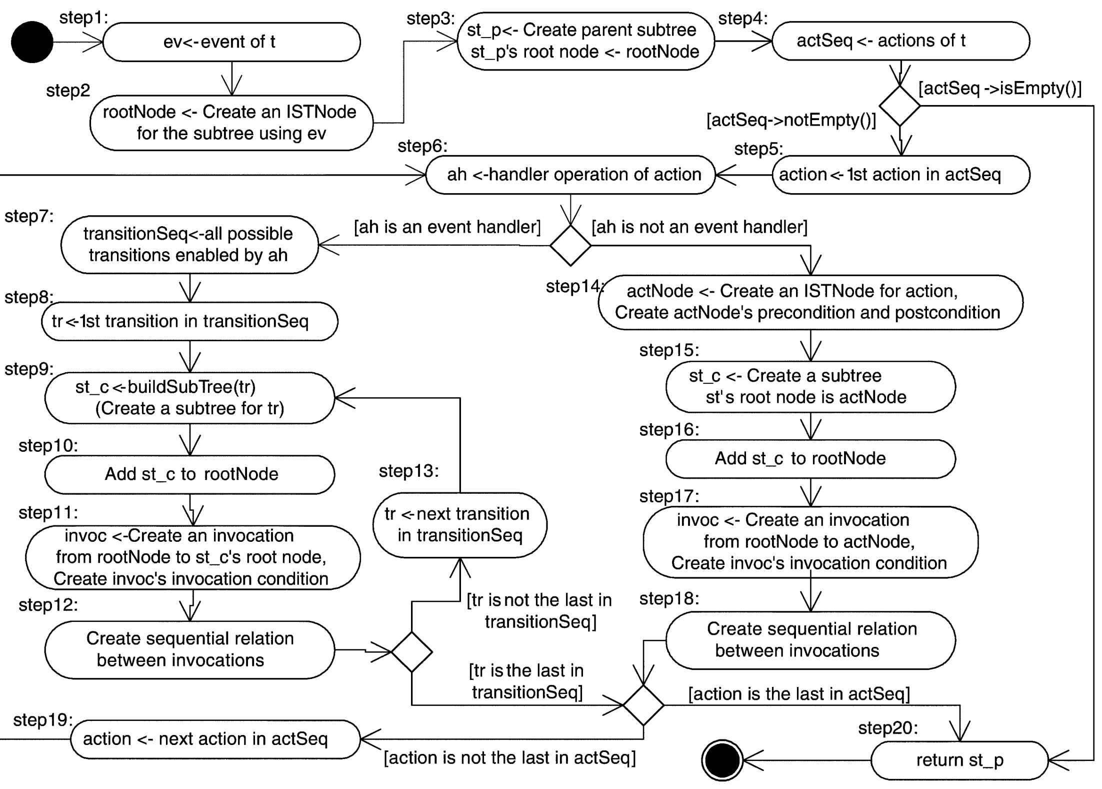

step1:
ev<-event of t
step2
rootNode <- Create an ISTNode for the subtree using ev
step3:
st_p<- Create parent subtree st_p's root node <- rootNode
step4:
actSeq <- actions of t
[actSeq ->isEmpty()]
[actSeq->notEmpty()]
step5:
action <- 1st action in actSeq
step6:
ah <-handler operation of action
step7:
transitionSeq<-all possible transitions enabled by ah
[ah is an event handler]
[ah is not an event handler]
step8:
tr <-1st transition in transitionSeq
step9:
st_c <-buildSubTree(tr) (Create a subtree for tr)
step10:
Add st_c to rootNode
step11:
invoc <-Create an invocation from rootNode to st_c's root node, Create invoc's invocation condition
step12:
Create sequential relation between invocations
step13:
tr <-next transition in transitionSeq
[tr is not the last in transitionSeq]
[tr is the last in transitionSeq]
step14:
actNode <- Create an ISTNode for action, Create actNode's precondition and postcondition
step15:
st_c <- Create a subtree st's root node is actNode
step16:
Add st_c to rootNode
step17:
invoc <- Create an invocation from rootNode to actNode, Create invoc's invocation condition
step18:
Create sequential relation between invocations
[action is the last in actSeq]
step19:
action <- next action in actSeq
[action is not the last in actSeq]
step20:
return st_p

Fig. 9 Building subtree (`buildSubTree()`), abstract algorithm

guards) written in OCL be transformed to become analyzable and comparable, a procedure referred to as *normalization*. A normalized constraint is said to be in a *normalized form*.

The OCL constraints are normalized to support a number of analyses:

– Constraint derivation: Transforming a postcondition to a form that explicitly shows the relation between the pre-state and the after-state of the operation.
– Consistency checking among OCL expressions: Check consistency and redundancy among constraints. For example, we need to compare a postcondition and the constraint on the current collective state and remove the redundant parts.

OCL offers a rich set of operations on the predefined OCL types. This provides the modeller with flexibility in writing and thus enhances its ease of use. However, this flexibility brings about variety. For instance, the same constraint can be written using different collection operations and this raises difficulties regarding the manipulation of OCL expressions. Normalization steps are required to unify the form of semantically equivalent constraints and compare constraints with different semantics. We identify five types of normalizations: Normalization of logical expressions (Sect. 4.4.1); Normalization of navigation paths (Sect. 4.4.2); Normalization of operations on OCL basic and collection types (Sect. 4.4.3); Normalization of postconditions (Sect. 4.4.4); Elimination of local variables (defined in OCL let expressions) and query operations[11] (Sect. 4.4.5).

The VSS case study is then used in Sect. 4.4.6 to illustrate some of the normalizations, and we discuss in Sect. 4.4.7 the advantages of normalized forms and their limitations.

4.4.1 Normalizing logical expressions

The main motivation is to ease the combination of OCL expressions and check for redundancy. Model constraints are OCL expressions of type Boolean, i.e., logical expressions. Typical normalizations for logical expressions are the conjunctive normal form (CNF) and disjunctive normal form (DNF). Any OCL expression can be converted into its equivalent CNF or DNF using standard Boolean algebra [24].

Operation preconditions and invocation conditions are transformed into DNF (Disjuncts are the different conditions under which operations and invocations can be executed or triggered); Class invariants are transformed into CNF (Conjuncts are the conditions that must hold at the same time in order to define a valid state); Operation postconditions are in the form of a conjunction of atomic formulae or a conjunction of `implies` expressions[12] (each implied part of the

[11] The only operations defined in the class diagram allowed in OCL expressions as they do not change the state of the system [1].

[12] Recall we assume postcondition are complete and precise. Therefore, in the case of alternative postconditions, we assume them to be in the form of `(A implies B) and (not A implies C)`,

implies expressions is a conjunction of atomic formulae). The reason why implies expressions are not transformed, for instance using Modus Ponens [24] is that, as discussed in Sect. 4.5, this facilitates the identification of the different conditions under which a subscenario can be executed. This requires, however, that OCL constructs like if expressions be transformed into implies expressions. Additionally, the predicate parts of implies expressions have to be transformed into CNF and then a last transformation ensures that in a conjunction of implies expressions, the predicate parts of all the expressions are explicitly mutually exclusive. (further details can be found in [28]).

4.4.2 Normalizing navigation path in OCL expressions

The complexity of comparing OCL expressions is in part due to navigation mechanisms offered by OCL: This allows modellers to write expressions that start from a context class and reach other classes through associations. As a result, two seemingly different navigation paths can result in the same collection of objects, and thus two seemingly different OCL expressions can be equivalent. Let us take an example from a Video Store System (see Sect. 4.1), where there are associations between classes Copy, Title and Reservation. The two following constraints can be written in the context of class Copy (i.e., self represents a Copy instance): self.reservation.title.name=''b'' and self.title.name=''b''. In both expressions attribute name of a Title instance is constrained. Those two expressions are equivalent, though it is not clear from the navigation paths that the Title instance involved in both constraints is the same. The goal of normalizing navigation paths is thus to be able to compare different OCL expressions and determine, for instance, whether they are equivalent.

The equivalence of navigation paths cannot be, most of the time, fully determined from a class diagram or any other UML diagram since the semantics of the associations has to be considered. Note that one of the main reasons for having equivalent paths in a class diagram is that redundant associations are added to the class diagram during design to improve performance [2]. As a consequence, the user is expected to provide information about redundant or equivalent paths through the class diagram associations: It is usually possible to enumerate all the equivalent navigation paths as their number is generally small.

Normalizing navigation paths is performed during the construction of the IST and can be fully automated using inputs from the user on equivalent navigation paths. We first unify (Sect. 4.4.2.1) the context object in the navigation paths of action expressions (in the case of associated state-dependent objects), the invocation conditions in the IST, and the operations' contracts. Navigation paths are then simplified by replacing them with their simplest equivalent paths (Sect. 4.4.2.2).

assuming a case with two alternatives. Disjunctions, if present, are transformed into implies expressions, e.g., (a or b) implies c is transformed into (a implies c) and (b implies c). Postconditions are therefore expected to be in CNF.

4.4.2.1 Unifying the context object in navigation paths The first step is to change the navigation path's context object. Recall that any OCL constraint is written in the context of a specific class and the reserved word self is used to refer to the context object [7]. A constrained element has a navigation path that starts from the context of the constraint, possibly followed by a sequence of association-ends (or class names) until the class that owns the property is reached. In this work, for each constrained element, we transform the context object of its navigation path to be the owning object. The rationale is that combining OCL expressions requires a common context object to enable comparisons. It is also preferable to choose the owning object as the common context object due to the fact that constrained elements in most model constraints we manipulate (such as guards and operation contracts for events and actions) already have the owning object as the context object, and accordingly this choice will minimize the effort of unifying the context object.

As an example, let us consider the transition between states Rented and OnHold of a Copy object (the owning object) triggered by the event return in Fig. 2. The action of the transition is a call to operation copyAvailable() on the oldest pending Reservation object. In Fig. 3, action copyAvailable() is the call event that triggers the transition from state Pending to Outstanding. In the statechart the guard of this transition is:

```
[  self.title.oldestPending = self
   and
   self.title.reservation->exists
     (v:Reservation| v.state=#Pending
     and v<>self)]
```

Note that the context object self in this guard refers to the oldest pending reservation object and it needs to be changed to the owning object (the Copy object). The navigation path self.title.oldestPending of the action expression on the transition in Fig. 2 shows how to navigate from the owning object to the oldest pending reservation object.

The changing of the context object is straightforward, self in the above guard can be replaced by the navigation path self.title.oldestPending. We obtain:

```
[  self.title.oldestPending.title.
     oldestPending =
       self.title.oldestPending
   and
   self.title.oldestPending.title.
     reservation->
   exists(v:Reservation|
   v.state=#Pending and v<>self.title.
     oldestPending)]
```

4.4.2.2 Simplifying navigation paths We simplify navigation paths by (1) removing redundancies within navigation paths and (2) replacing a navigation path by its simplest equivalent path. Both simplifications are possible once the user has provided relevant information regarding the class diagram association paths, which a tool cannot simply deduce from UML diagrams.

Removing redundancy within a navigation path. Assume there exists one association between class `A` and `B` and that the multiplicity at both association ends is `1` or `0..1`. In the context of class `A` the following OCL expression then holds: `self.b.a=self` (a similar expression exists in the context of class `B`). More generally an OCL constraint involving a navigation path of the form `prePath.b.a.postPath`, where the result of navigation path `prePath` is an instance of class `A`, is redundant and can be simplified into `prePath.postPath`. Note that such a simplification can be generalized to more than two classes provided that associations' multiplicities are `1` or `0..1`. For other multiplicities, the user input is required to specify whether certain paths in the class diagram can be simplified.

As an example, let us look at the two conjuncts of the guard in the previous example. The navigation paths of the two constrained elements are as follows (Fig. 1):

(1) `self.title.oldestPending.title.oldestPending`
(2) `self.title.oldestPending.title.reservation`

From the class diagram, the association end with the role name `oldestPending` has multiplicity `0..1` and the other association end has multiplicity `1`. So the following equations hold.

(a) `self.title.oldestPending.title = self.title`
(b) `self.title.oldestPending.title.oldestPending =self.title.oldestPending`

The previous guard can therefore be simplified into:

```
[  self.title.oldestPending = self.
     title.oldestPending
   and
   self.title.reservation->exists
     (v:Reservation| v.state=#Pending
     and v<>self.title.oldestPending)]
```

Note that, as shown by the first conjunct above, this first step can reveal constraints that are always true, and are eventually removed.

Replacing a navigation path with the simplest equivalent path. Two different situations can occur. First, different navigation paths starting with the same context that result in the same collection of objects are said to be equivalent since they provide different ways to retrieve the same information. For instance, using the previous example, paths `self.title` and `self.reservation.title` are equivalent. Then, one of the equivalent navigation paths can be considered the simplest (e.g., `self.title`) and can be used to replace any occurrence of the other equivalent navigation paths in OCL expression (e.g., `self.reservation.title`). In this case, the simplest navigation path is simply the shortest in terms of number of associations.

A second situation leading to possible simplifications is when two different collections of objects, resulting from two different navigation paths starting from the same context (say `path1` and `path2`) have an include relationship (in OCL, `path1->includesAll(path2)`). In this situation, OCL collection operations (such as `select`) can be applied on the larger collection to yield the smaller collection: one can find an OCL operation `op`, likely with an OCL expression as a parameter (such as `select`), such that `path1->op()=path2`. Then, any occurrence of `path1->op()` can be replaced by `path2`. In this case, the simplest navigation path is the one that does not involve an OCL collection operation.

In both situations, the user input is required to specify equivalent, simpler paths as the class diagram does not provide enough information to do so automatically: The semantics of associations need to be considered.

4.4.3 Normalizing operations on OCL types

OCL offers a rich set of predefined operations for so called basic types (e.g., `Integer`) and collection types (e.g., `Set`). Again, this provides flexibility but also results in different ways of expressing constraints. For instance, `aSet->size()=0` and `aSet->isEmpty()` are two syntactically different constraints that are semantically equivalent. Normalization is then necessary in order to compare operations on OCL basic and collection types.

The normalization of operations is based on the fact that many operations on OCL basic and collection types can be expressed through other operations. For instance, when two values `r1` and `r2` of type `Real` are compared, `r1<=r2` can be re-expressed as `(r1<r2)or(r1=r2)`, and `collection->isEmpty()` can be re-expressed as `collection->size()=0`. The normalization of operations on OCL types consists in changing OCL expressions such that only a subset of the operations, called the set of *core operations*, is used. Core operations must have the same expressive power as the complete set of operations. However, the choice of the core operations is not straightforward. Recall that our aim is to enable the test constraint derivation approach described in Sect. 4.5. This approach relies solely on the syntax to identify the semantic equivalence between two constraints. There are two extreme situations for the choice of a set of core operations: (1) it includes all OCL type operations, i.e., no operations are normalized; (2) it is the minimum set of operations, that is, no core operations can be re-expressed by any other core operations. (1) is obviously not a good choice. For our problem, (2) is not a good choice either as we have to keep some redundant operations. Indeed, `implies` expressions are not transformed into conjunctions and disjunctions using Modus Ponens to facilitate the derivation of test constraints (see Sect. 4.5). Moreover, operations that iterate over collection elements such as `select`, `collect`, and `sum` are not transformed into equivalent expressions using operation `iterate`, since they enhance the readability of the test constraints (more readable than their `iterate` counterpart). This is important as the resulting test constraints are likely to be presented to the user for further analysis, so as to detect possible remaining redundancies or impossible constraints. Of course this is a trade-off as keeping these iterative operations in addition to `iterate` may result into redundancies and inconsistencies that are not detected. We therefore advise not to use `iterate` in OCL expressions as it is usually possible, in the vast majority of cases we encountered, to use high level iterative operations such as `select` or `collect`. Therefore, we adopt an intermediate normalization strategy between solutions (1) and (2), and try to alleviate the consequences of not having a minimum core of operations by avoiding the use of `iterate`.

The normalization of operations can be automated and is performed on both the operation contracts and the guards. In the following, we only list example normalization rules due to space constraints. For a complete list the reader is referred to [31].

The normalization rules of operations on basic types are straightforward, as illustrated by the examples below.

– Boolean type: `b xor b2` is transformed into `not(b = b2)`.
– Real and Integer type: `r<>r2` is transformed into `not(r=r2)`.

The normalization of collection operations is much more complex. Two types of collection operations are distinguished here: Type (1) collection operations return a new collection, whereas Type (2) operations test the properties of the collection and return a Boolean or Integer value. The normalization of Type (1) operations is part of the step to identify the equivalence of two OCL expressions returning collections. The normalization of Type (2) collection operations transforms most of these operations into a test on a numeric property of the collection. The motivation is to ease consistency & redundancy checking between OCL expressions involving collection operations. Some examples of the normalization rules for Type (1) and Type (2) collection operations are provided below where : denotes on the right hand side, the normalized form of the expression on the left hand side.

– Type (1) collection operations
```
set->including(object) :
  set->union(Set{object})
set->excluding(object) :
  set - Set{object}
bag->reject(expr) :
  bag->select(not expr)
sequence->append(object) :
  sequence->union(Sequence{object})
```
– Type (2) collection operations
```
set->includes(object) :
  set->count(object) = 1
bag->includes(object) :
  bag->count(object) > 0
sequence->includes(object) :
  sequence->count(object) > 0
```

Furthermore, operation `select` is further normalized into multiple `select` operations when the OCL expression passed as a parameter is a conjunction. For instance, `path-> select(a and b and c)` is transformed into the equivalent expression `path->select(a)->select(b)->select(c)`. This normalization ensures that if `path->select(a and b and c)` appears in several OCL expressions (e.g., two different postconditions), possibly with different orders of conjuncts `a`, `b` and `c` (e.g., `a and c and b`), the normalized expression is the same[13] (i.e., `select(a)->select(b)->select(c)`). This normalization increases the chances of the constraint derivation algorithm to find equivalent OCL expressions (it uses the syntax of OCL expressions to compare them).

4.4.4 Normalizing PostConditions

As further described in Sect. 4.5, one important issue during constraint derivation is to transform a constraint on the after-values, as specified in an invocation condition, into a constraint on before-values using an operation postcondition. To enable the necessary substitutions, postconditions must be normalized to explicitly show the relations between the pre-state and the after-state of the operation: they must be transformed into a specific format that will enable such substitutions (Sect. 4.4.4.1) and they must be complemented (Sect. 4.4.4.2).

4.4.4.1 The standard equality form (SEF) We say an atomic formulae is in the *Standard Equality Form* if and only if it is written as an equation `l-value = r-value` where `r-value` is an OCL expression of only parameters, literals or values of properties (e.g., attributes) at the start of the execution of the operation (i.e., `@pre` values). A conjunction (or disjunction) of atomic formulae is in the *SEF* if and only if all its atomic formulae are in the SEF. An `implies` expression is in the *SEF* if and only if the implied part is a conjunction of atomic formulae in the SEF. And a postcondition, i.e., a conjunction of atomic formulae or `implies` expressions, is in the *SEF* if all its conjuncts (i.e., atomic formulae or `implies` expressions) are in the SEF.

Some examples and counter-examples of clauses in the Standard Equality Form are provided in Table 1 (where `attrib` denotes an attribute, `param` denotes a parameter).

[13] We have to chose a specific order for `a`, `b`, and `c`. A simple one is the alphabetical order of OCL expressions `a`, `b`, and `c`.

Table 1 Example clauses that comply with or violate the Standard Equality Form

Clauses in the SEF	Clauses not in the SEF
`attrib=10`	`attrib-10=0`
`attrib=param+1`	`attrib1@pre+1=attrib2`
`attrib1=attrib2@pre+1`	`attrib1=attrib2+1`
`set=Set{object1, object2}`	`asset->includes(obj)`
`set->size() = set@pre->size()+1`	`collection->isEmpty()`
`collection->size()=0`	

Note that clause `attrib1=attrib2+1` is not in the SEF as, using that clause only, one doesn't know whether `attrib2`'s value is changed by the operation. For instance, if another clause in the operation's postcondition is `attrib2=attrib3@pre`, then the SEF for the clause above is `attrib1=attrib3@pre+1`. If on the contrary `attrib2`'s value is not changed by the operation, then the SEF is `attrib1=attrib2@pre+1`. Note also that `collection->isEmpty()` is not in the SEF whereas the equivalent expression `collection->size()=0` is. The SEF is one of the main reasons for the normalization rules of OCL collection operations introduced in Sect. 4.4.2.

Finally, transforming a postcondition into its SEF can be automated as this simply entails replacing occurrences in `r-values` with `l-values` of other equations.

4.4.4.2 Complementing PostConditions Assume that operation `op1` has the following postcondition: `roleName=roleName@pre->append(obj)` where `obj` is a parameter of the operation and `roleName` is a navigation of an ordered association (i.e., `roleName` denotes a `Sequence`). Further assume that, following the node representing operation op1 in the invocation scenario, the invocation condition is `roleName->size()<3`.

When deriving the test constraint for the invocation scenario, we have to transform the invocation condition above (i.e., which is a constraint on the size of the sequence after the execution of `op1`) into a constraint on the size of the sequence before the execution of `op1`. As described in Sect. 4.5, this is done based on the operation postcondition. However, nothing in the operation's postcondition describes how the size of the sequence is changed: the size is implicitly increased by one because of the `append` operation. In order to allow the derivation of the constraint on the size of the sequence before the execution of the operation, we have to complement the postcondition and re-write it as follows: `roleName=roleName@pre->append(obj) and roleName->size()= roleName@pre->size() +1`. It is then possible to derive that `roleName->size() <2` must hold before the execution of the operation.

The problem is that we cannot presume, when complementing the postcondition, what are the other model constraints (invocation conditions) that need to be propagated based on the postcondition, i.e., how we should complement the postcondition. For instance, if instead of a constraint on the size of the sequence, we have a constraint on the first element in the sequence, we have to complement the postcondition in a different way and state that the first element in the sequence is not changed by the operation. To address this issue we have to complement the postcondition in a systematic way according to the OCL type and the operation used. In our example, we have to state three things: the size of the collection increases by one, the last element in the collection is the one added, and all the other elements remain the same. The postcondition then becomes[14]:

```
roleName=roleName@pre->append(obj)
and roleName->size()=roleName@pre->
  size()+1
and roleName->at(roleName->size())=obj
and roleName->size()>1 implies
        Sequence{1..roleName}->size()-1->
        forAll(index:Integer|roleName->at
        (index)=roleName@pre->at(index))
```

Note that what is added as a complement to the postcondition is inspired by the postcondition of the OCL collection operation (`append`) as stated in [7]. We define *postcondition complementing rules* for all operations in [31].

In order to complement operation postconditions in a systematic way, there must be a way to identify what changes have been modeled, and more importantly, to identify whether changes are modelled in a complete way (i.e., the postcondition unambiguously specifies what changes and what does not change). However, this is not an easy task because there is no set of rules on how to use OCL to model operation contracts. In other words, the flexibility provided by OCL makes it more complicated to analyze postconditions. Therefore, we assume that modellers use OCL in a specific way when they write operation contracts. We define specific forms for modeling a particular change to certain types of model elements (e.g., adding or deleting an element from a set) and we call these forms *contract patterns*. Such patterns are designed with the aim to be correct, complete, concise and easy to understand. Therefore, they do not only facilitate our analysis, they are also desirable in terms of properly defining contracts in OCL. For each contract pattern, we define a corresponding complementing rule that converts the postcondition of the operation into its normalized form (with complement postconditions added). When complementing a postcondition, we use these patterns to match the operation contract. If the contract matches a pattern, it is replaced by the pattern's corresponding complemented postconditions (e.g., instances of the append operation above are replaced with the four conjuncts of the complementing rule).

For the list of all possible changes to model elements, their contract patterns, and the corresponding complementing rules, refer to [31].

We now show a contract pattern example. Assume that class `className` has an `operation` that adds an `object`, passed as a parameter, to a `set`. The contract

[14] For the sake of simplifying our discussion, we left the `append()` in the expression, though it is supposed to be normalized.

pattern is as follows (parentheses show the normalized forms of excludes and includes):

```
context className::operation (object:
  objectTypeParameter)
/************* pattern ************* /
pre:  set->excludes(object)
  (set->count(object)=0)
post: set->includes(object)
  (set->count(object)=1)
    and set->size()=set@pre->size()+1
```

The complemented postcondition is shown below:

```
/**** Complemented postcondition **** /
post: set = set@pre->union(Set{object})
  and  set->size()=set@pre->size()+1
  and  set->count(object)=1
```

Several things need to be noted. First, the complemented postcondition is in standard equality form. Second, one more conjunct (the third one) is added to the complemented postcondition. This conjunct is necessary if we have to propagate an invocation condition that checks the value of set->count(object).

4.4.5 Eliminating local variables and query operations

Because let expressions allow the definition of local variables and functions, thus helping to write complex OCL expressions, every occurrence of such a variable or function must be replaced by its corresponding expression.

Query operations [1] do not have any impact on the collective state and return a value (the result keyword is used in the postcondition of the operation). We replace every call to a query operation with the expression assigned to result.

4.4.6 Examples in the VSS case study

If we now revisit the invocation leading to node 2.1 in the IST of Fig. 7, recall this was stated to be the conjunction of three elements, including the two listed below:

– The guard conditions:
```
self.title.oldestPending=self and
  self.title.reservation->
exists(v:Reservation|v.state=#Pending
  and v<>self)  (expr4)
```
– Operation copyAvailable() precondition:
```
self.state=#Pending and self.title.
  oldestPending=self (expr5)
```

Some upfront normalization is necessary to obtain the invocation condition c2.1 in Fig. 7 from these elements. First the navigation paths must be normalized, since we want to describe a constraint in the context of the Copy object whereas self in (expr4), for instance, refers to a Reservation object: the oldest pending reservation, as shown in the navigation path of call event copyAvailable() of the transition that is fired (Fig. 2). Therefore, self in (expr4) and (expr5) must be replaced with self.title.oldestPending (self here refers to the Copy object), and we obtain:

```
(expr4.1)   self.title.oldestPending.title.
              oldestPending=self.title.
              oldestPending
            and self.title.oldestPending.title.
              reservation->exists(v:Reservation|
              v.state=#Pending and not(v=self.
              title.oldestPending))
(expr5.1)   self.title.oldestPending.state=#
              Pending
            and self.title.oldestPending.title.
              oldestPending=self.title.
              oldestPending
```

Next, the navigation paths are simplified as there exist redundant paths in the class diagram. Redundant paths, as provided by the user are: self.title.oldestPending.title.oldestPending=self.title.oldestPending and self.title.oldestPending.title=self.title. As a result, (expr4.1) and (expr5.1) are transformed into:

```
(expr4.2) self.title.oldestPending=self.title.
            oldestPending
          and self.title.reservation->exists
            (v:Reservation|v.state=#Pending and
            not(v=self.title.oldestPending))
(expr5.2) self.title.oldestPending.state=#
            Pending and self.title.
            oldestPending=self.title.
            oldestPending
```

Another information provided by the user states that the oldest pending reservation for a title is always in state pending,[15] i.e., self.title.oldestPending.state=#Pending is always true. The above two expressions can then be transformed into:

```
(expr4.3)   self.title.reservation->exists
              (v:Reservation|
            v.state=#Pending and not(v=self.
              title.oldestPending))
(expr5.3)   true
```

Last, operation exists() is transformed using operation select(), and the result is the invocation condition c2.1 in Fig. 7.

4.4.7 Advantages and limitations

As discussed in previous sections, the normalization of the OCL expressions has a number of advantages with respect to our objectives of automating the derivation of test constraints. It unifies the expression of semantically equivalent model constraints (Sects. 4.2, 4.3 and 4.4.5) so that each distinct constraint can be uniquely expressed (in most cases).

[15] Note that this information could also be found in the class invariant of class Title.

It further converts each type of model constraints (preconditions, postconditions, and guards) into appropriate normal forms (Sect. 4.4.1) and transforms the postcondition (Sect. 4.4.4) to enable the automated test constraints derivation. Furthermore all the normalization steps themselves can be automated at this point. An additional benefit is that the normalized forms of postconditions provide guidelines (in the form of patterns) on how to write operation contracts that are complete and concise.

However, there is one limitation for the normalization technique in terms of detecting redundancies and inconsistencies among OCL constraints. For example, two normalized OCL constraints with a different syntax might be semantically redundant (e.g., `self.a>1` and `self.a>2`) or inconsistent (e.g., `self.b>10` and `not(self.b>5)`). Such relations are not easily detected even after the normalization steps have been performed. Theorem proving would be the most appropriate technique to check these relations [24]. Yet theorem-provers for OCL are not readily available. Even if there were such theorem provers, the cost and complexity of applying them will likely be very high.[16] Therefore, this research does not adopt any theorem proving technique. Instead, it relies solely on the syntactic equivalence to identify relations among OCL constraints. Nevertheless, we have observed that our technique based on normalization is still useful and practical for two reasons. First, in many cases, after the normalization is performed, two OCL constraints have exactly the same expression or one expression is a direct negation of the other thus easily enabling automated detection of redundancies and inconsistencies, respectively. For instance, after normalization, two constraints `aCollection->isEmpty()` and `aCollection->size()=0` will be identical (both will be normalized into `aCollection->size()=0`) so that they can easily be detected as redundant. Second, redundancies and inconsistencies in the test constraints do not prevent us from applying the test constraint derivation methodology. The only drawback is that the process may produce test constraints that consist of redundant or inconsistent components. One solution is that the tester manually identifies such relations among constraints. The normalization techniques make this work much easier. Moreover, it may not be necessary to identify and remove redundant and inconsistent postconditions during the test constraint derivation. During the subsequent constraint solving stage (which is the next step to derive test data after the test constraints are obtained), when we apply heuristic constraint solving techniques such as Genetic Algorithms (GA) [32], the redundancies within constraints might delay the convergence of the GA algorithms but will not prevent it from finding the solution. And in cases where there are inconsistencies among these constraints, GA algorithms will not converge, and we can then claim, with a certain degree of confidence, that there exist inconsistencies that make the test constraints unsolvable and check them more thoroughly.

[16] Recall that the test constraint derivation process iterates all the invocations in an invocation subscenario (Sect. 4.5) and theorem proving needs to be performed after the constraint propagation during each iteration. Moreover, the propagated constraints might be in a very complex form, thus further increasing the burden of the theorem prover.

4.5 Test constraints derivation

Recall from Sect. 3.1 that we distinguish associated state-dependent objects (their state depends on other objects' states) from orphan state-dependent objects. Indeed, since the behaviour of an associated state dependent object depends on other objects' state, some transitions in the TTS for the associated state-dependent object may require that other objects be in specific states. A driver executing the TTS then needs to suspend the execution to set the state of those other objects before resuming the execution (i.e., before triggering the transition that requires other object to be in specific states). In the case of an associated state-dependent object, it is then important to derive (under the form of a test constraint) the collective state before every transition in the TTS (this includes the initial state), in addition to the argument values for events and actions. In the case of an orphan state-dependent object, since the behaviour does not depend on any other object, we can derive a test constraint for the whole TTS, that is a constraint to be satisfied at the very beginning of the TTS. As a result, the driver does not have to suspend the execution.

An invocation subscenario represents a possible firing of a transition in the owning object's statechart. It starts with an edge (i.e., an `Invocation` instance) labelled with the conjunction of the transition's guard condition and the precondition of the transition's event handler, followed by a node (i.e., an `ISTNode` instance) representing the event of the transition. This is possibly followed by a sequence of edges/nodes representing the actions in the transition, and in transitions of other statecharts that are triggered as a result of the transition. Figure 10 shows an example IST[17]: details on events, actions and invocation conditions have been omitted and replaced by simple names (e.g., condition `c1` for the edge leading to node labelled `n1`). This IST is built for a TTS composed of four transitions which events are `n1`, `n2`, `n3` and `n4` (and guard conditions `c1`, `c2`, `c3` and `c4`, respectively). Figure 10 also highlights two invocation subscenarios (circled), one for event `n2` and one for event `n3` from a total of seven subscenarios in the whole IST: one subscenario for event `n1` (limited to `n1`), two for event `n2`, three for event `n3` (because of two alternative invocations at nodes `n3` and `n3.1`) and one for event `n4`.

Deriving a test constraint for an invocation subscenario—the constraint which, when satisfied, allows the execution of the event(s)/action(s) (sometimes by sending signals) in the subscenario—amounts to propagating constraints that appear in the tree branches of the subscenario (i.e., invocation conditions for edges and postconditions for nodes) onto the first edge of the subscenario

[17] This is not an abstract IST as **Error! Reference source Fig. 10.** is the IST in **Error! Reference source Fig. 7.** (the Video Store System case study) with a different layout and different labels.

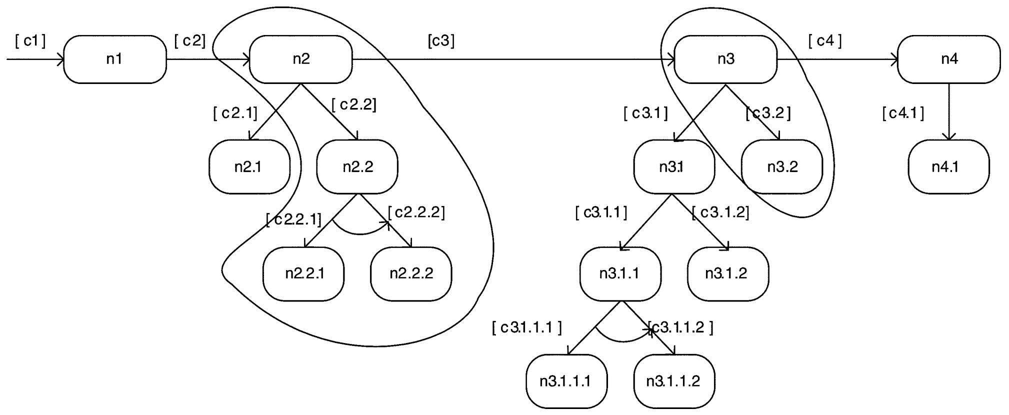


Fig. 10 An example IST

(e.g., edge labelled [c2] for event n2's subscenarios). This is done in a recursive, bottom-up way, starting with the last edge in the tree path representing the subscenario and ending with the first one (e.g., in Fig. 10, from the edge between n3 and n3.2 to the edge between n2 and n3). Note that the result of that process might be a set of constraints rather than a single constraint as discussed below.

We first describe how constraints are propagated in subscenarios from edge to edge (Sect. 4.5.1) and then describe how the corresponding OCL expressions are transformed (Sect. 4.5.2). We then illustrate the whole approach on the VSS case study (Sect. 4.5.3).

4.5.1 Propagating constraints in a subscenario

In the case of a subscenario that is reduced to a branch (circled subscenario for event n3 in Fig. 10), the bottom up process is illustrated by the activity diagram in Fig. 12. The general problem to be solved in this recursive process (core of the loop in Fig. 12) is, given an ISTNode and its incoming and outgoing Invocations (i.e., edges), to propagate the invocation condition in the outgoing invocation onto the invocation condition in the incoming invocation, using the node's postcondition. As illustrated in Fig. 11, we have to propagate invocation condition c2 (for invocation I2) onto invocation condition for invocation I1 using postcondition post. Informally,[18] this amounts to changing I1 invocation condition from c1 to c1 and f(post,c2), where f is the function, to be defined, that propagates c2 from I2 to I1 using post. The rationale is that, if c1 and a function of post and c2 (denoted f(post,c2)) is satisfied, then we can execute Op1 and Op2, i.e., c1 is true and after executing Op1, c2 is true, thus allowing the execution of Op2. The propagation thus consists in transforming a constraint on the after-state of operation Op1 (i.e., values after the execution of Op1), as stated in c2, into a constraint on

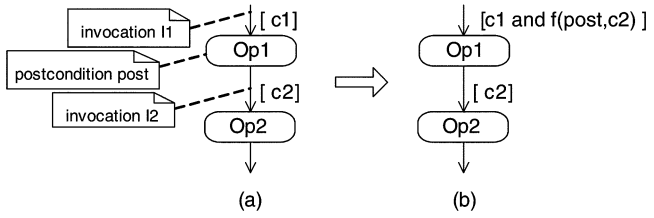


Fig. 11 Deriving test constraints (example 1)

the pre-state of operation Op1 (i.e., values before the execution of Op1), using postcondition post that provides the mapping between those after and pre-state. This is repeated (steps 3 to 5 in Fig. 12), starting from the bottom of the tree representing the subscenario, up until the tree top is reached.

For example, assume c2 equals to attribute>3 and that post equals to attribute=attribute@pre+1. Then, the propagation results in constraint attribute+1>3 that must be fulfilled in conjunction with c1. If the postcondition rather contains implies expressions (instead of a conjunction of atomic formulae), the propagation can lead to a *set* of constraints. In our example, condition c2 being the same, assume post equals to the following:

```
aCollection->size()=1 implies attribute=
  attribute@pre+1 and not(aCollection->size()
  =1) implies attribute=attribute@pre+2
```

Then, the constraint to be propagated and to be fulfilled in conjunction with c1 depends on the size of aCollection. If aCollection->size()=1 then the propagation results in aCollection->size()= 1 and attribute+1>3 whereas if not(aCollection-> size()=1) then the propagation results in not(aCollection->size()=1) and attribute+2>3. This illustrates that the result of the recursive process can be a set of constraints. Here, two different constraints can be fulfilled in order to execute the sequence of Op1 and Op2. In the general case, the constraints generated at a given step in the recursive process is the Cartesian product of the constraints already in the set of constraints and the implies-expressions in the postcondition.

[18] No invocation condition is changed in the IST, as suggested by the discussion. Separate constraints are generated during our IST analysis to save the results of the propagation. But this way of writing makes it easier to provide some intuition as to what is done in the recursive process.

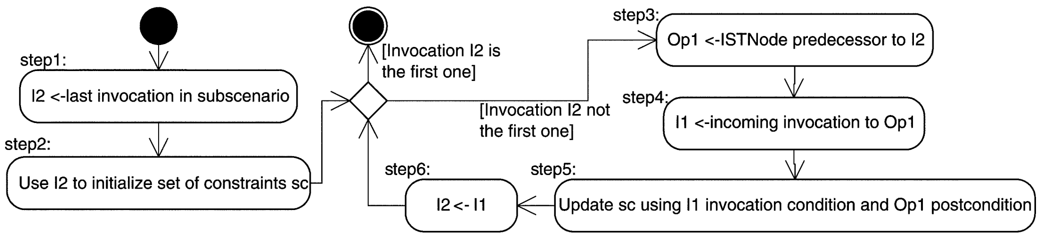

step1:
I2 <-last invocation in subscenario
step2:
Use I2 to initialize set of constraints sc
[Invocation I2 is the first one]
[Invocation I2 not the first one]
step3:
Op1 <-ISTNode predecessor to I2
step4:
I1 <-incoming invocation to Op1
step5:
Update sc using I1 invocation condition and Op1 postcondition
step6:
I2 <- I1

Fig. 12 Deriving test constraints, abstract algorithm

Note that if the invocation condition to be propagated using a postcondition is `true`, then the propagation results in `true`, as the postcondition has no impact on the execution of the second operation in the subscenario. In our example, if `c2=true`, satisfying `c1` is sufficient to execute both `Op1` and `Op2`, In other words, `f(post,true)=true`.

In the case of a subscenario that is a tree (e.g., subscenario for event `n2` in Fig. 10), i.e., it contains invocation sequences, the previous strategy can be used after a simple transformation of the tree: invocation sequences can be transformed into a path that specifies the exact same execution order: e.g., in the circled subscenario for event `n2` in Fig. 10, the transformation results in invocations between nodes `n2.2` and `n2.2.1` and between nodes `n2.2.1` and `n2.2.2`. The algorithm described above can then be applied without any adaptation.

Also, in the case of an orphan state-dependent object, since the behaviour does not depend on any other object, the invocation subtrees cannot contain invocation alternatives and the depth of the subtrees, measured as the number of nodes in paths of the tree is at most one (each subtree contains one subscenario). Then, another transformation of the IST allows us to derive a test constraint for the whole TTS, that is a constraint to be satisfied at the very beginning of the TTS, and not only for each event in the TTS.

These two similar IST transformations are very simple and are thus omitted in this article. The interested reader is referred to [28, 31] for more details.

4.5.2 Transforming OCL constraints

In order to detail how the propagation of an invocation condition, using a postcondition, is achieved (e.g., what function `f` is in Fig. 11), it is important to note that we make a number of assumptions on how each of the constraints that appear in a subscenario (i.e., postconditions and invocation conditions) is expressed in OCL. This is required in order to ease the derivation of test constraints. In Sect. 4.4, we present normalization steps that ensure these assumptions are met. More precisely, we assume a normalized invocation condition is in DNF and a normalized postcondition is in the Standard Equality Form (SEF). The SEF is a conjunction of equations[19] `l-value = r-value` where `l-value` is the value of a constrained element after the execution of the operation and `r-value` is an OCL expression possibly containing `@pre` values (i.e., values before the execution of the operation), arguments and literals. Alternatively, conjuncts in the SEF can contain `implies` expressions of the form `condition implies impliedPart`, in which case the conditions of the `implies` expressions are mutually exclusive and the implied parts are conjunctions of `l-value = r-value` equations.

Let us first consider the simple situation where the postcondition of operation `Op1` (Fig. 11), namely `post`, does not contain any `implies` expression. It is thus a conjunction of equations, as stated above. The propagation of invocation condition `I2`, onto `I1`, then replaces every occurrence in `I2`, of every `l-value` that appear in `post` with the corresponding `r-value` in which `@pre` postfixes are removed. This transformation of invocation condition `I2` then has to be fulfilled in conjunction with `c1` (invocation condition for `I1`). Reusing a previous example, assuming `c2` equals to `attribute>3` and `post` equals to `attribute=attribute@pre+1`, the propagation replaces `attribute` in `attribute>3` with `attribute@pre+1` after removing `@pre`, thus resulting in `attribute+1>3`.

If the postcondition contains `implies` expressions of the form `condition implies impliedPart`, the transformation is different. Different `implies` expressions state that the operation can be executed in different ways and then produces different results. Each `implies` expression describes one of these different executions: `condition` describes the condition under which the operation has to be executed in order to produce a result that satisfies `impliedPart`. Then, for each `implies` expression, `condition` must be satisfied in conjunction with the propagation result of `c2` and `impliedPart`. Reusing, once again, a previous example, assume `c2` equals to `attribute>3` and `post` equals to the following:

```
aCollection->size()=1 implies
  attribute=attribute@pre+1
and
not(aCollection->size()=1) implies
  attribute=attribute@pre+2
```

This results in the following set of two constraints (due to the two `implies` expressions):

```
{ aCollection->size()=1 and attribute+
  1>3, not(aCollection->size()=1) and
  attribute+2>3}
```

[19] The symbols `l-value` and `r-value` refer to the left-hand-side and the right-hand-side of a relational expression respectively. For instance, the `l-value` of a relation expression `a=b@pre+1` is `a` and the `r-value` is `b@pre+1`.

Note that in the two descriptions above we considered one invocation condition `c2` that needs to be propagated using the operation postcondition `post`, as we wanted the discussion to remain simple. However, as we have seen, during each execution of the loop in Fig. 12, we may have to consider a set of constraints (and not only one constraint), because of `implies` expressions in postconditions associated with node(s) below in the subscenario. We thus may need to apply the transformation above, where one constraint is propagated, on a set of constraints. This time, the invocation condition for invocation between nodes `Op1` and `Op2` is a set of conditions $\mathtt{cond}_1$, $\mathtt{cond}_2$, … $\mathtt{cond}_n$, and the propagation also results in a set of conditions `c1` and f(post, $\mathtt{cond}_1$), `c1` and f(post, $\mathtt{cond}_2$), … `c1` and f(post, $\mathtt{cond}_n$).

Last, note that this propagation of invocation conditions using postconditions may require inputs from the user. Indeed, we do not rely on a constraint solver for OCL expression when propagating constraints.[20] We only rely on the syntax of the OCL expressions (after normalization) and this may not be sufficient. For example, assume we have to propagate invocation condition `path->select(ocl_expression1)->size()>1` using postcondition `path->select(ocl_expression2)->append(obj)`. If `ocl_expression1` and `ocl_expression2` are syntactically identical, then we can derive (propagation) that `path->select(ocl_expression1)->size()>0` must be `true` before the execution of the operation. However, if using the syntax of `ocl_expression1` and `ocl_expression2` we cannot automatically identify that the same collection of elements is involved in the invocation condition and the postcondition, it is not possible to perform the propagation. In such a situation, the user must provide, under the form of an OCL expression, that part of the invocation condition and part of the postcondition are identical.

4.5.3 Example from the VSS case study

The constraint derivation algorithm presented above visits the nodes corresponding to the events in the TTS in order, from the first event node to the last (node 1 to node 4 in Fig. 7). For each trigger event node, the algorithm analyzes the corresponding subscenarios and derives the constraints that must be true for executing all the actions in each of them. The resulting constraints are then combined (conjunction) with the predicate condition of the event to form the complete test constraint for each invocation subscenario.

We show below, for each trigger event in the TTS, and each of the corresponding subscenario, the resulting constraint on the collective state. All these constraints must be fulfilled while executing the selected transition test sequence.

The first constraint specifies that the state of the copy must be either `ForRent` or `OnHold` for the first transition in the TTS to fire. Constraint number 7 additionally requires that the reservation associated with the copy is in state `Outstanding`. Constraints 2 and 3 (resp. 4 to 6) apply to event 2 (resp. 3) in the IST, i.e., event `return()` (resp. `holdExpire()`). These constraints specify that there must be one (constraints 3 and 5) or at least two (constraints 2 and 4) pending reservations for the copy's title for the event and the corresponding actions (in the subtree) to execute. Note that constraint number 6 is impossible (the second conjunct is the negation of the first one), as it corresponds to the situation where there is no pending reservation for the title, which contradicts the fact that the statechart for `Copy` stays in state `OnHold` on event `holdExpire()` in our sequence.

1. Event 1 has only one subscenario (made of node 1)
```
self.state=#ForRent or
self.state=#OnHold
```
2. Subscenario 1 for event 2 (nodes 2, 2.1, 2.1.1 and 2.1.2)
```
self.title.reservation->select
  (state=#Pending)->size()>0
and self.title.reservation->
  select(state=#Pending)->size()>1
```
3. Subscenario 2 for event 2 (nodes 2. and 2.2)
```
self.title.reservation->select
  (state=#Pending)->size()>0
and not self.title.reservation->
  select(state=#Pending)
->size()>1
```
4. Subscenario 1 for event 3 (nodes 3, 3.1, 3.1.1, 3.1.1.1 and 3.1.1.2)
```
self.title.reservation->select
  (state=#Pending)->size()>0
and self.title.reservation->
  select(state=#Pending)->size()>1
```
5. Subscenario 2 for event 3 (nodes 3, 3.1 and 3.1.2)
```
self.title.reservation->select
  (state=#Pending)->size()>0
and not self.title.reservation->
  select(state=#Pending)
->size()>1
```
6. Subscenario 3 for event 3 (nodes 3 and 3.2)
```
self.title.reservation->select
  (state=#Pending)->size()>0
and not self.title.reservation->
  select(state=#Pending)
->size()>0
```
7. Event 4 has only one subscneario (nodes 4 and 4.1)
```
(self.state=#ForRent or
self.state=#OnHold)
and self.reservation.state=
  #Outstanding
```

When executing the complete TTS, any of constraints 2 and 3 can be combined with any of constraints 4 and 5 for events `return()` and `holdExpire()`, respectively, resulting in four different executions of this specific TTS.

Let us illustrate the constraint derivation on one subscenario, i.e., the first subscenario for call event `return()`:

[20] We considered this would not be realistic considering the current state and applicability of this technology.

Nodes `2`, `2.1`, `2.1.1`, `2.1.2`. Note that in the description below, due to space constraints, only some of the normalization steps involved in the derivation are presented. Starting from the last invocation condition (`c2.1.2`) in the subscenario, the constraint derivation proceeds according to the following four steps:

Step 1: We propagate `c2.1.2` (which is `true`) using the postcondition of the operation in node `2.1.1`, thus producing `true` (the postcondition does not have any impact on the invocation condition). The condition to be satisfied for the execution of nodes `2.1.1` and `2.1.2` in sequence is then `c2.1.1 and true`, that is `c2.1.1`.

Step 2: We propagate `c2.1.1`, which is also `true` (`c2.1.1` and `c2.1.2` are equal) using the post condition of the operation in node `2.1`. The condition to be satisfied for the execution of nodes `2.1`, `2.1.1` and `2.1.2` is then `c2.1 and true`, that is `c2.1`.

Step 3: We propagate `c2.1`, that is:

```
self.title.reservation->select (v:
  Reservation|v.state=#Pending and not
  (v=self.title.oldestPending))->
  size()>0
```

which is normalized into:

```
self.title.reservation->select
  (v:Reservation|v.state=#Pending)->
select(v:Reservation|not(v=self.
  title.oldestPending))->
size()>0
```

The propagation then uses `return()` postcondition:

```
public Copy::return()
post:
( self.title.reservation@pre->select
  (v:Reservation|
v.state=#Pending)->size()>0
implies
self.state=#OnHold and
self.currentRental->size()=0
and Copy.allInstances->select(c:Copy|
  c.state=#OnHold)->size()=
Copy.allInstances@pre->select(c:Copy|
  c.state=#OnHold)->
size()+1
and Copy.allInstances->select(c:Copy|
  c.state=#Rented)->size()=
Copy.allInstances@pre->select(c:Copy|
  c.state=#Rented)->
size()-1
)
and
( not self.title.reservation@pre->
  select(v:Reservation|
v.state=#Pending)->size()>0
implies
self.state=#ForRent and
self.currentRental->isEmpty()
and Copy.allInstances->select(c:Copy|
  c.state=#ForRent)->size()=
Copy.allInstances@pre->select(c:Copy|
  c.state=#ForRent)->
size()+1
and Copy.allInstances->select(c:Copy|
  c.state=#Rented)->size()=
Copy.allInstances@pre->select(c:Copy|
  c.state=#Rented)->
size()-1
)
```

The constraint to be propagated (c2.1) constrains collection `self.title.reservation`. The propagating process therefore tries to match this collection with an `l-value` of a conjunct in the postcondition. Recall that the normalized postcondition is already in Standard Equality Form, and thus model elements that are modified by the operation appear as `l-values`. If an `l-value` starts with `self.title.reservation`, it means that the collection is modified by the operation. After checking the postcondition, no matches are found, i.e., `self.title.reservation` is not modified by operation `return()`. Therefore, the intermediate test constraint is propagated without changes (`f(post,c2.1)=c2.1`) and has to be combined (conjunction) with invocation condition `c2`, resulting in:

```
self.title.reservation->select
  (v:Reservation|v.state=#Pending)
    ->size()>0
and
self.title.reservation->select
  (v:Reservation|v.state=#Pending)
    ->select(v:Reservation|not(v=
  self.title.oldestPending))->
    size()>0
```

Step 4: Last, this OCL constraint is shown to the user. As an option, the user may then provide more information to the tool to simplify the resulting OCL expressions, in the same way relations between invocation conditions and postconditions during the constraint derivation process can be provided. In the above example, the user may provide the following equivalence:

```
self.title.reservation->select
  (v:Reservation|v.state=#Pending)
->select(v:Reservation|not(v=self.
  title.oldestPending))->
size()
= self.title.reservation->
select(v:Reservation|v.state=
  #Pending)
->size()-1
```

In other words, for a `Copy's Title`, the number of pending reservations, excluding the oldest pending one, is one less than the number of pending reservations. By combining this OCL expressions and the second conjunct (i.e., replacing an `l-value` with an `r-value`), the test constraint is transformed into:

```
self.title.reservation->select
  (v:Reservation|v.state=#Pending)
  ->size()>0
and
self.title.reservation->select
  (v:Reservation|v.state=#Pending)
  ->size()-1>0
```

We can observe that this latest constraint is easier to read and interpret. This is the condition that must be satisfied (e.g., by a test driver) in order to execute the transition between states `Rented` and `OnHold` in `Copy`'s statechart and execute the subscenario involving nodes 2, 2.1, 2.1.1, and 2.1.2 in the IST. By reading and interpreting it, we can deduce that there must be at least two pending reservations for the title. This can either be automated by a constraint solver, for example heuristics such as genetic algorithms [32], or deduced by the user from reading the test constraint.

4.6 Prototype tool

A prototype tool called Contract-based Constraint Derivation Tool (CBCDTool) was developed to show the feasibility of our approach towards automated support for deriving test constraints on test data from UML statecharts [28, 31]. CBCDTool takes as inputs information on (1) statecharts (including state invariants), and operations contracts, (2) the class diagram (operation signatures, attributes, associations), (3) the TTS, (4) equivalent navigation paths in the class diagram. As an output, the tool produces the constraints for the different transitions in the TTS.

The tool is implemented in Java and entails several packages (e.g., a package in charge of reading input data, including an XMI parser and an OCL parser). Two packages are of particular importance: the package including the UML metamodel (which has been slightly adapted from [1] for our purpose) and the package implementing the IST metamodel. More implementation details can be found in [28]. All the steps towards the derivation of constraints have been implemented except the normalization of logical expressions and postconditions (the current implementation assumes logical expressions and postconditions are already normalized). The reason is that there exist algorithms for the normalization of logical expressions and we have identified ways to automatically normalize postconditions (as explained in Sect. 4.4.4). Automating these two steps thus does not raise major problems, and we wanted to focus on the harder problem of the constraint derivation.

5 Conclusions

This article explores the automated support for deriving test requirements from UML statecharts given coverage criteria such as the all transitions, all transition pairs, full predicate, and all round-trip paths [3, 4]. These criteria all assume a test case to be in the form of a feasible sequence of transitions (Transition Test Sequence or TTS). The test requirements take the form of logical constraints and the test data derivation problem can be divided into two subproblems. The first subproblem is to derive test constraints on system states and arguments of events and actions for a given TTS. The second subproblem is then to generate actual test data values that satisfy these test constraints.

This article addresses the first subproblem as this is likely the most complex one, since various constraint-solving techniques can be readily adapted [32, 33]. It clarifies relevant issues and proposes a methodology to automate the derivation of test constraints for a given TTS. The methodology defines an appropriate model representation, the Invocation Sequence Tree (IST), to capture all possible sequences of actions triggered by a TTS. Information in the statecharts and class diagrams are extracted and modelled into ISTs, which are more amenable to the definition of algorithms. The methodology also identifies four types of normalizations for OCL expressions to support the analysis of model constraints written in OCL and the generation of test constraints. Note that these normalization rules can be also considered as guidelines for software engineers to write preconditions, postconditions, and invariants in OCL. Moreover, the methodology provides a number of algorithms that specify a sequence of precise steps to automatically generate test constraints. The algorithms involve (1) creating the IST from the model information, (2) performing normalization of OCL expressions, and (3) deriving test constraints on the system state and event/action arguments based on the analysis of invocation conditions and operation contracts for events and actions. A prototype tool is also implemented as a proof-of-concept. The tool is assessed through two case studies [28]—one of them is presented here—which provide an initial demonstration of the feasibility of automating the methodology.

The normalization of postconditions requires that modellers follow certain patterns while modeling operation contracts, which may restrict the applicability of our methodology, as they may not be followed in practice. However, we believe that these contract patterns correspond to good practice and should be encouraged.

Presently our methodology can identify redundancy or inconsistency between two OCL constraints only when they have exactly the same expression (eg. `a>b` and `a>b`) or one expression is the direct negation of the other (e.g., `a>b` and `not(a>b)`). Other types of redundancy (e.g., `constraintA implies constraintB`, hence `constraintB` is redundant in expression `constraintA and constraintB`) or inconsistency (e.g., `constraintA implies constraintB`, hence `constraintA` and `(not constraintB)` are inconsistent) cannot be automatically tackled and have to be identified manually. Automating this aspect would require an OCL theorem prover and even regular theorem provers would not be easy to use in practice. However, we also expect normalization to facilitate the manual analysis of constraints. In the worst case, if redundancies exist, this will simply slow down constraint solving algorithms. If, on the other hand, inconsistencies are present, that is the test constraint cannot be satisfied, then the constraint solving algorithm will not converge towards a result.

We also had to assume that the statecharts used for testing purposes are deterministic. This not only implies they are complete and well formed, but that concurrently executing statecharts do not trigger race conditions at run time. Such problems cannot be detected by state-based testing and must be addressed beforehand.

Additional case studies should of course be performed to evaluate our methodology. More specifically, empirical work can be carried out to investigate how people use OCL and to find a set of common modelling styles that are widely accepted. This could help in defining less restrictive modelling patterns so that our approach can be more widely applicable.

Recall that the test case derivation problem is divided into two subproblems. This article addresses the first problem, namely, the derivation of the test constraints. Such test constraints can then be used to derive actual test data either by providing guidance to the tester or by using constraint-solving techniques. Recall that our Video Store System (VSS) case study shows that infeasible test constraints might result during the test constraints derivation process. It is hoped that such constraints can be identified during this stage.

The second subproblem belongs to the realm of Constraint Satisfiability Problem (SAT) [32, 33] that requires the use of the constraint-solving techniques. There are a number of such techniques available, to name a few among the most popular ones: Linear Programming, Hill Climbing, Simulated Annealing, and Genetic Algorithms. These techniques all have their strengths and weaknesses. So there is no single best-for-all technique for an arbitrary SAT problem [33]. Which technique to choose really depends on the nature of the individual type of problem. Moreover, in order to apply a constraint-solving technique, our test constraints need to be transformed into the representations required by the technique. Therefore, further work needs to investigate all possible forms of test constraints as well as in what representations these forms of constraints can be encoded.

Last, alternatives to the use of OCL for specifying operations' behaviour is worth investigating. Action semantics is now part of the UML standard [1, 18, 34]. Where OCL specifies an operation's behaviour under the form of constraints (pre- and post-conditions), action semantics defines the semantics of a complete set of actions at a high level of abstraction (e.g., manipulating a collection of objects). However, action semantics does not specify any notation, and there already exist several action semantics conforming languages [34] that have not yet been standardized (as opposed to the OCL).

Acknowledgements This work was partly supported by a Canada Research Chair (CRC) grant. Lionel Briand and Yvan Labiche were further supported by NSERC operational grants. This work is part of a larger project on testing object-oriented systems with the UML (www.sce.carleton.ca/Squall). We are grateful to the editors and reviewers of this article for their thorough comments which helped improve the article in many ways.

References

1. OMG.: UML 1.4 Specification. Object Management Group, Complete Specification formal/01-09-67 (2001)
2. Bruegge, B., Dutoit, A.H.: Object-Oriented Software Engineering Using UML, Patterns, and Java, 2nd edn. Prentice-Hall, Englewood Cliffs, NJ (2004)
3. Binder, R.V.: Testing Object-Oriented Systems—Models, Patterns, and Tools. Object Technology. Addison-Wesley, Reading, MA (1999)
4. Offutt, A.J., Abdurazik, A.: Generating Tests from UML specifications. In: Proceedings of the 2nd International Conference on the Unified Modeling Language (UML'99), Fort Collins, CO, pp. 416–429 (1999)
5. Beizer, B.: Software Testing Techniques, 2nd edn. Van Nostrand Reinhold, NY (1990)
6. OMG.: UML 1.4 chapter 6—OCL Specification. Object Management Group, Complete Specification formal/01-09-67 (2001)
7. Warmer, J., Kleppe, A.: The Object Constraint Language. Addison-Wesley, Reading, MA. Errata at http://www.klasse.nl/english/boeken/errata.html (1999)
8. Meyer, B.: Object-Oriented Software Construction—2nd edn. Prentice-Hall, Englewood Cliffs, NJ (1997)
9. Mitchell, R., McKim, J.: Design by Contract, by Example. Addison-Wesley, Reading, MA (2001)
10. Chevalley, P., Thévenod-Fosse, P.: Automated Generation of Statistical Test Cases from UML State Diagrams. In: Proceedings of the International Computer Software and Applications Conference, Chicago, IL, pp. 205–214 (2001)
11. Chow, T.S.: Testing software design modeled by finite-state machines. IEEE Trans. Softw. Eng. **SE-4**(3), 178–187 (1978)
12. Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines—a survey. Proc. IEEE **84**(8), 1090–1123 (1996)
13. Bogdanov, K., Holcombe, M.: Statechart testing method for aircraft control Systems. Softw. Test. Verification Reliability **11**(1), 39–54, (2001)
14. Li, L., Qi, Z.: Test selection from UML statecharts. In: Proceedings of Technology of Object-Oriented Languages and Systems, pp. 273–279 (1999)
15. Hong, H.S., Kim, Y.G., Cha, S.D., Bae, D.H., Ural, H.: A test sequence selection method for statecharts. Softw. Testing Verification Reliability **10**(4), 203–227 (2000)
16. Kim, Y.G., Hong, H.S., Cho, S.M., Bae, D.H., Cha, S.D.: Test case generation from UML state diagrams. Proc. IEE Softw. **146**(4), 187–192 (1999)
17. OMG.: OCL 2.0 Specification. Object Management Group, Final Adopted Specification ptc/03-10-14 (2003)
18. OMG.: UML 2.0 Superstructure Specification. Object Management Group, Final Adopted Specification ptc/03-08-02 (2003)
19. Tanenbaum, A.S., Modern Operating Systems, 2nd edn. Prentice-Hall, Englewood Cliffs, NJ (2001)
20. Douglass, B.P.: Real Time UML, 3rd edn. Addison-Wesley, Reading, MA (2004)
21. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: A dynamic data race detector for multithreaded programs. ACM Trans. Comput. Syst. **15**(4), 391–411 (1997)
22. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide. Addison Wesley, Reading, MA (1999)

23. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual. Addison Wesley, Reading, MA (1999)
24. Huth, M.R.A., Ryan, M.D.: Logic in Computer Science, Modelling and Reasoning about Systems. Cambridge University Press, Oxford, UK (2000)
25. Finger, F.: Design and Implementation of a Modular OCL Compiler. Master Thesis, Dresden University of Technology, Dresden, Germany (2000)
26. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns—Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA (1995)
27. Briand, L.C., Labiche, Y., Yan, H.-D., Di Penta, M.: A controlled experiment on the impact of the object constraint language in UML-based development. In: Proceedings of the IEEE International Conference on Software Maintenance, Chicago, pp. 380–389 (2004)
28. Briand, L.C., Cui, J., Labiche, Y.: Towards Automated Support for Deriving Test Data from UML Statecharts. Carleton University, Ottawa, Canada, Technical Report SCE-03-13, http://www.sce.carleton.ca/Squall. (2003)
29. Mitchell, R.: Analysis by contract—Video store case study. University of Brighton, Brighton, UK, Technical Report, www.it.brighton.ac.uk/staff/rjm4 (1999)
30. Offutt, A.J., Xiong, Y., Liu, S.: Criteria for Generating Specification-Based Tests. In: Proceedings of the 5th International Conference on Engineering of Complex Computer Systems (ICECCS), Las Vegas, NV, pp. 119–129 (1999)
31. Cui, J.: Towards Automated Support for Deriving Test Data from UML Statecharts. Master Thesis, Carleton University, Ottawa, Canada, Systems and Computer Enginerring (2004)
32. Chambers, L.: Practical Handbook of Genetic Algorithms, vol. 1. CRC, Boca Raton, FL (1995).
33. Michalewicz, Z.: How to Solve it: Modern Heuristics. Springer-Verlag, Berlin Heidelberg New York (1999)
34. Mellor, S.J., Balcer, M.J.: Executable UML: A Foundation for Model-Driven Architecture. Addison-Wesley, Reading, MA (2002)

Lionel C. Briand is on the faculty of the Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, where he founded and leads the Software Quality Engineering Laboratory (http://www.sce.carleton.ca/Squall/Squall.htm). He has been granted the Canada Research Chair in Software Quality Engineering and is also a visiting professor at the Simula laboratories, University of Oslo, Norway. Before that he was the software quality engineering department head at the Fraunhofer Institute for Experimental Software Engineering, Germany.

Dr. Lionel also worked as a research scientist for the Software Engineering Laboratory, a consortium of the NASA Goddard Space Flight Center, CSC, and the University of Maryland. He has been on the program, steering, or organization committees of many international, IEEE conferences such as ICSE, ICSM, ISSRE, and METRICS. He is the coeditor-in-chief of Empirical Software Engineering (Springer) and is a member of the editorial board of Systems and Software Modeling (Springer). He was on the board of IEEE Transactions on Software Engineering from 2000 to 2004.

His research interests include: object-oriented analysis and design, inspections and testing in the context of object-oriented development, quality assurance and control, project planning and risk analysis, and technology evaluation. Lionel received the BSc and MSc degrees in geophysics and computer systems engineering from the University of Paris VI, France. He received the PhD degree in computer science, with high honors, from the University of Paris XI, France.

Yvan Labiche received the BSc in Computer System Engineering, from the graduate school of engineering: CUST (Centre Universitaire des Science et Techniques, Clermont-Ferrand), France. He completed a Master of fundamental computer science and production systems in 1995 (Université Blaise Pascal, Clermont Ferrand, France). While doing his Ph.D. in Software Engineering, completed in 2000 at LAAS/CNRS in Toulouse, France, Yvan worked with Aerospatiale Matra Airbus (now EADS Airbus) on the definition of testing strategies for safety-critical, on-board software, developed using object-oriented technologies.
In January 2001, Dr. Yvan Labiche joined the Department of Systems and Computer Engineering at Carleton University, as an Assistant Professor. His research interests include: object-oriented analysis and design, software testing in the context of object-oriented development, and technology evaluation. He is a member of the IEEE.

Jim (Jingfeng) Cui completed his BSc in Industrial Automation Control, from the School of Information and Engineering, Northeastern University, China. He received a Master of Applied Science (specialization in Software Engineering) in 2004 from the Ottawa-Carleton Institute of Electrical and Computer Engineering, Ottawa, Canada. While in his graduate study, he was awarded the Ontario Graduate Scholarship of Science and Technology. He is now a senior Software Architect in Sunyard System & Engineering Co.Ltd., China. His interest includes Object-Oriented Software Development, Quality Assurance, and Content Management System.