Towards Automated Support for Deriving Test Data from UML Statecharts

By

Jingfeng (Jim) Cui

A thesis submitted to

The Faculty of Graduate Studies and Research

In partial fulfilment of the requirements for the degree of

Master of Applied Science

Ottawa-Carleton Institute of Electrical and Computer Engineering

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario, Canada

January 2004

National Library
of Canada

Bibliothèque nationale
du Canada

Acquisitions and
Bibliographic Services

Acquisitions et
services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 0-612-89829-6

Our file Notre référence

ISBN: 0-612-89829-6

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this dissertation.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de ce manuscrit.

While these forms may be included in the document page count, their removal does not represent any loss of content from the dissertation.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

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 data, i.e., 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 two case studies that exemplify the steps and provide an initial validation. Though many of the steps are automated, some inputs are still required from the modeller and tester and further work is necessary to push the automation even further and solve a number of remaining issues. This thesis is a first attempt at clarifying the issues with the objective to support as much of the UML statechart notation as possible and to identify the analysis steps required in such an endeavour.

Acknowledgements

I would like to thank my thesis supervisors, Dr. Lionel Briand and Dr. Yvan Labiche, for their understanding, guidance, help, and feedback.

I would also like to thank my family and friends, especially my wife, for their tolerance, encouragement and continued support.

Table of Contents

Chapter 1 – Introduction	1
1.1 Motivation.....	1
1.2 Related Works.....	3
1.3 Objective	4
1.4 Thesis Contributions	5
1.5 Thesis Organization	6
Chapter 2 – Background Information	7
2.1 Specification-based Software Testing.....	7
2.2 UML and OCL.....	8
2.3 Analysis Model vs. Design Model.....	10
2.4 The Testability of UML Statechart	11
2.4.1 Statechart-related Definitions	12
2.4.2 Statechart Flattening	13
2.4.3 Non-determinism in Statechart	14
2.4.4 Assumptions about Event, Action and Activity.....	15
2.4.4.1 Event	15
2.4.4.2 Action.....	16
2.4.4.3 Activity	20
2.5 Design by Contract	22
2.5.1 Requirements on Postconditions.....	23
2.5.2 The Use of @pre Postfix.....	25
2.5.3 Change Specifications vs. Frame Rules.....	28
Chapter 3 – Methodology	30
3.1 Transition Test Sequence.....	30
3.2 Invocation Sequence Tree.....	31
3.2.1 Definitions.....	32
3.2.2 Metamodel of IST	38
3.2.3 Building the IST.....	39

3.2.3.1	Algorithms	39
3.2.3.2	Issues.....	43
3.2.3.3	Examples.....	45
3.3	Test Constraints Derivation	52
3.3.1	Types of State-dependent Objects	53
3.3.2	Deriving Test Constraints for Associated State-dependent Objects	54
3.3.3	Deriving Test Constraints for Orphan State-dependent Objects.....	62
3.4	Normalization of OCL Expressions.....	64
3.4.1	Eliminating Local Variables, Constants and Queries	67
3.4.2	Normalizing Logical Expressions.....	72
3.4.3	Normalizing Navigation Path of Constrained Elements.....	78
3.4.4	Normalizing Operations on OCL Types	84
3.4.4.1	Normalizing Operations on Basic Types	87
3.4.4.2	Normalizing Operations on Collection Types	89
3.4.5	Normalizing Postconditions.....	92
3.4.5.1	The Standard Equality Form.....	93
3.4.5.2	Complementing Postconditions	96
3.4.6	Advantages and Limitations	98
Chapter 4 –	Prototype Tool.....	101
4.1	Overview.....	101
4.2	Requirements of CBCDTool.....	102
4.3	Subsystem Decomposition.....	104
4.3.1	control Subsystem	104
4.3.2	umlModel Subsystem.....	106
4.3.2.1	foundation Subsystem	107
4.3.2.1.1	core Subsystem.....	108
4.3.2.1.2	dataTypes Subsystem.....	110
4.3.2.2	behavioralElements Subsystem.....	111
4.3.2.2.1	commonBehavior Subsystem.....	111
4.3.2.2.2	stateMachines Subsystem	113
4.3.3	ist Subsystem.....	113

4.3.4	ocl Subsystem	115
Chapter 5	– Case Studies	117
5.1	Case Study 1: VSS	117
5.2	Case Study 2: OrdSet	124
Chapter 6	– Conclusions	130
6.1	Thesis Limitations.....	131
6.1.1	Practical Limitations	131
6.1.2	Methodology Limitations.....	132
6.1.3	Implementation Limitations.....	132
6.2	Future Work	133
Appendix A	– Algorithms of CBCDTool.....	139
Appendix B	– Contract Patterns and Corresponding Complementing Rules	159

List of Figures

Figure 1. Execution order of event and actions	19
Figure 2. Elimination of entry and exit actions in a state	21
Figure 3. Postcondition of actions included in event handler.....	25
Figure 4. An awkward design of a statechart.....	32
Figure 5. An example IST.....	34
Figure 6. Illustration of an invocation.....	35
Figure 7. Alternative relation vs. sequential relation.....	36
Figure 8. Computing of the number of subscenarios.....	37
Figure 9. Metamodel of IST.....	38
Figure 10. Building IST (buildIST), overall abstract algorithm	41
Figure 11. Building subtree (buildSubTree), abstract algorithm	44
Figure 12. Statecharts and class diagram of the building IST example.....	46
Figure 13. IST for a given TTS (example 1)	47
Figure 14. IST for a given TTS (example 2)	48
Figure 15. Associated class and orphan class	53
Figure 16. Deriving test constraints (example 1).....	57
Figure 17. Deriving test constraints, abstract algorithm.....	58
Figure 18. Deriving test constraints (example 2).....	59
Figure 19. IST for an orphan state-dependent object, an example	63
Figure 20. Invocation sequence tree and constraint derivation (illustration).....	93
Figure 21. Overview of CBCDTool.....	103
Figure 22. Execution steps of CBCDTool	103
Figure 23. CBCDTool subsystem decomposition	105
Figure 24. control subsystem.....	105
Figure 25. umlModel subsystem	107
Figure 26. foundation subsystem	107
Figure 27. core subsystem class diagram.....	108
Figure 28. dataType subsystem	111

Figure 29. behavioralElements subsystem	111
Figure 30. commonBehavior subsystem.....	112
Figure 31. stateMachines subsystem	114
Figure 32. ist subsystem	114
Figure 33. ocl subsystem	115
Figure 34. Class diagram of entity classes in the VSS case study.....	119
Figure 35. Statechart of a Copy object in the VSS case study.....	119
Figure 36. Statechart of a Reservation object in the VSS case study.....	120
Figure 37. The IST of VSS	122
Figure 38. Class diagram in the OrdSet case study	125
Figure 39. Statechart of an OrdSet object in the OrdSet case study.....	126
Figure 40. The IST of OrdSet	128

List of Tables

Table 1. Existing works	4
Table 2. Truth table of an if-expression and its equivalent implies-expressions.....	74
Table 3. Truth table of two equivalent implies-expressions	75
Table 4. Examples of clauses in (or not in) the Standard Equality Form	94
Table 5. Enumeration values of UML data type classes.....	110

Chapter 1 – Introduction

1.1 Motivation

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) [UML01], 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 (most methodologies promote its use at that stage, e.g. [BD00]) to better specify object behavior, as well as during testing¹, where the implementation is tested (verified) against the specification (i.e., the statechart).

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 all possible behavior specified by the statechart) is often impossible. A number of papers [Binder99, Hong99, Offutt99] have proposed test strategies for UML statechart, based on several coverage criteria such as the all

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 the testing activity belongs to the realm of unit testing. If the behavior of a class-cluster, a subsystem or a component is modeled, the testing activity belongs to integration testing. If the whole system is modeled, then the testing activity belong to system testing.

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 feasible sequences of transitions. The generation of a set of transition sequences for a given coverage criterion can usually be automated [VT98, TSPM98, Hong99, Offutt99, Offutt03] so that the tester does not have to go through the tedious procedure of deriving paths manually to comply with the coverage criterion. Note however that the problem of checking whether a transition sequence is feasible is not addressed by these authors. 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 [Beizer90]. Solving the problem amounts to finding a solution to a set of Boolean predicate extracted from the control flow path. In our context, statechart-based testing, those predicates include guard conditions in transitions. This problem is further complicated in our context 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. Though this thesis focuses on the former subproblem, it also gives guidance on possible solutions to the latter.

1.2 Related Works

Existing works [VT98, TSPM98, Binder99, Hong99, Offutt99, Offutt03] discuss how to automatically generate test cases (transition sequences and in some cases input data) so that certain coverage criteria are achieved. However, the proposed approaches have the following limitations summarized in Table 1. First, the UML statechart notation is not fully supported. State diagrams used in [VT98, TSPM98, Offutt03] are not UML statecharts, though they correspond to UML statecharts with very simple features. For instance, state diagrams in these works can be regarded as statecharts that involve `call` or `signal` events, with guards but without actions (See Section 2.4.1 for definitions). Among the remaining three works, event type in [Hong99] is not specified whereas only change events are accounted in [Offutt99]. Although the UML defines seven different types of actions, actions are not taken into account except in [Offutt99, Hong99, Binder99], where actions are either assignment (to attributes), return, or call action. Last, guards are not considered in [Offutt99]. These are important limitations as they limit the applicability of the different approaches.

Second, since none of these works adopt the Design by Contract principle [Meyer97] in defining operations, only simple modifications are supported by events and actions, namely, modifications that are simple enough to be captured in the form of assignments to attributes. More importantly, existing works do not suit object-oriented systems well. For example, associations among classes cannot be represented in existing works and this might explain why existing works considered statechart in isolation

without accounting for the interactions among object statecharts. However, in order to test the combined behaviors of state-dependent objects, we need to take into account the interactions among object statecharts.

Existing Works	Statechart notation	Consider Guard	Event Types	Action Types	Consider Parameter	Use Contract	Consider Object Interaction
[VT98]	No	Yes	N/A	N/A	Yes	No	No
[TSPM98]	No	No	N/A	N/A	No	No	No
[Binder99]	Yes	Yes	call	call return	Yes	No	No
[Hong99]	Yes	Yes	Not specified	assignment	No	No	No
[Offutt99]	Yes	No	change	call	No	No	No
[Offutt03]	No	Yes	N/A	N/A	No	No	No

Table 1. Existing works

1.3 Objective

Our objective in this thesis can be defined as follows.

Given (1) a class to be tested (and classes reachable from the class through a sequence of associations, if any), (2) its statechart (and a set of interacting statecharts belonging to some of the associated classes, if any), and (3) a specific sequence of transitions to be tested, our objective is to determine *test constraints* (i) on the state of the system at different points in the transition 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.

In the remainder of this thesis, we use “model constraints” to refer to preconditions, postconditions or guard conditions, which are used in defining UML models, and use “test constraints” to refer to constraints that are derived for a specific

transition sequence from model constraints. When a discussion applies to both types of constraints, we simply use the term “constraints”.

The above objective is to be achieved in the context of the UML (statechart) and our goal is to account for most of the statechart features: Transitions can be triggered by any type of events, including events that can have parameters, such as call events; Transitions can have actions; and states can have entry and exit actions; The effects of event and actions on the system states are accounted for and they are captured by *operation contracts* (preconditions and postconditions); The guards and operation contracts may contain any form of OCL constraint [WK99], possibly involving predefined basic types, collection types, and user-defined model types. The interactions among statecharts are taken into account.

1.4 Thesis Contributions

This work is a first attempt at clarifying issues relevant to deriving test data for transition sequences from a UML statechart, with the objective to support as much of the UML statechart notation as possible. We propose a methodology to automate the first step towards automating test data generation for transition sequences, that is, the derivation of constraints to be satisfied in order for a transition sequence to execute. This first step requires that a set of normalization rules be defined to facilitate a number of analyses of OCL expressions so as to permit an efficient derivation of test constraints for transition sequences. We also provide guidelines on how to write good contracts in OCL to facilitate the derivation of test constraints. A prototype tool has been implemented to demonstrate the feasibility of automating our test constraint derivation process, and used on two case studies of non-trivial complexity.

1.5 Thesis Organization

Chapter 2 provides background knowledge on Design by Contract in a UML/OCL context and testability issues related to UML statecharts. Chapter 3 describes the methodology we employed to derive test constraints. Chapter 4 describes the prototype tool that implements the methodology. Chapter 5 reports the result of two case studies. Chapter 6 then concludes by summarizing the result and outlining future work.

Chapter 2 – Background Information

We first provide necessary background information about this research, clarify relevant issues, and state the assumptions we make in this research.

2.1 Specification-based Software Testing

This research belongs to the realm of specification-based testing as our main concern is to derive test constraints from UML statechart specifications. Specification-based testing is also called model-based testing or functional testing [MS01]. It is considered as black-box, as opposed to structural testing which is considered as white-box. In a specification-based testing, the test cases are derived solely from the software specification [MS01]. Deriving test data from specifications has several advantages [Offutt03]: (1) Testing activity can be shifted to early stage of software development process, making more effective planning and resource utilization possible; (2) It helps to detect faults in specifications such as inconsistent requirements before the specifications are implemented; (3) It is implementation-independent and thus is applicable to any implementation; (4) It supports system-level testing; (5) It has the potential to support the automated generation of test cases.

The specification-based test case generation requires that the specification itself must in the first place be correct. However, it would be very hard for a computerized program to detect specification errors. Firstly, missing functionalities can only be identified through the interactions among developers, clients and users. Secondly, to detect inconsistencies, the specification needs to be analyzed and converted to model

constraints, which are then checked to see if conflicts exist. However, the problem of solving arbitrary constraints is known to be intractable [Michale00]. Therefore, in this research, we assume that complete, correct, and consistent specifications are given. Moreover, in order to support the automated generation of test data, the specification has to be sufficiently formal. In this thesis, UML statecharts and Object Constraint Language (OCL) are employed for model specification.

2.2 UML and OCL

This section is a brief introduction to UML and OCL. For complete references, see [UML01] and [WK99].

UML is a semi-formal modeling language used for specifying, visualizing, constructing, and documenting object-oriented systems [RJB99]. It is the de facto industrial standard developed, maintained, and managed by the Object Management Group (OMG). UML is not a programming language. We cannot write a program in UML. Though some CASE tools provide code generation capabilities from UML models, the generated code is nothing more than a framework. Developers still need to write code to implement the methods.

OCL is the UML standard for specifying invariants, preconditions, postconditions, and other kinds of model constraints [WK99]. A constraint is defined as “a restriction on one or more values of (part of) an object-oriented model or system”. Ultimately, the model constraints need to be expressed in some language so that they can be verified. There are other alternatives for writing model constraints such as natural languages, implementation languages and various formal languages. However, natural languages are inherently ambiguous, implementation languages may not be suitably

expressive, and formal languages, such as Z, require strong mathematical background, and thus are difficult for regular (untrained) users. Another serious limitation of formal languages is that they are not applicable in a UML context. For instance, the class diagram notation is not supported. In comparison, OCL is an integral part of UML and it has been designed as a constraint language that is formal, while still easy to learn, and straightforward to construct and interpret [WK99]. The benefits of using OCL are manifolds. OCL helps to describe the model or system in an unambiguous way so that developers can communicate without misunderstanding; OCL can be used to specify operation contracts, class invariants and other types of constraints in a formal way, thus potentially enable the automated derivation of test data based on software specifications.

OCL is a declarative language that has no side effect [WK99]. That means the evaluation of OCL expressions does not change the value of any model element. Instead, an OCL expression returns a value based on the state of a model element or system. Consequently, unlike languages such as Java and C++, OCL does not provide any notion of control. In another word, OCL specifies *what* rather than *how*.

OCL is a typed language, therefore each OCL expressions has a type [WK99], which defines operations that can be applied on the type. Section 3.4.4 summarizes all the operations on each predefined OCL type.

OCL can be used for a number of purposes. In this thesis, we are mainly concerned with using OCL in the following context.

- To specify class invariants
- To specify preconditions and postconditions of operations
- To specify guards on statechart transitions

- To specify navigations

Class invariants, Operation preconditions and postconditions will be discussed in more details in Section 2.5.

Making use of OCL to support various software engineering activities has been an active research area since it was formally defined by OMG in Object Constraint Language Specification in 1997. Currently, OCL is mostly restricted to academic circles. There are several reasons for this. First, OCL is a relatively new language and therefore it is still undergoing changes and refinements. Second, any new language needs time and effort to learn. OCL is not an exception. Third, there is scarce tool support to demonstrate the power of OCL. This research is part of the effort to provide automated tool support that applies OCL in software quality assurance. We expect that OCL will be widely accepted and extensively applied in the software industry as increasing tool support becomes available.

2.3 Analysis Model vs. Design Model

Analysis and design are two essential activities in the software development life cycle. Analysis takes the requirement elicited from clients and users as input and generates the analysis model, which is supposed to be correct, complete, consistent, and verifiable [BD00]. The analysis model is composed of three individual models: the functional model represented by use cases, the analysis object model represented by class diagrams, and the dynamic model represented by statecharts and sequence or collaboration diagrams [BD00]. Many issues are still unsolved in the analysis stage. These include: subsystem decomposition if the system is complex, hardware/software platform, persistent data management, access control policy, the global control flow, etc

[BD00]. A system design resolves these issues. Then an object design is required to precisely define the classes and subsystem interfaces, to restructure the class model for extensibility, clarity and performance [BD00]. The result is the design model. The implementation then translates the design model into program source code.

A good practice is to derive test cases early in the software life cycle. In an Object-Oriented modeling context, we begin to derive test cases during the Analysis stage and then refine them during the Design and Implementation stages. The advantage of the Analysis model is that it focuses on the functional requirements of the system and thus is a high-level system specification. Accordingly, test cases derived from the Analysis model can be applied to different implementations and are suitable for regression testing. The strength of the Design model is that it is very close to the actual implementation and therefore most of the test cases derived from the Design model can be applied to the implementation directly. In comparison, test cases derived from Analysis models require extra effort to map them to a form that suits the actual implementation. The methodology we introduce in this research is applicable to both Analysis and Design models.

2.4 The Testability of UML Statechart

According to [UML01], "State machines can be used to specify behavior of various elements that are being modeled". Statechart diagrams are the diagrammatic representation of state machines. Statecharts 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.). From now on, this thesis will assume class statecharts only.

Nevertheless, conclusions and results obtained from class statecharts can easily be generalized to include other types of statecharts.

In the following, we first present the statechart-related definitions. Then we discuss the testability issue of the statechart.

2.4.1 Statechart-related Definitions

A state machine consists of a number of *transitions* and *states*.

A transition consists of an *event*, a *guard condition* (abbr. *guard*), and an *action* sequence. The following definitions are taken from [RJB99]. A transition specifies a directed relation between two states that when the specified event occurs and the specified guard is satisfied, the object will leave the first state, perform specified actions, and enter the second state. An event models external stimulus (input) to the state machine; a guard is a Boolean expression evaluated when an event occurs; an action models the response (output) of the state machine. A special type of action is the activity, which is an ongoing task that an object executes while it stays in a specific state. Testability-related issue of events, actions and activities will be discussed in Section 2.4.4.

Before giving the definition of states, several relevant definitions are introduced first. We refer to the *owning class* as the class that owns the state machine based on which the test case is specified. An *owning object* is an instance of the owning class. The *associated classes* are classes to which the owning class can navigate through associations, 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. Both an individual object and an object cluster can have states.

We can view an object state according to two levels of abstractions. At the lower abstraction level, the *concrete state* is defined as the combination of the object's attribute values plus its links to other objects. We then refer to the *collective state* as a set of the concrete states of all the objects in the object cluster. It should be noted that the collective state is a subset of the system state, which consists of the concrete states of all the objects in the software system. The concrete view of a state, which is sometimes referred to as the *primitive state* [Binder99], in most cases is too granular for statechart modeling [Binder99] (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 a so-called *abstract state* [Binder99]. Typically, state invariants associated with states in a statechart describe abstract states, whereas guards test the collective state and operation contracts model changes to the collective state. In the remaining part of the thesis, we will use “state” to denote the abstract states in statecharts and use “collective state” otherwise. The state definitions at both abstraction levels are unambiguous, executable, and thus testable.

Next we discuss the testability of UML statecharts.

2.4.2 Statechart Flattening

It is common practice to model the complex behavior of an object using composite state and concurrent substates. The use of such mechanisms helps to cope with complexity but makes it hard to generate test cases. It is not obvious how many distinct states and transitions are there in the statechart. In order to apply the state-based criteria mentioned in the introduction, it is necessary to remove all hierarchy and concurrency in the statecharts and obtain *flat statecharts*, in which every distinct state is represented by a

node and all possible transitions are shown explicitly [Binder99]. The transformation from hierarchical and concurrent statecharts into flat statecharts is explained in [Hong99] and in this thesis we assume there are such algorithms to flatten user-supplied statecharts. Though flattened statecharts might be highly complex, they are not visualized and are only used as internal representations by our algorithms. It should be noted that flattening a statechart may impact those OCL constraints that refer to composite states in the statechart because the composite 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. This type of transformation is straightforward and is not the purpose of this thesis. For example, a composite state invariant involved in a constraint can be replaced by the disjunction of all its substates invariants.

2.4.3 Non-determinism in Statechart

The semantics of UML statecharts allow for the possibility of non-determinism in state transitions: Multiple transitions may be *enabled* within a state machine [UML01]. For example, when two transitions originating from the same state are enabled by the same event, then only one of them will *fire*. If firing priorities are not specified, the selection of which transition to fire is non-deterministic. This research does not handle such cases as we assume the statecharts deterministic.

Non-determinism may also result from multiple state machines executing concurrently in the system. In this case, the state machines are owned by active objects, each has its own thread of control and is able to modify system state independently. If a state transition of one active object has a guard that depends on the system state and another active object can modify the system state by itself, then the evaluation of the

guard is not decidable. This type of non-determinism is not considered in this research either.

2.4.4 Assumptions about Event, Action and Activity

2.4.4.1 Event

According to the UML specification [UML01], there are five types of events: `call`, `signal`, `change`, `time` and `completion`² event. Among the above types of events, the `signal`, `change`, `time` and `completion` events are relatively simple in the sense that their corresponding event handlers only modify the object state (usually this is done by modifying the value of the attribute denoting the state). A `signal` or `call` event may carry parameters. The parameters are available to actions and thus their value may affect the collective state (defined in Section 2.4.1). A `call` event is more complex since its handler can modify the collective state. Note that this research does not account for the `completion` event because it represents the completion of an activity rather than an explicit event (UML01). Therefore its occurrence depends on when an activity completes its execution. In our state-based testing environment, we want the test driver to have full control over when transitions fire and this can only be done by creating explicit events to trigger transitions.

In a software design model, events are usually handled by event handler operations. This thesis does not consider completion event. Therefore we don't account

² Note that the term `completion` event is mentioned in [UML01] but is not a formally defined event type. Here we use this term to facilitate our discussion.

for triggerless transition, which is caused by the completion of an activity instead of an explicit trigger.

2.4.4.2 Action

Eight types of actions are defined in the UML specification [UML01]: assignment, call, create, destroy, return, send, terminate, and uninterpreted.

In this research, we do not consider the uninterpreted actions because they are language-specific and thus are not specified by UML. Their semantics will be precise only when they are combined with the semantics of the implementation language [UML01]. A return action returns the control to the object that invokes its execution and this has no effect on the collective states. The terminate action causes the self-destruction of the owning object of the state machine. Because the owning object will no longer exist after its execution, the terminate action is not interesting for this research and is thus not considered.

This thesis considers the following actions, which might change the collective state: assignment, call, send, create, and destroy. Among these types of actions only the call action is associated with an operation and therefore has its effect explicitly modeled in the operation contract. In this thesis it is desirable that the effects of all types of actions are captured by operation contracts. Our approach is to create a pseudo handler operation for each action that is not a call action and model the action's effect in the contract using OCL. In this way, each action will be associated with an operation, which we define as the *action handler*. This approach is reasonable because in an actual implementation, an event or action, no matter what type it is, is ultimately

realized by handler operations. With this approach, all other types of actions can be regarded as special cases of the `call` action.

Next, we show what the postcondition of each type of action handler will be. The semantics of actions discussed below is based on [UML01].

- The `assignment` action shows the modification to the collective state using an assignment expression. The action expression shows its effect on the collective state. For the action handler, the action expression (e.g. `a=a+1`) becomes its postcondition if we place `@pre` (the time expression) after each attribute occurring in the right-hand-side of the expression (e.g. the postcondition will be: `a=a@pre+1`).
- The `call` action is a synchronous method call to operations defined in the invoking object itself or another object. The `call` action already owns an operation that has a postcondition.
- The `send` action is an asynchronous one-way communication. It sends a signal to a receiver object, triggers some processing in that object and possibly changes the collective state. The effect of this type of action depends on the receiver object, which must provide information about how the signal is dealt with: If the signal is a `signal` event for the receiver, the `send` action might change the state of the receiver; otherwise, the receiver specifies an action handler operation for the signal. However, since the `send` action is asynchronous, the sender might not be able to know whether the signal is received and processed by the receiver. Therefore, the effect of a `send` action is non-deterministic in nature. We choose the queue, which is the most

commonly used data structure in implementing the signal mechanism, to explain this. During the signal communication, the sender places signal objects in the queue; the receiver (might be on a remote computer) fetches the signal objects from the queue and processes them. It is known that signal losses can occur in a distributed environment such as the loss of connections due to hardware failure. Moreover, even when the signal communication is assumed to be reliable, non-determinism can still exist. For example, if the sender sends signals faster than the receiver's processing speed, the signals will accumulate in the queue. Consequently, there might be a random delay in processing a signal after it is sent. Assume the processing of the signal modifies the collective state (e.g. before and after the signal is processed, the value of attribute x of object obj is 0 and 1 respectively), which in turn affects the evaluation of the guards on subsequent transitions (e.g. the guard of an outgoing transition of the current state is $[obj.x=1]$), then the evaluation of the guards is non-deterministic (if the signal hasn't been processed when the outgoing transition's guard is evaluated, the guard is false; otherwise, the guard is true). In order to account for send actions in this research, we have to assume that (1) the signal will always be received and processed by the receiver. (2) Furthermore, either there is no delay in handling the signal or, when delay is allowed, the processing of the signal does not modify the collective state. Such assumptions must be specified for each send action in the specification of class cluster statecharts.

- The `create` action will create an instance of a class and add a new link between the creating object and the created object. The action handler of a `create` action is a constructor operation that has a postcondition.
- The `destroy` action will delete an existing instance as well as all the links between the destroyed object and other objects. The action handler of a `destroy` action is a destructor operation that has a postcondition.

Actions can be attached to states as entry/exit actions or associated with transitions. 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 [RJB99]. This can be illustrated in Figure 1.

Figure 1. Execution order of event and actions

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.

Entry and exit actions can be eliminated without changing the semantics of a statechart. An entry action can be moved out the state and attached to all incoming transitions; an exit action can be moved out the state and attached to all outgoing transitions [RJB99]. This can be done in the following ways.

- Remove the entry action in the state.

- Append the entry action to the end of the action sequence on each incoming transition. If there is no action on an incoming transition, add the entry action after the “/”.
- Remove the exit action in the state.
- Prepend the exit action to the beginning of the action sequence on each outgoing transition.

For example, in Figure 2 (a), a state has entry and exit actions, three incoming transitions and two outgoing transitions. It can be converted to its semantic equivalent counterpart in Figure 2 (b).

In this research, we assume statecharts have already been transformed according to the above procedure. As a result all actions are associated with transitions only.

Another assumption in this section is that an action has only one target object. According to [RJB99], 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 (like “broadcasting”). Each object receives and handles this action copy independently and thus constituting a complex concurrent system. Since in our testing context, concurrencies are removed (Section 2.4.2), we do not consider such situation.

2.4.4.3 Activity

Events and actions are atomic so their effect is deterministic [RJB99]. 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. In practice, however, non-determinism resulted in this way exemplifies an inappropriate modeling strategy as one cannot guarantee the system’s state integrity. We, therefore, assume either of the following two cases about the activity:

- An activity does not modify collective state (e.g., cause transitions or modify attribute values) at any point in time. For instance, a security-monitoring object entering its Alarm state will execute the `soundAlarm` activity, which does not change the system state. This type of activity can be ignored when deriving test cases for coverage criteria.

Figure 2. Elimination of entry and exit actions in a state

- An activity may modify collective state during its execution. However there is some recovery mechanism to restore the system state back to the state before the activity execution. For instance, a `withdrawCash` transaction activity will restore the initial balance value if the transaction is cancelled before completion. In this case, either the activity completes execution without interruption and triggers a completion transition to a new state, or the initial state is restored when its execution is interrupted. In both cases, the result is

deterministic. Statechart containing this type of activity need to be tested by two types of test cases. The first type of test cases includes an event that cancels the activity before its completion. The effect of the activity is ignored. The second type of test cases includes a completion transition, which is triggered by the completion of the activity in a state. The effect of the activity is deterministic. However, as we already discussed, this situation is not accounted for in our research. Therefore, in this case, the activity's effect on collective state can also be ignored.

2.5 Design by Contract

This research assumes that the Design by Contract principle [Meyer97] is followed.

In [WK99] the author describes the Design by Contract principle as follows. “In object-oriented terms, a *contract* is a means to establish the responsibilities of an object clearly and unambiguously”. A contract specifies a service supplied by an object, which is called the *supplier*. An object that uses the service is called *client*. Each service has a contract that specifies two things [WK99]:

- The condition under which the service will be provided
- A specification of the result of the service, given that the conditions are fulfilled

The services offered by the supplier are operations that can be performed by the supplier. For each operation, the obligations of the client (rights of the supplier) are specified by *preconditions* [WK99]. A precondition must be true right before the operation is executed. The obligations of the supplier (rights of the client) are specified

by *postconditions* [WK99]. A postcondition must be true right after the completion of the operation.

An invariant is another kind of model constraint, which must always be satisfied by all instances of a *classifier* (A classifier is a metaclass in the UML metamodel, which comes in several specific form including class, data type, interface, stereotype, etc.) [UML01]. This thesis only focuses on class invariant. Therefore, the class invariant, though implicitly, is part of the pre- and postcondition.

Operation contracts are operation specifications used by implementer as implementation guide. The operation is implemented according to the operation postcondition. In the following, we discuss requirements on postconditions so that they offer testable operation specifications.

2.5.1 Requirements on Postconditions

In a practical analysis or design model, postconditions serve as the specification of operations and they must satisfy certain requirements so that operations can be implemented correctly.

Requirement 1:

The postconditions must be complete. The postconditions are complete if 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. Incomplete postconditions are considered incorrect. Moreover, the condition (possibly resulting from a different set of inputs or system state) under which those changes take place must also be specified. Furthermore, in the context of UML statechart, if a `call` event can be recognized by

multiple states, then each state has a corresponding condition identified in the postcondition. If a `call` event triggers multiple transitions on a single state, then the guard on each transition has a corresponding condition identified in the postcondition.

Requirement 2:

The postconditions of a `call` event are not redundant with the postconditions of actions. In UML, the action semantics does not indicate where the effect of an action should be modeled, in the postcondition of the object that invokes the action, in the postcondition of the object executing the action, or both. Where the action computation is done depends on the implementation of the statechart. When using the state design pattern [Gamma94], the actions are invoked by the *event handler* methods. As a result, the event postcondition includes the actions' postconditions. The effect of an action is therefore modelled by the postconditions of both the event handler and the action itself. Such redundancy might cause the postcondition of the event handler to be very complex, thus difficult for the modellers to write. In order to reduce the complexity of postconditions, and to remove redundancy among postconditions of different operations, 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 would not affect the modelling of the overall effect of a transition. To explain this, we use an example as follows. Figure 3 shows two states and a transition. The transition has an event `ev()` and two actions: `act1()` and `act2()`. The event handler of `ev()` does three things: assign a literal value to attribute `a`, invoke action `act1()`, and invoke action `act2()`. Therefore the postcondition of the event handler includes the postconditions of transition actions, as shown in the figure. The overall effect of the

transition is a conjunction of postconditions of the event handler and actions. After removing redundancy for conjuncts “b=2 and c=3”, the result is “a=1 and b=2 and c=3”. In another modeling style the transition actions’ postconditions are not included in the postcondition of the event handler. In this style, the postcondition of `ev()` is “a=1”. The overall effect of transition is now “a=1 and b=2 and c=3”, which has no redundancy and is exactly the same as the overall effect of the transition in the first modeling style.

Figure 3. Postcondition of actions included in event handler

2.5.2 The Use of @pre Postfix

According to [UML01], an object property is one of the following:

- An attribute, whose type can be any OCL types
- An association-end, whose type is either a class or a collection type
- An operation specified by property `isQuery`

The last property includes the first two properties as an operation with the property `isQuery` is a query operation³ (or simply query) [UML01], which returns a value (may be attributes and/or association-ends) as the result.

³ A query operation has no side effects, which means it does not modify the system state.

A postcondition expression can refer to two sets of values for each property of an object [UML01].

- The value of a property at the start of the operation or method, defined as the *pre-value*
- The value of a property upon completion of the operation or method, defined as the *after-value*

To refer to the pre-value in the postcondition, one has to use the @pre postfix after the property name. It should be noted that this postfix can only be used in postconditions. A property name without this postfix refers to the after-value. Based on the definitions of pre-value and after-value, we define the *pre-state* of an operation to be the collective state of an object cluster before executing the operation, and the *after-state* to be the collective state of an object cluster after executing the operation.

To tell whether a property is changed or not, we need to discuss different situations as follows:

- If the type of the property is a basic OCL type such as Integer, Real, Boolean or String, it is unchanged if and only if its after-value is the same value as its pre-value. Otherwise, it is changed.
- If the type of the property is a class, it is unchanged if and only if its after-value is the same reference as its pre-value. Otherwise, it is changed.
- If the type of the property is a Set, it is unchanged if and only if its after-value contains the same elements as its pre-value. Otherwise, it is changed.
- If the type of the property is a Bag, it is unchanged if and only if its after-value contains the same elements as its pre-value and the number of each

element is the same in both after-value and pre-value. Otherwise, it is changed.

- If the type of the property is a Sequence, it is unchanged if and only if its after-value contains the same elements and in the same order as its pre-value. Otherwise, it is changed.

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. This contextual instance is referred to as the *context object* [WK99]. When written inside constraints such as preconditions, postconditions, and guards, 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, probably followed by a sequence of association-ends until the class that owns the property is reached. Note that in a statechart, a `call` action expression also has a navigation path, which starts from a context object, probably followed by a sequence of association-ends until the class that owns the operation is reached. According to [UML01], the following properties may not be the same.

<code>self.a.b</code>	-- take the after-value a of self, say x, and then
	-- take the after-value b of x
<code>self.a.b@pre</code>	-- take the after-value a of self, say x, and then
	-- take the pre-value b of x
<code>self.a@pre.b</code>	-- take the pre-value a of self, say x, and then
	-- take the after-value b of x
<code>self.a@pre.b@pre</code>	-- take the pre-value a of self, say x, and
	-- then take the pre-value b of x

Therefore, when referring to the pre-value of a property in the postcondition, the name of all the association-ends must have `@pre` as the postfix.

2.5.3 Change Specifications vs. Frame Rules

In general, postconditions model changes to object properties. According to [MM02] 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*. Frame rules are important to a full understanding of the meaning of an operation [MM02]. 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. A tradeoff is to specify frame rules only where ambiguities exist without frame rules. For instance, `Class1` has Integer attributes `a` and `b` and an operation `op()`. What `op()` does is to assign the pre-value of attribute `b` plus one to attribute `a`. It does not change the value of attribute `b`. The following is an incomplete postcondition of `op()` since the implementers won't know whether the values of `a` and `b` change and if so, how they change:

```
context Class1::operation()
post: self.a = self.b + 1
```

In this case, the frame rule must be used to resolve ambiguity. The following postcondition is a correct and complete specification of the operation:

```
post: self.a = self.b + 1
      and
      self.b = self.b@pre
```

In general, ambiguity occurs when a postcondition compares the after-value of a changed property with the after-values of other properties. Without frame rule, it would be impossible to tell what property is changed and what is not.

On the other hand, for postconditions that compare the after-value of a changed property with literals and/or operation arguments, there is no ambiguity. In this case, a common practice is to adopt a convention that all operations come with an implicit frame rule [MM02], which states that, if a postcondition does not specify that some property changes, it is assumed not changed. Unexpected changes are thus symptoms either of incomplete postconditions or bugs.

In this research, we adopt the convention of implicit frame rules while still allowing for explicit frame rules to resolve ambiguity.

Chapter 3 – Methodology

This section describes the methodology adopted by this research in order to solve the problem of deriving test constraints on the test data for a given transition test sequence (TTS). The methodology is presented in a structured manner. Section 3.1 precisely defines the form of a TTS. Section 3.2 introduces an appropriate data structure, the invocation sequence tree (IST), necessary to capture the interactions among state-dependent objects (Section 2.4.1). The data structure is devised from the TTS and class-cluster statecharts. This is followed by a description of the procedure we devised to produce test constraints from ISTs. Finally we provide the normalization rules of OCL expressions that we use to ease the generation of test constraints.

3.1 Transition Test Sequence

All the proposed criteria based on statechart diagrams assume a test case to be in the form of a feasible TTS in the statechart. 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* [Offutt99], which is a sequence of inputs that puts the system in a pre-state.

When the statechart contains guards, the TTS also specifies a condition predicate (i.e., Boolean expression that may contain logical operators) for each event, and the way those predicates are derived from the corresponding guards depends on the coverage criterion. Using a simple example, when one chooses to cover all transitions in a

statechart, in the case of guards, one has to choose one disjunct in the Disjunctive Normal Form (Section 3.4.2) of the guard condition. A number of more thorough criteria exist to test guards [Offutt99] but a sequence of transitions is always denoted as follows:

`@state0@event0[pred0]/actions0@state1@event1[pred1]/actions1@...`

where `pred0`, `pred1`, ... are the predicates derived from the corresponding guards 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 clauses.

3.2 Invocation Sequence Tree

The main motivation of this section is to facilitate the derivation of test constraints by defining an appropriate data structure (IST).

In a state-dependent system that contains interacting objects, there exist a phenomenon we term as the *invocation chain*. When the owning object is executing a transition, it may send events to other objects. If the target object has a state-dependent behavior, the events may trigger a transition, which might in turn send events to yet another object.

In this thesis, we assume that an invocation chain does not form a cycle, that is, as a result of triggering a transition in an object statechart, an event is sent to that same object (possibly indirectly by another object), thus producing a cycle. We consider this to be a rare and awkward design as this results in an event triggering more than one transition at a time (recall that we do not consider concurrent statecharts here (Section 2.4.2)). For instance, Figure 3 illustrates an awkward statechart design for a class. When an object of the class is in `state1` and it dispatches event `e1()`, several different

effects are possible. Depending on the evaluation of guards (g_1 , g_2 , and g_3), the object may be in `state2` with output “ $a=0$ ”, or in `state3` with output “ $a=1$ ”, or in `state1` with output “ $a=2$ ”. In addition, an awkward but possible scenario is that the state transitions may form an infinite loop with certain value set (in this case, g_1 is “ $a=2$ ”, g_2 is “ $a=0$ ”, g_3 is “ $a=1$ ”) of the guards. In this case, the object has no stable response to event $e_1()$.

Figure 4. An awkward design of a statechart

Since we assume invocation chains are cycle free, they 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, or when the triggered transition does not have actions.

3.2.1 Definitions

An IST is built as an acyclic digraph. A node denotes an event/action and an edge an *invocation*. An invocation is directed and connects two nodes: the *predecessor* and *successor*, the latter being invoked after the former. For instance a transition with an event and a sequence of two actions results into a node for the event and nodes for the actions, and invocation between the event and the two nodes. An action sent to (or invoked on) another object is also a trigger event if it triggers transitions on that object.

As discussed in Section 2.4.4, this research studies five types of actions, among which only the `call` and the `send` action can act as trigger events to other state-dependent objects. In comparison, the `assignment`, `create` or `destroy` actions won't be trigger events for other objects.

The IST shows all possible *invocation scenarios* that may occur during the execution of the TTS. Some scenarios are feasible, others are not depending on guard for instance. An invocation scenario consists of a sequence of *invocation subscenarios* (abbr. subscenario). An invocation subscenario is associated with an event and it consists of a sequence of actions invoked after the event. Figure 5 shows an example IST, in which, `event1` has two subscenarios and `event2` has one subscenario. The total number of scenarios is two. An event node and all the nodes that can be invoked directly or indirectly after it form an *invocation subtree* (abbr. subtree). The subtree models all the invocation subscenarios that can be invoked when a specific event occurs. The root of the subtree (corresponding to an event in the owning class statechart) is called the *ancestor* and all the other nodes (corresponding to actions in the owning class statechart or events/actions in other classes) in the subtree are *descendants* of the ancestor.

Figure 5. An example IST

The invocations are labelled with conditions, called *invocation conditions*. An invocation between a predecessor and a successor node, and its invocation condition (enclosed in a pair of square brackets) are illustrated in Figure 6. It states what condition must be fulfilled to execute the event/action associated with the successor node. An invocation condition may involve a maximum of three parts, a precondition, a state condition and a guard, depending on the following three situations:

- An incoming invocation to a node representing an event in the TTS. The corresponding invocation condition is the conjunction of the event precondition and the guard of the transition.
- An incoming invocation to a node representing an action that does not trigger any transition in another statechart. The corresponding invocation condition is simply the precondition of the action.
- An incoming invocation to a node representing an action that triggers a transition in another statechart. The corresponding invocation condition is the conjunction of the precondition of the action, the guard of the triggered

transition (in the other statechart), and the state condition (state invariant) of the triggered transition.

Figure 6. Illustration of an invocation

Note that if an action invoked on another object enables two or more transitions on that object, each transition is represented by an *invocation alternative* in the invocation sequence tree. Edges leaving the same action node have an *alternative relation* (XOR) to each other by default, which means there must be one and only one executed invocation out of all the alternatives. Two invocation alternatives have the same predecessor node, two different successor nodes that have the same expression, and mutually exclusive invocation conditions. In contrast, the other type of relation in an IST is the *sequential relation*. Under this relation, for two invocation chains that originate from the same predecessor action, one invocation chain is executed first and when it terminates the other invocation chain will start. Two invocation chains have sequential relation if their respective successor actions are adjacent actions in an action sequence on a transition. Recall that in this research, actions within the same action sequence are assumed to be independent of each other (Section 2.4.4) and as a result we analyze them according to their written order. Directed arcs are used to specify the order between two adjacent invocation chains. Such an arc is defined to be an *invocation order*.

An example to illustrate these two relations are shown in Figure 7. Instances of Class X and Y have state-dependent behavior. Operation m1 is a trigger event operation for instance x of class X. Class Y has two operations m2 and m3. Event `self.m1` triggers a transition on x as follows.

```
self.m1() [g]/self.y.m2();self.y.m3()
```

In Figure 7, invocations are labelled with number to facilitate our discussion. Invocations 1 and 2 are alternatives as they have the same predecessor node and the expressions of their successor nodes are the same. Note that the two invocations have mutually exclusive invocation conditions (condition `[c1]` and condition `[not c1]`). If both alternative and sequential relations exist among the outgoing invocations from the same predecessor action (`self.m1`), for each invocation alternative (the two invocations with the same successor action `self.y.m2`), an arc must be drawn to connect it to the edge that invokes the next action (`self.y.m3`) in the action sequence. The two invocation arcs 1-3 and 2-3 connect invocation 1 to 3 and 2 to 3 respectively.

Figure 7. Alternative relation vs. sequential relation

The number of invocation subscenarios for a trigger event can be calculated by traversing the invocation subtree in a bottom-up fashion. First, we look at two simple cases in Figure 8. Diagram (a) shows a simple case where two invocations have an alternative relation. Diagram (b) shows a simple case where two invocations have a

sequential relation. Numbers in the parentheses are numbers of subscenarios in the subtree that start with the corresponding event/action node as a root. The number of subscenarios triggered by event in (a) is the addition of the number of subscenarios of all its successor nodes; and the number of subscenarios triggered by event in (b) is the product of the number of subscenarios of all its successor nodes. With this observation, we can compute the number of subscenarios for more complex cases. Two examples are provided in diagrams (c) and (d).

Figure 8. Computing of the number of subscenarios

3.2.2 Metamodel of IST

We model the structure of invocation sequence trees as described in the previous section by means of the class diagram in Figure 9 (metamodel).

Figure 9. Metamodel of IST

This metamodel does not only help us to precisely define what an invocation sequence tree is but also allows us to define algorithms in a precise but abstract manner. These algorithms build an IST from the UML models (class diagram and statecharts) and a given TTS and derive test constraints from the IST (Appendix A). There are six classes in the metamodel (Figure 9). An IST is composed of a sequence of SubTrees corresponding to each transition in the TTS. Each SubTree has a root, which is an ISTNode. An ISTNode may have a number of SubTrees. It also has a precondition (precond) and a postcondition (postcond), which are both ISTConstraints. An ISTNode has one incoming Invocation (incomingInvoc) and a number of outgoing Invocations (outgoingInvoc). Each Invocation has zero or one predecessor (predecessor) and a successor (successor), both of which are

ISTNodes. It also has an invocation condition (`invocCond`) that is an ISTConstraint. An Invocation might have a number of previous Invocations (`previous`) and a number of next Invocations (`next`) due to the sequential relations between invocations. For example, in Figure 8 (d), the leftmost invocation (from event to `action1`) has two next invocations, while the rightmost invocation (from event to `action2`) has two previous invocations. Both `ISTEvent` and `ISTAction` are ISTNodes. An `ISTEvent` has a sequence of `SubScenarios`, which consists of a sequence of `ISTActions`.

3.2.3 Building the IST

3.2.3.1 Algorithms

To build an IST for a given TTS, we need the UML statechart of the class under test and, if the statechart contains actions that are invoked on instances of other classes (i.e., some actions have a navigation path that navigates from the statechart owning class to other classes), we also need the UML class diagram (to show the associations among classes) and the statecharts of classes whose transitions might be triggered. We say a class is relevant to the owning class (which owns the statechart under test) if its transitions might be enabled when an instance of the owning class executes the TTS. For instance, assume `classA` is the owning class and when an instance of `classA` executes the TTS, one of its transitions has a `call` action that is invoked on an instance of `classB` and enables a transition. Then `classB` is relevant to `classA`. Note that the relevance of a class is transitive, i.e., if `classB` is relevant to `classA`, and `classC` is relevant to `classB`, then `classC` is also relevant to `classA`.

The creation of an IST is to map information contained in UML statecharts and class diagram (if other classes than the owning class are relevant) to the IST. This mapping is performed in a systematic way, following a sequence of steps (for detailed algorithm, see Appendix A.3). In this section, we first describe the steps at a high level of abstraction, during which we also describe the issues to be resolved in order to automate this task. Then we illustrate how to build an IST with a simple but representative example following these steps.

The tree-construction steps are illustrated, at a high level of abstraction, with the activity diagrams in Figure 10 and Figure 11. Note that the algorithms listed here are intended to show how the tree structure is constructed; therefore the creation of invocation condition and operation contracts as well as the normalization steps are omitted. Such information can be found in the detailed algorithms (Appendix A).

In the overall algorithm (called `buildIST`) described in Figure 10, first an IST with no subtree is built (Step 1). Next we iteratively process each transition (from the first to the last) in the TTS. For each transition t , another algorithm `buildSubTree` is invoked to build a subtree for t (Step 3), a sequence of invocation subscenarios is created for the subtree (Step 4), and then an invocation is created whose successor is the subtree root node and whose invocation condition⁴ is the conjunction of the guard and the precondition (if any) of t 's event (Step 5). Finally, the created subtree is added to the IST (Step 6). This process terminates when all the transitions in the TTS have been processed, otherwise, the next transition is obtained (Step 7) and the process goes to Step 3.

⁴ Note that for such an invocation, the state condition in the invocation condition is implicitly satisfied by the TTS, so it is omitted from the invocation condition.

Figure 10. Building IST (buildIST), overall abstract algorithm

Figure 11 shows the abstract algorithm of `buildSubTree(t)`, which recursively builds a subtree for the transition `t`. As stated before (Section 3.2.1), transition `t`'s event `ev` corresponds to the root node (type is `ISTNode`) of its subtree, so first the algorithm creates an `ISTNode` called `rootNode` for `ev` (Step 1 and 2). Then, the parent subtree `st_p` for `t` is created and its root node is `rootNode` (Step 3). Next, the algorithm obtains the transition action sequence (`actSeq`) of `t` (Step 4), assigns the first action in `actSeq` to `action` (Step 5), and then assigns the action's handler operation to `ah` (Step 6). After that, `ah` is checked to see whether it's an event handler or not because this will lead to different processing. These two situations are described separately below:

- *Situation 1.* `ah` is an event handler, i.e., it triggers transitions in another statecharts.

Then `action` might enable a number of transitions on another object and the algorithm follows Steps 7.1.1 to 7.1.7. First, all transitions that might be enabled are obtained and stored in sequence `transitionSeq` for further processing (Step 7.1.1). Next the algorithm iterates all the transitions and for each transition `tr`, it does the following: (1) Builds a child subtree `st_c` for `tr` (Step 7.1.3, note that this is a recursive call); (2) Add the subtree to `rootNode`'s subtree sequence (Step 7.1.4); (3) Create an invocation from `rootNode` to the root node of `st_c` so that the link between the parent subtree and the child subtree is established. Also create the invocation condition that is the conjunction of the guard, the state condition, and the precondition of `ah` (Step 7.1.5); (4) Create sequential relations between each of the invocations that have been created for the previous action in `actSeq` to this invocation, which is created for the current action in `actSeq` (Step 7.1.6). After that, if `tr` is not the last transition, then set it to be the next transition in `transitionSeq` (Step 7.1.7) and go to Step 6. Otherwise, the processing for current action is complete, and the algorithm goes on to process the next action in `actSeq`.

- *Situation 2.* `ah` is not an event handler.

Then `action` does not enable any transition and the algorithm follows Steps 7.2.1 to 7.2.5: (1) An `ISTNode` called `actNode` is built for `action` (Step 7.2.1); (2) The child subtree `st_c` is created and its root node is `actNode` (Step 7.2.2); (3) The subtree is then added to `rootNode`'s subtree sequence (Step 7.2.3); (4) The algorithm creates an invocation from `rootNode` to the

root node of `st_c` so that the link between the parent subtree and the child subtree is established. It also creates the invocation condition that is the precondition of `ah` (Step 7.2.4); (5) Create sequential relations between each of the invocations that have been created for the previous action in `actSeq` to this invocation, which is created for the current action in `actSeq` (Step 7.2.5). The algorithm then goes on to process the next action in `actSeq`.

If `action` is not the last action in `actSeq`, then set `action` to be the next action in `actSeq` and continue with Step 6 (Step 8.1). Otherwise, all the action in `actSeq` have been processed, and the parent subtree `st_p` is returned (Step 8.2).

3.2.3.2 Issues

There are two issues related to the subtree-building algorithm that need to be solved. First, the information on whether or not a `call` action will raise a `call` event, i.e., triggers a transition in a statechart, can be obtained from the statechart (by checking the name of events on transitions) of the class that defines the action operation. However, it would be inefficient if the statechart information has to be retrieved each time we check an operation (taken into account that there might be a number of statecharts, each of which has a number of transitions). In our implemented tool (Chapter 4), we add a Boolean attribute `isEventHandler` to the `Operation` metaclass and we set this attribute value when loading the statechart information at the beginning of processing. In this way, the statechart information only needs to be loaded once.

Figure 11. Building subtree (buildSubTree), abstract algorithm

Second, in Step 7.1.1 of the algorithm in Figure 11, the algorithm retrieves all possible transitions that might be enabled by a given action (action). This information

can be retrieved in two different ways. In the first situation, the last association end in the action's navigation path implicitly denotes the state of the target object. Let's take an example from the Video Store System case study (Section 5.1). In the statechart of the `Copy` class, one transition from state `Rented` to `OnHold` has the action `self.title.oldestPending.copyAvailable()`. The last association end of this action's navigation path is `oldestPending`, which specifies that the state of the target object is `Pending`. In this case, we get transitions (with their event name to be the same as the action name) from all the outgoing invocations of that specific state.

In other situations, the association end does not specify the state of the target object and in this case, all transitions with the same event name as the action name are obtained. However, such state information cannot be obtained from either the UML class diagram or statecharts as there is nowhere in the UML metamodel to store such information. Therefore, we need to extend the UML metamodel to accommodate this state information. We add an attribute `participantState` to the `AssociationEnd` metaclass in the UML core package to store the state invariant of its participant. This information is to be provided by the user when the class diagram is loaded (Section 4.2).

3.2.3.3 Examples

To illustrate the algorithms detailed above, consider the example in Figure 12, which shows a class diagram (Figure 12 (d)) containing three classes (`X`, `Y`, and `Z`) and their respective statecharts (Figure 12 (a) to (c)). Assume `X` is the owning class that will execute a given TTS and `Y` and `Z` are relevant classes. To facilitate the discussion, we assume `call` events and `call` actions only in these statecharts (from the statecharts, we

can see that operations e_{x1} to e_{x4} , e_{y1} , e_{y2} , e_{z1} are event handlers and operations a_{x1} and a_{z1} are action operations). Also, to keep the discussion simple, we assume event handler operations and action operations have the same name as their events/actions respectively (e.g., the name of the handler operation for event e_{x1} is also e_{x1}).

We first consider the following TTS to be executed by an instance of X:

$@S_{x1}@e_{x1}[g_{x1}]@S_{x2}@e_{x2}[g_{x2}]@S_{x1}$

(d) class diagram of X, Y, Z

Figure 12. Statecharts and class diagram of the building IST example

Figure 13 is the corresponding IST of the TTS. In constructing this IST, algorithm `buildIST` (referred to as Algorithm1 in the following discussion) first builds an IST `ist` without subtrees (Algorithm1, Step 1). Next, it takes the first transition in the TTS and passes it as an argument to algorithm `buildSubTree` (referred to as Algorithm 2 in the following discussion)(Algorithm 1, Step 2), which then considers the event and actions of the transition. For the first transition, the event is e_{x1} , the guard is g_{x1} (Algorithm 2, Step 1). The algorithm creates a corresponding `ISTNode` node (node e_{x1})

for Event e_{x1} (Algorithm 2, Step 2). It then builds a parent subtree whose root node is e_{x1} (Algorithm 2, Step 3). Since the transition has no action (Algorithm 2, Step 4), the root node has neither outgoing invocations nor subtrees. The parent subtree is then returned to algorithm `buildIST` (Algorithm 2, Step 8.2), which then proceeds with the following tasks. Creates the subscenarios for the subtree (Algorithm 1, Step 4). There is one subscenario, which contains one `ISTEvent` node e_{x1} , but no `ISTAction` node. Create an incoming invocation (`ISTConstraint`) for whose invocation condition is a conjunction of the guard and the precondition of e_{x1} [g_{x1} and pre_e_{x1}] (Algorithm 1, Step 5). Now, the subtree construction is complete and it is added to `ist` (Algorithm 1, Step 6). In the next step, the algorithm proceeds to the next transition, builds another subtree and adds it to `ist`. It then terminates because all transitions have been processed.

Figure 13. IST for a given TTS (example 1)

Now let's consider a more complex TTS shown below (note that, now in this TTS, the two transitions both have actions):

$@S_{x1}@e_{x3}[g_{x3}]/self.a_{x1}@S_{x2}@e_{x4}[g_{x4}]/self.y.e_{y1}@S_{x1}$

The corresponding IST is shown in Figure 14. In the following illustration of the execution steps, we mainly focus on the steps of algorithm `buildSubTree` since algorithm `buildIST` executes in the same way as in the previous example.

Figure 14. IST for a given TTS (example 2)

Building Subtree for Transition: $e_{x3} [g_{x3}] / self.a_{x1}$

After the first transition is passed to algorithm `buildSubTree`, the algorithm (referred to as recursive level one⁵, or simply level-1 process) creates the `rootNode` (e_{x3}) for Event e_{x3} (Algorithm 2, Step 1&2) and then builds a parent subtree whose root node is `rootNode` (Algorithm 2, Step 3). Next the process obtains the transition's action sequence, which consists of one action (Algorithm 2, Step 4). Then, the action is obtained from the sequence and the handler operation of the action is analyzed (Algorithm 2, Step 5&6). Since the action (`self.ax1`) is invoked on the owning object, it is not a trigger event⁶. So the process proceeds with Steps 7.2.1 to 7.2.5. The result is that: an `actNode` is created for the action; a subtree whose root node is `actNode` is created and the subtree is added to `rootNode`; an invocation is created whose predecessor is `rootNode` and successor is `actNode`, and the invocation condition is

⁵ We use recursive level n to denote a process that has been recursively invoked $n-1$ times.

⁶ Recall that we assume an IST is an acyclic digraph so an action sent by the owning object to itself cannot enable transitions on itself (Section 3.2).

the precondition of the action handler (`pre_ax1`); no sequential relations are established since there is no previous action to this action in the action sequence. Because there is no other action in the action sequence, the subtree is then returned (Algorithm 2, Step 8.2) to algorithm `buildIST`, which then creates the subscenarios for the subtree (Algorithm 1, Step 4). There is one subscenario, which contains one `ISTEvent` node `ex3`, and one `ISTAction` node `self.ax1`.

Building Subtree for Transition: `ex4 [gx4] / self.y.ey1`

When the second transition is passed to algorithm `buildSubTree`, the algorithm executes in a similar way except that this time, the action (`self.y.ey1`) is a trigger event invoked on another object and enables several transitions. So the algorithm (level-1) proceeds with Steps 7.1.1 to 7.1.7. Recall that to solve the second issue addressed earlier in this section requires information about whether or not an action's navigation path specifies the state of the target object that executes this action (Section 3.2.3.2). Here we assume that the path `self.y` does not specify the state. So any transition in class `Y`'s statechart with event name `ey1` might be triggered. There are two such transitions, the first from state `Sy1` to `Sy2` (`ey1 [gy1]`) and the second from state `Sy2` to `Sy1` (`ey1 [gy2] / self.z.ez1; self.z.az1`) (Level-1, Step 7.1.1).

The level-1 process iterates the two transitions one at a time.

It starts by recursively invoking itself (Recall that `buildSubTree` is a recursive algorithm) and passes the first transition (`ey1 [gy1]`) as an argument to the new execution process (level-2). The level-2 process creates a `rootNode` for `ey1` (`self.y.ey1`) (Level-2, Step 2), creates a subtree whose root is `rootNode` (Level-2, Step 3). Since

the transition has no action, the subtree construction is complete and the level-2 process terminates and returns the subtree to its level-1 process (Level-2, Step 8.2).

The level-1 process then adds this subtree to `rootNode` (e_{x4}) (Level-1, Step 7.1.4), creates an invocation from `rootNode` to the subtree's root node (`self.y.ey1`) (Level-1, Step 7.1.5). The invocation condition is g_{y1} and pre_e_{y1} and `self.state=Sy1`, which is a conjunction of guard, e_{y1} 's precondition, and the state condition (the source state of the transition). No sequential relations are established since there is no previous action to this action in the action sequence (Level-1, Step 7.1.6).

Now the processing of the first transition is complete and the level-1 process proceeds to the second transition ($e_{y1} [g_{y2}]/self.z.e_{z1};self.z.a_{z1}$) (Level-1, Step 7.1.7) by invoking itself and passing the transition as the argument (Level-1, Step 7.1.3). The new execution process (level-2) creates the `rootNode` (`self.y.ey1`) (Level-2, Step 2), and the subtree whose root node is `rootNode` (Level-2, Step 3). In the next step, it obtains the action sequence `actSeq`, which is `self.z.ez1;self.z.az1` (Level-2, Step 4). It then iterates the two actions. The first action (`self.z.ez1`) (Level-2, Step 5) in the sequence is a trigger event (Level-2, Step 6) for class Z and thus algorithm steps 7.1.1 to 7.1.7 are followed. Assume that the navigation path `self.z` does specify the state of the target instance of class Z to be S_{z1} (Recall that such information is stored in the attribute `participantState` that is added to the core, Section 3.2.3.2). As a result, although in the statechart of class Z there are two transitions whose event name is e_{z1} , only the one ($e_{z1} [g_{z1}]$) that leaves state S_{z1} is enabled. So `transitionSeq` contains one transition (Level-2, Step 7.1.1).

The level-2 process invokes itself and passes the first transition (Level-2, Step 7.1.2) in `transitionSeq` as the argument. The new executing process (level-3) creates a `rootNode` (`self.z.ez1`) (Level-3, Step 2); creates the subtree whose root is `rootNode` (Level-3, Step 3). Since the transition has no action (Level-3, Step 4), the level-3 process terminates and returns the subtree to level-2 process (Level-3, Step 8.2).

The level-2 process then adds the subtree to its own `rootNode` (`self.y.ey1`) (Level-2, Step 7.1.4) and creates an invocation from its `rootNode` to the root node of the subtree (`self.z.ez1`) (Level-2, Step 7.1.5). The invocation condition is `gz1` and `pre_ez1` and `self.state=Sz1`, which is a conjunction of guard, `ez1`'s precondition, and the state condition. Again, no sequential relations are created for this invocation because there is no previous action to this action in the action sequence (Level-2, Step 7.1.6).

Now every transition in `transitionSeq` has been processed, the level-2 process proceeds to the next action (`self.z.az1`) in the action sequence (Level-2, Step 8.1). This action is not a trigger event (Level-2, Step 6), so algorithm Steps 7.2.1 to 7.2.5 are followed. The level-2 process creates an `actNode` (`self.z.az1`) for the action (Level-2, Step 7.2.1) and creates a subtree whose root node is `actNode` (Level-2, Step 7.2.2). The subtree is then added to `rootNode` (`self.y.ey1`) (Level-2, Step 7.2.3). Next, the level-2 process creates an invocation from the `rootNode` to the root node of the subtree (`self.z.az1`) whose invocation condition is the precondition (`pre_az1`) of the action (Level-2, Step 7.2.4). A sequential relation is created from its previous invocation (from node `self.y.ey1` to `self.z.ez1`) to this invocation because this invocation's successor node corresponds to an action (`self.z.az1`) that has a previous

action (`self.z.ez1`) in the action sequence of the transition in Y's statechart (`ey1` [`gy2`]/`self.z.ez1`;`self.z.az1`) (Level-2, Step 7.2.5).

Now all actions in the `actionSeq` have been processed. Thus the level-2 process terminates and returns the subtree (with root node `self.y.ey1`) to level-1 process (Level-2, Step 8.2). The level-1 process adds the subtree to `rootNode` (`ex4`) (Level-1, Step 7.1.4) and then creates an invocation from `rootNode` to the subtree's root node (`self.y.ey1`) (Level-1, Step 7.1.5). The invocation condition is `gy2` and `pre_ey1` and `self.state=Sy2`, which is a conjunction of guard, `ey1`'s precondition, and the state condition (the source state of the transition). No sequential relations are established since there is no previous action to this action in the action sequence (Level-1, Step 7.1.6).

Now both transitions possibly enabled by action `self.y.ey1` have been processed and there is no next action on transition `ex4` [`gx4`]/`self.y.ey1`, the level-1 process will return the subtree (with root node `ex4`) to algorithm `buildIST` (Algorithm2, Step 8.2), which then creates the subscenarios for the subtree (Algorithm1, Step 4). There are two subscenarios: the first consists of one `ISTEvent` node `ex4` and one `ISTAction` node `self.y.ey1` and the second consists of one `ISTEvent` node `ex4` and a sequence of three `ISTAction` nodes `self.y.ey1`, `self.z.ez1`, and `self.z.az1`.

3.3 Test Constraints Derivation

In this section, we describe how to derive test constraints on the test data (the collective state and event/action arguments) so that a specific invocation subscenario in

the IST can be properly executed. For detailed algorithms on deriving test constraints for an invocation subscenario and for the IST, refer to Appendix A.

3.3.1 Types of State-dependent Objects

We identify two types of state-dependent objects under test and they require different processing (and thus different algorithms) with respect to constraints derivation. In the following, we describe these two types of objects and distinguish the differences in deriving test constraints for them.

An *associated state-dependent object* is an object whose behavior depends on the state (at both levels of abstraction) of other objects. Such an object must be an instance of an *associated class*, which is part of a class cluster and can navigate to other classes in the cluster along associations. An *orphan state-dependent object* is an object whose behavior does not depend on the state of other objects. It must be an instance of an *orphan class*, which has no associations with the navigable association end that allows it to navigate to other classes. Figure 15 shows two associated classes, ClassA and ClassB, and one orphan class ClassC. Assume all three classes own a state machine. From the class diagram, we can tell that instances of ClassA and ClassB must be associated state-dependent object and instances of ClassC must be orphan state-dependent object.

Figure 15. Associated class and orphan class

In order to execute a Transition Test Sequence (TTS) on an object whose state-dependent behavior is specified by a statechart, it is important to describe what the initial state (before the execution of the TTS) of the object and what values should be selected for arguments in events and actions. Recall that our objective is to provide test constraints describing that initial state and these argument values. The ultimate objective, once initial test inputs satisfying those constraints have been found, is to build a test driver that sets the object state to its initial value and then execute the TTS using adequate (i.e., satisfying constraints) argument values. This automatic execution is easy to implement for an orphan state dependent object since its behavior does not depend on any other object state: i.e., once the initial state is set, and provided that adequate argument values are used, the test driver can execute the TTS automatically. However, this automatic execution of the TTS by a driver is more complicated for an associated state-dependent object. Indeed some transitions in the TTS may require that other objects be in specific states. The driver then needs to suspend the execution of the TTS to set the states of these other objects before resuming the execution (i.e., before triggering the transition that requires specific other objects' states). In the case of an associated state-dependent object, it is then important to derive (under the form of a constraint) the collective state (i.e., the state of all the objects involved) before every transition (this includes the initial state), in addition to the argument values for events and actions.

3.3.2 Deriving Test Constraints for Associated State-dependent Objects

An invocation subscenario is a branch in the invocation sequence tree. It starts with an edge (i.e., `Invocation` in the IST metamodel) followed by a node (i.e., an `ISTNode` in the metamodel), representing the invocation of the event. This first

edge/node is then possibly followed by a sequence of edges/nodes representing the invocation of the actions in the transition (see Figure 18 for an example).

Deriving a test constraint for an invocation subscenario, i.e., the constraint which, when satisfied, allows the execution of the event and actions in the subscenario, amounts to propagating constraints that appear in the tree branch (invocation constraints on edges and postconditions on nodes) onto the first edge of the tree branch. This is done in a recursive way from bottom to top, thus starting with the last edge and ending with the first one. Note that the result of that process might be a set of constraints rather than a single constraint (this is discussed below). The general problem to be solved in this recursive process is then, 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 Figure 16, we have to propagate invocation condition `c2` (for invocation `I2`) onto invocation condition for invocation `I1` using postcondition `post`. Informally⁷, this amounts to changing `I1` invocation condition into `c1` and $f(\text{post}, c2)$, where f is the function, to be defined, that propagates `c2` using `post`. The rationale is that, if `c1` and $f(\text{post}, c2)$ is satisfied, then we can execute `Op1` and `Op2`.

As stated above, the propagation function uses an invocation condition and a postcondition (`c2` and `post` in Figure 16). Its purpose is to transform a constraint on after-values (i.e., values after the execution of `Op`), as stated in `c2`, into a constraint on

⁷ In fact invocation condition for invocation `I1` is not changed, as suggested by the discussion. We use that mechanism to provide some intuition as to what is done in the recursive process. The subscenario is not changed in any way. The propagation of constraints is built on a separate structure, i.e., a set of constraints.

before-values (i.e., values before the execution of Op), using postcondition post that provides the mapping between those after and before-values.

For example, assume `c2` equals `attribute>3` and that post equals `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 clauses), the result is very different. `c2` being the same, assume post equals 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 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 (this is illustrated with a more complete example below).

Figure 16. Deriving test constraints (example 1)

The overall recursive process is illustrated, at a high level of abstraction with the activity diagram in Figure 17: detailed algorithms (pseudo code) can be found in Appendix A.11. As stated before, the process starts with the last invocation in the subscenario: this invocation is called I2 (invocation condition $c2$). The next step is to initialize the set of constraint, called SC, with I2 invocation condition. If the subscenario contains only one invocation, i.e., the transition does have an event but does not have any action, the process stops, and the constraint to be satisfied to execute the subscenario is invocation condition $c2$, i.e., the conjunction of the transition guard condition and the event precondition. If I2 is not the first invocation, i.e., there are actions in the transition, we retrieve I2 predecessor ISTNode Op, as well as Op's incoming invocation I1 (steps 3 and 4 in Figure 17). Note that the names used here for invocations and ISTNodes (i.e., names used in Figure 17) match those used in Figure 16. The set of constraints SC is updated according to its current contents, I1 invocation condition $c1$ and Op postcondition (more details on this step are provided below). Last, the algorithm recursively progresses up in the tree branch.

Figure 17. Deriving test constraints, abstract algorithm

To further illustrate the recursive process detailed above, consider the example subscenario in Figure 18 (a). The execution of the algorithm in Figure 17 is illustrated in Figure 18 (b). First, set of constraints SC is initialized with the last invocation condition, i.e., invoc3 . The algorithm then enters the loop as this is not the first invocation condition. SC is updated according to invoc2 and postcondition postA (we assume it does not contain any implies-expression). During the second iteration of the loop, the algorithm encounters a postcondition (i.e., postB) with two implies expressions. This results into two constraints in SC . Last, the algorithm encounters a postcondition (i.e., postC) with three implies-expression. The three alternative possible executions of event are combined with the two alternative executions of the rest of the subscenario, thus producing 6 constraints.

Figure 18. Deriving test constraints (example 2)

In order to detail how the propagation of an invocation condition, using a postcondition, is achieved (i.e., what f is in Figure 16 and Figure 18), 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 Section 3.4, we present normalization steps that ensure these assumptions are met. Typically, a normalized invocation condition is in DNF and a normalized postcondition is in the Standard Equality Form (see Section 3.4.5). The Standard Equality Form is a conjunction of equations $l\text{-value} = r\text{-value}$ where $l\text{-value}$ is the value of a constrained element after the execution of the operation and $r\text{-value}$ is an OCL expression containing only $@pre$ values (i.e., values before the execution of the operation), arguments and literals. Alternatively, conjuncts in the Standard Equality Form can contain implies-expressions of the form $condition \text{ implies } impliedPart$, in which case the conditions of

the implies-expressions are mutually exclusive and the implied parts are conjunctions of l-value = r-value equations. For instance, the following is an example of a postcondition in the Standard Equality Form:

```

attribute1 = attribute2@pre * argument + 1
and
aCollection->size = 0
and
attribute1 = 1 implies result = 1
and
not (attribute = 1) implies result = 0

```

Let us first consider the simple situation where the postcondition does not contain any implies expression. It is thus a conjunction of equations, as stated above. The propagation of invocation condition I2 then replaces every occurrence, in I2, of every l-value that appear in Op's postcondition 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. Reusing a previous example, assuming c2 equals attribute>3 and post equals attribute=attribute@pre+1, the propagation results in constraint attribute+1>3.

If the postcondition contains implies-expressions of the form condition implies impliedPart, the transformation is a little bit 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 of c2 and impliedPart. Reusing, once again, a previous example, assume c2 equals attribute>3 and post equals 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 constraints:

```

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

```

Note that in the two descriptions above we consider one invocation condition *c2* that needs to be propagated using the operation postcondition, as we wanted the discussion to remain simple. However, as we have seen, during each execution of the loop in Figure 17, we have to consider a set of constraints (and not only one), because of the implies-expressions in postcondition. The transformation above is in fact realized for each constraint in the set of constraints and *c2* can be considered a placeholder for them.

It must also be noted that when a UML model element is constrained in two different OCL expressions, for instance a guard and a postcondition, the OCL syntax used to identify that element must be the same in the two expressions. In other words, it is required that model elements must have unique syntactic form. The reason is that when propagating constraints in an invocation subscenario, no semantic analysis is performed (adding some semantic analysis to this process will be considered in future work), and the process only relies on the syntax of the OCL expression to identify constrained elements. Consider the following example. The postcondition of operation *op1* and the invocation condition [*c2*] in Figure 16 are, respectively:

```

post: aSet->select(attr1=1 and attr2=2)->including(obj)
[aSet->select(attr1=1)->select(attr2=2)->size = 1]

```

The same model element is constrained in the postcondition (*aSet->select(attr1=1 and attr2=2)*) and in the guard (*aSet->*

`select(attr1=1)->select(attr2=2)`): i.e., the subset of `aSet` containing only objects whose `attr1` and `attr2` values are 1 and 2, respectively. The two constraints, `aSet->select(attr1=1 and attr2=2)` and `aSet->select(attr1=1)->select(attr2=2)`, are thus semantically equivalent but syntactically different. In this case, the test constraint derivation approach will fail because it cannot detect the effect of the operation (modeled by the postcondition) on the evaluation of the invocation condition. For the derivation of test constraints to perform as expected, the constrained element in the postcondition must be written in the same way (i.e., with the same syntax) as in the guard. In other words, the postcondition should be: `aSet->select(attr1=1)->select(attr2=2)->including(obj)`. Note that what matters is that both constraints contain the same syntax for the constrained element: i.e., the postcondition remaining unchanged, the guard can be written as `aSet->select(attr1=1 and attr2=2)->size=1`. In this work, we require that modellers ensure they use the same syntactic form for each constrained element in different constraints⁸.

3.3.3 Deriving Test Constraints for Orphan State-dependent Objects

Recall that an orphan state-dependent object is an object whose behavior does not depend on any other object state, i.e., the corresponding class does not have any association navigable to other classes. As a result, actions in transitions are necessarily operations of the object itself. Moreover, since multiple paths (subscenarios) in subtrees

⁸ Later in Section 3.4.4, we'll show that reducing the predefined OCL type operations to a minimum core set of operations, which has the same expressive power as the complete set of type operations, does not guarantee the unique syntactic form of a model element. Therefore, this uniqueness has to be ensured by the modeller.

for events in the IST are the result of multiple transitions possibly fired by actions on other objects (see Section 3.2.3), which is not possible in the case of an orphan state-dependent object, subtrees in an orphan IST amount to single branches. This is illustrated in Figure 19 (a), which shows subtrees for events `event1` and `event2` that only contain one path. This results in only one sequence of events/actions executed for a given TTS: In Figure 19 (a), `event1.action1.event2.action2.action3.event3`. For the purpose of test constraint derivation, the IST for an orphan state-dependent object can thus be simplified to reflect a unique sequence of events/actions, as illustrated in Figure 19 (b): The last action in the subtree for a given event in the TTS is linked to the next event.

As a result, the algorithm for deriving the test constraint to be fulfilled to execute a complete TTS for an orphan state-dependent object is not necessarily different from the one described in the previous section. It also proceeds in a recursive way from the bottom (i.e., the last event node in the TTS), up in the IST, and consists in the same kind of transformation of invocation conditions using postconditions. It is thus not detailed here.

Figure 19. IST for an orphan state-dependent object, an example

3.4 Normalization of OCL Expressions

The test constraints derivation based on an IST requires that model constraints (operation contracts, guards) written in OCL are in certain forms, defined as the *normalized forms*, in order to facilitate the analyses of both types of constraints. The transformation of OCL model constraints into their normalized forms is called *normalization*. We can distinguish between several types of normalizations for a model constraint expression.

First, normalization is necessary to unify the form of semantically equivalent model constraints for comparison purposes. A model constraint written in OCL has a constrained element⁹ of a certain type (pre-defined OCL types or user-defined model types) on which a set of operations is executed in a specified sequence. In order to compare two OCL constraints, it is necessary to first identify their constrained elements (as two constraints are comparable only when they affect the same constrained element) and then analyze the operations applied on them. A detailed analysis may reveal that two constraints are the same (they affect the same constrained element with semantically equivalent operations), consistent (they affect different constrained elements), redundant (one constraint semantically implies the other), or inconsistent (they affect the same constrained element and they logically contradict each other). However, extra work is needed before comparing constraints written in OCL because of several facts. First, local variables, constants and queries can be used in OCL constraint expressions. These constructs need to be replaced by their corresponding definitions (the definition of a query is its postcondition). Then, we need to identify whether constrained elements are

⁹ A constrained element is a model element or set of model elements that are affected by a constraint [UML01]).

equivalent across constraints¹⁰. However, this is not simple. OCL provides a navigation mechanism, which allows modellers to write expressions that starts from a context class and reaches other classes through associations. In a UML class diagram, two seemingly different navigation paths (Section 2.5.2) can result in the same collection of objects (Section 3.4.3). Furthermore, OCL offers a rich set of predefined operations. Though they provide the modeller with flexibility in writing model constraints and thus enhances the usability of OCL, these operations allow for the same constraint to be expressed in more than one way, thus make it difficult to identify relations among constraints. For instance, the same model constraint can be written using different collection operations (e.g., `aSet->size=0` and `aSet->isEmpty` are two syntactically different constraints that are semantically equivalent). For all the above reasons, normalization is performed before we derive the test constraints.

Second, normalization is needed to transform OCL model constraints into normal forms such as Conjunctive Normal Form (CNF) and Disjunctive Normal Form (DNF) [HR00]. This type of normalization is performed on both the operation contracts and the invocation conditions. Operation preconditions and invocation conditions must be in DNF; and operation postconditions are in the form of a conjunction of clauses or a conjunction of implies-expressions¹¹ (Section 3.4.2).

Besides, normalization is also required to transform the operation postconditions into a form that shows the relations between the pre-state and the after-state (Section

¹⁰ Two equivalent constrained elements refer to the same model element or set of model elements affected by constraints.

¹¹ A conjunction of implies-expressions is equivalent to a CNF because of the semantic equivalence between logical expressions “a implies b” and “(not a) or b”.

2.5.2) of the operation. For example, a postcondition of a `call` action is `self.a=self.b@pre+1`. It shows that attribute `a`'s after-value (part of the operation's after-state) equals attribute `b`'s pre-value (part of the operation's before-state) plus one. This type of normalization is performed on the operation contracts only.

In general, the OCL model constraints are normalized to facilitate two types of analyses:

1. Test constraints derivation: Since operation postconditions show the relations between the pre-state and the after-state, they allow the easy transformation from test constraints on the after-state to test constraints on the pre-state. Using the previous `call` action example, if the test constraints define its after-state to be `self.a>1`, then this constraint can be transformed to be `self.b@pre+1>1`. Thus, we can obtain the test constraints on the action's pre-state as `self.b+1>1`, which means, if `self.b>0` is true before executing the action, then `self.a>1` is also true after the execution.
2. Consistency checking among OCL expressions:
 - Check the consistency between an event or action's postconditions and the test constraints on the collective state: Are the postconditions conflicting with the test constraints? Are the test constraints satisfied by the postconditions?
 - Check consistency among constraints: Are two OCL expressions redundant/conflicting?

The subsections below are organized as follows. Section 3.4.1 describes the elimination of local variables, constants and queries. Section 3.4.2 explains required

normal forms for different types of model constraints. Section 3.4.3 then discusses the identification of equivalent navigation path to facilitate the identification of equivalent constrained elements. And Section 3.4.4 exemplifies the normalization of operations for different types of model elements so that the comparison of constraints is possible. Finally, Section 3.4.5 illustrates transforming postconditions to allow test constraint derivation.

3.4.1 Eliminating Local Variables, Constants and Queries

Local variables, constants and queries need to be eliminated from model constraints to enable their comparison. This is necessary for the test constraint derivation and consistency checking among constraints (Section 0). This step is performed during the stage of importing input files (described in Section 4.2) and can be fully automated. For instance, if a local variable `var` is defined as `self.a`, the equivalence between the two constraints `self.a>1` and `var>1` can be identified only after `var` is replaced by its definition (`self.a`). In the following, we discuss the elimination of the three constructs.

Eliminating Local Variable

OCL allows the use of the `let`-expression to define a local variable to represent an attribute or an operation that is used more than once in a model constraint, e.g., an invariant or pre- or postcondition [UML01]. Such a variable is a *local variable* because its scope is the constraint. Examples of local variables are:

```
context Person inv:
let a:Integer = self.age
a >= 0
```

```
context Account::debit(amount:Real)
let b:Real = self.getBalance()
pre: b >= amount
post: b = b@pre - amount
```

The first example shows an Integer local variable `a` defined to be the attribute `age` of class `Person`. The second example shows a Real local variable `b` defined as the operation `getBalance()` of class `Account`.

The elimination of let-expressions is straightforward. We replace all occurrences of the variable name with the expression that define them in the scope of its context.

Local variables may also be used in different consecutive OCL expressions, typically in a transition's action sequence. In this case, the local variable is a shorthand for a query in order to keep the action expressions simple. For example, in the statechart for a `Reservation` class in the Video Store System case study (Chapter 5), one of the transitions from state `Pending` to state `Outstanding` has the following actions.

```
v = self.title.getSecondOldestPending();
self.title.changeOldestPending(v);
v.beOldestPending()
```

Here `v` is a local variable defined by a query. To eliminate this type of local variable, every occurrence of the variable is replaced with the query that defines it. For example, after elimination the previous action sequence becomes:

```
self.title.changeOldestPending(self.title.
    getSecondOldestPending());
self.title.getSecondOldestPending().beOldestPending()
```

Eliminating Constants

Constants may be used in model constraint expressions. For example, in the `OrdSet` case study (Chapter 5), constant `max_accepted_resizes` defines the maximum times that an `OrdSet` object can be resized. The elimination of constants is done by replacing each occurrence of the constant with its value. The definitions of constants are provided by users, e.g., in a text file to support the elimination of constants.

Eliminating Queries

A query does not have any impact on the system state [UML01]. It simply returns a value or a set of values (the `result` keyword is used in the postcondition of the operation). One can consider two kinds of queries depending on whether the returned value depends on certain conditions, i.e., whether the query postcondition contains an if- or implies-expression. If it does, we call the query a *conditional query*. Otherwise it is referred to as a *simple query*. It is important to distinguish these two kinds of query since, as we will see below, their elimination is different. For a simple query, its elimination is straightforward: we replace every occurrence of the query with what is assigned to `result` in its postcondition. Therefore, the remainder of this subsection focuses on the elimination of conditional query only.

First note that an if-expression can be transformed into two implies-expressions that can themselves be transformed into DNF, as illustrated in Section 3.4.2. Let us look at an example. Assume the following is the postcondition of a conditional query, say `aQuery()`. The postcondition originally contains an if-expression (the one on the left), which is converted into two implies-expressions with mutually exclusive predicates (the one in the middle), which is further transformed into DNF (the one on the right). We therefore assume, in the rest of this discussion, the postcondition of any conditional query is in DNF.

post:	post:	post:
if c1	c1 implies	(c1 and result=expr1)
then result=expr1	result=expr1	or
else result=expr2	and	((not c1) and result=expr2)
endif	(not c1) implies	
	result=expr2	

The elimination of conditional queries depends on the type of model constraints. We discuss it below.

Eliminating Conditional Queries Inside Preconditions and Guards

A precondition or guard may contain conditional queries. For example, the following guard on the left contains the conditional query `aQuery()` as described above. We eliminate `aQuery()` inside the guard by replacing the guard with the query's postcondition, in which any occurrence of a result assignment expression¹² (e.g. "`result=expr1`" in the first postcondition disjunct) in the query's postcondition is replaced by the guard expression (i.e. "`aQuery()>expr`"), where the occurrence of the query name (i.e. "`aQuery()`") is replaced by the right-hand-side of the result assignment expression (i.e. "`expr1`"). The resulting guard is on the right.

<code>[aQuery() > expr]</code>	<code>[(c1 and expr1>expr)</code>
	<code>or</code>
	<code>((not c1) and expr2>expr)]</code>

Eliminating Conditional Queries inside Postconditions

A conditional query can be used in a postcondition in two different ways:

1. Conditional queries are used as the predicate part of and if-expression or implies-expression. As explained in Section 3.4.2, if-expressions are replaced by implies-expression, so only implies-expressions are accounted for below. Let us consider the following example. The postcondition on the left contains a conditional query `aQuery()` (same as above), and `expr3` and `expr4` are two Boolean expressions. We eliminate `aQuery()` with its postcondition and the result is shown on the right:

¹² A result assignment expression refers to a postcondition expression of the form "`result=...`".

<pre> post: aQuery() implies expr3 and not aQuery() implies expr4 </pre>	<pre> post: (c1 and expr1) or ((not c1) and expr2) implies expr3 and not((c1 and expr1) or ((not c1) and expr2)) implies expr4 </pre>
--	--

This postcondition can be further transformed. The predicate part of the second implies-expression can be transformed into CNF according to the De Morgan Laws and the Double Negation Rule, then further transformed into DNF according to the Distributive Law (Section 3.4.2) as follows:

$((\text{not } c1) \text{ and } (\text{not } \text{expr2})) \text{ or } (c1 \text{ and } (\text{not } \text{expr1}))$

Furthermore, the first implies-expression and the transformed second implies-expression can be transformed according to the following equivalence (Section 3.4.2), where a, b and c are Boolean expressions:

$(a \text{ or } b) \text{ implies } c \Leftrightarrow (a \text{ implies } c) \text{ and } (b \text{ implies } c)$

Therefore, the previous postcondition can be further transformed to be:

```

post:
(c1 and  expr1) implies expr3
and
((not c1) and expr2) implies expr3
and
((not c1) and (not expr2)) implies expr4
and
(c1 and (not expr1)) implies expr4

```

2. The conditional query `aQuery()` (same as above) is used as the new value of a model element in the postcondition (with name `post1`¹³) on the left. In this case, we eliminate `aQuery()` inside `post1` by replacing `post1` with the query's postcondition, in which any occurrence of a result assignment expression (e.g. "`result=expr1`" in the first postcondition disjunct) is

¹³ OCL allows a model constraint to have a name [UML01]. Here we give the name `post1` to the postcondition to distinguish it from the query's postcondition.

replaced by `post1` (i.e. “`expr=aQuery()`”), where the occurrence of the query name (i.e. “`aQuery()`”) is replaced by the right-hand-side of the result assignment expression (i.e. “`expr1`”). The resulting guard is on the right.

<pre>post post1: expr = aQuery()</pre>	<pre>post post1: c1 implies (expr = expr1) and (not c1) implies (expr = expr2)</pre>
--	--

3.4.2 Normalizing Logical Expressions

The purpose of normalizing logical expressions is to ease the test constraint derivation (Section 0) and consistency checking. This step is performed during the stage of building the IST and can be fully automated (Section 4.2).

According to [UML01], OCL logical expressions are OCL relational expressions connected by logical operators such as `and`, `or`, `xor` and `implies`. A *clause* is a logical expression that contains no logical operators and is basically a relational expression. A *predicate* is a logical expression that is composed of clauses and zero or more logical operators.

Typical normal forms for logical expressions are the Disjunctive Normal Form (DNF) and Conjunctive Normal Form (CNF). An expression is in DNF if it is a disjunction (sequence of ORs) of one or more disjuncts, each of which is a conjunction of clauses. An expression is in CNF if it is a conjunction (sequence of ANDs) consisting of one or more expressions in DNF. Using the Distributive Law [HR00], logical expressions in CNF can be converted into DNF or vice versa:

- (1) $a \text{ or } (b \text{ and } c) \Leftrightarrow (a \text{ or } b) \text{ and } (a \text{ or } c)$
- (2) $a \text{ and } (b \text{ or } c) \Leftrightarrow (a \text{ and } b) \text{ or } (a \text{ and } c)$

Using the De Morgan Laws [HR00], logical expressions in the form of a negation of a CNF/DNF can be converted into the disjunction/conjunction of the negated conjuncts/disjuncts:

$$\begin{aligned}\text{not } (a \text{ or } b) &\Leftrightarrow (\text{not } a) \text{ and } (\text{not } b) \\ \text{not } (a \text{ and } b) &\Leftrightarrow (\text{not } a) \text{ or } (\text{not } b)\end{aligned}$$

Using the Double Negation Rule [HR00], double negation of a logical expression can be eliminated:

$$\text{not } (\text{not } a) \Leftrightarrow a$$

Model constraints considered in this work include class invariants, operation contracts, and guards. In order to facilitate the test constraint derivation, this work requires that each type of model constraints are normalized to forms as follows:

- Class invariants: conjunctions of clauses
- Operation preconditions: DNF
- Operation postconditions: conjunctions of clauses and/or implies-expressions
- Invocation conditions in IST: DNF

Without losing generality, this work assumes modellers write model constraints in normal forms (if not, a model constraint can always be transformed into a normal form [HR00]). In practice, an invariant should already be in the form of a conjunction of clauses because it states what conditions must hold true at the same time in order to define a valid instance state. An operation precondition is either in DNF or a conjunction of clauses. Therefore, this work also assumes that invariants and operation preconditions are already in their required form, accordingly, there is no need to perform any normalization of logical expressions in them.

In the remainder of this section, we'll focus on normalizing logical operations in postconditions. An operation postcondition often contains different clauses that must all be satisfied when the operation completes, and is thus a conjunction of clauses (e.g., an attribute's value is updated *and* a link between two objects is changed). Sometimes a postcondition may contain structures that are more complex. For instance, it is common that the evaluation of a postcondition depends on the inputs of the operation (e.g., operation's arguments) or the current system state. Such a situation is easily expressed in OCL with logical operators `implies` or `if-then-else`. There are three steps in normalizing the logical operations inside postconditions and they are described below.

Step 1: Replacing if-expressions by implies-expressions

The normalization step first eliminates the if-expressions by converting them into implies-expressions in the postcondition. For example, the following two postconditions are equivalent (we use `post1` and `post2` to distinguish two postconditions in Table 2).

<pre> post post1: if c1 then expr1 else expr2 endif </pre>	<pre> post post2: c1 implies expr1 and (not c1) implies expr2 </pre>
--	--

c1	c1 implies expr1	(not c1) implies expr2	post1	post2
T	expr1	T	expr1	expr1
F	T	expr2	expr2	expr2

Table 2. Truth table of an if-expression and its equivalent implies-expressions

Step 2: Transforming the predicate part of implies-expressions into a conjunction of clauses

The predicate part of the implies-expression might be in a complex form such as DNF, CNF or may simply not be normalized. Since any logical expression can be

converted into CNF, and then into DNF [HR00], the predicate part of an implies-expression can always be converted into DNF when it is not simply a clause. If the predicate of an implies-expression is in DNF, the normalization step needs to transform the implies-expression so that its predicate only consists of a conjunction of clauses. For instance, we may encounter the following postcondition, where *c1* and *c2* are conjunctions of clauses.

```
post post1: (c1 or c2) implies expr
```

Such implies-expression can be split into several (one for each disjunct) implies-expressions so that the predicate part for each implies-expression is a conjunction of clauses (in our example, simply a clause). The result is as follows:

```
post post2:
c1 implies expr
and
c2 implies expr
```

The equivalence between the two postconditions (*post1* and *post2*) can be shown using the following truth table.

c1	c2	c1 or c2	c1 implies expr	c2 implies expr	post1	post2
F	F	F	T	T	T	T
F	T	T	T	expr	expr	expr
T	F	T	expr	T	expr	expr
T	T	T	expr	expr	expr	expr

Table 3. Truth table of two equivalent implies-expressions

Step 3: Identifying alternative postconditions

After step 1 and step 2, the general form of a postcondition is a conjunction of clauses and/or implies-expressions whose predicate part and implied part are both conjunctions of clauses. The next step is to re-express the implies-expressions. This step replaces all the original implies-expressions with new implies-expressions whose

predicates are truth combinations of predicates of all the original implies-expressions. The number of the new implies-expressions is two to the power of the number of original implies-expressions. In this way, the predicate parts of any two new implies-expressions are mutually exclusive. We define the implied parts of these implies-expressions as the *alternative postconditions*. If there is no implies-expression in the postcondition, the number of alternative postcondition is one.

Let us consider the following example,

```
post:
  c1 implies expr1
  and
  c2 implies expr2
  and
  c3 implies expr3
  and
  expr4
```

The original implies-expressions in the postcondition can be re-expressed as $2^3 = 8$ new implies-expressions as follows. Note that the implied part of the first conjunct is Boolean value `true`. It denotes that the operation does not modify the collective state when none of `c1`, `c2` or `c3` is true and thus it has the weakest postcondition, which is `true`.

```
post:
  ((not c1) and (not c2) and (not c3)) implies true
  and
  (c1 and (not c2) and (not c3)) implies expr1
  and
  ((not c1) and c2 and (not c3)) implies expr2
  and
  (c1 and c2 and (not c3)) implies (expr1 and expr2)
  and
  ((not c1) and (not c2) and c3) implies expr3
  and
  (c1 and (not c2) and c3) implies (expr1 and expr3)
  and
  ((not c1) and c2 and c3) implies (expr2 and expr3)
  and
  (c1 and c2 and c3) implies (expr1 and expr2 and expr3)
  and
  expr4
```

Note that clause `expr4` is not included in any `implies-expression`. Since now the predicates of all the new `implies-expressions` are mutually exclusive, `expr4` can be combined in conjunction with the implied parts of each `implies-expression`. The postcondition can be re-expressed as follows.

```

post:
  ((not c1) and (not c2) and (not c3)) implies expr4
and
  (c1 and (not c2) and (not c3)) implies (expr1 and expr4)
and
  ((not c1) and c2 and (not c3)) implies (expr2 and expr4)
and
  (c1 and c2 and (not c3)) implies (expr1 and expr2 and expr4)
and
  ((not c1) and (not c2) and c3) implies (expr3 and expr4)
and
  (c1 and (not c2) and c3) implies (expr1 and expr3 and expr4)
and
  ((not c1) and c2 and c3) implies (expr2 and expr3 and expr4)
and
  (c1 and c2 and c3) implies (expr1 and expr2 and expr3 and expr4)

```

An observation is that, if a postcondition contains `if-expressions` or `implies-expressions` before normalization, then after normalization, it will be a conjunction of mutually exclusive `implies-expressions`. Otherwise, the postcondition is a conjunction of clauses.

It should be noted that if an operation is a query and its postcondition is a conjunction of `implies-expressions`, we further transform the postcondition into DNF in the following way: Replace each `implies-expression` by the conjunction of its predicate part and implied part, and change all the logical AND operators that connect the `implies-expression` to logical OR operator. Let us look at the following example, where `c`, `expr1` and `expr2` are all conjunctions of clauses. The postcondition on the left can be transformed into the postcondition on the right (assume `expr1` is mutually exclusive to

expr2, which is reasonable because when modeling a postcondition, we expect that two mutually exclusive conditions would lead to two mutually exclusive results).

<pre> post: c implies expr1 and (not c) implies expr2 </pre>	<pre> post: (c and expr1) or ((not c1) and expr2) </pre>
--	--

On the other hand, if an operation is not a query or it is a query whose postcondition does not have implies-expressions (which means it is a conjunction of clauses), then we do not transform the postcondition into DNF. The implies-expressions provide necessary information for deriving test constraints (Section 0).

3.4.3 Normalizing Navigation Path of Constrained Elements

The goal is to uniquely identify constrained elements, i.e., a model element or collection of model elements contained in an OCL expression so that relations among constraints (are they redundant, consistent, or conflicting?) are identified. In general, both the navigation path and the name of the model element (e.g., an attribute) are needed for identifying a constrained element. For instance, assume in the Video Store System case study (Chapter 5), we have a constraint as follows: where *self* refer to an instance of class *Copy*:

```
self.reservation.title.name="b" and self.title.name="b"
```

The relation between the two conjuncts is unclear. However, further analysis reveals that the title of the reservation that is on hold for a copy is exactly the same as that copy's title. Hence, the first conjunct's navigation path (*self.reservation.title*) refers to the same *Title* instance as what is referred to by the navigation path of the second conjunct (*self.title*). Since the two conjuncts also constrain the same

property (in this case the attribute name) of the `Title` instance, they are redundant to each other. We say two constrained elements are equivalent when they have the same navigation path and they refer to the same object property.

This type of normalization is necessary when the class under test is part of a class cluster and its instances' behavior depends on the collective state (Section 2.4.1). Normalizing navigation paths is performed during the tree construction stage of the IST and can be fully automated using a helper file provided by the user that specifies equivalent navigation paths (Section 4.2). This step first unifies the context object in navigation path of action expressions, constrained elements in invocation conditions in the IST, and constrained elements in operations' contracts. Navigation paths are then simplified by replacing them with equivalent simplest paths.

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 [WK99]. A constrained element has a navigation path that starts from the context of the constraint, possibly followed by a sequence of association-ends until the class that owns the property is reached (Section 2.5.2). 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 unifying the context object of each constrained element's navigation path helps to identify equivalent constrained elements. 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 (such as guards and operation contracts for events and actions) already have the owning object as the context object of

their navigation paths, 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 Figure 35. The action of the transition is a call to operation `copyAvailable()` on the oldest pending reservation object. In Figure 36, the 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 Figure 35 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)
]
```

Simplifying Navigation Paths

Unifying the context object of all the navigation paths is a necessary but not a sufficient step to identify whether two navigation paths are equivalent. There are two reasons for this. First, after the context object of an OCL expression is changed to the owning object, redundancy *within* a navigation path might occur (e.g., the navigation path of the first conjunct in the above guard). Moreover, there might exist redundancy *among* navigation paths, for example, in a class diagram produced during the Design stage, additional associations are added due to performance considerations, thus introducing redundancy in paths in the class diagram [BD00]. We simplify the navigation paths by (1) removing redundancies within a navigation path and (2) replacing a navigation path by its simplest equivalent path, to remove the two kinds of redundancy introduced above.

Removing Redundancy within a Navigation Path

Assume there exists an 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.

Now 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 (the class diagram is in Figure 34):

```
(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 equation holds.

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

Therefore, we can simplify the previous guard using these two equation (by replacing all occurrences of the l-value of the equation by the r-value)¹⁴ and the resulting guard is:

```
[
  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 example above, this first step can reveal constraints that are always true. Therefore, extra checking is performed to identify and remove such constraints.

Replacing a Navigation Path with Simplest Equivalent Path

Different navigation paths starting with the same context that result in the same collection of objects are said to be redundant since they provide several different ways to retrieve the same information. For instance, in Figure 34 paths `self.title` and `self.reservation.title`, which both have the `Copy` object as context, are redundant. Though at the Analysis stage redundant navigation paths are often avoided,

¹⁴ 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`.

they may be added at the Design stage when performance issues are considered [BD00]. In such a situation, 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 our work, the simplest navigation path is simply the shortest, i.e., the path that navigates the least number of associations.

This replacing of a navigation path with a simplest equivalent path can be extended to situations where the paths, starting with the same context, result in collections that have an include relationship: In Figure 34, for example, given that `Copy` is the context, the collection resulting from `self.title.reservation` includes the collection resulting from `self.reservation`. The former collection contains all the `Reservation` objects for the title of the `Copy` object whereas the latter collection contains the `Reservation` object for which the `Copy` object is held (in this case, the `Reservation` object is in state “Outstanding”). In such a situation, there exists an operation (or operations) that can be applied to the biggest collection and results in the smaller collection. In Figure 34 for instance, collection resulting from the following two OCL expressions are identical:

```
(1) self.title.reservation->
    select (v:Reservation | v.state=#Outstanding)
(2) self.reservation
```

Then any occurrence of the navigation path followed by the operation that result in a smaller collection (in our example, (1)) can be replaced by the other navigation path (in our example, (2)).

It is clear that the identification of these situations (the simplest, first one), where simplifications are possible, cannot be fully automated since the semantics of the

associations has to be accounted for. In our work, we thus require these situations to be identified by the user. After unifying the context object for navigation paths (see previous discussion), it is usually possible to exhaustively enumerate all the equivalent navigation paths because their number is generally small. In this thesis, information about equivalent navigation paths is provided in a file, which defines transformation rules. Each rule defines a navigation path in terms of its equivalent simplest path. When simplifying the navigation path of a constrained element, these rules are applied in an iterative way. If no rule can be applied after all the rules are attempted, then the navigation path is already the simplest. Otherwise, a new round of iteration must be performed after all the rules have been iterated. This process terminates when after a new round of iterations no rules can be applied any more.

3.4.4 Normalizing Operations on OCL Types

Recall that in order to identify the relations (see beginning of this chapter) among constraints, we need to (1) uniquely identify each constrained element and (2) uniquely identify the operations that affect constrained elements (Section 3.4). The first step is addressed in the previous section. In this section, we discuss how to uniquely identify operations on constrained elements that are of predefined OCL types¹⁵ (basic or collection types).

The normalization of operations is based on the fact that many operations of predefined OCL types, both basic and collection types, can be expressed through more basic operations. For instance, when two values r_1 and r_2 of type `Real` are compared,

¹⁵ We do not normalize operations in user-defined model types.

$r_1 \leq r_2$ can be re-expressed as $(r_1 < r_2)$ or $(r_1 = r_2)$, 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 have the same expressive power as the complete set of operations.

The choice of the core operations is worth discussion. Note that our aim is to enable the test constraint derivation approach described in Section 3.3.2. This approach relies solely on the syntax to identify the semantic equivalence between two constrained model elements. There are two extreme situations: (1) the core operations include all OCL type operations so no operations are normalized; (2) find the minimum set of operations, that is, no core operations can be re-expressed by other core operations.

(1) is not a good choice because usually a reduced set of core operations helps to reduce the effort in analyzing the resulting test constraints. For example, it is useful to re-express operation `isEmpty` with `size=0` so that we know the two test constraints `aSet->isEmpty` and `aSet->size=0` are equal.

(2) is a good choice for our problem in general, except for some OCL types. First, the `implies` operation of the Boolean type is necessary for the test constraint derivation process. Indeed, the test constraint derivation algorithm expects the operation postcondition to be either a conjunction of clauses or a conjunction of `implies`-expressions (Section 0)¹⁶. Therefore the `implies` operation, though not primitive, must be kept in the core operation set. Second, iterative collection operations, such as the

¹⁶ This is ensured by normalizing logical expressions inside the postconditions (Section 3.4.2) and by eliminating queries (Section 3.4.1).

operations `select`, `collect`, and `sum`, enhance the readability of the resulting test constraints (more readable than their `iterate` counterpart), which might be presented to the user for further analysis. Though these operations can be re-expressed by the primitive operation `iterate`, the re-expressed form is much more complex and thus harder to understand by a human. The only reason why we may want to re-express these operations with `iterate` is that we hope this re-expressing can unify the syntactic form of two semantically equivalent constrained elements. However, this is not always the case, i.e., two semantically equivalent but syntactically different expressions involving `select` (or `collect`, `sum`) may have different syntactic normalized forms. Consider for instance, the two following semantically equivalent but syntactically different constrained elements: `aSet->select(attr1=1 and attr2=2)` and `aSet->select(attr1=1)->select(attr2=2)`. After re-expressing `select` with the operation `iterate`, the two constrained elements are as follows, respectively:

```

aSet->iterate(elem;result=Set{|
  if (attr1=1 and attr2=2) then
    result->union(Set{elem})
  else
    result
  endif
})

aSet->iterate(elem;result=Set{|
  if (attr1=1) then
    result->union(Set{elem})
  else
    result
  endif
})->iterate(elem;result=Set{|
  if (attr2=2) then
    result->union(Set{elem})
  else
    result
  endif
})

```

We can see that the two normalized forms look very different in syntax. Therefore we conclude that re-expressing iterative collective operations with `iterate` does not necessarily unify the syntactic form of semantically equivalent model elements¹⁷ while complicating the resulting test constraint. Therefore, we keep the iterative collection operations `select`, `collect`, and `sum` in the core operation set.

The normalization of operations can be automated and is done on both the operation contracts and the guards when operation contracts and statecharts are imported by our prototype tool (Section 4.2). Though the normalized expressions will usually be more complex, this added complexity is acceptable because normalized OCL expressions are only manipulated by the tool and will not be visible to the user.

Two types of operations are identified, and we provide the rules for the normalization in the next subsections.

- Operations on basic-types
- Operations on collection-types

The normalization rules are structured: First the type name is given; Then the core operations are defined; Finally all the operations that can be re-expressed with core operations are listed with the normalized expression.

3.4.4.1 Normalizing Operations on Basic Types

The normalization rules of basic operations are straightforward, as shown below.

Boolean type

- Core operations

¹⁷ This is why this research requires modelers to use the unique syntatic form to refer to a model element in any constraint (Section 3.3.2)

`=, and, or, not, implies`

- Operations re-expressed through core operations

```
b xor b2 ⇒ not (b = b2)
if b then b1 else b2 endif ⇒ (b implies b1)
                           and (not b) implies b2
```

Real type

- Core operations

`=, >, +, *, /, -(unary), floor, round`

- Operations re-expressed through core operations

```
r <> r2 ⇒ not (r = r2)
r - r2 ⇒ r + (-r2)
r < r2 ⇒ r2 > r
r >= r2 ⇒ r > r2 or r = r2
r <= r2 ⇒ r2 > r or r2 = r
r.abs() ⇒ (r > 0 implies result=r)
          and
          (not (r > 0) implies result=-r)
r.max(r2) ⇒ (r > r2 implies result=r)
            and
            (not (r > r2) implies result=r2)
r.min(r2) ⇒ (r > r2 implies result=r2)
            and
            (not (r > r2) implies result=r)
```

Integer type

- Core operations

`=, >, +, *, /, -(unary)`

- Operations re-expressed through core operations

```
i <> i2 ⇒ not (i = i2)
i - i2 ⇒ i + (-i2)
i < i2 ⇒ i2 > i
i >= i2 ⇒ i > i2 or i = i2
i <= i2 ⇒ i2 > i or i2 = i
i.abs() ⇒ (i > 0 implies result=r)
          and
          (not (i > 0) implies result=-r)
i.max(i2) ⇒ (i > i2 implies result=i)
            and
            (not (i > i2) implies result=i2)
```

```

i.min(i2) ⇒ (i > i2 implies result=i2 )
           and
           (not (i > i2) implies result=i)
i.div(i2) ⇒ (i/i2 > 0 implies result=(i/i2).floor())
           and
           (not (i/i2 > 0) implies result=-((-i/i2).floor()))
i.mod(i2) ⇒ i - (i.div(i2)*i2)

```

String type

- Core operations

```
=, size, concat, toUpper, toLower, substring
```

All operations on string type are core operations and there is no operations need to be re-expressed.

Enumeration type

- Core operations
- =
- Operations re-expressed through core operations

```
enumeration <> enumeration2 ⇒ not (enumeration = enumeration2)
```

3.4.4.2 Normalizing Operations on Collection Types

The normalization of collection operations is much more complex than the normalization of basic type operations. Before proceeding we first provide the required background information on collection operations. The collection type system in OCL includes abstract type `Collection` and its concrete subtypes `Set`, `Bag` and `Sequence`. Some of the operations on OCL collections are common to all concrete subtypes and are thus defined by `Collection` (e.g., operation `isEmpty()` returns true if the collection contains no element), whereas others are specific to the concrete subtypes (e.g., type `Sequence` is the only one that provides an operation that results in an element at a given position). The latter category of operations also includes some operations that have the same name but different semantics for each subtype: For

instance, Set, Bag and Sequence all define the excluding operation but the operation will behave differently for each subtype.

It is thus often necessary to know the concrete type of a collection when collection operations are applied. The concrete collection type can be identified as follows:

- (1) If the collection is obtained by navigating associations, human help is needed to specify the concrete type of a collection as this requires information on the semantics of the associations traversed. And this is possible since there is limited number of normalized navigation paths from the state machine context class to all other classes in the class cluster. If not specified, according to [UML01], by default a single navigation results in a Set, combined navigations result in a Bag, and navigation over ordered associations results in a Sequence.
- (2) If the collection is obtained by a collection operation that returns a new collection. The collection operation specifies the type of the resulting collection: for instance, `aSet->including(obj)` returns a Set object, and `aSet-> collect (expr)` returns a Bag object [UML01].

Two types of collection operations are identified in this thesis: Type (1) operations that return a new collection, Type (2) operations that 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 collection-type constrained elements, which is discussed in the beginning of Section 3.4. 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 checking between OCL expressions involving collection operations. There are two collection operations that return the numeric properties of the collection: The `size` operation returns the number of all the elements in the collection. The `count` operation returns the number of occurrences of a specific element in the collection. Later in Section 3.4.4.2, we will show how most Boolean collection operations can be re-expressed with operations `size`, `count`, and `select`.

We now provide the normalization rules for Type (1) collection operations¹⁸ and Type (2) collection operations in sequence.

Type (1) collection operations on Set

- Core operations

```
iterate, union, intersection, -, select, collect, sum,
asSequence, asBag
```

- Operations re-expressed through core operations

```
set->including(object) ⇒ set->union(Set{object})
set->excluding(object) ⇒ set - Set{object}
set->symmetricDifference(set2) ⇒ (set-set2)->union(set2-set)
set->reject(expr) ⇒ set->select(not expr)
```

Type (1) collection operations on Bag

- Core operations

```
iterate, union, excluding, intersection, select, collect, sum,
asSequence, asSet
```

- Operations re-expressed through core operations

```
bag->including(object) ⇒ bag->union(Bag{object})
bag->reject(expr) ⇒ bag->select(not expr)
```

¹⁸ There is an operation common to all concrete collection subtypes, namely, `any(expr:OclExpression)`. This operation returns any element in the collection for which `expr` evaluates to true [UML01]. Since result of this operation is non-deterministic, it is not considered in this thesis.

Type (1) collection operations on Sequence

- Core operations

`iterate, union, excluding, subsequence, select, collect, at, sum, asBag, asSet`

- Operations re-expressed through core operations

`sequence->append(object) ⇒ sequence->union(Sequence{object})`
`sequence->prepend(object) ⇒ Sequence{object}->union(sequence)`
`sequence->first() ⇒ sequence->at(1)`
`sequence->last() ⇒ sequence->at(sequence->size())`
`sequence->including(object) ⇒ sequence->union(Sequence{object})`
`sequence->reject(expr) ⇒ sequence->select(not expr)`

Type (2) collection operations

- Core operations

`size, count, select, includesAll, excludesAll, isUnique`

- Operations re-expressed through core operations

`collection->isEmpty() ⇒ collection->size = 0`
`collection->notEmpty() ⇒ collection->size > 0`
`set->includes(object) ⇒ set->count(object) = 1`
`bag->includes(object) ⇒ bag->count(object) > 0`
`sequence->includes(object) ⇒ sequence->count(object) > 0`
`collection->excludes(object) ⇒ collection->count(object) = 0`
`collection->exists(expr) ⇒ collection->select(expr)->size > 0`
`collection->one(expr) ⇒ collection->select(expr)->size = 1`
`collection->forAll(expr) ⇒ collection->size>0 and
collection->select(not expr)->size=0`

3.4.5 Normalizing Postconditions

Recall from Section 4.3 that the general issue in this backward process is, using the example in Figure 20 (a), to transform operation after-state constraint `invocCond2` into a constraint on the state before the execution of `myOp`, that is a constraint to be combined with `invocCond1`. `f(post, invocCond2)` is the re-writing of `invocCond2` given that some of the elements constrained in `invocCond2` are likely

changed by `myOp` according to `myOp` postcondition `post`. For instance, if `invocCond2` equals `self.attrib>1` and `post` equals `self.attrib=self.attrib@pre+2`, then `f(post, invocCond2)` equals `self.attrib+2>1`. When this is recursively done from the bottom of the invocation subscenario (i.e., a branch in the invocation sequence subtree) to the top, we eventually (when we reach the root node of the invocation subtree) derive the constraint that allows the execution of the subscenario.

To enable the necessary substitution between postconditions and invocation conditions, postconditions must be normalized: they must be transformed into a specific format that will allow substitutions (Section 3.4.5.1) and they must be complemented (Section 3.4.5.2).

Figure 20. Invocation sequence tree and constraint derivation (illustration)

3.4.5.1 The Standard Equality Form

We first provide in this section definitions of the Standard Equality Form that must be used when normalizing postcondition, and then justify its use.

Definition: Standard Equality Form for a postcondition

Let us assume a postcondition is written as a conjunction of clauses or implies-expressions whose implied parts are also conjunctions of clauses.

Such a postcondition is in the Standard Equality Form if and only if all its conjunctions of clauses (possibly in an implied part) are in the Standard Equality Form.

Definition: Standard Equality Form for a conjunction of clauses

A conjunction of clauses is in the Standard Equality Form if and only if all its clauses are in the Standard Equality Form.

Definition: Standard Equality Form for a clause

A clause 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 at the start of the execution of the operation (i.e., @pre values).

Some examples and counter-examples of clauses in the Standard Equality Form are provided in Table 4.

Clauses in the Standard Equality Form	Clauses not in the Standard Equality Form
attribute=10 attribute=parameter+1 attribute=otherAttribute@pre+1 set=Set{object1, object2} set->size = set@pre->size+1 collection->size=0	attribute-10=0 attribute@pre+1=otherAttribute a=b+1 asset->includes(obj) collection->isEmpty

Table 4. Examples of clauses in (or not in) the Standard Equality Form

Note from Table 4 that clause $a=b+1$ is not in the Standard Equality Form as, using that clause only, one doesn't know whether model element b's value is changed by the operation. For instance, if another clause in the operation's postcondition is

$b=c@pre$, then the Standard Equality Form for the clause above is $a=c@pre+1$. If on the contrary b 's value is not changed by the operation, then the Standard Equality Form is $a=b@pre+1$. Note also that $collection->isEmpty$ is not in the Standard Equality Form whereas the equivalent expression $collection->size=0$ is. The Standard Equality Form is one of the main reasons for the normalization rules of OCL collection operations introduced in Section 4.4.2.

This Standard Equality Form is necessary to ease the transformation of an invocation condition after the execution of an operation into a constraint on pre-state (Section 2.5.2). Once postconditions are transformed into the Standard Equality Form, one is typically in the following situation:

- The post condition includes a clause that describes, under the form of an equation, the new value of a model element (i.e., upon completion of the execution of the operation) in terms of values of model elements before the operation executes. For instance, $att1$ and $att2$ being two attributes of type Integer, $att1=att2@pre+2$ (1).
- The invocation condition to be considered after the operation constrains an element that appears in one of the postonditions clauses as an l-value. For instance, $att1>5$ (2).

Using expressions (1) and (2) above, it is easy to derive a condition that must be fulfilled before the execution of the operation: the r-value in (1) replaces in (2) all the occurrences of the l-value in (1), the $@pre$ is removed, and we obtain $att2+2>5$.

Finally, note that transforming a postcondition into its Standard Equality Form can be automated as this entails replacing occurrences in r-values with l-values of other clauses.

3.4.5.2 Complementing Postconditions

Assume that, in the invocation scenario in Figure 20, operation myOp 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., the OCL collection type of `roleName` is `Sequence`). Further assume that, following the node representing operation myOp in the invocation scenario, the invocation condition (i.e. `invocCond2`) 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 the myOp) into a constraint on the size of the sequence before the execution of myOp. As described in Section 0, 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 facilitate/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 used and the OCL type 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:

```

roleName=roleName@pre->append(obj)
and roleName->size=roleName@pre->size+1
and roleName->at(roleName->size)=obj
and 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 basically an expression describing the result (or postcondition) of the OCL collection operation as stated in [WK99] (Appendix A). We adapt these results for this work and call the adaptations *postcondition complementing rules*.

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., all changes caused by an operation are modelled and it is clear what is changed and what is not). However, this is not an easy task because there is no set of rules for how to use OCL to model operation contracts. The flexibility provided by OCL implies that trying to exhaustively enumerate all possibilities might be impossible. Therefore, we assume that modellers use OCL in a guided manner when they write operation contracts. We define specific forms for modeling a particular change to certain types of model elements and we call these forms *contract patterns*. Such patterns are designed with the aim to be correct, complete,

concise and easy to understand and use. For each modeling 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.

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

3.4.6 Advantages and Limitations

As discussed in previous sections, the normalization of the OCL expressions has a number of advantages. It unifies the expression of semantically equivalent model constraints (Section 3.4.1, 3.4.3, and 3.4.4) so that each distinct constraint can be uniquely expressed. It further converts each type of model constraints (preconditions, postconditions, and guards) into appropriate normal forms (Section 3.4.2) and transforms the postcondition (Section 3.4.5) to enable the automated test constraints derivation. Furthermore most of the normalization steps themselves can be automated at this point (as demonstrated in our prototype tool, Chapter 4). A potential benefit is that the normalized forms of postconditions provide suggestions (in the form of patterns) on how to write good operation contracts so that it's complete and concise and amenable to testing.

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. Yet theorem-proving systems (i.e. theorem provers) for OCL are not readily available. Even there are such theorem provers, the cost of applying them can conceivably be very high¹⁹. 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, the normalization technique is still useful for two reasons. First, in some 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 `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. The tester can manually identify such relations among constraints. The normalization techniques make this work much easier. Moreover, it might 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

¹⁹ Recall that the test constraint derivation process iterates all the invocations in an invocation subscenario (Section 0) 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 increase the burden of the theorem prover.

are obtained), when we apply heuristic constraint solving techniques such as Genetic Algorithms (GA), the redundancies within constraints might delay the convergence of the algorithms but will not prevent it from finding the solution. And in cases where there are inconsistencies among these constraints, algorithms like GA will not converge, and we can claim with a certain degree of confidence, though not for sure, that there exist inconsistencies that make the test constraints unsolvable.

Chapter 4 – 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.

4.1 Overview

CBCDTool is implemented in Java, and implements all the core functionalities except for part of the normalization steps as explained in Sections 3.4.2 and 3.4.5. In its current version the tool does not have a Graphic User Interface (GUI) and users need to interact with the tool through command-line arguments. However, since the tool is a proof-of-concept, a GUI is not an essential part and it can easily be added in a later version. The tool contains 377 classes, in which 57 classes and 2690 Line of Code (LOC) are implemented for the purpose of this thesis and the remaining classes are reused from an OCL compiler for OCL version 1.3 [Dresden02]. There are four main subsystems in the tool, namely: `umlModel`, `ist`, `control`, and `ocl`. The `umlModel` subsystem contains the UML class diagram and statechart diagram metamodels. The `ist` subsystem contains the IST metamodel. The `control` subsystem contains classes for the test constraint derivation algorithm. The `ocl` subsystem is borrowed from Finger's OCL compiler [Finger00]. Moreover, additional classes from the same source are reused to facilitate the analysis of OCL expressions.

The organization of this chapter is as follows. Section 5.2 specifies the requirements of the CBCDTool and describes the inputs and outputs of the tool. Section

5.3 explains the subsystem decomposition and describes each subsystem in detail. Section 5.4 describes how to use CBCDTool to perform the task of deriving test constraints for UML statecharts.

4.2 Requirements of CBCDTool

We would like the CBCDTool to meet the following functional requirements:

- Import and parse files that define UML models, such as the class diagram and statechart diagram; Import the file that specifies transition test sequences (TTS).
- Import and parse helper files (Section 3.4.3), which provide necessary information that guide the tool to proceed, such as equivalent navigation paths, subset relations among collections.
- Build ISTs from UML models and a specific TTS. Perform normalization steps. Create subscenarios for each event node in an IST.
- Derive test constraints for the IST.

Figure 21 is an overview of CBCDTool. The inputs required by the tool consist of the following:

- The operation contract file, which defines the precondition and postcondition of each operation involved in the UML statecharts under consideration.
- The UML statechart specification file, which defines a set of UML statecharts.
- The UML class specification file, which defines a class cluster. For each class, both the attributes and operation signatures are defined. The file also specifies all the associations among classes. Each association has two association ends with a role name and a multiplicity.
- The TTS specification file, which specifies the TTSs considered for testing

- Helper files provided by user, i.e., a file that specifies the equivalent navigation paths in the class diagram and a file that specifies the subset relation among collections obtained from navigating associations.

Figure 21. Overview of CBCDTool

The sequence of steps that the CBCDTool follows to derive test constraints are shown in Figure 22.

Figure 22. Execution steps of CBCDTool

The first step during the execution of CBCDTool is to read the input files and instantiate the corresponding metamodels (Section 4.3.1). When reading the operation contract input file, normalization is performed by eliminating local variables defined by

let-expressions (Section 3.4.1) and eliminating if-expressions (Section 3.4.2). Then CBCDTool builds an IST from the model information and each specified TTS. Within this step, several normalization steps are performed, including: eliminating local variables defined by query operations in the action sequence (Section 3.4.1); normalizing the navigation path of action expressions, OCL expressions in action handler's contracts, and OCL expressions in invocation conditions (Section 3.4.3); normalizing the operations in operation contracts and invocation conditions (Section 3.4.4); normalizing the postconditions (Section 3.4.5). The tool then creates invocation subscenarios for each IST, and finally derives test constraints for each of the invocation subscenario in the IST. Such constraints can then be used to derive test data for the selected TTS.

All these tasks have been automated except for the normalization of logical expressions and postconditions. These functionalities will be added to the utility class `ConstraintNormalizer` in the `control` subsystem if it is implemented.

4.3 Subsystem Decomposition

Following the system analysis and design principle of “tight cohesion within a subsystem” and “loose coupling among subsystems”, the CBCDTool system can be decomposed into four subsystems, as shown in Figure 23. Each of the subsections provides a synopsis of these subsystems.

4.3.1 control Subsystem

The control subsystem (Figure 24), is responsible for the overall control flow of the application and contains eleven classes. One class, `CbcdController`, is the control class that defines the main control flow of our approach. Five classes,

Contract, EquivalentPath, ClassCluster, StateMachineCluster, TTS and ISTBuilder are entity classes that reads data from input files and store the data so they can be tackled by control classes.

Figure 23. CBCDTool subsystem decomposition

Figure 24. control subsystem

In the current version of the tool, both the helper files and model files are text files that require particular format. It is possible to extend the tool to import model files

in other formats such as the standard XMI format [XMI00]. The `Contracts` class loads and stores operation contracts. The `Equivalent` class is an entity class that loads user-supplied helper files into the system. The `ClassCluster` class loads and stores the class diagram of a class cluster. The `StateMachineCluster` class loads and stores the statecharts for a state machine cluster. The `TTS` class loads and stores the TTS. Three other control classes, `ISTBuilder`, `ConstraintNormalizer`, and `ConstraintBuilder` execute detailed control steps. The `ISTBuilder` class builds the IST from information stored in the above four entity classes. During the tree-building process, the class creates an instance of `ConstraintNormalizer` class to perform the normalization of OCL expressions. The `ConstraintBuilder` class derives test constraints based on the IST created by `ISTBuilder` class.

4.3.2 umlModel Subsystem

The `umlModel` subsystem is an adaptation from the official OMG UML standard version 1.4 [UML01]. The adaptation consists of only those subsystems and corresponding metaclasses that are of interest to this work. The `umlModel` subsystem is shown in Figure 25. It contains three subsystems, namely: `foundation`, `behaviouralElements` and `modelManagement`. The `modelManagement` package defines grouping units, such as `Model`, `Package`, and `Subsystem`, for other model elements and thus it is not directly relevant in the context of `CBCDTool`. Therefore, only the first two subsystems are implemented.

Figure 25. umlModel subsystem

4.3.2.1 foundation Subsystem

The foundation subsystem is the language infrastructure that specifies the static structure of models [UML01]. As illustrated in Figure 26, the foundation subsystem is decomposed into three subsubsystems: core, extensionMechanisms, and dataTypes. In the context of CBCDTool, we are only interested in subsystems core and dataTypes (further described in the following on subsections) and not the extensionMechanisms subsystem that specifies how UML model elements can be extended with new semantics [UML01].

Figure 26. foundation subsystem

4.3.2.1.1 core Subsystem

The following description of the core subsystem is taken from [UML01]. The core subsystem is the most fundamental of the subsystems that compose the UML Foundation subsystem. It defines the basic abstract and concrete metamodel constructs needed for the development of object models. Abstract constructs include `ModelElement`, `GeneralizableElement`, and `Classifier`. Concrete constructs include `Class`, `Attribute`, `Operation`, and `Association`.

Figure 27 is the class diagram for the core subsystem. It is adapted from [UML01] and only shows those constructs that are relevant to CBCDTool.

Figure 27. core subsystem class diagram

The abstract constructs used are: `ModelElement`, `GeneralizableElement`, `Namespace`, `Classifier`, `Feature`,

StructuralFeature, and BehavioralFeature; and the concrete constructs used are: Class, Association, AssociationEnd, Attribute, Operation, and Parameter.

According to [UML01], a ModelElement “is an element that is an abstraction drawn from the system being modeled”. All other modeling metaclasses are either direct or indirect subclasses of ModelElement. A GeneralizableElement is a model element that participates in a generalization relationship. A Namespace is a ModelElement that owns other model elements whose name must be unique within the Namespace. A Classifier is a model element that describes a collection of behavioral and structural Features, such as Attribute and Operation. It has a name, which is unique in the Namespace enclosing the Classifier. A Feature is a property encapsulated within a Classifier. In the metamodel, a Feature declares a behavioral or structural characteristic of a Classifier. A BehavioralFeature refers to a dynamic feature of a ModelElement, such as an Operation. An Operation is a service that can be requested from a Classifier. It has a signature, which includes the actual Parameters. A Parameter is an unbound variable that can be changed, passed, or returned. It is also used in the specification of events. An Attribute is a named placeholder within a classifier that describes a range of values. An associationEnd is extended by adding a String attribute stateOfParticipant to it. This attribute is used by the buildSubTree algorithm to identify whether an associationEnd specifies the state of its participant and if yes, what the state is (Section 3.2.3).

It should be noted that additions to the standard UML core subsystem have been made to two classes: `Class` has a Boolean attribute `isStateMachine` added, which specifies whether instances of the `Class` have a state-dependent behavior described by a statechart; `Operation` has Boolean attribute `isEventHandler`, which denotes if an instance of `Operation` is an event handler. These two attributes are not part of the UML metamodel but are necessary for CBCDTool to derive test constraints.

4.3.2.1.2 dataTypes Subsystem

The `dataTypes` subsystem specifies the data types that are used to define UML [UML01]. In CBCDTool six enumeration classes are reused from the standard UML `dataTypes` subsystem (Figure 28). The possible enumeration values for each enumeration data type class are reported in Table 5.

Class	Enumeration Value
AggregationKind	0 – none, 1 – aggregation, 2 – composition
ChangeableKind	0 – changeable, 1 – frozen, 2 – addOnly
Multiplicity	lower – lower bound of the multiplicity range upper – upper bound of the multiplicity range
OrderingKind	0 – unordered, 1 – ordered
ScopeKind	0 – instance, 1 – classifier
VisibilityKind	0 – public, 1 – protected, 2 – package, 3 – private

Table 5. Enumeration values of UML data type classes

Figure 28. dataType subsystem

4.3.2.2 behavioralElements Subsystem

The behavioralElements subsystem (Figure 29), contains five subsystems, namely: commonBehavior, stateMachines, activityGraphs, useCases, and collaborations. Only the first two subsystems are relevant to CBCDTool; the other three were not implemented.

Figure 29. behavioralElements subsystem

4.3.2.2.1 commonBehavior Subsystem

The following description of the commonBehavior subsystem is taken from [UML01]. The commonBehavior subsystem (Figure 30) is the most fundamental subsystem that constitutes the behavioralElements subsystem. It specifies the core

concepts required for subsystems in behavioralElements. We adapted the metamodel to facilitate our implementation by associating Action to Operation. The reason is that in this research, we assume every action has an action handler operation (Section 2.4.4.2).

The Action class is an abstract metaclass that inherits from the ModelElement class from the core subsystem. It is a specification of an executable statement that forms an abstraction of a computational procedure that results in a change in the state of the model [UML01]. There are seven concrete action types. However only five of them are of interest in CBCDTool (see discussions in Section 2.4.4), namely: CallAction, SendAction, CreateAction, and DestroyAction and Assignment. An Action can be part of an ActionSequence, which consists of a collection of actions that are performed sequentially as an atomic unit [UML01].

Figure 30. commonBehavior subsystem

4.3.2.2.2 stateMachines Subsystem

The `stateMachines` subsystem is shown in Figure 31. Again, only classes of interest to CBCDTool are reused from the standard UML `stateMachines` subsystem.

A `StateMachine` class is a specification that describes all possible behaviors of some dynamic model element [UML01]. It has a context which is a `ModelElement`. Since this work assumes flat statecharts (statecharts being flattened [Binder99]), there will be no state hierarchy. Therefore a `StateMachine` has a collection of `States`. A `State` class inherits from the abstract `StateVertex` class and it has a collection of incoming and outgoing `Transitions`. A `StateMachine` also has a collection of `Transitions`. Each `Transition` has a source and a target `State`, an `Event`, a `Guard`, and an `Action` (defined in `commonBehavior` subsystem). An `Event` is “a specification of a type of observable occurrence whose occurrence generates an event instance that takes place at an instant in time with no duration” [UML01] and it has four subclasses, namely, `CallEvent`, `SignalEvent`, `ChangeEvent`, and `TimeEvent`.

4.3.3 ist Subsystem

The `ist` subsystem (Figure 32), is the Invocation Sequence Tree (IST) metamodel and contains six classes. The metaclasses in this subsystem are already discussed in Section 3.2.2.

Figure 31. stateMachines subsystem

Figure 32. 1st subsystem

4.3.4 ocl Subsystem

The ocl subsystem is reused from [Finger00] for the purpose of parsing OCL expressions. This subsystem is generated from the grammar file [Finger00] for OCL version 1.3 using the SableCC compiler compiler [Gagnon98]. The generated subsystem consists of a `parser` subsystem, which in turn consists of four subsystems, namely: `node`, `lexer`, `parser`, and `analysis`. The `node` subsystem contains all the classes necessary for defining the typed Abstract Syntax Tree (AST) of an OCL expression. The `lexer` subsystem contains classes necessary for lexical processing of an OCL expression. The `parser` subsystem contains classes that are used to parse an OCL expression and build its corresponding typed AST. The `analysis` subsystem contains interfaces and classes that are used to define tree-walkers (designed according to the Visitor design pattern [Gamma94]) for AST. A tree-walker is a class that visits all the nodes of an AST in a predefined order [Gagnon98].

Figure 33. ocl subsystem

There are a number of utility classes defined in the `ocl` subsystem, namely, `OclTree`, `ASTTextGenerator`, `NodeFinder`, `TypeOperationNormalizer`, `RedundancyRemover`, and `OclTree`.

and RedundancyRemover. The first two classes are adapted from [Finger00] (by removing unused methods) while the remaining classes are for the purpose of the CBCDTool. The OclTree class contains methods that create and return the AST of an OCL expression. The ASTTextGenerator class contains methods that build and return the string representation of a tree node in an AST. The NodeFinder class contains methods that find all the nodes of a specific type in an AST. This is necessary for the normalization of OCL expressions (discussed in Section 3.4). The TypeOperationNormalizer class contains methods that normalize operations in an OCL expression. The RedundancyRemover class has methods that remove redundancies in an OCL expression.

Chapter 5 – Case Studies

We report here the use of our methodology on two case studies that are selected because they are representative of two typical situations where statecharts are used. The first one is a Video Store System (VSS) that consists of a class cluster whereas the second one is an `OrdSet` that is a container class implementing a data structure. By studying these two different examples, we wish to provide evidence of the generality of our methodology. Our prototype tool CBCDTool produces the results of the case studies and they match the constraints derived by manual analysis, thus provides an initial validation of our methodology. However, the two normalization steps of normalizing logical expressions and normalizing postconditions, are not yet implemented. Therefore, the constraints we supply to the CBCDTool are normalized by hand before loading them into the tool. Besides these two limitations, other functionalities such as building the IST, deriving test constraints for an invocation subscenario, and the rest of normalization techniques are all tested against the two case studies. The automation of the two normalization steps, as well as their applicability on our two case studies will be our future research endeavor.

5.1 Case Study 1: VSS

The VSS is a UML specification for a video store management system adapted from [Mitchell99]. We choose to study a cluster of six classes whose instances are entity

objects²⁰, namely, Copy, Reservation, Title, Rental, Member, and TimePoint. This case study has the following characteristics:

- A variety of collection operations are involved in both the guards and the operation postconditions.
- The class Copy whose statechart is under test is an associated class, and there are interactions among instances of Copy and instances of other state-dependent classes (Reservation).

The class diagram for entity classes in the VSS is depicted in Figure 34.

Instances of Copy in the VSS have state-dependent behaviors and Figure 35 depicts the corresponding statechart: State OnHold specifies that the Copy object is available for a Reservation of 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 the Reservation and Title objects. Such effect must be considered since the behavior of the Copy object might also depend on the state of its related Reservation and Title objects. The Reservation object owns a statechart as well, as depicted in Figure 36: A Reservation object becomes Outstanding when a Copy is available. Title objects do not have state-dependent behavior.

²⁰ An entity object represents persistent information that is tackled by the system [BD00].

Figure 34. Class diagram of entity classes in the VSS case study

Figure 35. Statechart of a Copy object in the VSS case study

Figure 36. Statechart of a Reservation object in the VSS case study

We assume we want to test the following TTS in the statechart for 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), and a reservation for the corresponding title is made before the copy is returned (state OnHold follows state Rented on 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 event holdExpire). The second reservation does not expire and the copy is rented.

The invocation sequence tree corresponding to the above TTS is shown in Figure 37. The nodes denote events/actions and the edges denote invocations. Nodes 1-4 represent trigger events in the TTS to be tested. They are ancestor nodes of invocation subtrees, which can be seen to the right, capturing possible invocation orders on other instances. Edges leaving the same action node have an alternative relation by default (e.g. invocation labelled 2.1 and 2.2). If two edges leaving the same predecessor node have successor nodes that correspond to two adjacent actions in the action sequence, the invocation chain after one successor node is executed after that of the other successor node. In Figure 37, directed *arcs* are used to specify the order between two adjacent invocation chains (e.g., 2.1.1, 2.1.2). All the invocation conditions in the invocation sequence tree are already normalized, as described in Section 3.4.

We can see that the number of subscenarios for each event node is as follows: one for event node 1, two for event node 2, three for event node 3 and one for event node 4.

The algorithm we use to derive constraints traverses in a forward manner from the first event node in the invocation sequence tree to the second before last trigger event in the transition test sequence. For each trigger event node, we analyze its invocation sequence subscenarios and derive the constraints that must be true for executing all the actions in each of the subscenarios. The resulting constraints are then combined (conjunction) with the predicate condition of the event (see Section 3.3.2) to form the complete constraint for each invocation subscenario.


```

1. i = 1 //Event 1 has one sub-scenario.
result->at(i)->at(j) = {self.state=#ForRent or
                        self.state=#OnHold}

2. i = 2, j = 1
result->at(i)->at(j) = self.title.reservation->select
  (v|v.state=#Pending)->size > 0 and self.title.reservation->
  select(v|v.state=#Pending)->size > 1

3. i = 2, j = 2
result->at(i)->at(j) = self.title.reservation->select
  (v|v.state=#Pending)->size >0 and not self.title.reservation->
  select(v|v.state=#Pending)->size > 1

4. i = 3, j = 1
result->at(i)->at(j) = self.title.reservation->select
  (v|v.state=#Pending)->size > 0 and self.title.reservation->
  select(v|v.state=#Pending)->size > 1

5. i = 3, j = 2
result->at(i)->at(j) = self.title.reservation->select
  (v|v.state=#Pending)->size >0 and not self.title.reservation->
  select(v|v.state=#Pending)->size > 1

6. i = 3, j = 3
result->at(i)->at(j) = self.title.reservation->select
  (v|v.state=#Pending)->size >0 and not self.title.reservation->
  select(v|v.state=#Pending)->size > 0

7. i = 4, j = 1
result->at(i)->at(j) = (self.state=#ForRent or self.state=#OnHold)
                      and self.reservation.state=#Outstanding

```

Constraint numbered 1 (resp. 7) applies to event 1 (resp 4) in our IST (Figure 37), i.e., event rent. The constraint specifies that the state of the copy must be either #ForRent or #OnHold for the event to execute (constraint numbered 7 additionally requires that the copy has a corresponding outstanding reservation, which implies that the copy is current in state #OnHold). Constraints numbered 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 the last 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.

When executing the complete transition test sequence, any of constraints 2 and 3 can be combined with any of constraints 4 and 5 for events `return` and `holdExpire`, respectively. Then either the result of the first event (i.e., `return`), given the selected constraint, implies or contradicts the selected constraint of the second event (i.e., `holdExpire`). For instance, given that the number of pending reservations is reduced by one when a copy is returned²¹, choosing constraints 3 and 4 (or 5) is possible only if the test driver creates a reservation for the title after `return` is triggered but before `holdExpire` is triggered. This would correspond to the concurrent execution of two use cases (i.e., use cases `returnCopy` and `makeReservation`). From a general standpoint, it is worth noting that:

- Test constraints may appear to be incompatible.
- Executing a TTS may require that the test driver simulate concurrent executions of use cases.

5.2 Case Study 2: OrdSet

Class `OrdSet` is a container class that provides operations for bounded, ordered sets. The specification for an `OrdSet` consists of a UML class diagram (Figure 38) and a statechart (Figure 39) adapted from [Antoniol02]. The case study is chosen because:

- It can be used to demonstrate the derivation of test constraints on event/action arguments.

²¹ `holdExpire` on a copy in state `OnHold` triggers `expire` on the reservation for that copy (Fig. 1), that itself triggers `copyAvailable` on the oldest pending reservation (Fig. 2), which reduces the number of pending reservations by one (see “Normalizing postconditions”).

- The `OrdSet` is an orphan state-dependent class that has no navigable associations to other classes.

The adapted version of the `OrdSet` class contains methods that create and modify sets: constructor `OrdSet(int n)`, operator `+=(int n)`, `remove(int n)` and `resize()`. In order to cover the state-dependent behavior of the `OrdSet`, we choose its initial cardinality to be three. (i.e. the `OrdSet` initially has three empty slots). When the set is full and there is a new `Integer` added to the set, resizing, if allowed, is performed. The number of allowed resizing times must not exceed the constant `max_accepted_resizes`, which equals to one in this case study.

Figure 38. Class diagram in the `OrdSet` case study

The statechart of `OrdSet` is shown in Figure 39. The object can be in any of four distinct states. Initially, it is in the `Empty` state. When the number of elements (`Integer` objects) in the set is less than its capacity `_set_size`, it is in the `PartiallyFilled` state. When the number of elements in the set equals to its capacity, it is in the `Filled` state. When the set is in the `Filled` state and there is a new `Integer` added to the set and the set has already been resized once, then it goes to the `Overflow` state.

It should be noted that since we adopt the Design by Contract principle, it is necessary to make sure that an `Integer` is not already in the `OrdSet` before adding it to

the set. Similarly, it is necessary to make sure that an Integer is already in the OrdSet before removing it from the set. Such checking is modeled as the precondition of the operation `operator+=(n:Integer)` (`pre: self->count(n)=0`) and operation `remove(n:Integer)` (`pre: self->count(n)=1`) respectively.

Figure 39. Statechart of an OrdSet object in the OrdSet case study

We consider a specific transition sequence to execute:

```

@Empty
@operator+=(n0) [true]
@PartiallyFilled
@operator+=(n1) [_last<_set_size-1]
@PartiallyFilled
@operator+=(n2) [_last=_set_size-1]
@Filled
@operator+=(n3) [_resized_times<max_accepted_resizes]/self.resize(
)
@PartiallyFilled
@remove(n4) [_last > 1]
@PartiallyFilled
@operator+=(n5) [_last<_set_size-1]
@PartiallyFilled
@operator+=(n6) [_last<_set_size-1]
@PartiallyFilled
@operator+=(n7) [_last=_set_size-1]
@Filled
@operator+=(n8) [_resized_times >=max_accepted_resizes]
@Overflow
  
```

This sequence corresponds to the following scenario: After a set instance of the OrdSet class is created, add one Integer object to the set to put it in state

PartiallyFilled. Then add two other Integers to the set one by one to put it in state Filled. Then add another Integer so that the set will perform a resizing operation and enter state PartiallyFilled again. Next remove any of the existing Integer from the set. Then add three Integers consecutively until the set is in state Filled. Finally, put the set in state Overflow by adding one more Integer.

The invocation sequence tree (IST) for this TTS is shown in Figure 40. The invocation condition of an invocation in the IST is a conjunction of three parts: the operation's precondition, the state condition, and the guard. For instance, the invocation condition on the invocation from node1 to node2 is a conjunction: `self.state=#PartiallyFilled` is the state condition, `self->count(n1)=0` is the precondition of operation `operator+=(n:Integer)`, and `self.last<self.set_size-1` is the guard. All the constraints in the invocation sequence tree are already normalized. Since `OrdSet` is an orphan state-dependent class, its IST has only one scenario (Section 3.3.1), which consists of a sequence of event/action nodes:

```
node1, node2, node3, node4, node4.1, node5,
node6, node7, node8, node9
```

The algorithm for deriving the test constraint to be fulfilled to execute the complete TTS for an orphan state-dependent object is similar to the one for associated state-dependent objects. It also proceeds in a recursive way from the bottom (i.e., the last event node in the TTS, which is node9 here), up in the IST until the first event node in the TTS has been processed, which is node1 here), and consists in the same kind of transformations of invocation conditions using postconditions.

Figure 40. The IST of OrdSet

The resulting test constraints sequence consists of one constraint element, which is a conjunction of eleven clauses.

```

{
self->union(Set{n0})->union(Set{n1})->union(Set{n2})->
union(Set{n3})->excluding(n4)->union(Set{n5})->
union(Set{n6})->union(Set{n7})->count(n8) = 0
and
self->union(Set{n0})->union(Set{n1})->union(Set{n2})->
union(Set{n3})->excluding(n4)->union(Set{n5})->
union(Set{n6})->count(n7) = 0
and
self->union(Set{n0})->union(Set{n1})->union(Set{n2})->
union(Set{n3})->excluding(n4)->count(n6) = 0
and
self->union(Set{n0})->union(Set{n1})->union(Set{n2})->
union(Set{n3})->excluding(n4)->count(n5) = 0
and
self->union(Set{n0})->union(Set{n1})->union(Set{n2})->
union(Set{n3})->count(n4) > 0
and
self->union(Set{n0})->union(Set{n1})->union(Set{n2})->
count(n3) = 0
and
self->union(Set{n0})->union(Set{n1})->count(n2) = 0
and
self->union(Set{n0})->count(n1) = 0
and
__last > -3
and
__resized_times > 0 or __resized_times = 0
and
1 > __resized_times
}

```

This constraint specifies that in order for an OrdSet object to execute the TTS, the following condition must be satisfied when the object is created:

- `__resized_times` must be zero
- `__last` must be greater than -3
- `n0`, `n1`, `n2`, `n3` are distinct Integers; `n4` equals one and only one Integer from `n0` to `n3`; and `n0` to `n3` (excluding `n4`), `n5`, `n6`, `n7`, and `n8` are distinct Integers.

It is easy for the user to find values that satisfies this test constraint. For instance, the following value set is a solution to the test constraint.

```

n0=1,  n1=2,  n2=3,  n3=4,  n4=1,  n5=5,  n6=6,  n7=7,  n8=8,
__resized_times=0

```

Chapter 6 – Conclusions

This thesis explores the automating support of deriving test data derivation from UML statecharts given coverage criteria such as the all transitions, all transition pairs, full predicate, and all round-trip paths [Binder99, Hong99, Offutt99]. These criteria all assume a test case to be in the form of feasible sequences of transitions. 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 Transition Test Sequence (TTS). The second subproblem is then to generate actual test data values that satisfy these test constraints.

This thesis addresses the first subproblem. 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. In doing so, we account for possible interactions (through events sent) between instances of different classes that have state-dependent behaviors. Information in the statecharts and class diagrams are extracted and modelled into IST, which is more amenable to the definition of algorithms. The methodology also identifies five types of normalizations of OCL expressions to support the analysis of constraints written in OCL and their combinations to generate test constraints. Note that these normalization rules can be also considered as guidelines to write preconditions, postconditions and invariants in OCL. Moreover, the methodology provides a number of algorithms that specify a sequence of precise steps to solve the problem. 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, which provide an initial demonstration of the feasibility of automating the methodology. Results show that the methodology is promising in that most of the steps defined in the algorithm can be automated and the result corresponds to what is obtained through manual test constraint derivation.

6.1 Thesis Limitations

There are several limitations of this thesis and we divide them into three categories: practical limitations, methodology limitations, and limitations of the prototype implementations.

6.1.1 Practical Limitations

Though our methodology has shown to work on two representative case studies, there is no guarantee that they will work on all state-dependent systems. More case studies need to be performed to test the generality of our methodology, especially the normalization of OCL expressions. However, statecharts that are both non-trivial and rich in UML statechart features are not readily available. Actually, the Video Store System case study has been greatly extended from its original version [Mitchell99] to reach its current form.

In addition, the normalization of postcondition 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 corresponds to good practice and should be encouraged.

6.1.2 Methodology Limitations

Presently our methodology can identify redundancy or inconsistency between of 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 $\text{not } (a > b)$). Other types of redundancy (e.g., $\text{constraintA implies constraintB}$, hence constraintB is redundant) or inconsistency (e.g., $\text{constraintA implies constraintB}$, hence constraintA and (not constraintB) are inconsistent) cannot be automatically tackled and have to be identified manually. However, we also expect normalization to facilitate the manual analysis of constraints.

6.1.3 Implementation Limitations

Though potentially our methodology might be fully automatable, this research does not explore this aspect in a complete way due to time constraints. Currently, the implemented prototype tool does not support the normalization of logical expressions and postconditions. Though not implemented, both type of normalization have been described in great detail. For the former type of normalization there are off-the-shelf algorithms that perform logical expression transformations (e.g. an algorithm is described in [HR00] to convert any logical expression into its equivalent CNF). The implementation of the latter type of normalization requires two algorithms. The first one converts the operation contracts into Standard Equality Form. The second algorithm analyzes the contracts

(which is in Standard Equality Form) and complements the postcondition according to contract patterns and their corresponding complementing rules (Appendix B).

6.2 Future Work

Additional case studies should be performed to evaluate our methodology. An ideal case study would be a complete and correct software specification (in UML notation) modelling a non-trivial system, which consists of a class cluster, several of which have state-dependent behaviors. Moreover, the specification should follow the Design by Contract principle [Meyer97] and should define operation contracts using OCL. It would be more interesting if the contracts involved rich navigation expressions and collection operations. Such case studies would be very useful to assess the generality of our methodology and can help us to improve it.

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 can help us in defining less restrictive modelling patterns so that our approach can be more widely applicable.

Fully automated tool support for our methodology needs to be explored. Specifically, the normalization of postconditions and logical expressions needs to be added to the prototype tool.

Recall that the test case derivation problem is divided into two subproblems. This thesis 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 (Section 5.1) shows that infeasible test constraints (e.g. the sixth test constraint for VSS) might result during the test constraints derivation process. It is hoped

that such constraints can be identified during this stage. The solution, to address the second subproblem mentioned above, will be the subject of future work. In the following we describe the second subproblem briefly and point out possible research directions.

The second subproblem belongs to the realm of Constraint Satisfiability Problem (SAT) [Michale00] 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 (LP), Hill Climbing (HC), Simulated Annealing (SA), and Genetic Algorithms (GA). These techniques all have their strengths and weaknesses. So there is no single best-for-all technique for an arbitrary SAT problem [Michale00]. Which technique to choose really depends on the nature of the individual type of problems. 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.

For instance, both LP and GA can work on potential solutions encoded as Integers and Real numbers [Michale00]. Suppose that our test constraints can be converted into one of the above two representations, and further assume that these constraints are linear constraints. Then we choose LP instead of GA since LP is a deterministic optimization technique that guarantees to find the solution (if there is one). In comparison, GA is a heuristic technique, which means we cannot predict its result each time we execute it. It is likely that GA produces two different solutions to the same SAT out of two executions, moreover, GA may not find solutions to a feasible SAT. However, if our test constraints appear to be non-linear, then LP is not applicable. Further assume that we can devise a

suitable fitness function for our test constraint problem, we may then consider GA as an alternative. An argument in favor of using GA is that it was shown to perform pretty well for large, complex and non-linear SATs [Michale00].

References

- [Antoniol02] G. Antoniol, L. C. Briand, M. Di Penta and Y. Labiche, "A Case Study Using the Round-Trip Strategy for State-Based Class Testing," *Proc. (to appear) 13th IEEE International Symposium on Software Reliability Engineering*, Annapolis (MD, USA), November 12-15, 2002.
- [Binder99] R. V. Binder. *Testing Object-Oriented Systems, Models, Patterns, And Tools*. 2000. Addison Wesley Longman, Inc.
- [Beizer90] B. Beizer, *Software Testing Techniques*, Van Nostrand Reinhold, 2nd Ed., 1990
- [BD00] B. Bruegge, A. H. Dutoit. *Object-Oriented Software Engineering, Conquering Complex and Changing Systems*. Prentice Hall, 2000
- [Dresden02] Dresden University of Technology. "Dresden OCL Toolkit"
<http://dresden-ocl.sourceforge.net/>
- [Finger00] F. Finger. *Design and Implementation of a Modular OCL Compiler*. Master Thesis, Dresden University of Technology, 2000
- [Gagnon98] E. Gagnon. SableCC: an object-oriented compiler framework. Master Thesis, McGill University, Montreal, 1998.
- [Gamma94] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison Wesley Longman, Inc. 1994.
- [Hong99] Y. G. Kim, H. S. Hong, S. M. Cho, D. H. Bae, S. D. Cha. Test Case Generation from UML State Diagrams. IEE proceedings, Software, vol.146(4), pp187-192,1999
- [HR00] M. Huth, M. Ryan. *Logic in Computer Science, Modelling and reasoning about systems*. Cambridge University Press, 2000
- [Meyer97] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997
- [Mitchell99] Richard Mitchell. Analysis by Contract, Video Store Case Study, version2.4. June,1999

- [Michale00] Z. Michalewicz *How to solve it: Modern Heuristics*. Springer-Verlag, 2000
- [MM02] R. Mitchell, J. McKim. *Design by Contract, by Example*. Addison Wesley, 2002
- [MS01] J. D. McGregor, D. A. Sykes. *A Practical Guide to Testing Object-Oriented Software*. Addison-Wesley, 2001
- [Offutt03] J. Offutt, S. Liu, A. Abdurazik, P. Ammann. Generating Test Data from State-based Specifications. *Software Testing, Verification and Reliability* 2003;13:25-53.
- [Offutt99] A. J. Offutt and A. Abdurazik, "Generating Tests from UML specifications," *Proc. UML'99*, Fort Collins, CO, pp. 416-429, October, 1999
- [RJB99] J. Rumbaugh, I. Jacobson, G. Booch. *The Unified Modeling Language Reference Manual*. Addison Wesley Longman, Inc. 1999
- [TSPM98] B. Y. Tsai, S. Stobart, N. Parrington, I. Mitchell. An Automatic Test Case Generator Derived from State-Based Testing. Asia Pacific Software Engineering Conference. December 02-04, 1998. TaiPei, Taiwan.
- [UML01] *Unified Modeling Language (UML), Version 1.4*. OMG. 2001
- [VT98] M. E. R. Vieira, G. H. Travassos. An Approach to Perform Behavior Testing in Object-Oriented Systems. Technology of Object-Oriented Languages and Systems. September 22-25, 1998. Beijing, China.
- [WK99] J. Warmer, A. Kleppe. *The Object Constraint Language, Precise Modeling with UML*. Addison Wesley Longman, Inc. 1999
- [XMI00] OMG, "OMG XML Metadata Interchange (XMI) Specification", Version 1.1, November 2000.

Appendices

Appendix A – Algorithms of CBCDTool

The algorithms presented below build the IST for a given TTS using information obtained from UML models (class diagram and statecharts for the class cluster) and then derive test constraints using the IST. By using a mix of control structures and OCL navigation paths in the metamodel, we define the algorithms precisely but still remain at an abstract, logical level.

A.1 CbcdController::start

This function controls the steps of test constraints derivation.

Input:

args: command line arguments that specifying the input file names

Output:

Algorithm

```
function start(args:Sequence(String))
var:
constraint :Sequence(Sequence(String)); //test constraints for an IST
begin
    //create an equivalent path object and reads data from the file
    //that defines equivalent navigation paths
    equivalentPath := new EquivalentPath(args->at(1));
    //create a contract object and reads data from the file that defines
    //operation contracts
    contract := new Contract(args->at(2));
    //create a class cluster object and reads data from the file that
    //defines the class diagram
    classCluster := new ClassCluster(args->at(3));
    //create a state machine cluster object and reads data from the file
    //that defines a cluster of state machines
    stateMachineCluster := new StateMachineCluster(args->at(4));
    //create a transition test sequence object and reads data from the
    //file that defines the transition test sequences
    tTS := new TTS(args->at(5));
    //create an IST builder object
    iSTBuilder := new ISTBuilder(equivalentPath,contract,classCluster,
        stateMachineCluster,tTS,args->at(6));
    //build the IST
    iSTBuilder.buildIST();
    //create a constraint builder object
```

```

    constraintBuilder := new ConstraintBuilder(ist);
    //derive test constraints for the IST
    constraint := constraintBuilder.deriveConstraint();
end

```

A.2 ISTBuilder::ISTBuilder

This is the constructor for ISTBuilder.

Input:

```

ep:  EquivalentPath; //equivalent path object
contr:Contract;      //contract object
cc:  ClassCluster;   //class cluster object
smc:  StateMachineCluster; //state machine cluster object
tts:  TTS;           //transition test sequence object

```

Output:

```

function ISTBuilder(ep:EquivalentPath,contr:Contract,cc:ClassCluster,
                    smc:StateMachineCluster,tts:TTS)
begin
    equivalentPath := ep;
    contract := contr;
    classCluster := cc;
    statemachineCluster := smc;
    tTs := tts;
end

```

A.3 ISTBuilder::buildIST

This function builds IST using information from the TTS object `tTs`, the `ClassCluster` object `cc`, the `Contract` object `contract`, the `StateMachineCluster` object `smc`, and the `EquivalentPath` object `ep`.

Input :

```
context    :String    //the owning class of the statechart under test
```

Output:

```
function buildIST(context:String)
```

var:

```

st      :SubTree;
ist     :IST;
t       :Transition;
ev      :Event;
guard,precond,invocCond :String;
invoc   :Invocation

```

subfunction:

```
ISTBuilder::buildSubTree(t:Transition,
                        rootPath:String,
                        context:String):SubTree
```

Create a subtree for a transition `t`. `context` is the owning class of the statechart under `test`. `rootPath` is the navigation path from `context` to the class that defines the operation associated with the action on transition, which corresponds to the subtree root node.

begin

```

//create an IST object
ist := new IST();
//create a constraint normalizer object, pass classCluster and
//stateMachineCluster to the constructor
constraintNormalizer := new ConstraintNormalizer(
    classCluster,stateMachineCluster);
//iterate each transition in the TTS, build corresponding
//subtree and add the subtree to the IST object ist
for i:=1 to tts.transition->size do
    //get the ith transition
    t := tts.transition->at(i);
    ev := t.event;
    guard := t.guard.expression;

    if ev->isEmpty() then
        //if ev does not exist, an exception is thrown to indicate an
        //error; we do not consider completion transition (no explicit
        //trigger event) in this work
        throw exception;
    else
        //build subtree for transition t
        st := buildSubTree(t,"self",context);

```

```

//add st to the ist subtree sequence
ist.subTree->append(st);
//create an invocation whose predecessor is null and successor
//is the root of st
invoc := new Invocation(null,st.root);
//get precondition of ev's handler operation (if any)
if ev.oclIsTypeOf(CallEvent) then
    precondition := ev.operation.precondition;
else
    precondition := null;
end if

//create invocation's invocation condition
invocCond := null;
if guard->notEmpty then
    invocCond := guard;
end if
if precondition->notEmpty then
    invocCond := invocCond + "and" + precondition;
end if

//set invocation's invocation condition to be invocCond
if invocCond->notEmpty then
    invoc.invocCond := new ISTConstrint(invocCond);
else
    invoc.invocCond := null;
end if
//create subscenarios for st
createSubScenarioForSubTree(st);

end if //if ev->isEmpty()

end for //for i:=1 to tTS.transition->size
return ist;
end

```

A.4 ISTBuilder::buildSubTree

Input:

```
t :Transition;
rootPath :String; //the navigation path from context to the class that
                  //defines the operation associated with the action,
                  //which corresponds to the subtree's root node
context :String; //the owning class of the statechart under test,
                 //which is the unified context class for all OCL
                 //constraints after normalizing the navigation path.
```

Output:

```
st :SubTree;
function buildSubTree(t:Transition,rootPath:String,context:String)
    :SubTree
var:
ev :Event
rootNode :ISTNode;
st :SubTree
act :Action;
action:CallAction;
ah :Operation;
actSeq :Sequence(CallAction);
invocsForAction :Sequence(Sequence(Invocation));

subfunction:
ISTBuilder::normalizeRootNode(rootNode:ISTNode,rootPath:String,
                             ev:Event)
    Normalize the rootNode's expression and creates rootNode pre- and
    postconditions whose navigation paths are normalized.

ISTBuilder::buildSubTree_EventHandler(action:CallAction,context:String,
                                     rootPath:String,enclosingSubTreeRoot:ISTNode)
    :Sequence(Invocation)
    This function builds a sequence of subtrees for an action whose
    action handler is a trigger event. It obtains all the transitions
    that might be enabled when the target object receives the action,
    and creates one subtree for each transition.

ISTBuilder::buildSubTree_NonEventHandler(action:CallAction,
    context:String,rootPath:String,enclosingSubTreeRoot:ISTNode)
    :Sequence(Invocation)
    This function builds one subtrees for an action whose action handler
    is not a trigger event.

begin
    //get the event of the transition
    ev := t.event;
    //get the action of the transition
    act := t.action;

    //create root node for subtree
    rootNode := new ISTNode(ev.expression);
    //create a subtree for rootNode
    st := new SubTree(rootNode);
```

```

if not ev.ocIsTypeOf(CallEvent) then
    //if ev is not of type CallEvent, it does not have pre- and
    //postcondition
    rootNode.precond := null;
    rootNode.postcond := null;
else
    //normalize the navigation path of rootNode's expression, create
    //its pre- and postcondition, and normalize their navigation
    //paths
    normalizeRootNode(rootNode, rootPath, ev);
end if

if act->isEmpty then
    //the transition has no action, return the subtree
    return st;
else
    //the transition has action(s), create an action sequence
    if act.ocIsTypeOf(ActionSequence) then
        //act is an action sequence
        actSeq := act.action;
    else
        //act is a single action
        actSeq := Sequence{act};
    end if

    //iterate all the actions in the action sequence
    //1. if action is a trigger event
    //get possible transitions for each action, create:
    //(1)an actionSubtree for the transition
    //(2)an invocation from rootNode, which is the root node of the
    //current subtree, to root, which is the root node of the
    //created actionSubtree
    //2. if action is not a trigger event, create:
    //(1)an root node
    //(2)an actionSubtree whose root is root
    //(3)an invocation from rootNode to root
    for i:=1 to actSeq->size do
        //get the ith action in actSeq
        action := actSeq->at(i);

        //get the action handler operation. Note that in this work,
        //all types of action classes have an association to an
        //Operation class to ease the data storage and retrieval.
        //For actions that are not call action, a pseudo-operation
        //object is created for it during the process of loading
        //statechart data.
        ah := action.operation;
        if ah.isEventHandler then
            //if action handler is a trigger event:
            //obtain all the transitions that might be enabled and
            //create an actionSubtree for each transition. The subtree
            //root is the action's corresponding ISTNode.
            invocsForAction->at(i) := buildSubTree_EventHandler(action,
                context, rootPath, rootNode);
        else
            //action handler is not trigger event.

```

```

        //create an Action node and create the subtree whose root
        //is this action node.
        invocsForAction->at(i) := buildSubTree_NonEventHandler(
            action, context, rootPath, rootNode);
    end if    // if ah.isEventHandler
end for // for i:=1 to actSeq->size
//create sequential-relation to link every element in
//invocsForAction->at(i-1) to every element in
//invocsForAction->at(i)
for j:=1 to invocsForAction->at(i-1)->size do
    for k:=1 to invocsForAction->at(i)->size do
        invocsForAction->at(i-1)->at(j).next :=
            invocsForAction->at(i)->at(k);
    end for
end for
end if    // if act->isEmpty
return st;
end

```

A.5 ISTBuilder:: normalizeRootNode

This function normalizes the root node expression and creates its pre- and postconditions whose navigation paths are normalized.

Input:

rootNode :ISTNode
rootPath :String
ev :Event

Output:

```
function normalizeRootNode(rootNode:ISTNode,rootPath:String,ev:Event)
```

var:

```
evh :Operation //event handler operation  
precond,postcond :String  
paraList :Sequence(String); //list of parameters  
argList :Sequence(String); //list of arguemnts
```

subfunction:

```
ConstraintNormalizer::normalizeNavPath(constraint:String,  
                                       path:String):String
```

Normalize navigation path of a constraint by replacing the "self" in constraint with the argument path, simplify the navigation path, and then return the constraint as a string.

```
ISTBuilder::replaceParaWithArg(s:String, paraList:Sequence(String),  
                               argList:Sequence(String)):String
```

Iterate paraList, for each element in paraList, use its corresponding element in argList to replace all its occurrences in s, then return s.

begin

```
//get ev's handler evh  
evh := ev.operation;  
//get pre and postcondition of evh  
precond := evh.precondition;  
postcond := evh.postcondition;  
//normalize navigation path of rootNode expression, including the  
//navigation path of its arguments  
rootNode.expression := constraintNormalizer.  
    normalizeNavPath(rootNode.expression,rootPath);  
//normalize navigation path of pre- and postcondition  
if precond->notEmpty() then  
    precond := constraintNormalizer.  
        normalizeNavPath(precond,rootPath);  
end if  
if postcond->notEmpty() then  
    postcond := constraintNormalizer.  
        normalizeNavPath(postcond,rootPath);  
end if  
//get formal parameters of event handler  
paraList := evh.parameter.expression;  
if paraList->notEmpty() then
```



```

    //get arguments of event
    argList := ev.parameter.expression;
    //replace formal parameters in precondition and postcondition
    //with arguments
    if precondition->notEmpty() then
        precondition := replaceParaWithArg(precondition, paraList,
            argList);
    end if
    if postcondition->notEmpty() then
        postcondition := replaceParaWithArg(postcondition, paraList,
            argList);
    end if
end if

//create rootNode's pre- and postcondition
if precondition->notEmpty() then
    rootNode.precondition := new ISTConstraint(precondition, "precondition");
else
    rootNode.precondition := null;
end if
if postcondition->notEmpty() then
    rootNode.postcondition := new ISTConstraint(postcondition, "postcondition");
else
    rootNode.postcondition := null;
end if
end

```

A.6 buildSubTree_EventHandler

This function builds a sequence of subtrees for an action whose action handler is a trigger event. It obtains all the transitions that might be enabled when the target object receive the action, and creates one subtree for each transition.

Input:

action :Action
context :String
rootPath :String
enclosingSubTreeRoot :ISTNode //root node of the enclosing subtree

Output:

seqInvoc :Sequence(Invocation) //the sequence of invocation is ret

```
function ISTBuilder::buildSubTree_EventHandler(  
    action:CallAction,  
    context:String  
    rootPath:String,  
    enclosingSubTreeRoot:ISTNode  
):Sequence(Invocation)
```

var:

```
actPath :String //normalized navigation path of the action  
targetClass :String //class name of the object, which is the target of  
//action. The object can be reached from context  
//by navigating the normalized navigation path  
sm :StateMachine //the state machine of targetClass  
ase :AssociationEnd//the association end (to the target) of the last  
//association in a navigation path  
stateName :String //state name  
transitionSeq :Sequence(Transition) //the sequence of transitions that  
//might be enabled by action  
st :SubTree //an invocation subtree  
trans :Transition  
seqInvoc :Sequence(Invocation) //the sequence of invocations that  
//connect enclosingSubTreeRoot to each  
//subtree for action
```

subfunction:

```
ISTBuilder::getNavPath(actExpr:String):String  
    Returns the navigation path of an action expression actExpr as a  
    String
```

```
ClassCluster::getTargetClassName(path:String,  
    context:String):String  
    Return the name of the class, which is reached by navigating along  
    path from context
```

```
StateMachineCluster::getStateMachine(className:String):StateMachine  
    Return the state machine object describing the behavior of class  
    className
```

```

ClassCluster::getLastAssociationEndForPath(path:String, context:String)
                                   :AssociationEnd
    Return the association end object of path

AssociationEnd::getStateOfParticipant():String
    Return the attribute stateOfParticipant of an associationEnd. This
    information is provided when importing input files at the beginning
    of the overall process.

StateMachine::getTransitionsByEventName(eventName:String)
                                   :Sequence(Transition)
    Return the sequence of all the transitions that might be enabled by
    eventName

StateMachine::getState(stateName:String):State
    Return the state object specified by stateName

State::GetOutgoingTransitionsByEventName(eventName:String)
                                   :Sequence(Transition)
    Return the sequence of all the outgoing transitions from the state
    that are enabled by the specified event eventName

ISTBuilder::createInvocCondForEvent(invoc:Invocation, actPath:String,
                                   trans:Transition)
    Create the invocation condition for invoc

begin
    //normalize the navigation path of the action expression
    //using rootPath
    action.expression := constraintNormalizer.
        NormalizeNavPath(action.expression, rootPath);
    //get the normalized navigation path of the action. The
    //context of the path is always context, which is the class
    //that owns the statechart under test
    actPath := getNavPath(action.expression);

    //find the class of the object that executes the action
    //using the information of (1)navigation path of action
    //expression and (2)the context class name context
    targetClass := classCluster.getTargetClassName(actPath, context);
    //get the state machine of the target class
    sm := stateMachineCluster.getStateMachine(targetClass);
    //get the association end of the last association in the
    //action's navigation path
    ase := classCluster.getLastAssociationEndForPath(
        actPath, context);
    //get the state of the participant object
    //Recall Section 3.2.3, users are required to provide
    //information whether an association end implies the state
    //of its participant object. This information is stored in
    //a String type attribute "stateOfParticipant" of the
    //association end.
    stateName := ase.getStateOfParticipant();
    if stateName->isEmpty() then
        //the navigation path does not imply the state of
        //the target object;
        //get all the transitions enabled by the action

```

```

        transitionSeq := sm.getTransitionsByEventName
            (ah.expression);
    else
        //the navigation path implies the state of the target
        //object;
        //get all the outgoing transitions from the state
        //(specified by stateName) enabled by the action
        transitionSeq := sm.getState(stateName).
            getOutgoingTransitionsByEventName(ah.expression);
    end if

    //iterate all the transitions that might be enabled
    //create an actionSubtree for each transition. The subtree
    //root is the action's corresponding ISTNode.
    for j:=1 to transitionSeq->size do
        //get jth transition
        trans := transitionSeq->at(j);
        //build the actionSubtree for the jth enabled transition
        //trans. actPath is the normalized navigation path for
        //action expression, context is the class that owns the
        //statechart under test
        st := buildSubTree(trans,actPath,context);

        //add st to enclosingSubTreeRoot's subtree sequence
        enclosingSubTreeRoot.subTree->append(st);

        //create an invocation from enclosingSubTreeRoot to the subtree
        //root.
        //The invocation is one of enclosingSubTreeRoot's outgoing
        //invocations, and is subtree root's incoming
        //invocation
        invoc := new Invocation(enclosingSubTreeRoot,st.root);

        //add invoc to the seqInvoc
        //Purpose: to prepare for the construction of the
        //sequential relations (Section 3.2.1) between current
        //invocation and all the invocations that belong to the
        //previous action in the action sequence.
        SeqInvoc.append(invoc);

        //create invocation condition for invoc
        createInvocCondForEvent(invoc,actPath,trans);

    end for // j:=1 to transitionSeq->size
    return seqInvoc;
end

```

A.7 ISTBuilder::createInvocCondForEvent

The function creates the invocation condition for an action whose action handler is an event handler operation.

Input:

```
invoc :Invocation //invocation whose invocation condition needs to
                        //be created
actPath :String //normalized navigation path of an action expression
trans :Transition //a transition enabled by the action
```

Output:

```
function createInvocCondForEvent (invoc:Invocation,
                                actPath:String,
                                trans:Transition)

var:
stateCond :String //String-form state condition of invoc
precond    :String //String-form precondition of invoc
guard      :String //String-form guard
invocCond  :String //String-form invocation condition of invoc
subfunction:
ConstraintNormalizer::normalizeNavPath(constraint:String,
                                       path:String):String
    Normalize navigation path of a constraint by replacing the "self" in
    constraint with the argument path, simplify the navigation path, and
    then return the constraint as a string.

ConstraintNormalizer.removeRedundancy(constraint:String):String
    Return the constraint with redundancy removed

begin
    //get stateCondition
    stateCond := actPath + ".state=" + trans.source.name;
    precond   := trans.event.operation.precondition;
    if precond->notEmpty() then
        invocCond := stateCond + "and" + precond;
    else
        invocCond := stateCond;
    end if
    if trans.guard->notEmpty() then
        //if guard exists
        //normalize navigation path in guard expression
        guard := trans.guard.expression;
        guard.expression := constraintNormalizer.
            normalizeNavpath(guard.expression,actPath);
        invocCond := invocCond + "and" + guard.expression;
    end if
    //remove redundancy in invocCond
    invocCond := constraintNormalizer.
        removeRedundancy(invocCond);
    //set the invocation condition
    invoc.invocCond := new ISTConstraint(invocCond);
end
```

A.8 ISTBuilder::buildSubTree_NonEventHandler

This function builds one subtree for an action whose action handler is not a trigger event.

Input:

```
action    :Action
context   :String
rootPath  :String
enclosingSubTreeRoot :ISTNode
```

Output:

```
seqInvoc :Sequence(Invocation)
function buildSubTree_NonEventHandler (
    action:CallAction,
    context:String,
    rootPath:String,
    enclosingSubTreeRoot:ISTNode
):Sequence(Invocation)
```

var:

```
actNode  :ISTNode    //an ISTNode corresponding to action
st       :SubTree    //a subtree whose root is actNode
invoc    :Invocation
```

subfunction:

```
ConstraintNormalizer::normalizeNavPath(constraint:String,
    path:String):String
```

Normalize navigation path of a constraint by replacing the "self" in constraint with the argument *path*, simplify the navigation path, and then return the constraint as a string.

```
ISTBuilder::getNavPath(actExpr:String):String
```

Returns the navigation path of an action expression *actExpr* as a String

```
ISTBuilder::replaceParaWithArg(s:String, paraList:Sequence(String),
    argList:Sequence(String)):String
```

Iterate *paraList*, for each element in *paraList*, use its corresponding element in *argList* to replace all its occurrences in *s*, then return *s*.

```
ISTBuilder::normalizeRootNode(rootNode:ISTNode,rootPath:String,
    ev:Event)
```

Normalize the *rootNode*'s expression and creates *rootNode* pre- and postconditions whose navigation paths are normalized.

begin

```
//create an action node, pass action's expression to
//the constructor
actNode := new ISTNode(action.expression);
```

```
//create a subtree
st := new SubTree(actNode);
```

```

//add st to enclosing subtree root node's subtree sequence
enclosingSubTreeRoot.subTree->append(st);

//normalize the navigation path of actNode's expression, create
//its pre- and postcondition, and normalize their navigation
//paths
normalizeRootNode(actNode,rootPath,action);

//create an invocation from enclosingSubTreeRoot to actNode
invoc := new Invocation(enclosingSubTreeRoot,actNode);
//set the invocation condition
invoc.invocCond := new ISTConstraint(actNode.precond);

//add invoc to the seqInvoc
//Purpose: to prepare for the construction of the
//sequential relations (Section 3.2.1) between current
//invocation and all the invocations that belong to the
//previous action in the action sequence.
seqInvoc->at(1) := invoc;

return seqInvoc;
end if

```

A.9 ConstraintBuilder::deriveConstraint

Input

ist :IST

Output

ci :Sequence(Sequence(String))

```
function deriveConstraint(ist:IST):Sequence(Sequence(String))
```

```
var
```

```
v: Sequence(SubTree);
```

```
st :SubTree
```

subfunction

```
ConstraintBuilder::constraintForSubTree(st:SubTree):Sequence(String)
```

```
    Returns a sequence of constraints for subtree st
```

begin

```
    //create the constraint sequence for IST
```

```
    ci := new Sequence();
```

```
    //obtain IST's subtrees
```

```
    v := ist.subTree;
```

```
    //iterate all the subtrees and obtain constraints for each subtree
```

```
    for i:=1 to v->size do
```

```
        st := v->at(i);
```

```
        ci->append(constraintForSubTree(st));
```

```
    end for
```

```
    return ci;
```

```
end
```


A.10 ConstraintBuilder::constraintForSubTree

Input

st :SubTree

Output

cst :Sequence(String)

```
function ConstraintBuilder::constraintForSubTree(st:SubTree)
                                         :Sequence(String)
```

```
var
```

```
v :Sequence(SubScenario);
```

```
ss :SubScenario
```

```
subfunction
```

```
ConstraintBuilder::constraintForSubScenario(ss:SubScenario)
                                         :Sequence(String)
```

 Returns a sequence of constraints for subscenario ss

```
begin
```

```
    //create the constraint sequence for subtree
```

```
    cst := new Sequence();
```

```
    //get all the subscenarios for the subtree
```

```
    v := st.root.subScenario();
```

```
    //Iterate all the subscenarios and get the constraint sequence for
```

```
    //each subscenario
```

```
    for i:=1 to v->size do
```

```
        ss := v->at(i);
```

```
        cst->append(constraintForSubScenario(ss));
```

```
    end for
```

```
    return cst;
```

```
end
```

A.11 ConstraintBuilder::constraintForSubScenario

Input

ss :SubScenario

Output

css :Sequence(String)

```
function constraintForSubScenario(ss:SubScenario):Sequence(String)
```

```
var:
```

```
istAction :Sequence(ISTAction);
```

```
istNodeSeq :Sequence(ISTNode);
```

```
act :ISTNode;
```

```
constraint,postcond,invocCond,clause,lhs,rhs :String;
```

```
css,lhsSeq,rhsSeq,tempSeq,clauseSeq :Sequence(String);
```

subfunction

```
ConstraintBuilder::getImpliesLhsSeq(s:String):Sequence(String)
```

Returns a sequence of Strings that are left-hand-side of all the implies-expression conjuncts

```
ConstraintBuilder::getImpliesRhsSeq(s:String):Sequence(String)
```

Returns a sequence of Strings that are right-hand-side of all the implies-expression conjuncts

```
ConstraintBuilder::getClauseSeq(s:String):Sequence(String)
```

Returns the sequence of all the conjuncts in a string, which is a conjunction of clauses.

```
ConstraintBuilder::getLHS(equation:String):String
```

Returns the left-hand-side of an equality

```
ConstraintBuilder::getRHS(equation:String):String
```

Returns the right-hand-side of an equality

```
ConstraintBuilder::replace(s:String,s1:String,s2:String):String
```

replace every occurrence of s1 in s by s2, then remove time expression "@pre" from the result.

begin

```
//create the sequence, which is the ISTEEvent node followed by the
```

```
//sequence of ISTAction nodes in the subscenario ss
```

```
istNodeSeq := ss.istAction->prepend(ss.istEvent);
```

```
//get size of the sequence
```

```
n := istNodeSeq->size;
```

```
//get the last ISTNode in the sequence
```

```
act := istNodeSeq->last;
```

```
//initialize the constraint variable to be the invocation condition
```

```
//of the last invocation in the subscenario
```

```
constraint := act.incomingInvoc.invocCond.expression;
```

```
//create the constraint for the subscenario
```

```
css := new Sequence();
```

```
//add constraint to css
```

```
css->append(constraint);
```

```
//iterate nodes in istNodeSeq in a backward way, from the second
```

```
//last node to the first node
```

```

for i:=n-1 to 1 do
  //get the ith node
  act := istNodeSeq->at(i);
  //get postcondition
  postcont := act.postcond.expression;
  //get invocation condition of incoming invocation
  invocCond := act.incomingInvocation.invocCond.expression;

  //the type of a postcondition is either "impliedConjunction" or
  //"clauseConjunction"
  if postcond.form = "impliedConjunction" then
    //get the sequence of LHS and RHS of all the implies-
    //expressions in the postcondition
    lhsSeq := getImpliesLhsSeq(postcond);
    rhsSeq := getImpliesRhsSeq(postcond);

    //create a temporary constraint sequence
    tempSeq := new Sequence();

    for j:=1 to rhsSeq->size do
      //get the jth implies-expression's RHS, which is a
      //conjunction of clauses
      clauseSeq := rhsSeq->at(j);
      //iterate each clause in the clauseSeq
      for k:=1 to clauseSeq->size do
        //get the kth clause, which is in standard equality form
        clause := clauseSeq->at(k);
        //get the LHS and RHS of the clause
        lhs := getLHS(clause);
        rhs := getRHS(clause);

        //for kth clause, iterate all the constraints in css
        for m:=1 to css->size do
          //replace every occurrence of lhs in css's mth
          //constraint by rhs, and then remove "@pre" from the
          //result
          css->at(m) := replace(css->at(m), lhs, rhs);
        end for
      end for
      //iterate all the constraints in css and combine each
      //constraint with invocation condition and the jth implies-
      //predicate
      for m:=1 to css->size do
        constraint := css->at(m) + "and" + invocCond +
          "and" + lhsSeq->at(j);
        tempSeq->append(constraint);
      end for
    end for

    //assign the temporary constraint sequence to css
    css := tempSeq;

  else //the type of the postcondition is "clauseConjunction"
    //get the sequence of clauses of the postcondition
    clauseSeq := getClauseSeq(postcond);
    //iterate each clause, which is in standard equality form
    for j:=1 to clauseSeq do

```

```

clause := clauseSeq->at(j);
//get the LHS and RHS of the clause
lhs := getLHS(clause);
rhs := getRHS(clause);

//for jth clause, iterate all the constraints in css
for k:=1 to css->size do
    //replace every occurrence of lhs in css's kth
    //constraint by rhs, and then remove "@pre" from the
    //result
    css->at(k) := replace(css->at(k),lhs,rhs);
    //combine kth element in css with invocation condition
    css->at(k) := css->at(k) + "and" + invocCond;
end for
end for
end if //the type of postcondition
end for //for all nodes in istNodeSeq
end

```

Appendix B – Contract Patterns and Corresponding Complementing Rules

This appendix lists all possible types of changes that might be applied to each type of model elements. A contract pattern is a specific form for modeling a particular change to certain types of model elements (Section 3.4.5.2). It serves as guidance for writing good and testable contracts. Such patterns are designed with the aim to be correct, complete, concise and easy to understand and use. For each contract pattern, we define a corresponding complementing rule that transforms the original postcondition and adds complement postconditions to the operation (Section 3.4.5.2). 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.

A contract pattern usually consists of two parts: a precondition and a postcondition. The complementing rules take into account the preconditions when complementing the postconditions. Therefore, information from preconditions is already encompassed in the postconditions. As a result, algorithms that derive test constraints only need to handle the postconditions.

In the following, possible changes, their contract patterns, and corresponding complementing rules are listed according to the type of the model elements. Appendix B.1 is about user-defined class types; Appendix B.2 is about predefined OCL basic types; And appendix B.3 is about predefined OCL collection types).

B.1 Changes to a model element of a user-defined class type

A user-defined class-type model element is an object. Changes to an object assign it an object reference. The object can be (1) the keyword `self`, which refer to the statechart owning object (2) a class type attribute of the owning object (3) a class type argument of the operation (4) an element chosen from a collection by the `select` operation (5) an element chosen from a sequence by the `first`, `last` and `at` operation.

The pattern of a contract that models such a change is as follows:

```
context className::operation ()  
  
/***** pattern *****/  
post: object = anObject
```

The complemented postcondition remains the same as the contract pattern.

```
/***** Complemented postcondition *****/  
post: object = anObject
```

B.2 Changes to a model element of a predefined basic type

A basic type model element is either an attribute of an object or an argument of the operation. Changes to a basic type model element assign it a new value, whose type is the same as that of the model element. The new value could be any combination of attribute pre-values, arguments and literals.

The contract pattern that models a change to a basic type argument is as follows:

```
context className::operation ()  
  
/***** pattern *****/  
pre: basicTypeArg = oldBasicTypeValue  
post: basicTypeArg = newBasicTypeValue
```

The complemented postcondition remains the same as the contract pattern.

```
/***** Complemented postcondition *****/  
pre: basicTypeArg = oldBasicTypeValue  
post: basicTypeArg = newBasicTypeValue
```

The contract pattern that models a change to a basic type attribute is as follows:

```
context className::operation ()

/***** pattern *****/
pre: attribute = oldBasicTypeValue
post: attribute = newBasicTypeValue
```

The change to an attribute value also affects the size of certain collections. These collections are sub-collections that are obtained by applying the `select` operation on two collections: (1) the collection obtained by navigating a sequence of associations from the owning class to the class that owns the attribute. (2) the collection that contains all instances of the class that owns the attribute. For each collection, two sub-collections are selected. One sub-collection contains all objects with the attribute having the old attribute value, the size of which decreases by one. The other contains all objects with the attribute having the new attribute value, the size of which increases by one. The complemented postcondition is shown as follows:

```
/***** Complemented postcondition *****/
post:
attribute = newBasicTypeValue
and
navigationPath->select(attribute = oldBasicTypeValue)->size
    = navigationPath@pre->select(attribute =
        oldBasicTypeValue)->size + (-1)
and
navigationPath->select(attribute = newBasicTypeValue)->size
    = navigationPath@pre->select(attribute =
        newBasicTypeValue)->size + 1
and
className.allInstances->select(attribute = oldBasicTypeValue)
    ->size = className.allInstances@pre->select
        (attribute = oldBasicTypeValue)->size + (-1)
and
className.allInstances->select(attribute = newBasicTypeValue)
    ->size = className.allInstances->select
        (attribute = newBasicTypeValue)->size + 1
```

B.3 Changes to a model element of a predefined Collection type

In general, modifications to a Collection type object may be to empty a Collection; to add elements to a Collection; to remove elements from a Collection; or to assign a collection of objects to a Collection. For one of the concrete Collection types, the Sequence, the modification may also relate to the index of elements in the Sequence. When defining the complementing rule for a contract pattern that models changes to a sequence, we need to establish the relation between the object at an index in the sequence's after-value and (1) its index in the sequence's pre-value or (2) the argument of the operation. The purpose is to propagate the invocation condition (a constraint on the after-value of the sequence) that refers to an object at a specific index in a sequence to be a constraint on the pre-value of the sequence. An invocation condition before the propagation might refer to an object at three different types of indexes, the beginning, the end and any index (including the first and the last indice) in the sequence. Examples are:

- (1) `[sequence->first.attribute coreRelationalOperator additiveExpression]`
- (2) `[sequence->last.attribute coreRelationalOperator additiveExpression]`
- (3) `[sequence->at(j).attribute coreRelationalOperator additiveExpression]`

A variable `j` is introduced in the complemented postcondition to represent the index of an object specified in the invocation condition. For instance, if an operation adds object to the end of sequence, then its complemented postcondition is:


```

post:
(sequence->size > j implies (
  sequence = sequence@pre->Union(Sequence{object})
  and
  sequence->size = sequence@pre->size + 1
  and
  sequence->count(object) = sequence@pre->count(object)+1
  and
  sequence->at(sequence@pre->size +1) = object
  and
  sequence->at(j) = sequence@pre->at(j)
))
and
(sequence->size = j implies (
  sequence = sequence@pre->Union(Sequence{object})
  and
  sequence->size = sequence@pre->size + 1
  and
  sequence->count(object) = sequence@pre->count(object)+1
  and
  sequence->at(sequence@pre->size +1) = object
  and
  sequence->at(j) = object
))

```

It should be noted that this postcondition is transformed from the postcondition as follows²²:

```

post:
sequence = sequence@pre->Union(Sequence{object})
and
sequence->size = sequence@pre->size + 1
and
sequence->count(object) = sequence@pre->count(object)+1
and
sequence->at(sequence@pre->size +1) = object
and
(sequence->size > j implies sequence->at(j) = sequence@pre->at(j))
and
(sequence->size = j implies sequence->at(j) = object)

```

Notice that the predicate part of the last two implies-expressions (in italic font) are two mutually exclusive conditions that check the index *j*, which is the index of the object referenced inside the invocation condition. We can use the value of *j* to replace its

²² This transformation is done during the normalization of the logical expressions (Section 3.4.2). It is required by the test constraint derivation approach that the postcondition is either a conjunction of clauses or a conjunction of implies-expressions.

occurrence inside the implemented postcondition when propagating the invocation condition. The above invocation condition example (1) references the first object in the sequence, so $j=1$. After propagating the invocation condition based on the postcondition, the resulting invocation condition has two alternatives. The first alternative is:

```
[sequence->size > 1 and
sequence->at(1)=sequence@pre->at(1) and
(sequence->at(1).attribute coreRelationalOperator
additiveExpression)
]
```

It can be further transformed to be:

```
[sequence->size > 1 and
(sequence@pre->at(1).attribute coreRelationalOperator
additiveExpression)
]
```

The second alternative is:

```
[sequence->size = 1 and
sequence->at(1)=object and
(sequence->at(1).attribute coreRelationalOperator
additiveExpression)
]
```

It can be further transformed to be:

```
[sequence->size = 1 and
(object.attribute coreRelationalOperator additiveExpression)
]
```

Both alternatives are feasible results after propagating the invocation condition.

Similarly, the above invocation condition (2) references the last object, so $j=sequence->size$. After propagating the invocation condition based on the postcondition, the resulting invocation condition has two alternatives. The first alternative is:

```
[sequence->size > sequence->size and
sequence->at(sequence->size)=sequence@pre(sequence->size) and
(sequence->at(sequence->size).attribute coreRelationalOperator
additiveExpression)
]
```

The second alternative is:

```
[sequence->size = sequence->size and  
sequence->at(sequence->size)=object and  
(sequence->at(sequence->size).attribute coreRelationalOperator  
additiveExpression)  
]
```

It can be further transformed to be:

```
[object.attribute coreRelationalOperator additiveExpression]]
```

It is obvious that only the second alternative is a feasible result after propagating the invocation condition. The first alternative has an infeasible conjunct `sequence->size>sequence->size`.

The above invocation condition (3) references the object at any index in the sequence. After propagating the invocation condition based on the postcondition, the resulting invocation condition has two alternatives. The first alternative is:

```
[sequence->size > j and  
sequence->at(j) = sequence@pre->at(j) and  
(sequence->at(j).attribute coreRelationalOperator  
additiveExpression)  
]
```

It can be further transformed to be:

```
[sequence->size > j and  
(sequence@pre->at(j).attribute coreRelationalOperator  
additiveExpression)  
]
```

The second alternative is:

```
[sequence->size = j and  
sequence->at(j)=object and  
(sequence->at(j).attribute coreRelationalOperator  
additiveExpression)  
]
```

It can be further transformed to be:

```
[sequence->size = j and  
object.attribute coreRelationalOperator additiveExpression)]
```

If `j=sequence->size`, then only the second alternative is feasible. Otherwise, both alternatives are feasible.

In the following we list the complementing rules according to the type of changes to a Collection. If the complemented postconditions vary for different concrete Collection type, each concrete Collection type has a corresponding complementing rule.

B.3.1 Empty a Collection

When a Collection is emptied, the contract pattern that models such a change is as follows:

```
context className::operation ()

/***** pattern *****/
post: aCollection->size = 0 (aCollection->isEmpty())
```

The complemented postcondition remains the same as the contract pattern.

```
/***** Complemented postcondition *****/
post: aCollection->size = 0
```

B.3.2 Add an object to a Collection

The contract patterns and their corresponding complemented postconditions vary for each concrete type of Collection, so we discuss them separately.

B.3.2.1 Add an object to a Set

The contract pattern that models such a change is as follows:

```
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:

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

B.3.2.2 Add an object to a Bag

The contract pattern that models such a change is as follows:

```
context className::operation ([object:objectTypeParameter])

/****** pattern *****/
post: bag = bag@pre->including(object)
```

The complemented postcondition is:

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

B.3.2.3 Add an object to a Sequence

There are three indice inside a Sequence to add the object, the beginning, the end and anywhere in the middle. These cases are discussed separately.

Add the object to the end of the Sequence

The contract pattern that models such a change is as follows:

```
context className::operation ([object:objectTypeParameter])

/****** pattern *****/
post: sequence = sequence@pre->append(object)
```

The complemented postcondition is:

```
/****** Complemented postcondition *****/
post:
(sequence->size > j implies (
  sequence = sequence@pre->Union(Sequence{object})
  and
  sequence->size = sequence@pre->size + 1
  and
  sequence->count(object) = sequence@pre->count(object)+1
  and
  sequence->at(sequence@pre->size +1) = object
  and
  sequence->at(j) = sequence@pre->at(j)
))
and
(sequence->size = j implies (
  sequence = sequence@pre->Union(Sequence{object})
  and
  sequence->size = sequence@pre->size + 1
  and
  sequence->count(object) = sequence@pre->count(object)+1
  and
  sequence->at(sequence@pre->size +1) = object
  and
  sequence->at(j) = object
))
))
```

Add the object to the beginning of the Sequence

The contract pattern that models such a change is as follows:

```
context className::operation ([object:objectTypeParameter])
/****** pattern *****/
post: sequence = Sequence{object}->union(sequence@pre)
```

The complemented postcondition is:

```
/****** Complemented postcondition *****/
post:
(j > 1 implies (
  sequence = Sequence{object}->union(sequence@pre)
  and
  sequence->size = sequence@pre->size + 1
  and
  sequence->count(object) = sequence@pre->count(object)+1
  and
  sequence->at(1) = object
  and
  sequence->at(j) = sequence@pre->at(j-1)
))
and
```

```

(j = 1 implies (
  sequence = Sequence{object}->union(sequence@pre)
  and
  sequence->size = sequence@pre->size + 1
  and
  sequence->count(object) = sequence@pre->count(object)+1
  and
  sequence->at(1) = object
  and
  sequence->at(j) = object
))

```

Insert the object to *i*th index of the Sequence, *i* is an argument of the operation

The contract pattern that models such a change is as follows:

```

context className::operation (i:Integer
  [,object:objectTypeParameter])

/***** pattern *****/
post:
sequence->subsequence(i+1,sequence@pre->size+1) =
  sequence@pre->subSequence(i,sequence@pre->size)
and
sequence->size = sequence@pre->size + 1
and
sequence->at(i) = object

```

The complemented postcondition is:

```

/***** Complemented postcondition *****/
post:
(i>j implies (
  sequence = sequence@pre->subSequence(1,i-1)->
    union(Sequence{object})->union(sequence@pre->
      subSequence(i,sequence@pre->size))
  and
  sequence->size = sequence@pre->size + 1
  and
  sequence->count(object) = sequence@pre->count(object)+1
  and
  sequence->at(i)= object
  and
  sequence->at(j) = sequence@pre->at(j)
))
and
(j=i implies (
  sequence = sequence@pre->subSequence(1,i-1)->
    union(Sequence{object})->union(sequence@pre->
      subSequence(i,sequence@pre->size))
  and
  sequence->size = sequence@pre->size + 1

```

```

    and
    sequence->count(object) = sequence@pre->count(object)+1
    and
    sequence->at(i)= object
    and
    sequence->at(j) = object
  ))
and
(j>i implies (
  sequence = sequence@pre->subSequence(1,i-1)->
    union(Sequence{object})->union(sequence@pre->
      subSequence(i,sequence@pre->size))
  and
  sequence->size = sequence@pre->size + 1
  and
  sequence->count(object) = sequence@pre->count(object)+1
  and
  sequence->at(i)= object
  and
  sequence->at(j) = sequence@pre->at(j-1)
))

```

B.3.3 Add a Collection of objects to a Collection

The contract patterns and their corresponding complemented postconditions vary for each concrete type of Collection, so we discuss them separately.

B.3.3.1 Add a set of objects to a Set

The contract pattern that models such a change is as follows:

```

context className::operation ([aSet:setTypeParameter])

/***** pattern *****/
pre:  set->excludesAll(aSet)
post:
set->includesAll(aSet)
and
set->size=set@pre->size + aSet->size

```

The complemented postcondition is:

```

/***** Complemented postcondition *****/
post:
set = set@pre->union(aSet)
and
set->size=set@pre->size + aSet->size

```


B.3.3.2 Add a set/bag of objects to a Bag

The contract pattern that models such a change is as follows:

```
context className::operation ([aBag:bagTypeParameter |
    aSet:setTypeParameter])

/***** pattern *****/
post:
bag = bag@pre->union((aBag|aSet))
and
bag->size=bag@pre->size + (aBag|aSet)->size
```

The complemented postcondition remains the same.

```
/***** Complemented postcondition *****/
post:
bag = bag@pre->union((aBag|aSet))
and
bag->size=bag@pre->size + (aBag|aSet)->size
```

B.3.3.3 Add a sequence of objects to a Sequence

There are three indice inside a Sequence to add another Sequence, the beginning, the end and anywhere in the middle. These cases are discussed separately.

Add another Sequence to the end of a Sequence

The contract pattern that models such a change is as follows:

```
context className::operation ([aSequence:sequenceTypeParameter])

/***** pattern *****/
post:
sequence = sequence@pre->union(aSequence)
and
sequence->size=sequence@pre->size + aSequence->size
```

The complemented postcondition is:

```
/****** Complemented postcondition *****/
post:
((sequence@pre->size>j or sequence@pre->size=j) implies (
    sequence = sequence@pre->union(aSequence)
    and
    sequence->size=sequence@pre->size + aSequence->size
    and
    sequence->at(j) = sequence@pre->at(j)
))
and
(j > sequence@pre->size implies (
    sequence = sequence@pre->union(aSequence)
    and
    sequence->size=sequence@pre->size + aSequence->size
    and
    sequence->at(j) = aSequence->at(j-sequence@pre->size)
))
```

Add another Sequence to the beginning of a Sequence

The contract pattern that models such a change is as follows:

```
context className::operation ([aSequence:sequenceTypeParameter])

/****** pattern *****/
post:
sequence = aSequence->union(sequence@pre)
and
sequence->size=sequence@pre->size + aSequence->size
```

The complemented postcondition is:

```
/****** Complemented postcondition *****/
post:
((aSequence->size>j or aSequence->size=j) implies (
    sequence = aSequence->union(sequence@pre)
    and
    sequence->size=sequence@pre->size + aSequence->size
    and
    sequence->at(j) = aSequence->at(j)
))
and
(j > aSequence->size implies (
    sequence = aSequence->union(sequence@pre)
    and
    sequence->size=sequence@pre->size + aSequence->size
    and
    sequence->at(j) = sequence@pre->at(j-aSequence->size)
))
```

Insert another Sequence to the ith index of a Sequence, i is an argument of the operation

The contract pattern that models such a change is as follows:

```
context className::operation (i:Integer
    [, aSequence:sequenceTypeParameter])

/***** pattern *****/
post:
sequence = sequence@pre->subSequence(1,i-1)->
    union(aSequence)->union(sequence@pre->
    subSequence(i,sequence@pre->size)
and
sequence->size=sequence@pre->size + aSequence->size
```

The complemented postcondition is:

```
/***** Complemented postcondition *****/
post:
(j < i implies (
    sequence = sequence@pre->subSequence(1,i-1)->
    union(aSequence)->union(sequence@pre->
    subSequence(i,sequence@pre->size)
    and
    sequence->size=sequence@pre->size + aSequence->size
    and
    sequence->at(j) = sequence@pre->at(j)
))
and
((j>i or j=i) and (i+aSequence->size>j or i+aSequence->size=j)
    implies (
    sequence = sequence@pre->subSequence(1,i-1)->
    union(aSequence)->union(sequence@pre->
    subSequence(i,sequence@pre->size)
    and
    sequence->size=sequence@pre->size + aSequence->size
    and
    sequence->at(j) = aSequence->at(j-i)
))
and
(j > i + aSequence->size implies (
    sequence = sequence@pre->subSequence(1,i-1)->
    union(aSequence)->union(sequence@pre->
    subSequence(i,sequence@pre->size)
    and
    sequence->size=sequence@pre->size + aSequence->size
    and
    sequence->at(j) = sequence@pre->at(j-i-aSequence->size)
))
))
```

B.3.4 Create and add an object to a Collection.

The object satisfies a property specified by Boolean expression `expr`, which test the object's properties against `expr1`, which is an argument of the operation. The contract patterns and their corresponding complemented postconditions vary for each concrete type of Collection, so we discuss them separately.

B.3.4.1 Create and add the object to a Set

The contract pattern that models such a change is as follows:

```
context className::operation (expr1:oclExpression)

/***** pattern *****/
pre: not set->exists(expr)
post:
set->exists(expr)
and
set->size = set@pre->size + 1
and
className.allInstances->size = className.allInstances@pre->size+1
and
className.allInstances->select(expr)->size
    = className.allInstances@pre->select(expr)->size + 1
```

The complemented postcondition is:

```
/***** Complemented postcondition *****/
post:
set = set@pre->union(className.allInstances->
    select(elem|expr and elem.oclIsNew))
and
set->select(expr) = className.allInstances->
    select(elem|expr and elem.oclIsNew)
and
set->select(expr)->size = 1
and
set->size = set@pre->size + 1
and
className.allInstances->size = className.allInstances@pre->size+1
and
className.allInstances->select(expr)->size =
    className.allInstances@pre->select(expr)->size + 1
```

B.3.4.2 Create and add the object to a Bag

The contract pattern that models such a change is as follows:

```
context className::operation (expr1:oclExpression)

/***** pattern *****/
post:
bag->size = bag@pre->size+1
and
bag->select(expr)->size = bag@pre->select(expr)+1
and
className.allInstances->size = className.allInstances@pre->size+1
and
className.allInstances->select(expr)->size =
    className.allInstances@pre->select(expr)->size + 1
```

The complemented postcondition is:

```
/***** Complemented postcondition *****/
post:
bag = bag@pre->union(className.allInstances->
    select(elem|expr and elem.oclIsNew))
and
bag->select(expr) = bag@pre->select(expr)->
    union(className.allInstances->
        select(elem|expr and elem.oclIsNew))
and
bag->size = bag@pre->size+1
and
bag->select(expr)->size=bag@pre->select(expr)+1
and
className.allInstances->size = className.allInstances@pre->size+1
and
className.allInstances->select(expr)->size =
    className.allInstances@pre->select(expr)->size + 1
```

B.3.4.3 Create and add the object to a Sequence

There are three indices inside a sequence to add the object, the beginning, the end and anywhere in the middle. These cases are discussed separately.

Create and add the object to the end of the Sequence

The contract pattern that models such a change is as follows:

```
context className::operation (expr1:oclExpression)

/***** pattern *****/
post:
sequence = sequence@pre->append(className.allInstances->select
    (elem|expr and oclIsNew))
and
className.allInstances->size = className.allInstances@pre->size+1
and
className.allInstances->select(expr)->size =
    className.allInstances@pre->select(expr)->size + 1
```

The complemented postcondition is:

```
/***** Complemented postcondition *****/
post:
(sequence@pre->size+1>j implies (
    sequence = sequence@pre->append(className.allInstances->select
        (elem|expr and oclIsNew))
    and
    sequence->select(expr) = sequence@pre->select(expr)->
        union(className.allInstances->select(elem|expr and oclIsNew))
    and
    sequence->last = className.allInstances->
        select(elem|expr and oclIsNew)
    and
    sequence->size = sequence@pre->size+1
    and
    sequence->select(expr)->size = sequence@pre->select(expr)+1
    and
    className.allInstances->size = className.allInstances@pre
        ->size+1
    and
    className.allInstances->select(expr)->size =
        className.allInstances@pre->select(expr)->size + 1
    and
    sequence->at(j) = sequence@pre->at(j)
))
and
(j = sequence@pre->size + 1 implies (
    sequence = sequence@pre->append(className.allInstances->select
        (elem|expr and oclIsNew))
    and
    sequence->select(expr) = sequence@pre->select(expr)->
```

```

        union(className.allInstances->select(elem|expr and oclIsNew))
    and
    sequence->last = className.allInstances->
        select(elem|expr and oclIsNew)
    and
    sequence->size = sequence@pre->size+1
    and
    sequence->select(expr)->size = sequence@pre->select(expr)+1
    and
    className.allInstances->size = className.allInstances@pre
        ->size+1
    and
    className.allInstances->select(expr)->size =
        className.allInstances@pre->select(expr)->size + 1
    and
    sequence->at(j) = className.allInstances->
        select(elem|expr and oclIsNew)
))

```

Create and add the object to the beginning of the Sequence

The contract pattern that models such a change is as follows:

```

context className::operation (expr1:oclExpression)

/***** pattern *****/
post:
sequence = sequence@pre->prepend(className.allInstances->select
    (elem|expr and oclIsNew))
and
className.allInstances->size = className.allInstances@pre->size+1
and
className.allInstances->select(expr)->size =
    className.allInstances@pre->select(expr)->size + 1

```

The complemented postcondition is:

```

/***** Complemented postcondition *****/
post:
(j > 1 implies (
  sequence = sequence@pre->prepend(className.allInstances->select
    (elem|expr and oclIsNew))
  and
  sequence->select(expr) = className.allInstances
    ->select(elem|expr and oclIsNew)->
    union(sequence@pre->select(expr))
  and
  sequence->first = className.allInstances->
    select(elem|expr and oclIsNew)
  and
  sequence->size = sequence@pre->size+1
  and
  sequence->select(expr)->size=sequence@pre->select(expr)+1
  and
  className.allInstances->size = className.allInstances@pre
    ->size+1
  and
  className.allInstances->select(expr)->size =
    className.allInstances@pre->select(expr)->size + 1
  and
  sequence->at(j) = sequence@pre->at(j-1)
))
and
(j = 1 implies (
  sequence = sequence@pre->prepend(className.allInstances->select
    (elem|expr and oclIsNew))
  and
  sequence->select(expr) = className.allInstances
    ->select(elem|expr and oclIsNew)->
    union(sequence@pre->select(expr))
  and
  sequence->first = className.allInstances->
    select(elem|expr and oclIsNew)
  and
  sequence->size = sequence@pre->size+1
  and
  sequence->select(expr)->size=sequence@pre->select(expr)+1
  and
  className.allInstances->size = className.allInstances@pre
    ->size+1
  and
  className.allInstances->select(expr)->size =
    className.allInstances@pre->select(expr)->size + 1
  and
  sequence->at(j) = className.allInstances->select(elem|expr
    and oclIsNew)
))

```


Create and insert the object to the ith index of the Sequence

The contract pattern that models such a change is as follows:

```
context className::operation (i:Integer, expr1:oclExpression)

/***** pattern *****/
post:
sequence = sequence@pre->subSequence(1,i-1)->
    union(Sequence{className.allInstances->select(elem|expr and
    oclIsNew)}->union(subSequence(i,sequence@pre->size))
and
className.allInstances->size = className.allInstances@pre->size+1
and
className.allInstances->select(expr)->size =
    className.allInstances@pre->select(expr)->size+1
```

The complemented postcondition is:

```
/***** Complemented postcondition *****/
post:
(i > j implies (
    sequence = sequence@pre->subSequence(1,i-1)->union(Sequence
    {className.allInstances->select(elem|expr and oclIsNew)}->
    union(subSequence(i,sequence@pre->size))
    and
    sequence->select(expr) = sequence@pre->subSequence(1,i-1)->
    select(expr)->union(className.allInstances->select(elem|expr
    and oclIsNew))->union(subSequence(i,sequence@pre->size))->
    select(expr)
    and
    sequence->at(i) = className.allInstances->
    select(elem|expr and oclIsNew)
    and
    sequence->size = sequence@pre->size+1
    and
    sequence->select(expr)->size=sequence@pre->select(expr)+1
    and
    className.allInstances->size=className.allInstances@pre->size+1
    and
    className.allInstances->select(expr)->size =
    className.allInstances@pre->select(expr)->size+1
    and
    sequence->at(j) = sequence@pre->at(j)
))
and
(j = i implies (
    sequence = sequence@pre->subSequence(1,i-1)->union(Sequence
    {className.allInstances->select(elem|expr and oclIsNew)}->
    union(subSequence(i,sequence@pre->size))
    and
    sequence->select(expr) = sequence@pre->subSequence(1,i-1)->
    select(expr)->union(className.allInstances->select(elem|expr
    and oclIsNew))->union(subSequence(i,sequence@pre->size))->
    select(expr)
```

```

and
sequence->at(i) = className.allInstances->
  select(elem|expr and oclIsNew)
and
sequence->size = sequence@pre->size+1
and
sequence->select(expr) ->size=sequence@pre->select(expr)+1
and
className.allInstances->size=className.allInstances@pre->size+1
and
className.allInstances->select(expr) ->size =
  className.allInstances@pre->select(expr) ->size+1
and
sequence->at(j) = className.allInstances->
  select(elem|expr and oclIsNew)
))
and
(j > i implies (
  sequence = sequence@pre->subSequence(1,i-1) ->union(Sequence
    {className.allInstances->select(elem|expr and oclIsNew)}->
    union(subSequence(i,sequence@pre->size)))
  and
  sequence->select(expr) = sequence@pre->subSequence(1,i-1) ->
    select(expr) ->union(className.allInstances->select(elem|expr
    and oclIsNew)) ->union(subSequence(i,sequence@pre->size)) ->
    select(expr)
  and
  sequence->at(i) = className.allInstances->
    select(elem|expr and oclIsNew)
  and
  sequence->size = sequence@pre->size+1
  and
  sequence->select(expr) ->size=sequence@pre->select(expr)+1
  and
  className.allInstances->size=className.allInstances@pre->size+1
  and
  className.allInstances->select(expr) ->size =
    className.allInstances@pre->select(expr) ->size+1
  and
  sequence->at(j) = sequence@pre->at(j-1)
))

```

B.3.5 Add a Collection to a Collection.

Objects in the Collection to be added satisfy property `expr`, which tests objects' attributes against an argument `expr1` of the operation.

Such operations use the `select` operation on a Collection to get a sub-Collection that satisfies `expr` and then add this sub-Collection to another Collection.

The Collection on which the “select” operation works is either known to the operation or is passed in as an argument of the operation.

The contract patterns and their corresponding complemented postconditions vary for each concrete type of Collection, so we discuss them separately.

B.3.5.1 Add a set of objects to a Set

The contract pattern that models such a change is as follows:

```
context className::operation(expr1:oclExpression
    [, aSet:setTypeParameter])

/***** pattern *****/
post:
set = set@pre->union(aSet->select(expr))
and
set->select(expr)=set@pre->select(expr)->union(aSet->
    select(expr))
```

The complemented postcondition is:

```
/***** Complemented postcondition *****/
post:
set = set@pre->union(aSet->select(expr))
and
set->select(expr)=set@pre->select(expr)->
    union(aSet->select(expr))
and
set->size = set@pre->size + aSet->select(expr)->size
    +(-set->intersection(aSet->select(expr))->size)
and
set->select(expr)->size = set@pre->select(expr)->size +
    aSet->select(expr)->size+(-set->intersection(aSet->
    select(expr))->size)
```

B.3.5.2 Add a bag/set of objects to a Bag

The contract pattern that models such a change is as follows:

```
context className::operation(expr1:oclExpression
    [, (aBag:bagTypeParameter|aSet:setTypeParameter)])

/***** pattern *****/
post:
bag = bag@pre->union(aBag->select(expr))
and
bag->select(expr)=bag@pre->select(expr)->union(aBag->
    select(expr))
```

The complemented postcondition is:

```
/****** Complemented postcondition *****/
post:
bag = bag@pre->union(aBag->select(expr))
and
bag->select(expr)=bag@pre->select(expr)->union(aBag-
->select(expr))
and
bag->size = bag@pre->size + aBag->select(expr)->size
and
bag->select(expr)->size = bag@pre->select(expr)->size +
aBag->select(expr)->size
```

B.3.5.3 Add a sequence of objects to a Sequence

There are three indice inside a sequence to add another sequence. They are discussed separately.

Add a sequence of objects to the end of a Sequence

The contract pattern that models such a change is as follows:

```
context className::operation(expr1:oclExpression
[, aSequence:sequenceTypeParameter])

/****** pattern *****/
post:
sequence = sequence@pre->union(aSequence->select(expr))
and
sequence->select(expr)=sequence@pre->select(expr)->
union(aSequence->select(expr))
```

The complemented postcondition is:

```

/***** Complemented postcondition *****/
post:
((sequence@pre->size>j or sequence@pre->size=j) implies (
  sequence = sequence@pre->union(aSequence->select(expr))
  and
  sequence->select(expr)=sequence@pre->select(expr)->
  union(aSequence->select(expr))
  and
  sequence->size=sequence@pre->size+aSequence->select(expr)->size
  and
  sequence->select(expr)->size = sequence@pre->select(expr)->size
  +
  aSequence->select(expr)->size
  and
  sequence->at(j)=sequence@pre->at(j)
))
and
(j > sequence@pre->size implies (
  sequence = sequence@pre->union(aSequence->select(expr))
  and
  sequence->select(expr)=sequence@pre->select(expr)->
  union(aSequence->select(expr))
  and
  sequence->size=sequence@pre->size+aSequence->select(expr)->size
  and
  sequence->select(expr)->size = sequence@pre->select(expr)->size
  +
  aSequence->select(expr)->size
  and
  sequence->at(j) = aSequence->select(expr)->
  at(j-sequence@pre->size)
))

```

Add a sequence of objects to the beginning of a Sequence

The contract pattern that models such a change is as follows:

```

context className::operation(expr1:oclExpression
  [,aSequence:sequenceTypeParameter])

/***** pattern *****/
post:
sequence = aSequence->select(expr)->union(sequence@pre)
and
sequence->select(expr)= aSequence->select(expr)->
union(sequence@pre->select(expr))

```

The complemented postcondition is:

```
/****** Complemented postcondition *****/
post:
((aSequence->select(expr)->size>j or aSequence->
    select(expr)->size=j) implies (
    sequence = aSequence->select(expr)->union(sequence@pre)
    and
    sequence->select(expr)= aSequence->select(expr)->
    union(sequence@pre->select(expr))
    and
    sequence->size = sequence@pre->size+aSequence->select(expr)-
    >size
    and
    sequence->select(expr)->size = sequence@pre->select(expr)->size
    +
    aSequence->select(expr)->size
    and
    sequence->at(j) = aSequence->select(expr)->at(j)
))
and
(j > aSequence->select(expr)->size implies (
    sequence = aSequence->select(expr)->union(sequence@pre)
    and
    sequence->select(expr)= aSequence->select(expr)->
    union(sequence@pre->select(expr))
    and
    sequence->size = sequence@pre->size+aSequence->select(expr)-
    >size
    and
    sequence->select(expr)->size = sequence@pre->select(expr)->size
    +
    aSequence->select(expr)->size
    and
    sequence->at(j) = sequence@pre->at(j-aSequence->
    select(expr)->size)
))
```

Insert a sequence of objects to the *i*th index of a Sequence

The contract pattern that models such a change is as follows:

```
context className::operation(expr1:oclExpression, i:Integer
    [, aSequence:sequenceTypeParameter])

/****** pattern *****/
post: sequence = sequence@pre->subSequence(1,i-1)->
    union(aSequence->select(expr))->
    union(sequence@pre->subSequence(i, sequence@pre->size))
```

The complemented postcondition is:

```

/***** Complemented postcondition *****/
post:
(j < i implies (
  sequence = sequence@pre->subSequence(1,i-1)->
    union(aSequence->select(expr))->
    union(sequence@pre->subSequence(i,sequence@pre->size)
  and
  sequence->size=sequence@pre->size+aSequence->select(expr)->size
  and
  sequence->at(j) = sequence@pre->at(j)
))
and
((j>i or j=I) and (i+aSequence->select(expr)->size>j or
  i+aSequence->select(expr)->size=j) implies (
  sequence = sequence@pre->subSequence(1,i-1)->
    union(aSequence->select(expr))->
    union(sequence@pre->subSequence(i,sequence@pre->size)
  and
  sequence->size=sequence@pre->size+aSequence->select(expr)->size
  and
  sequence->at(j) = aSequence->select(expr)->at(j-i)
))
and
(j > i + aSequence->select(expr)->size implies (
  sequence = sequence@pre->subSequence(1,i-1)->
    union(aSequence->select(expr))->
    union(sequence@pre->subSequence(i,sequence@pre->size)
  and
  sequence->size=sequence@pre->size+aSequence->select(expr)->size
  and
  sequence->at(j)=
    sequence@pre->at(j-i- aSequence->select(expr)->size)
))

```

B.3.6 Delete an object from a Collection.

The object is specified by an object reference. The contract patterns and their corresponding complemented postconditions vary for each concrete type of Collection, so we discuss them separately.

B.3.6.1 Delete an object from a Set.

The contract pattern that models such a change is as follows:

```
context className::operation([object:objectTypeParameter])

/***** pattern *****/
pre:  set->includes(object)
post:
  set->excludes(object)
  and
  set->size=set@pre->size+(-1)
```

The complemented postcondition is:

```
/***** Complemented postcondition *****/
post: set = set@pre-Set{object}
  and
  set->size=set@pre->size+(-1)
  and
  set->count(object) = 0
```

B.3.6.2 Delete all occurrences of an object from a Bag.

The contract pattern that models such a change is as follows:

```
context className::operation([object:objectTypeParameter])

/***** pattern *****/
pre:  bag->includes(object)
post: bag->excludes(object)
```

The complemented postcondition is:

```
/***** Complemented postcondition *****/
post:
  bag = bag@pre->excluding(object)
  and
  bag->size = bag@pre->size+(-bag@pre->count(object))
  and
  bag->count(object)= 0
```

B.3.6.3 Delete an object from a Sequence.

There are various ways to delete an object from a sequence. These are listed separately.

Delete all occurrences of an object from a Sequence.

The object is specified by an object reference.

The contract pattern that models such a change is as follows:

```
context className::operation([object:objectTypeParameter])

/***** pattern *****/
pre: sequence->includes(object)
post: sequence->excludes(object)
```

The complemented postcondition is:

```
/***** Complemented postcondition *****/
post:
sequence = sequence@pre->excluding(object)
and
sequence->size = sequence@pre->size-sequence@pre->count(object)
and
sequence->count(object) = 0
```

Delete the last object from a Sequence

The contract pattern that models such a change is as follows:

```
context className::operation()

/***** pattern *****/
post: sequence = sequence@pre->subSequence(1,
sequence@pre->size-1)
```

The complemented postcondition is,

```
/***** Complemented postcondition *****/
post:
sequence->size = sequence@pre->size + (-1)
and
sequence = sequence@pre->subSequence(1,sequence@pre->size-1)
and
sequence->at(j) = sequence@pre->at(j)
```

Delete the first object of a Sequence

The contract pattern that models such a change is as follows:

```
context className::operation()

/***** pattern *****/
post: sequence = sequence@pre->subSequence(2,sequence@pre->size)
```

The complemented postcondition is:

```
/****** Complemented postcondition *****/
post:
sequence->size = sequence@pre->size + (-1)
and
sequence = sequence@pre->subSequence(2, sequence@pre->size)
and
sequence->at(j) = sequence@pre->at(j+1)
```

Delete the object at the *i*th index in a Sequence.

The contract pattern that models such a change is as follows:

```
context className::operation(i:Integer
    [,object:objectTypeParameter])

/****** pattern *****/
post:
sequence = sequence@pre->subSequence(1,i-1)->
    union(sequence@pre->subSequence(i+1, sequence@pre->size))
```

The complemented postcondition is:

```
/****** Complemented postcondition *****/
post:
(i>j implies (
    sequence->size = sequence@pre->size + (-1)
    and
    sequence = sequence@pre->subSequence(1,i-1)->
        union(sequence@pre->subSequence(i+1, sequence@pre->size))
    and
    sequence->at(j) = sequence@pre->at(j)
))
and
((j>i or j=i) implies (
    sequence->size = sequence@pre->size + (-1)
    and
    sequence = sequence@pre->subSequence(1,i-1)->
        union(sequence@pre->subSequence(i+1, sequence@pre->size))
    and
    sequence->at(j) = sequence@pre->at(j+1)
))
```

B.3.7 Delete a Collection of objects from a Collection

The Collection is specified by a reference. The contract patterns and their corresponding complemented postconditions vary for each concrete type of Collection, so we discuss them separately.

B.3.7.1 Delete a set of objects from a Set

The contract pattern that models such a change is as follows:

```
context className::operation (aSet:setTypeParameter)

/***** pattern *****/
pre: set->includesAll(aSet)
post:
set->excludesAll(aSet)
and
set->size=set@pre->size+(-aSet->size)
```

The complemented postcondition is:

```
/***** Complemented postcondition *****/
post:
set = set@pre - aSet
and
set->size=set@pre->size+(-aSet->size)
```

B.3.7.2 Delete a bag/set of objects from a Bag

In UML, there is no bag operation defined to model this type of changes. We do not define contract patterns and complementing rules for such changes.

B.3.7.3 Delete a sequence of objects from a Sequence

In UML, there is no sequence operation defined to model this type of changes. We do not define contract patterns and complementing rules for such changes.

B.3.8 Delete an object or a collection of objects from a Collection

The object or objects to be deleted satisfy a property specified by `expr`, which test the object's properties against `expr1`, which is an argument of the operation.

B.3.8.1 Delete an object or a set of objects from a Set

The contract pattern that models such a change is as follows:

```

context className::operation (expr1:oclExpression)

/***** pattern *****/
pre: set->exists(expr)
post:
not set->exists(expr)
and
set->size = set@pre->size+(-set@pre->select(expr)->size)

```

The complemented postcondition is:

```

/***** Complemented postcondition *****/
post:
set = set@pre->select(not expr)
and
set->select(expr)->size = 0
and
set->size = set@pre->size+(-set@pre->select(expr)->size)

```

B.3.8.2 Delete an object or a bag of objects from a Bag

The contract pattern that models such a change is as follows:

```

context className::operation (expr1:oclExpression)

/***** pattern *****/
pre: bag->exists(expr)
post:
not bag->exists(expr)
and
bag->size = bag@pre->size+(-bag@pre->select(expr)->size)

```

The complemented postcondition is:

```

/***** Complemented postcondition *****/
post:
bag = bag@pre->select(not expr)
and
bag->select(expr)->size = 0
and
bag->size = bag@pre->size+(-bag@pre->select(expr)->size)

```

B.3.8.3 Delete an object or a sequence of objects from a Sequence

The contract pattern that models such a change is as follows:

```
context className::operation (expr1:oclExpression)

/***** pattern *****/
pre:  sequence->exists(expr)
post:
not sequence->exists(expr)
and
sequence->size=sequence@pre->size+(-sequence@pre->
    select(expr)->size)
```

The complemented postcondition is:

```
/***** Complemented postcondition *****/
post:
sequence = sequence@pre->select(not expr)
and
sequence->select(expr)->size = 0
and
sequence->size = sequence@pre->size+(-sequence@pre->
    select(expr)->size)
```