

Trace-Based Object-Oriented Application Testing

Diplomarbeit

der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Michael Freidig

October 2003

Leiter der Arbeit:

Prof. Dr. S. Ducasse

Prof. Dr. O. Nierstrasz

Institut für Informatik und angewandte Mathematik

Abstract

Testing the behavior of object-oriented systems is an important activity in the software development and maintenance process. It validates an expected behavior against an observed behavior. A behavioral test is an assertion over a set of messages and objects states that occur during the execution of a system. Testing behavior is especially important for object-oriented legacy systems where current behavior is the only thing we can trust because a specification is often missing.

There are two problems with testing behavior of object-oriented systems. First there exists no common form to express a hypothesis about an expected behavior and to validate it against an actual program behavior. This has the consequence that behavioral tests are carried out manually by stepping through an execution with a debugger and asserting behavioral properties by visually inspecting states, arguments and messages in the context of the execution history of the system. Second it is a priori not clear what kind of behavior should be tested and how it is represented in terms of message passing and state changes. This causes additional friction when setting up tests for behaviors that occur over and over again in different systems.

In this thesis the concept of trace-based object-oriented testing is introduced. It supports the specification of an expected behavior in the form of a formal expression and an automatic test of whether an expected behavior occurs in previously recorded execution trace. A prototype tool TESTLOG on the basis of the logic language SOUL is developed that supports trace-based object-oriented testing in the form of a logic query over a trace. As a validation of the concept behavioral tests for different types of behaviors that frequently occur in object-oriented systems are designed and documented in the form of a pattern language.

The use of the computational power of a logic language for behavioral testing solves the problem of automatically identifying if an expected behavior occurs in an execution trace. A set of pre-defined logic rules serves as a language to compose complex behavioral tests such that a tester can take advantage of the intrinsic rule abstraction facility of SOUL. In order to identify recurring behavioral concepts we classify behavior and try to abstract general purpose templates for different types of behavior in order to obtain reusable behavioral test artifacts.

Acknowledgments

I thank Markus Gaelli and Stéphane Ducasse for reviewing this document.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Overview of Trace-based Object-Oriented Testing	2
1.3	Goal	2
1.4	Contribution	3
1.5	Structure	3
2	Object-Oriented Testing	5
2.1	What is Software Testing	5
2.2	Different Kinds of Tests	6
2.2.1	Scopes of Tests	7
2.3	Problems with Testing Behavior	8
2.3.1	Specifying Behavioral Tests	8
2.3.2	Testing Object States	9
2.3.3	Templates for Behavioral Tests	9
3	Trace-based Object-Oriented Application Testing	11
3.1	The Concept of Trace-based Object-Oriented Testing	11
3.2	Using Dynamic Information	12
3.3	Language Support for Trace-based Testing	12
3.4	Example	14
3.5	Benefits of Trace Based Testing	16
3.6	Related Work	16
4	Tests for Different Types of Behavior	19

4.1	List of Patterns for Behavioral Tests	19
4.2	Message Sequences	19
4.3	State-Based Testing	25
4.4	Recursion	30
4.5	Case Study: Importing MOOSE Models	36
4.6	Conclusion	39
5	Implementation	41
5.1	TESTLOG: A Prototype Tool	41
5.1.1	SOUL Syntax and Symbiosis with Smalltalk	42
5.2	Representation of the Event Trace	43
5.2.1	Reification of the Event Model	44
5.2.2	Reification of Object States	45
5.3	Queries over the Event Trace	46
5.3.1	Bank Example	47
5.4	Basic Logic Queries over the Event Trace	48
5.4.1	Event Querying	49
6	Conclusion	57
6.1	Summary	57
6.2	Further Work	57
6.2.1	Test Libraries	57
A	Logic Code For Tree Pattern Matcher	59
A.1	Event Tree Pattern Matcher	59
	Bibliography	61

List of Figures

1.1	Overview of trace-based testing	3
2.1	Engineering view on testing	6
3.1	Moose model reification	15
4.1	Example Scenario	21
5.1	Architecture of TESTLOG	41
5.2	Object-oriented event model	44
5.3	Reified State Model	46
5.4	Bank Model	47

Chapter 1

Introduction

1.1 Motivation

Software systems are human made artifacts that expose complex behavior. In order to validate whether an actual behavior conforms to an expected behavior it is necessary to test a software system. Behavior in object-oriented systems emerges from dynamic artifacts such as message sends, object creations and state changes, that occur during an execution. There are situations where a tester wants to make a statement about program behavior affecting numerous message sends and object-states. This is for example the case when testing a collaboration between multiple objects and one is interested in the sequence of messages exchanged and object states at intermediate steps of a scenario. In this document program behavior is considered as an ordered set of execution events and event attributes. A behavioral test is defined as checking for the occurrence of a subset of events that satisfy a certain predicate.

In contrast to unit testing, where the focus lies on testing behavior through application of a stimulus and asserting a resultant state, testing behavior through an analysis of multiple message sends and object states is a different approach for testing behavior. The main reason for introducing a new concept is that one cannot test every behavior that is of interest with a unit testing pattern. Let's consider for example the observer pattern as described in [ABW98]. In order to adequately test the observer pattern one must take into account the collaboration between a subject and its registered observers. When a subject receives a change message, it responds with an updated message to every registered observer that in turn takes an appropriate action. A test of this collaboration requires an analysis of messages exchanged between objects that is not feasible with a unit test. Because it is painful to manually analyse a behavior that spawns over many steps of an execution, there is an urgent need to create a language that enables a tester to formally codify a test and automatically evaluate whether it passes or fails.

Testing program behavior can be carried out during an execution such as in [Gué03]. Another possibility is to record an execution trace and examine it after the execution has finished as it was done in [Duc99]. The approach chosen in this work is to first record an execution trace and later

analyse it. This gives the opportunity to iteratively refine a statement about program behavior by trace analysis without reexecuting the program several times. Because behavioral tests are performed by analysing an execution trace, the concept introduced in this document is called "Trace-Based Object-Oriented Testing".

There are different usage scenarios for testing object-oriented behavior by analysing a trace. We can test for example the collaborations among multiple objects and object states at intermediate steps of a computation. For example the domain of behavioral design patterns we could make a precise statement about the object interactions that occur in the pattern. This is important, because behavior of design patterns in its current state is described informal as natural language text. For execution of single methods pre- and postconditions can be asserted. Furthermore every application exposes domain specific behavior that has its representation in the microworld of interacting objects and offers an interesting target for testing.

1.2 Overview of Trace-based Object-Oriented Testing

In order to test the behavior of object-oriented systems a conceptual approach called "*Trace Base Object-Oriented Testing*" and a prototype tool with the name *TESTLOG* are developed. The concept for testing behavior is based recording an execution trace and reifying messages and object states in an event model. Then a tester runs a logic query over the execution trace in order to test a behavioral property. If a test passes or fails is then determined whether a query over the execution trace finds a solution or not. In figure 5.2 we see a general overview of the concept. A user stimulates a system to produce a behavior. Then he forms a conjecture about an expected behavior and expresses it in form of a logic query. A test is performed by evaluating whether an expected behavior matches an observed behavior located in the trace. In order to perform a regression test, the system is reexecuted so that it produces the same trace again. This can be achieved by coding test drivers that allow one to repeatedly apply the same stimulus.

1.3 Goal

The general goal of this thesis is to find ways to use dynamic information for testing object-oriented systems. The reuse of dynamic information should facilitate testing by shifting away a tester's focus from the time consuming construction of tests by coding to a precise diagnosis and examination of a system's behavior. This is especially important in the context of a maintenance scenario where every change can heavily impact and alter a system's behavior.

To reach this goal a language for specifying behavioral tests as an expression over an execution trace was developed. Its usability is demonstrated by giving different examples of behavioral tests. To get a better understanding of object-oriented behavior, different kinds of behaviors are identified and described in terms of basic execution events. The concept of trace-based object-oriented testing is validated by creating a prototype tool that implements a language for

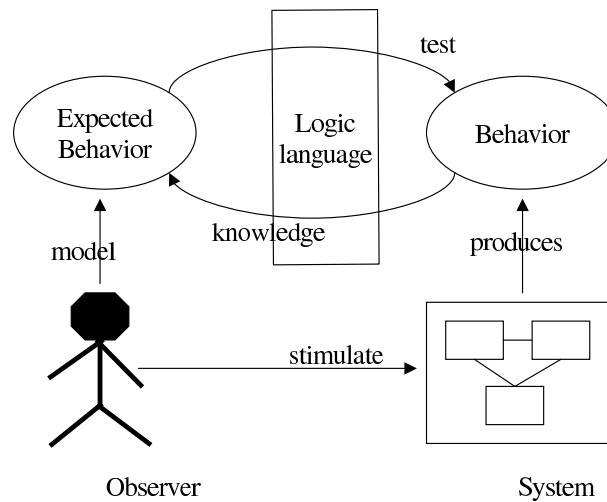


Figure 1.1: Overview of trace-based testing

specifying behavioral tests and supports automatic execution of them.

1.4 Contribution

This thesis introduces a new concept for testing that focuses on behavioral aspects of an object-oriented system. Trace-based object-oriented testing is established as a concept for carrying out behavioral tests by declaratively codifying an expected behavior and checking its conformance against an observed behavior.

In order to support trace-based object-oriented testing an event model to represent dynamic information and a language to express behavioral test was developed. The event model is representing message sends and objects states. A recorder reifies entities in the event model during system execution. In order to express behavioral tests, a language on top of the logic meta language SOUL is developed. It supports different layers of reasoning about behavior from simple event querying to complex behaviors in application systems. The language is open and extensible because it is based on the rule architecture of SOUL.

On the basis of trace-based object-oriented testing a list of patterns for testing different types of behavior is discovered and described.

1.5 Structure

This document is structured as follows. Chapter 2 gives a general background about testing and object-oriented testing and discusses some problems associated with testing object-oriented

programs. In Chapter 3 the concept of trace-based object-oriented testing is introduced. In Chapter 4 trace-based object-oriented testing is applied to different types of behavior. Chapter 5 introduces the prototype tool TESTLOG and some of its general purpose queries. Chapter 6 gives the conclusion.

Chapter 2

Object-Oriented Testing

2.1 What is Software Testing

A good introduction into the concepts of software testing can be found in [Bin99]. This chapter classifies software testing as a research and an engineering discipline.

When we consider testing as an engineering discipline, we are talking about an activity in the software engineering process that is concerned with designing test cases based on specifications of an application and automatically executing test cases against an implementation of the specified application as visualized in 2.1. A test case thereby specifies the pretest state of an implementation under test, the input provided for the test and the expected result. The design of test cases is supported by generic models such as equivalence class partitioning of input values, decision tables, cause effect graphs or state machines and procedures of how to generate test cases from those models.

After having specified test cases in terms of test design models a further activity is the implementation of a test automation system. The main advantages of having automated tests is reproducibility and preciseness. Reproducibility allows one to repeat a test case after a change in the software as a regression test. The preciseness of automated test is much higher than that of manual tests because automated test require a description in terms of an executable language such as a testscript in a scripting language.

Test execution then deals with executing a test case for an implementation under test and obtaining a pass fail status by comparing an expected result against an actual result. Besides evaluating a pass or fail status another goal while executing test is the evaluation of coverage. Coverage is the percentage of elements exercised by a given test suite. Elements that can be considered for coverage are statements, branch, methods or states in a state event model.

Besides being an engineering discipline testing is also a research discipline. The goal of testing as a research discipline is to find models for testing and test automation. A good introduction on testing as a research discipline is given in [Har00].

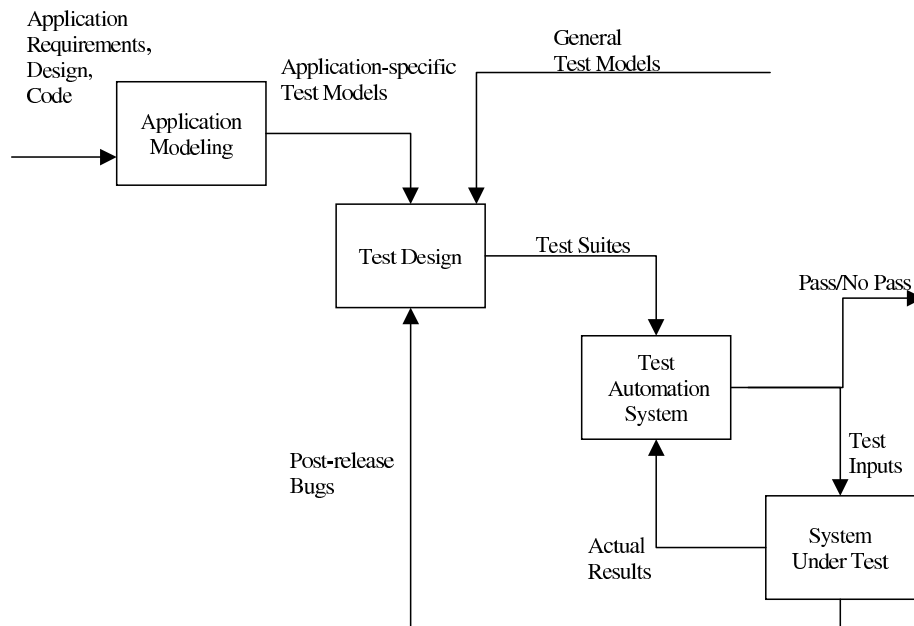


Figure 2.1: Engineering view on testing

There are many branches of research about testing. An important one is concerned with deriving test cases by statically analyzing source code artifacts such as in [BM00]. Although those approaches provide sophisticated algorithms to generate test inputs so that a certain coverage is achieved their usage is limited because expected values cannot be computed.

Another subdiscipline is concerned with deriving test cases from formal specifications such as in [Bar97]. The main drawback of this approach however is that existing implementations are rarely based on formal specifications so that this approach is seldom usable. There is a lot of work in research on object-oriented testing that focuses on automatically creating test cases for a class by statically analyzing the structure of a class such as in [KGH⁺95].

Other researchers address the problem of selecting regression test cases after a change has been made to the software for example in [RHD00]. The goal of this work is to reduce the cost of testing imposed by creating and executing a complete regression test suite.

2.2 Different Kinds of Tests

The general term *testing* denotes a wide range of activities in the software development process each with different scope and goals. In addition to executing a system and validating its response

there are the following kinds of testing.

Installation Test The goal of an installation test is to validate whether the components of a systems can be deployed on a target platform so that the systems is still running in the new environment. It should also be checked whether every component can be uninstalled again.

Documentation Test A documentation test deals with checking the completeness, correctness and the understandability of every element of the documentation. Documentation generally includes manual and online help.

Performance Test A performance tests measures the response times of a system, typically in the context of a concurrent user simulation. In case of unsatisfactory response times the challenge is to identify the components that contribute most to the delay and optimize them.

Recovery Test A recovery test simulates the failure of the whole or a part of a system with the goal to recover from the failure in a short time and without loss of data. Recovery tests are important for critical systems in the finance business because of a potentially large economical impact.

Acceptance Test An acceptance test includes the end user in the testing process. The goal of an acceptance test is to validate the usability of the software for the domain of the end user.

2.2.1 Scopes of Tests

The scope of a test is the collection of software components to be verified. Software testing is typically categorized by the scope of the implementation under test. Scope is traditionally designated as unit, integration or system.

Unit Test The scope of a unit test in object-oriented programming typically is a class or a method. Because classes often have dependencies to other classes the smallest individually executable unit is a cluster of interdependent classes. A unit test requires a tester to set up a set of instances in a state so that a test message or stimulus produces an intended response.

Integration Testing The scope of an integration test is a complete system or subsystem of software units. The units are interdependent and must cooperate to meet some requirement. Integration testing exercises interfaces among units within the specified scope to demonstrate that the units are collectively operable. Integration testing begins early in the programming of object-oriented software because single class typically has relationships to other classes an inherits features from superclasses.

System test A system test targets a whole application or a set of application that are connected . Tests focus on capabilities and characteristics that are present only with the entire system. System scope tests may be categorized by the kind of conformance they seek to establish. functional (input/ output), performance (response time and resource utilization) , stress or load (response under maximum or overload).

2.3 Problems with Testing Behavior

Testing an object-oriented system means to validate a property of its behavior [Bin99]. In order to understand the difficulties associated with testing behavior, it is first necessary to understand how to characterize it. Pure object oriented languages expose two dynamic properties or execution events that are representing behavior a the lowest level of granularity. It is message passing between objects and changes of an object's state. Messages are passed from a sender to a receiver with an ordered list of arguments and a method in the class hierarchy of the receiver is executed. State changes can be described as change of the value of an instance variable. The change of an object's state within the context of a single operation described as pre- and postcondition is the concept that is generally understood as behavior and targeted for unit testing.

In addition to single operations one can also consider sets of execution events that are representing higher level behavioral concepts. For example one could think of a collaboration that is described as a set of messages exchanged among different objects. Behavioral tests should also take into account this form of behavior. As a behavioral test we understand therefore the general concept of examining properties of a set of execution events. This general definition subsumes the set of behavioral tests that are carried out as unit tests and target a single operation and the tests that take into account a whole scenario consisting of many messages and states.

2.3.1 Specifying Behavioral Tests

A general form of a behavioral test can target a scenario consisting of many messages and object states. An example of a behavioral test that targets a scenario is given in Chapter 3. The current practice to test a scenario is to use a debugger and step through an execution of a program. Although it is possible to test a scenario using a debugger together with inspectors, it is an uncomfortable procedure because it requires a tester to control a program by stepping through its execution and in parallel perform a test of an expected behavior. Besides that it is also difficult to filter out those messages and states, that are representing a behavior of interest among the whole set of messages that are sent during an execution. In contrast to unit testing, there exists no language to declare and codify a scenario based test. This means that a tester has to form a conjecture about an expected behavior in his head and validate it manually against the behavior he observes during an execution. This has the consequence that it is difficult to reproduce scenario based tests because they are not made explicit by being written down in terms of a well defined language. This fact makes it also hard to share knowledge among different testers because the

don't have a common basis to exchange scenario based tests.

2.3.2 Testing Object States

Object-oriented behavior is a symbiosis between message passing and state changes. When testing a scenario where many object states occur, one has the need to make assertions about states and state changes. Some common tasks that a tester has to accomplish are:

- Accessing an objects state before and after an operation
- Test whether a link between two objects exists, is established or detached
- Access the state of an object in the recursive state of another object

When carrying out a trace-based test one would also like to have the possibility to assert state as it can be done with assertion in the source code or with SUnit tests. Therefore it is essential to provide a set of abstraction that allow a test developer to express state based assertions about object states that occur in a trace.

2.3.3 Templates for Behavioral Tests

When practicing object-oriented testing and programming one can observe that there are behaviors that occur over and over again in different situations. For example a type of behavior, that frequently occurs, is passing an object as an argument and adding it to the state of a receiver. When testing behavior one is confronted with the question which strategy is applicable to test a certain form of behavior. When adding an object in the recursive state of an other object a strategy would be to query an object state an see whether the newly added object is returned.

If one could achieve sufficient knowledge about what kind of behaviors occur, one could create test templates for them that embody a specific test strategy. In analogy to design patterns that help a designer to solve a specific design problem such a behavioral testing pattern could be used to solve a specific testing problem. Good candidates for behavioral testing patterns are those that concern behavior occurring with high frequency such as for example

- A collaboration between objects
- Establishment and detachment of links between objects
- Querying and returning an object from the recursive state of another object
- Recursion over a composite structure

The benefit of templates, that support a tester in solving small testing problems, is already shown in [Sil00] where a set of templates for unit testing are demonstrated. The identification of behavioral patterns and associated strategies for testing them could be the basis for an assistant that proposes a developer a strategy how to test a certain form of behavior. Such an assistant was first proposed by [Bro75] and could help to increase software development productivity because a big fraction of time while developing software is spent with testing.

Chapter 3

Trace-based Object-Oriented Application Testing

3.1 The Concept of Trace-based Object-Oriented Testing

Trace-based object-oriented testing is a concept, that supports the specification and execution of different kinds of behavioral tests. It is based on querying an execution trace with a logic language in order to assert a behavioral property. If a logic query succeeds, then the test passes otherwise it fails. Trace-based testing supports tests for the behavior of single operations and also for scenarios consisting of many events. Some key behavioral tests that can be carried out are

- Postconditions of arbitrary operations that occur in the trace
- Establishment and detachment of links between objects
- Message sequence taking into account causality among messages
- Any behavioral property that can be expressed in terms of messages and object states and can be specified as a recursive set over a set of execution events

Trace-based tests are performed in two steps. First a user exercises a system, so that the behavior that is the target for testing is produced. In order to execute a system one could rely on a graphical user interface or use existing test drivers. While the system is executing, a recorder records every message send and object state before and after the execution of a method. Everything that is recorded is then stored in a dynamic information model that serves as a target of analysis of a behavioral property of interest. In a second step, a tester expresses a behavioral test by specifying a query over the recorded execution trace. Such a query specifies an expected behavior in terms of message sends and object states. A test succeeds whenever the expected behavior can be identified in an execution trace.

3.2 Using Dynamic Information

Trace-based object-oriented testing is entirely based on dynamic artifacts such as messages, objects and their states. In order to test program behavior, dynamic information is more effective compared to static analysis because of the following reasons:

- A trace acts like a program slice with respect to control flow, since it limits the scope of our investigation to the particular scenario executed and so provides focus in the investigation.
- It is always precise with respect to the executed scenario since we know exactly which method has been invoked on which object.
- It is easy to obtain compared to static control flow analysis which proves difficult for large programs.
- It provides information not obtainable from any static analysis such as the number of instances and the multiplicity of relationships between objects.

The main argument against the use of dynamic information is its incomplete coverage of the code. But this very property is also its advantage. A program trace provides information about the behavior of the system in a scenario exercising a certain functionality, and so helps us to tie functionality to behavior.

3.3 Language Support for Trace-based Testing

Because of the huge number of message sends and object states that can be observed in the execution of a nontrivial system, manual evaluation of behavioral tests is not feasible. A better choice would be the use of computational power to carry out behavioral tests with the goal, to automate the evaluation of a behavioral test by carrying out a *computation over an event trace*.

Besides automatic evaluation we need a language that provides appropriate abstractions to specify behavioral tests. A logic language would be an appropriate basis for such a requirement. A computation over an event trace would then be described as a logic query and the pass or fail semantics of a behavioral test established by a success or failure of the query. Other properties of a logic language such as pattern matching, unification of variables with terms and the ability to build abstractions with logic rules are usable in our domain. However there are additional requirements that are not covered by standard implementations of logic languages such as accessing an object-oriented model that is representing an execution trace or unifying objects with variables which is an important requirements when reasoning about states of objects. A language that is supporting this requirement is SOUL (Smalltalk Open Unification Language) [Wuy01]. SOUL is a logic language implemented in Smalltalk and supports unification of logic variables with objects and allows the user to query object-oriented models without having to represent them as logic facts.

A language that supports trace-based testing must also support a set of domain specific requirements. Trace-based object-oriented testing is mainly concerned with identifying events or sets of events according to events attributes that are representing a behavior. By thinking about what kind of generic behavioral artifacts occur in object-oriented systems we can generate requirements for basic logic queries that can identify them.

Below is a incomplete list of generic behavioral artifacts that frequently occur in object-oriented systems.

- A message is passed from a sender object to a receiver object with arguments.
- An object is returned to the sending context after each method execution.
- New objects are created
- Messages can be nested, so that a call tree is formed
- Objects have state that can change
- Objects are passed around as arguments

Having identified basic behavioral artifacts we can now specify requirements for queries that can identify them.

- Identify a message according to its selector, sender or receiver object and the objects that are passed as arguments
- Identify the creation of new objects of different classes
- Pattern match a call tree
- Identify an object state at a certain point of an execution
- Allow the detection of state changes
- Observe the history of an object as it is created, passed around as argument or serves as sender or receiver

Every test query for complex behavior can then be composed from a few basic queries.

3.4 Example

This section introduces an example for trace-based object-oriented application testing. It will show how a scenario based test can be used to make a statement about *how* a certain result is produced over several steps. The example deals with a scenario that occurs during an import of Smalltalk classes into the MOOSE reengineering model [DLT00]. The MOOSE reengineering model provides a model for representing source code artifacts that is similar to the UML or MOF but with the extension that it is also representing method invocations and attribute accesses [OMG99]. MOOSE provides importers for various source code languages such as C++, JAVA or Smalltalk. The task of an importer is to parse source code entities and produce a reified representation of them in the MOOSE model.

A MOOSE importer exposes interesting and complex behavior that is produced after calling the `importModel` method on the importers facade and contains method parse tree traversal, Smalltalk meta model access, object creation and the reification of model entities and the establishment of links between them. An import of a small model produces more than ten thousand message sends that are representing the complete behavior of an import operation.

The behavioral test we consider in the example concentrates on a slice of the importer behavior that deals with reifying a class entity, Figure 3.1 visualizes this behavior as a message sequence chart. The scenario starts with an `#importClass:` message sent to the importer facade. The system then produces a whole cascade of message sends. Just those that are interesting for the test are visualized. The reification of a class subsumes the messages `#reifyClass:`, `#ensureClassEntity:` and `#addObject:`. In method `#ensureClassEntity:` a new model entity is created and added to the model with `#addObject:`.

A behavioral test for this scenario is important because the importer performs the import over several intermediate steps that need to be correct. For example we would like to test whether the message `#reifyClass:` really produces a message `#addObject:` that adds an object that is representing the class passed as an argument or that the object returned by `ensureClassEntity` is the same object that is passed as an argument to `#addObject:`.

Lets think of the following behavioral test for the scenario in Figure 3.1.

"The class with name `#TheRoot` that is passed as an argument to `#reifyClass` is reified as a model entity and added to the model according to the scenario in Figure 3.1"

The next step in carrying out a behavioral test is to codify a statement about behavior in form of a logic query, so that it can be automatically checked against an execution trace.

```

1 event(?e1, selector(?e1, [#reifyClass]),
2 contains(<event(?e2, selector(?e2, [#ensureClassEntity])),
3     contains(<event(?e3, selector(?e3, [#addObject:]))>>)),
4 arguments(?e1, [1], ?class),
5 return(?e2, ?reifiedEntity),
6 resurrectReceiverAfterEvent(?e3, ?modelAfterAddObject),
7 includes(?modelAfterAddObject, ?reifiedEntity),

```

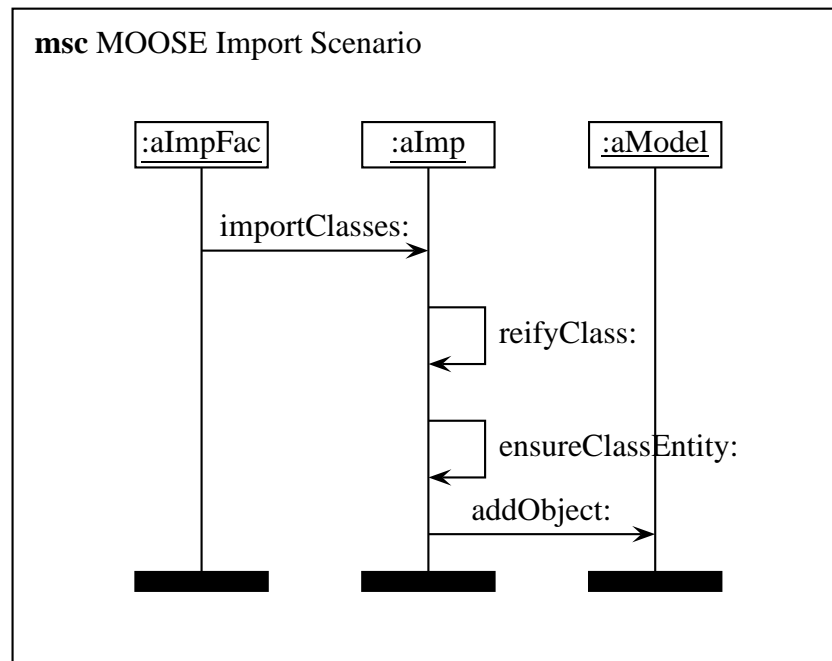


Figure 3.1: Moose model reification

8 equals([?class name], [?reifiedEntity name])

In lines 1 to 3 the trace is queried for a scenario fragment that starts with a message with selector `#reifyClass:` and includes further messages with selectors `#ensureClassEntity` and `#addObject` as shown in Figure 3.1. The variables `?e1`, `?e2` and `?e3` unify with events that are representing the corresponding messages in the trace. The variable `?class` is unified with the Smalltalk class object and `?reifiedEntity` references the reified class entity object of the MOOSE model and the object that is return by the method `#ensureClassEntity`. In line 6 the state of the model after the reified entity has been added is reconstructed in order to get the state with the original object as they were during the execution. In line 7 and 8 two assertion are performed. In line 7 it is checked whether the model includes the reified entity and in line 8 it is asserted whether the reified entity is representing the class that was passed as argument by comparing their name attributes.

This examples shows a scenario based test implemented as a logic query that takes into account multiple message sends and states. In addition to validate a postcondition, this test also targets the correct message exchange among multiple collaborators. In order to test the postcondition a reference to an object at an intermediate step of the computation is obtained and later used to perform a check. In order to test whether an object is included in the recursive state of another object, that state is reconstructed from the recorded trace so that the same objects with the same identities are accessible. This example shows general concepts that can be observed over and over again when performing trace-based tests.

- Specify a test as a logic query
- Identify a scenario in a trace
- Test the message exchange taking into account causality between messages
- Reconstruct an original state as it was during the last execution of the scenario
- Make a statement about objects at different position in a trace
- Validate a postcondition

3.5 Benefits of Trace Based Testing

Trace-based testing enables a tester to specify a scenario based test consisting of multiple operations by writing a logic query that declares an expected behavior. Scenario based tests can take into account the causality and sequence of message sends, object states at intermediate steps of a computation and the occurrence of the same object at different positions in the trace. The use of a query language frees a tester from the burden of controlling a program during an execution and manually inspecting message sends and states, so that behavioral testing can be carried out more efficiently than with a debugger.

Trace-based testing can be applied to different levels of abstraction. At the lowest level of behavioral abstraction is a single execution event. An event is representing a time interval that starts with sending a message to an object and ends when the control is returned to the sending object. We don't consider asynchronous messages here. At the highest behavioral abstraction level we can observe behavioral aspects of a whole system.

A logic language is also a convenient platform to create a rule base that provides templates for different types of behavior, because of its intrinsic abstraction and pattern matching facilities. A set of general purpose rules will help a tester to setup all kinds of behavioral tests just by parametrizing them for a specific situation. Such a rule base can consist of rules that are representing tests for generic behavioral artifacts and such that are implemented for domain specific behavior.

3.6 Related Work

There is little work on dynamic analysis and program testing the behavior of object-oriented systems. In a pioneering paper [JE94] the authors argue that testing object-oriented software should not focus on units but on the message exchange between them in a scenario, however they do not provide a computational infrastructure to do this.

Caffeine [Gué03] is a JAVA based tool that uses the debug API to capture execution events and uses also a Prolog variant to express and execute queries on a dynamic trace. The main difference

to TESTLOG is that Caffeine has a linear representation of a trace so that it is not possible reason about nested events. Caffeine is also missing state reification so that constraints on state cannot be expressed. Its main context is not testing but reasoning about dynamic properties in reverse engineering.

A second similar tool is OPIUM [Duc99] that allows a user to validate a prolog trace using a set of debugging queries. Prolog is used as a base language and as meta language to reason about events. The main usage scenario of OPIUM is the implementation of a high level debugger for Prolog that allows forward navigation to the next event that satisfies a certain condition.

Other work that is based on event models and computations over an event trace to test program behavior can be found in [Aug98][Aug95]. However it is based on procedural programming languages and does not take into account the specific behavioral aspects of object-oriented languages such object as creation and the state of objects. Furthermore the author does not reason about what kind of behaviors can occur in a program and how to test them.

Chapter 4

Tests for Different Types of Behavior

4.1 List of Patterns for Behavioral Tests

We present a list of patterns to document different types of behavioral test, because patterns have already been useful in [Bin99] for the documentation of various models to test object-oriented software. Each pattern documentation adheres to a template of sections as follows:

- **Intent:** Describes in brief memorizable form what it is all about.
- **Behavior:** Gives an overview what kind of behavior in object-oriented systems we intend to test. The main purpose of this section is to reveal the behavioral archetype under investigation.
- **Errors:** In this section some typical errors that could occur in the behavior under consideration are described.
- **Concept:** Under this section general purpose concepts for testing a specific kind of behavior are introduced. If possible rules and queries that can be used as templates are created. The intended usage of concepts is illustrated with examples

4.2 Message Sequences

Intent. Provide a set of generic queries to test message sequences. Provide pattern matching for message trees.

Behavior. A scenario is a sequence of interactions between multiple objects. A scenario starts with an initial event upon which a cascade of events are produced as response of a system. Based on that initial event, the response produced by the system can be different based on its state and the actual input.

Scenarios are behavioral archetypes that occur frequently in object-oriented systems because the object-oriented programming paradigm forces the developer to distribute the implementation of a service among different classes, each with its own responsibility. This has the consequence that different objects have to collaborate with each other in order to provide a service, so that often a complex cascade of messages is produced upon sending a message to a receiver. This is for example the case for the Facade pattern where a message sent to the facade interface produces a message exchange among numerous objects.

Scenarios can be found at different levels of abstraction. At a low level of abstraction a scenario describes the interaction between a small number of objects, such as for example in design patterns, then at higher abstraction levels the scope of a scenario can be a whole subsystems of an application that interact with each other.

While in procedural languages testing focuses on executing different paths in a procedure, in object-oriented systems the interaction between different objects are more interesting to test than single methods because most of the methods have a small number of statements and not more than a single path. The usability of scenarios as testable artifacts is already stated in [Bin99] p.290, where tests based on message sequence specifications are performed.

Errors. Testing message sequences is important because numerous errors can occur in an interaction between objects.

- Messages are sent that must not be sent
- A message is sent before another message, but must be sent afterwards
- A collaboration is not implemented as it was specified

Problems. A trace can consists of several hundred messages, containing such that are the target for a test and others that are not relevant. Accessors for example are called with a high frequency but are not relevant for testing a scenario. Therefore it is difficult to identify a scenario by manually stepping through a trace.

Concept. We provide a set of rules that allow a user to write tests for different forms of scenarios. We start with two simple forms of queries and present then a general form that can be used for any form of a scenario. The first form deals with testing whether a single message send produces another message send in its context. The second form targets testing a chain of messages and the third form allows one to express a generic scenario test.

Message Implication. Test whether a message send implies another message send in its context

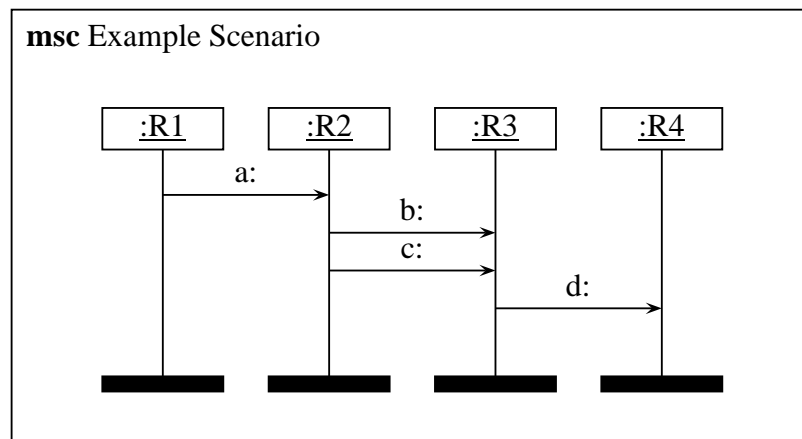


Figure 4.1: Example Scenario

```
event(?e1, ?query1, contains(event(?e2, ?query2))
```

In this query the variables `?query1` and `?query2` are queries that match event attributes of the two events to be queried. The query will succeed if there exists two events, so that `?query1` matches the first event, `?query2` matches the second event and the first event includes the second event in the event model. If the query succeeds then the variables `?e` and `?e1` are substituted with the corresponding event objects from the trace.

In order to test whether the message with selector `#a:` implies a message with selector `#d:` 4.1 we write the following query.

```
event(?e1, selector(?e1, [#a]),
  contains(event(?e2, selector(?e1, [#d])))
```

Message Path. Test if a chain of messages exists. We call a chain of message implications a *message path*. A message path can be used for example to test a chain of responsibilities.

```
messagePath(<?eventList>)
```

The query succeeds if every event query in `?eventList` matches and if for any two events in `?eventList` the event that occurs first in `?eventList` contains the event that follows in `?eventList`. If the query succeeds every event variable `?e` in the event term is substituted with the event object that matched. Single message implication could also be written as message path of length two.

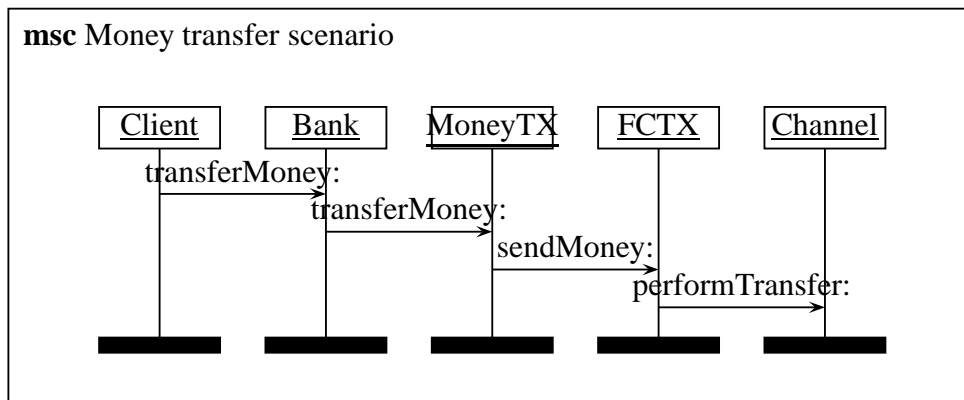
In order to test whether a message path with the selector `#a:`, `#b:` and `#d:` occurs in a scenario of Figure 4.1 we write the following query:

```
messagePath(<
  event(?e1, selector(?e1, [#a:])),
  event(?e2, selector(?e2, [#b:])),
  event(?e3, selector(?e3, [#d:]))>
)
```

The query succeeds because a message path can be found in the trace and the SOUL query interpreter produces the following result.

```
SOUL found
1 solutions in 101 ms for:
if event(?e) messagePath(
<event(?e1, selector(?e1, [#a:])),
event(?e2, selector(?e2, [#b:])),
event(?e3, selector(?e3, [#d:]))>
)
[?e1-->[#a:]]
[?e2-->[#b:]]
[?e3-->[#d:]]
```

We now show how a message path query can be used to test an implementation of the chain of responsibility design pattern. In the chain of responsibility pattern a request is passed along a chain of handlers until an object handles it. We would like to check whether the correct handlers are invoked on the right objects. Let's imagine the following scenario where a client of a bank wants to transfer money to a foreign country. He first sends a request to a facade object of the bank system. The request is delegated to the money transaction subsystem then to the transaction system that is responsible for handling foreign currency transaction and finally the request is handled by a channel that sends the money to the destination country.



In this query we first get the references to the handler objects, for example by accessing instances in the running system and then write a message path query to check the proper implementation of the chain of responsibility. In order to test this fragment of behavior we can write a message path query expression as follows:

```

getReferences(?bankFacade, ?MoneyTransactionSystem, ?FCSys,
?FCChannel),
messagePath(
<event(?e, selectorAndReceiver([#transferMoney:], ?bankFacade)),
event(?e1, selectorAndReceiver([#transferMoney:],
?MoneyTransactionSystem)),
event(?e2, selectorAndReceiver([#sendMoney:], ?FCSys)),
event(?e3, selectorAndReceiver([#performTransfer:], ?FCChannel))>
)

```

Scenario. The generic form of a scenario consisting of more than one message is an ordered and labelled tree. For each tree node there are a set of attributes that can be matched as it is described in 5. In order to match a scenario in the trace, we specify the structure of a subtree to be matched and for each tree node a query that matches its attributes. A scenario pattern is a recursive term that specifies a pattern tree to be matched in the trace. In order to define the recursive pattern, an query for a pattern event can contain a list of child events. The match succeeds if there exists an ordered embedding of the scenario pattern in the event tree.

```

event(?e, ?q,
contains(<event(?e1, ?nodeQuery), ?restOfEvents>)),

```

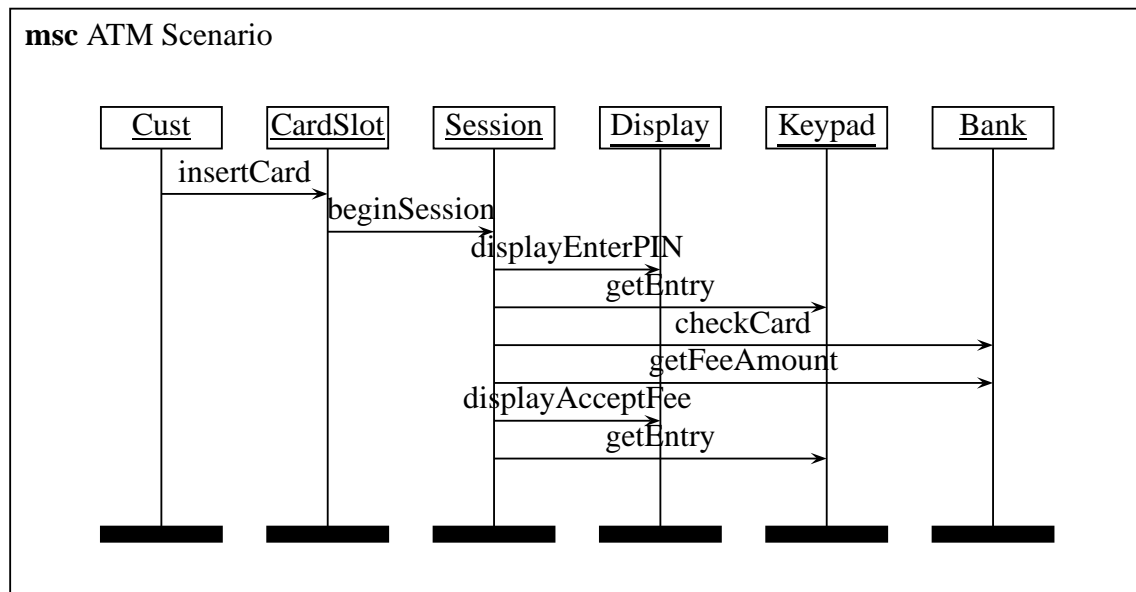
For the example in Figure 4.1 the scenario pattern matching term for the whole scenario looks as follows:

```
event(?e1, selector(?e, [#a:])), contains(<
  event(?e2, selector([#b:])),
  event(?e3, selector(?e2, [#c:], contains(<
    event(?e4, selector(?e4, [#d:]>))
  >))
)>)
```

By parametrizing a pattern matching expression we can test different variants of a scenario by taking advantage of the intrinsic iteration over a list in evaluation of a logic query. Imagine for example that instead of selector `#d:` the selectors `#g:` and `#h:` occur in different variants of the scenario above. Variants of a scenario are typically produced in context of conditional branches in method or in the context of polymorphic behavior. The example given is a very simple form of a parametrization of a scenario pattern, more complex forms are thinkable, i.e. one could parametrize event attribute queries or event whole subtrees of the pattern tree.

```
member(?s, <[#d:], [#g:], [#h:]>),
event(?e1, selector(?e, [#a:])), contains(<
  event(?e2, selector([#b:])),
  event(?e3, selector(?e2, [#c:], contains(<
    event(?e4, selector(?e4, [?s])>))
  >))
)>)
```

Now we shift to a more complex example that deals with testing a whole scenario. Often a developer is confronted with the situation where a the message sequence in a scenario is specified in a graphical form and he wants to make sure whether this specification is consistent with an implementation. The implementation of this scenario typically contains additional message that makes it hard to validate the design scenario. By using the pattern matching facility we can test the correct implementation of a design scenario more comfortable by writing an appropriate pattern matching query. The example deals with establishing a session with a hypothetical automatic teller machine (ATM).



In order to test the implementation of this scenario we assume that the developer of the implementation chooses the same method names as they were given in the message sequence chart. The expression to test this scenario the looks as follows. As one immediately sees there is an isomorphic mapping from the message sequence chart to a term expressing an event pattern tree. This query succeeds whenever there is an embedding of the scenario expression into the event hierarchy tree.

```

event(?e, selector(?e, [#insertCard:]),
contains(<
  event(?e1, selector(?e1, [#beginSession:],
contains(<
  event(?e2, selector(?e2, [#displayEnterPin:]))
  event(?e3, selector(?e3, [#getEntry:]))
  event(?e4, selector(?e4, [#checkCard:]))
  event(?e5, selector(?e5, [#getFeeAmount:]))
  event(?e6, selector(?e6, [#displayAcceptFee:]))
  event(?e7, selector(?e7, [#getEntry:]))
>))
>))
)

```

4.3 State-Based Testing

Intent. Describe behavior in terms of state changes. Provide queries for recursive state inspection and state changes. Show how to test the states of objects by using expression over an execution trace.

Behavior. There are many forms of behavior that can be tested by inspecting the state of one or more objects. One is the change of an object's state after the execution of an operation. State changes manifest themselves in establishment and detachment of links between objects instances or the change of primitive type values. In contrast to focus on a single operation the change of state of a single object through its lifetime can also be tested by asserting a sequence of states.

Errors. A couple of errors that can be detected through testing state are:

- Violation of a postcondition.
- Incorrect initialization of objects after creation.
- Return of a wrong object from a method

Problems. When testing state a tester is confronted with a set of task that are time consuming to carry out without a convenient infrastructure, but are supported by trace based testing.

- Access simultaneously the state of an object before and after an operation in order to test a change in state.
- Make an assertion over the recursive state of an object.
- Test whether a link between two instances exists.
- Access to a nested object in a recursive state

Concept. In this paragraph several general purpose concepts are presented that support the creation of state based tests.

Pre- and Postconditions. Postconditions are a statement about a state of an object before and after an operation and are therefore an adequate means to describe behavior in terms of state changes. Using postconditions for testing is already advocated in [MP00]. Trace-based testing is also an efficient means to validate a conjecture about a postcondition when examining legacy code, because with a single logic query it can be shown that a postcondition holds for every method execution that satisfies a precondition.

To validate postconditions by specifying an expression over an execution trace we need to accomplish the following steps.

- Identify the execution of an operation that is a target for postcondition validation
- Ensure the precondition
- Validate the postcondition

The general form of a logic rule for postcondition validation looks as follows:

```
validatePostcondition(
  event(?e, ?eventQuery),
  precondition(?e, ?preState, ?preconditionQuery),
  postcondition(?e, ?preState, ?postState, ?postconditionQuery))
```

Three queries are passed as arguments. The first query identifies the event for which pre- and postconditions should be ensured. The second argument is the precondition and the third argument is the postcondition.

Lets imagine a simple example of incrementing a bounded counter. The precondition asserts that the counter has not yet reached its upper bound. The postcondition asserts that the counter value has been incremented by one. A query to validate this postcondition looks as follows.

```
validatePostcondition(
  event(?e, selector(?e, [#increment:]),
  precondition(?e,
    ?preState,
    not(greatherThan([?preState value], ?upperBound)
  ))
  postcondition(?e,
    ?preState,
    ?postState,
    equals([?preState value + 1], [?postState value])
  )
)
```

Another example deals with a hypothetical bank application that provides a service for transferring money that exposes the following behavior. A debit account is debited if the balance is greater than the amount to be transferred.

```
validatePostcondition(
  event(?e, selector(?e, [#transferamount:to:]),
  precondition(?e,
    ?preState,
    and(argument(?e, [1], ?amount), greatherThan([?preState balance],
    ?amount))),
  postcondition(?e,
    ?preState,
    ?postState, and(argument(?e, [1], ?amount),
    equals([?postState balance],
    [?preState balance - ?amount])))
  )
```

Encapsulated States. When testing states there are various situations where we need to break up encapsulation.

- An internal state is not accessible through a components interface
- If we want to know how to navigate to an internal state starting from a root object

Let's imagine that an object is encapsulated but nevertheless we want to validate its state. For this purpose we specify an expression that is similar to an OCL navigation expression in order to navigate to a certain object in the recursive state of a root object. The navigation expression consists of a sequence of instance variable names that is used to stepwise access objects through a reflective method that takes the name of an instance variable as argument and returns the object at the specified slot value. In the Smalltalk VisualWorks system it is called `instVarAt:` which takes an index that corresponds to the position of the instance variable as it is defined by the objects class. With another method `instVarIndexFor:` that takes the name of an instance variable we query the index of the instance variable. We call the sequence of instance variables used to access a nested state an *accesspath* from a root object to a nested object.

```
nestedObjectAt(<>, ?s, ?s).
nestedObjectAt(<?firstInstVar |?restInstVar>, ?object,
?nestedObject) if
  objectAt(?object, ?firstInstVar, ?includedObject),
  nestedStateAt(?restInstVar,?includedObject, ?nestedObject)
```

For example in order to access an nested path through an access path #a, #b, #c we would write the following query

```
event(?e, selector(?e, [#xyz])),
  resurrectReceiverBeforeEvent(?e, ?receiver),
  nestedObjectAt(<[#a], [#b], [#c]>, ?rootObject, ?nestedObject)
```

If we know an access path we can query an object at the specified position, but sometime we would like to know whether an object is included in the recursive state and retrieve its access-path. For example if we call a method of a component that adds an object somewhere in the components internal state we want to know where the object has been added we can use this. The corresponding query would look as follows.

```

event(?e, selector(?e, [#addObject])),
argument(?e, [1], ?addedObject),
ressurectReceiverAfterEvent(?e, ?rootObject),
includesObject(?rootObject, ?addedObject, ?accessPath)

```

If we know how to access an object through an access path, we can test whether it is the same object as another object. For example we could test whether the object passed as an argument is the same object located at certain access path.

```

event(?e, selector(?e, [#addObject])),
argument(?e, [1], ?addedObject),
ressurectReceiverAfterEvent(?e, ?rootObject),
nestedObjectAt(<[#a], [#b], [#c]>, ?rootObject, ?nestedObject),
equals(?addedObject, ?nestedObject)

```

Links between Instances. An object has a link to another object whenever it is possible to access an object through navigation along a path of instance variables. Links between objects occur because objects are representing a single aspect of a domain and a whole model consists of many objects that are linked together. Some frequently used tests are whether a link between to objects exist or in an operation a link between objects is established or detached.

In order to test the existence of a link or a linkpath between a first and a second object accessible from a model root when can test whether a first object is included in the recursive state of the model root and the first object includes the second object in its recursive state.

```

existsLink(?fromObject, ?toObject, ?rootObject) if
  includesObject(?rootObject, ?fromObject),
  includesObject(?fromObject, ?toObject)

```

This rule gives us the basis to create two new rules that are use for testing whether a link is established or detached by a single operation. The semantics should be understandable when analysing the logic code.

```
establishesLink(?event, ?fromObject, ?toObject) if
  ressurectReceiverBeforeEvent(?event, ?r1),
  ressurectReceiverAfterEvent(?event, ?r2),
  not(existsLink(?fromObject, ?toObject, ?r1)),
  existsLink(?fromObject, ?toObject, ?r2)

detachesLink(?event, ?fromObject, ?toObject) if
  ressurectReceiverBeforeEvent(?event, ?r1),
  ressurectReceiverAfterEvent(?event, ?r2),
  existsLink(?fromObject, ?toObject, ?r1),
  not(existsLink(?fromObject, ?toObject, ?r2))
```

4.4 Recursion

Intent. Express queries to test the traversal of a composite object structure with a visitor. Identify and test single or composite events that occur during recursion.

Behavior. In contrast to procedural languages, recursion in object-oriented systems is concerned with traversing hierarchical object structures. Such structures typically occur in the presence of a composite pattern and a visitor that visits every element. In context of the visitor design pattern the visitor performs node type specific operations that are either independent of other nodes or depend on the result produced while visiting child nodes in the composite tree. A typical example for the second case is when an interpreter evaluates an expression and the result from evaluating subexpression is used to compute the expressions result.

Although different visitors and composite structures produce different behaviors we can abstract from them and write generic queries which tests whether the structure is properly traversed or not or for identifying event patterns during recursion that are representing a certain behavior.

Errors. Several errors can occur when recursing a composite structure

- Recursive structure is not properly traversed
- Wrong operation executed on the visitor
- Wrong results produced during recursion on composite structure, i.e. in the context of interpreting an expression.

A recursive structure is not properly traversed when nodes in the composite structure are not visited or the sequence of nodes visited does not conform to a specified traversal scheme. For a visitor the wrong operation in the visitor could be executed or the double dispatch with the object could fail if the visitor does not send the message back to the object. Furthermore results produced in intermediate steps of the recursion could be wrong and lead to an abnormal behavior.

Problems. When testing recursive behavior one is confronted first with the problem of setting up a composite structure, so that the recursion can be performed. This already requires programming effort. Then testing whether every node is traversed makes it necessary to enumerate every object in the recursive state of the composite root and check whether it is traversed.

Finally it is quite difficult to check intermediate result in the recursion because this requires a tester to step through the execution identify the event for which she wants to test a certain property, write the result down then probably check another event and so on. Applying a unit test pattern can test the final result of the computation but not in intermediate steps, so that it may occur that the result computed is accidentally correct, but the wrong computation has been performed.

Concept. We provide several query templates that serve for testing the recursive behavior in the context of a composite structure recursion. The task of testing this type of behavior can be decomposed in subtasks as follows

- Ensure complete tree traversal
- Ensure correct operation on nodes
- Specify behavioral assertions for operations that are performed on the composite structure

First we must make sure that every node that must be traversed by a visitor, is really traversed. Then we require that the correct node type dependent operations are performed by a visitor. Finally, in order to test the behavior that emerges from recursing a composite structure we identify a set of events and express a predicate over that event set.

Testing the traversal of composite structures is expressed as a query with two parts: First we define a query that tests whether every object of a component class in the recursive state of a root object is traversed in the recursion. We do this by identifying an event for the start of the recursion and then query every object in the recursive state of the receiver that belongs to a class in a composite hierarchy, i.e. in the ProgramNode hierarchy for abstract syntax trees.

In a second part of the query we test whether every node receives a recursive selector. A recursive selector is a selector that is recursively called for every object in the composite structure typical example are `eval:` or `acceptVisitor:`. For every event with a recursive selector we require that it contains the recursive events to the components of the receiver. If this was not the case the recursion would not be isomorphic to the composite structure so that recursive events could be missing or occur in the wrong order.

Check if the tree traversal is properly performed. The rule `objectsInRecursiveState` binds the variable `?objects` to every object found in the recursive state of another object belonging to a given set of classes.

```

recursesCompositeStructure(
  ?root,
  ?recursiveSelector,
  ?compositeClasses) if
objectsInRecursiveState( ?object,
  member([?object class], ?compositeClasses),
  ?root
),
event(?e, selectorAndReceiver(?e, ?recursiveSelector, ?object)),
ressurrectReceiverBeforeEvent(?e, ?receiver),
forall(objectsInState(?component,
  member([?object class], ?compositeClasses),
  ?receiver)
),
event(?e,
  contains(event(?ei,
    selectorAndReceiver(?ei, ?recursiveSelector, ?component)
  )
)
)
)
)

```

Now that we have enough confidence that the object structure is properly traversed we focus on testing operations that are performed during the traversal. We classify operations that are performed during traversal in such that are performed on a single node and others depend on the previous traversal of sub nodes such as it occurs when an interpreter traverses a parse tree or a bottom up pattern matching on a structure is performed.

A frequently used pattern to perform operations on object in a composite structure is the visitor pattern. As a visitor traverses objects a type dependent operation is called on the visitor by performing a double dispatch between the node and the visitor. In concrete implementations of the visitor pattern, the operation that is associated with a type can be identified by analysing method names that encode the type name.

By encoding the naming convention developers use as a rule we can predict which operation in the visitor should be executing for each type. A commonly used naming convention for methods, i.e. is `#forNameOfTypeX: anObjectOfTypeX`. Together with a query that checks if a double dispatch between an object and a visitor is done, we can check whether ever object in a composite structure performs a double dispatch with a visitor.

Evaluate if the event is a double dispatch between a visitor and the argument passed to its accept method.

```

visitorDoubleDispatchEvent(?e, ?selector) if
  nonvar(?e),
  receiver(?e, ?r),

```

```

argument(?e, [1], ?arg),
event(?e,
  implies(event(?ei,
    selectorReceiverArguments(?ei, ?selector, ?arg, <?r>))
  )
)

```

The event `?e` is representing the execution of the `#accept:` method of an object that is visited. The first argument to the `accept` is the visitor. The visitor is then the receiver of the next message that performs an operation dependent of the type of the visited object. This message is denoted by `?ei`. The first argument to this message is the object that was originally visited. The variable `?selector` holds the selector name for carrying out the type dependent operation.

When recursing object structures we many times observe dependencies between operations on a specific node and its sub nodes. For example when an interpreter is interpreting a message expression in Smalltalk all argument expressions are evaluated first and the results of the evaluation are passed as arguments to the message expression evaluation. Another example is a bottom up tree pattern matching process, so that a pattern node is only matched if all of the child nodes have been matched before.

To identify a set of events on a composite substructure we can write a pattern matching expression that matches events with receiver objects of the recursive selectors. A test is then expressed as predicate over the matched events.

For better understandability of this concept we show an example of an interpreter that evaluates a simple arithmetic expressions consisting of the product and sum operators and constants. The interpreter is implemented as a visitor that traverses the arithmetic expression and evaluates it. Below we show the code for the visitor.

```

EvalVisitor>>forConst: aConst
^aConst value

EvalVisitor>>forProduct: aProduct
^((aProduct left accept: self) *
  (aProduct right accept: self))

EvalVisitor>>forSum: aSum
^((aSum left accept: self) +
  (aSum right accept: self))

```

Lets imagine that we have the following arithmetic expression that is evaluated.

$(3+4) * 5 * 6$

Within the expression evaluation the term $3 + 4$ is evaluated. Instead of debugging the visitor traversal we would like to write an expression on the trace that checks whether the result of

evaluating $3 + 4$ is correct. To do that we write a pattern matching expression that matches the evaluation of a Sum term, query the returned values from evaluating the constant expressions and then check whether the result returned from evaluation the Sum equals the sum of the constant expressions.

```
event(?e, and(selectorAndClass(?e, [#accept:], [Sum]),
  ressurectReceiverBeforeEvent(?e, ?r)),
event(?e, selectorAndReceiver(?e, [#accept:], ?r),
  contains(<event(?e1,
    selectorAndReceiver(?e1, [#accept:], [?r left])),
    event(?e2, selectorAndReceiver(?e2, [#accept:], [?r right]))>)),
add([?e1 return], [?e2 return], ?sum),
equals(?sum, [?e return])
```

For this simple example, we can also test whether the visitor recurses the composite structure of the arithmetic expression.

```
arithmeticRecurseComposite if
  getExpression(?r),
  allSubclassesList([ArithmeticExpression], ?subclassList),
  recursesCompositeStructure(?r, [#accept:], ?subclassList)
```

The rule `getExpression` returns the root object `?r` of the composite structure. The variable `?subclassList` is unified with the classes `Sum`, `Prod` and `Const`. The last line of the rule the performs the check whether every object of the composite structure starting from `?r` is visited.

Here is the example whether for every node of the arithmetic expression a `doubleDispatch` with the visitor is performed.

```
arithmeticVisitorDoubleDispatch if
  getExpression(?r),
  allSubclassesList([ArithmeticExpression], ?subClasses),
  objectsInRecursiveState(?o, member([?o class], ?subClasses), ?r),
  getVisitorSelector([?o class name], ?vSelector),
  event(?e, and(selectorAndReceiver(?e, [#accept:], ?o),
    visitorDoubleDispatchEvent(?e, ?vSelector)))
```

First every object of the composite structure is queried and bound to the variable `?o`. Then for every event `?e` that is representing an acceptance of a visitor it is checked whether `?e` initiates a double dispatch event with the visitor. The query `getVisitorSelector` returns the selector of a visitor that performs a type dependent operation.

Testing a Smalltalk Meta Interpreter. We take the meta interpreter for Smalltalk [Til00] as more complex example to test recursive behavior. The meta interpreter for Smalltalk is an interpreter written in Smalltalk that interprets Smalltalk code. It is mainly used as coverage tool and fine grained dynamic analysis. The goal is to test whether a fragment of Smalltalk code is correctly interpreted. Given is the following method in Smalltalk

```
foo
  z:= x doSomething: self y.
```

When the meta interpreter is correctly implemented we expect that the result of evaluating the expression `self y` is passed as argument to the selector `doSomething`, the instance variable `x` is the receiver of the selector `doSomething` and after evaluation `z` holds the returned value from the method call.

The meta interpreter first parses the source code and creates an abstract syntax tree. Each node then recursively receives the message `eval:` with the context for evaluation as argument. In order to test the expression evaluation the meta interpreter is instrumented and the trace is generated by sending an object the message `foo`, so that the expression is evaluated with the meta interpreter. Now we can write some tests for correct expression evaluation.

Get references to the node objects in the parse tree

```
getParseTreeNode(?assignment, ?messageExpression, ?argument) if
  event(?methodEval, selector(?methodEval, [#valueWithReceiver:])),
  resurrectReceiverBeforeEvent(?methodEval, ?messageExpression),
  equals(?assignment, [?messageExpression statements at: 1]),
  equals(?messageExpression, [?assignment value]),
  equals(?argument, [?messageExpression arguments at: 1])
```

Test if the result of the evaluation of the argument expression `self y` is passed as argument to the evaluation of the expression `x doSomething:`

```
testArgumentEvaluation if
  getParseTreeNode(?assignment, ?messageExpression, ?argument),
  event(?evalSelfy, selectorAndReceiver(?evalSelfy, [#eval:], ?argument)),
  event (?perform,
    selectorAndReceiver(?perform, [#perform:receiver:arguments:class]),
    argument(?perform, [3], ?messageArguments),
    includes(?messageArguments, [?evalSelfy return])
```

Test if after method execution the instance variable ?z has the result returned from evaluating the expression x doSomething.

```
testAssignment if
  getParseTreeNode(?assignment, ?messageExpression, ?argument),
  event(?evalMessageExpression,
    selectorAndReceiver(?evalMessageExpression, [#eval:],
      ?messageExpression)),
  event(?foo, selector(?foo, [#foo])),
  resurrectReceiverAfterEvent(?foo, ?receiver),
  instVarValue(?receiver, ['z'], ?z),
  equals(?z, [?evalMessageExpression return])
```

Test if the receiver of x doSomething is the value of the instance variable x.

```
testReceiverEvaluation if
  getParseTreeNode(?assignment, ?messageExpression, ?argument),
  event(?foo, selector(?foo, [#foo])),
  resurrectReceiverBeforeEvent(?foo, ?receiver),
  event (?perform, selectorAndReceiver(?perform,
    [#perform:receiver:arguments:class]),
    argument(?perform, [2], ?performOnReceiver),
    instVarValue(?receiver, ['x'], ?x),
    equals(?x, ?performOnReceiver)
```

4.5 Case Study: Importing MOOSE Models

We take the import of source code models into the FAMIX model as an example for testing a model import with trace-based testing. FAMIX is an object-oriented model for representing source code entities and is part of the reengineering environment MOOSE. MOOSE provides an import facility that reads source code entities such as classes, methods and invocations from a file in CDIF format or from a Smalltalk image. We will concentrate on the import of models from the Smalltalk image that is implemented using the introspection capabilities of Smalltalk and a parse tree visitor to import the fine grained structure of methods such as accesses and invocations.

The concept we use for testing whether entities are imported in a model is first to identify a location in the trace where FAMIX model entities of a certain type are created in order to get a reference to an object representing this newly created entity. Later we test whether this FAMIX

model entity really exists in the model. For example we can identify where in the import process a new model entity for a class is created and then test whether this entity exists in the model.

The most important entity in the FAMIX meta model is the class **Class**. It is representing a class with a name in a programming language. In Smalltalk VisualWorks classes are organized in packages and bundles. Packages and bundles are concepts of the Store repository and allow a developer to hierarchically organize classes and compose applications from packages. The MOOSE importer for Smalltalk can import a FAMIX model from the set of classes that are located in a bundle. We would like to test whether a MOOSE importer properly imports a FAMIX model from a Smalltalk Store bundle. In order to do this, we first query the classes located in a bundle with a logic query, then query the trace generated during the import to observe the creation of a new entities and finally check if the set of classes is properly imported. In further test we can then check if other entities that are dependent of classes are properly imported.

Below are two queries that test whether every class in a bundle is imported in a MOOSE model. The first one performs a check for a single class. The second performs the test for every classes in a set using a `forall` query.

```
classEntityReifiedAndInModel(?c) if
  nonvar(?c),
  event(?e, and(selector(?e, [#ensureClassEntityFor:]),
    argument(?e, [1], ?c))),
  includesInRecursiveState([MSEModel currentModel], [?e return])
```

The variable `?c` is bound to a class that resides in the Smalltalk image. Creation of FAMIX model entities is performed by the method `#ensureClassEntityFor`. This method takes a Smalltalk class as an argument and returns a reified FAMIX entity. We locate the execution of the method `#ensureClassEntityFor` in the trace so that the argument matches the value of the variable `?c`. After that we test whether the imported MOOSE model contains the newly created entity in its recursive state. The expression `[#MSEModel currentModel]` refers to the last model that has been imported.

```
importerReifiesEveryClassInBundle(?bundleName) if
  forall(classInBundle(?bundleName, ?c),
    classEntityReifiedAndInModel(?c))
```

We now check whether the query `classEntityReifiedAndInModel(?)` succeeds for every class in a bundle. The query `classInBundle(?c)` unifies `?c` with every class that can be found in a

Smalltalk Store bundle with value of `?bundleName`. The query `forall` checks whether a predicate passed as a second argument is true for every solution passed by the first query. We can also note that this test is independent of which classes are in the bundle so that any bundle can serve as a source for testdata.

A rule that should be implemented by the MOOSE importer is, that if a class is imported its meta class is imported too. We can express a test for this behavior adding a new rule.

```
classAndMetaClassReifiedAndInModel(?c) if
  classEntityReifiedAndInModel(?c),
  classEntityReifiedAndInModel([?c class])
```

We simply compose this new rule by calling the query `#classEntityReifiedAndInModel(?x)` twice, once for the class and a second time for its meta-class. In order to test whether this rule is true for every class, we can again write a rule with a `forall` statement. We abstract the rule `#importerReifiesEveryClassInBundle` a bit, so that it can be used for arbitrary tests on a set of classes.

```
testEveryClassInBundle(?bundleName, ?test(?c)) if
  forall(classInBundle(?bundleName, ?c),
  ?test(?c))
```

The term `?test(?c)` is a higher order query that is passed as an argument to the rule above and can express any predicate dependent on the set of classes. Now we can perform the test for whether every class and its meta class are imported in the model by passing the query `classAndMetaClassReifiedAndInModel(?c)` as argument.

```
testEveryClassInBundle([ReferenceModel],
  classAndMetaClassReifiedAndInModel(?c))
```

As a last example we show how we can test whether a link between two imported entities exist, i.e. the link between a reified class and its reified instance variables. First we query every reified instance variable entity and class entity and then check whether there exists a link between them.

```

classAndInstanceVarEntity(?c, ?cEntity, ?ivEntity) if
  event(?e, and(selector(?e, [#ensureClassEntityFor:]),
    argument(?e, [1], ?c))),
  event(?e1, and(selector(?e1, [#ensureInstVarFor:]),
    argument(?e1, [1], ?c))),
  equals(?cEntity, [?e return]),
  equals(?ivEntity, [?e1 return])

```

In MOOSE instance variables are reified calling [#ensureInstVarFor:] where the first argument is the class and the returned object is the reified instance variable. We bind the reified class entity to the variable ?cEntity and the reified instance variable to the variable ?ivEntity for all execution of [#ensureInstVarFor:].

```

existsLinkBetweenClassAndInstanceVariables(?c) if
  nonvar(?c),
  classAndInstanceVarEntity(?c, ?cEntity, ?ivEntity),
  existsLink(?cEntity, ?ivEntity, [MSEModel currentModel])

```

We now check whether for every pair ?cEntity and ?ivEntity a link exists between those two objects within the MOOSE model. This must be true according to the FAMIX model. Finally we perform the test again for every class.

```

testEveryClassInBundle([ReferenceModel],
existsLinkBetweenClassAndInstanceVariables(?c))

```

4.6 Conclusion

Object-oriented behavior has two dimensions. One dimension of behavior is the change of object states, the other one is the sequence and containment of messages passed. Because testing is the act of making a precise statement about behavior, both dimensions need to be taken into account. Trace-based testing allows one to test any form of behavior by querying the execution trace with a logic query. Because it is based on Prolog, every recursive set of events can be computed, so that

every form of program behavior that is expressible in the event model can be tested. However the huge number of messages that can be found in the trace makes it necessary to rely on tree matching algorithms that have a time complexity that is not NP-complete.

Trace-based testing is suitable for the engineer to express a precise statement about program behavior in a declarative and compact form of a logic query. This offers a big advantage over manually controlling and tracing an execution. Behavioral tests that are often needed can be represented as logic rules and be reused for a particular test by passing parameters. This was shown for the visitor pattern or for testing pre- and postcondition. In this context, it is a big advantage that SOUL supports passing higher order queries as parameters as one can do it with higher order function in functional languages.

Objects have state, are senders and receivers, are passed around as arguments and returned to the sending context after the execution of a method. This makes it often necessary to make a statement about the equality of two objects, i.e. when one wants to express that an object passed as argument is bound to an instance variable of another object. Because SOUL allows one to unify variables with objects and objects can be resurrected from the trace it is possible to express arbitrary equality between object, also in the case when events are temporally distant.

Because trace-based testing is carried out as postmortem trace-analysis the whole evolution of a certain program state can be observed without having to rerun a program several times. In order to develop a test for a certain behavior a tester can first query the trace in order to acquire knowledge about the actual program behavior. This can occur in parallel to an analysis of the source code during in context of a reverse engineering activity.

Trace-based testing is suitable to create ad-hoc tests of behavior. Because it is not necessary to decompose a software in to units and code unit tests and conjecture expected values from analysing the source code, creation of behavioral test is frictionless and therefore faster. This has the benefit that during a maintenance cycle ad hoc tests can be written that focus on a specific aspect of system behavior.

The main drawback of trace-based testing is the fact that it is necessary to execute the software. Because software that is built from scratch has a long way to go until it is integrated and executable, the focus for trace-based testing are systems that are at least minimally operable. Because recording a trace delivers just a slice of the possible program behavior, it is best used together with a coverage analyzer to iteratively refine coverage to an adequate level.

Chapter 5

Implementation

5.1 TESTLOG: A Prototype Tool

TESTLOG is a prototype tool that serves as a testbed to study and iteratively develop the concept of trace-based object-oriented testing. Its computational model is that of a logic query. A logic query is composed of logic terms and unifies logic variables with events from the trace. The pass or fail semantics of a test can therefore easily expressed in terms of the semantics of a logic query: If a logic query fails then a test fails and if a logic query produces a least one result then the test succeeds.

The underlying medium to express terms and logic rules is SOUL [Smalltalk Open Unification Language]. SOUL is a full prolog written in Smalltalk with an extension mechanism that allows a programmer to integrate Smalltalk expressions into logic rules and use logic variables in Smalltalk expressions. TESTLOG is implemented on top of SOUL as a layered architecture.

SOUL	Domain Specific Queries	
	Behavioral Archetypes	
	Tree Pattern Matching	
	Basic Event Queries	
	Event Reification	
SM	Reified Events and States	

Figure 5.1: Architecture of TESTLOG

The bottom layer comprises an object-oriented model that represents the event trace. The trace

is stored in the Smalltalk image and accessible via a singleton pattern. At the next abstraction level TESTLOG provides queries to access single events and object states. This layer serves two purposes: First it serves as a reification layer of an object-oriented model into the logic environment of SOUL by binding objects from the model to logic variables, second it provides basic queries on the event trace for querying events according to their attributes. Also part of this layer are queries that deal with accessing properties of state such as whether an object is included in the recursive state of the events receiver object.

The pattern matching layer supports the execution of a pattern matching query on the event tree. As tree pattern matching we understand the process of checking the occurrence of a substructure, the pattern tree, in a larger structure, the target tree. The primary usage of tree pattern matching is to test for collaborations.

5.1.1 SOUL Syntax and Symbiosis with Smalltalk

We give a short introduction into SOUL because it is necessary to understand its concepts for the following sections where different types of queries are introduced. SOUL is a full prolog with several syntactical and semantical enhancements that allows a tight integration with an object-oriented language.

As in prolog, code is written as rules and computations are performed by writing queries. An extension over standard prolog is that logic variables can be unified with Smalltalk objects. This means that there exists a binding of a logic variable to a Smalltalk object as it is represented in the virtual machine. The following example of a query binds the root of the Smalltalk class hierarchy to the variable ?c. The term in angle brackets is evaluated in Smalltalk and the result is passed as argument to the query.

```
classWithName(?c, ['Object'])
```

By looking at the implementation of the rule `classWithName` we can see the syntax of rules where the head of the rule is separated from the body with word 'if'. On the third line of the body we again see a term in square brackets. It contains Smalltalk code that is evaluated by the Smalltalk VM. In the Smalltalk code the logic variable ?Class is used. This mechanism allows a programmer to pass variables from a logic environment into the Smalltalk environment. The concept that Smalltalk and SOUL can work together is called *symbiosis*.

```
classWithName(?Class,?ClassName) if
  not(and(var(?Class),string(?ClassName))),
  class(?Class),
  equals(?ClassName,[ Soul.MLI current classNameOf: ?Class ])
```

5.2 Representation of the Event Trace

Before we can express a precise statement about behavior we must find a form for representing the recorded trace. The key concept we use for representing dynamic information is the event. An event is a time interval starting with execution of a method and ending with passing control back to the calling method. On the basis of a recorded execution trace we reify an event model that allows us to express ordering and containment relations between events. With each event the following attributes are provided

- The sender object of a message
- The receiver object of a message
- The received selector
- A list of arguments that are passed
- A snapshot of the complete recursive state of the receiver before and after a method execution so that we are capable of reasoning about state changes

On the set of events two basic relations are established, which may hold between two arbitrary events. An event may precede another event and an event may be included in another event. Event precedence is established by a temporal ordering of event and event containment is defined by the nesting of message sends.

If an event e precedes another event e_1 we write $e < e_1$ and if an event e_1 is included within another event e it is expressed as $e_1 \text{ in } e$.

A list of general axioms is satisfied by any events a, b, c in the set of events.

Mutual exclusion of relations:

$$a < b \rightarrow \text{not}(a \text{ in } b)$$

Non commutativity:

$$a < b \rightarrow \text{not}(b < a)$$

$$a \text{ in } b \rightarrow \text{not}(b \text{ in } a)$$

Transitivity:

$$(a < b) \text{ and } (b < c) \rightarrow (a < c)$$

Distributivity:

$$(a \text{ in } b) \text{ and } (b < c) \rightarrow (a < c)$$

$$(a < b) \text{ and } (c \text{ in } b) \rightarrow (a < c)$$

=1

5.2.1 Reification of the Event Model

The event model is created from a previously recorded execution trace. For each recorded message in the execution trace an event object in the event model is created. Not only messages to ordinary objects but also messages to classes are considered, so that an instance creation is recorded. Because of the call semantics of method execution the set of events is ordered. An event includes other events whenever a message send is nested within another message send, so that the event tree is isomorphic to the call tree. Events are represented as an object-oriented model as follows.

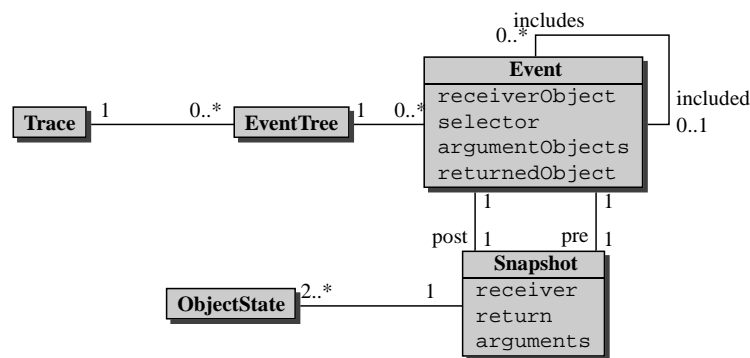


Figure 5.2: Object-oriented event model

The class `Trace` is the root of the model and serves as single access point for reification in the logic layer. The trace can contain many event trees, because the trace may not be recorded within a single calling context but within different ones such there can be many call trees. As experienced by trying out the recorder in realistic instrumentation scenario the number of event trees is one order of magnitude smaller than the set of messages recorded.

Each event has several attributes. The attributes `receiverObject` and `argumentObjects` are representing references to the corresponding objects when the message that is represented by a certain event has been executed. Every event references the events it includes at the next level of the call tree. In order to make a statement about object states that occurred during an execution a snapshot of the receivers recursive state before and after the event is taken. A snapshot of an object that is representing a reified object state of an object is represented as an object of the class `ObjectState`. Object states and object identity are completely separated in the model. An object identity will remain the same during the whole lifetime of an object, however an object state may change.

In the following we describe more formally the properties of event trees in order to show how the event inclusion relationship is created.

We first define some sets

- Let E be the set of events
- The set $sender(E)$ is the set of all senders
- The set $receiver(E)$ the set of all receivers

Let T be an arbitrary event tree and $root(T)$ its root. The following properties then hold on the event tree.

- $e = root(T)$ if $sender(e)$ not in $receiver(E)$. This states that the root may not be included by another event. The term $sender(e)$ denotes the sender object of e . This means the first message send that is recorded is the one from $sender(e)$ to e .
- T is a tree with $T = (V, F, root(T))$ where $V \subset E$ and $root(T) \in E$. F is binary relation (u, v) so that: $(u, v) \in F$ if $sender(u) = receiver(v)$
- $(root(T), v) \in F^*$ where F^* is transitive closure of F . This means that all nodes in T are reachable via the root of T .

For establishing an ordering of the trees we define now a forest $F = \langle T_1, \dots, T_n \rangle$ as an ordered sequence of trees with nodes from E so that two tree roots $root(T_i) < root(T_j)$ for $i < j$.

5.2.2 Reification of Object States

For each event we reify the complete recursive state of the receiver before and after a method execution as a new meta object. In this reified state we preserve the structure of the recursive state and the identity of the object that are included in it. This is necessary in case we would like to make a statement about the recursive state of an object that needs to take into account object identities. If we want to test whether an object is included in the recursive state of another object at a certain event in the trace, we need to compare to object identities. However the recursive state of an object can change during an execution, therefore we need to preserve it.

One strategy to preserve an objects state would be to create a deep copy of an object with the same structure but newly create objects in it. However with this strategy we lose the ability to make a statement about objects for different events and states. For example we could no longer express that the same object that is passed as an argument is added to the recursive state of another object by comparing the two object references. Therefore another strategy is chosen: A graph that is isomorphic to the recursive state of the receiver is created and at each node of this graph a reference to the original object is maintained.

A formal definition for the state reification strategy is given here. First we define the state and the recursive state of an object. The state of an object o is a set of object defined as follows

$$state(o) = (\cup instVar(o, i)) \cup (\cup variableValue(o, i))$$

In the above formula the terms $instVar(o, i)$ denotes the value of the i th instance variable of object o and $variableValue(o, i)$ denotes the i th slot value in case o is a variable object. We need to consider variable sized objects because that's how the Smalltalk systems provides named slots and indexed slots in the case of variable subclasses.

And the *recursive state* is recursively defined as

$$recstate(o) = (\cup recstate(o_i)) \cup o \forall o_i \in state(o)$$

Then we define as the reified recursive state a Graph $G = (V, E)$ that is isomorphic to the recursive state, so that each node v in G references the corresponding object in the original recursive state, denoted by $object(v)$. For every tuple (o, p) and $p \in recstate(o)$ exist $(v1, v2)$ in E so that $object(v1) = o$ and $object(v2) = p$.

We define now two operations that deal with recursive state. The first operation we described above we call *reification* of a recursive state. The inversion of reification restores the recursive state of the receiver exactly as it was a certain point of the execution, so that a message can be sent to it. We call this operation *resurrection* of an object state from its reified state, because we bring back to life the original object. Figure 5.3 shows the class model for reified object states.

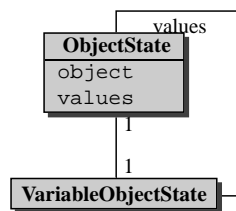


Figure 5.3: Reified State Model

5.3 Queries over the Event Trace

In this section we will introduce different types of logic queries. They serve as components for constructing complex rules and queries about behavior. There are three classes of queries. Some deal with identifying sets of event, others are about pattern matching the trace and still another group is concerned with expressing properties about state. Furthermore queries of different classes can be combined to form complex expressions. We will explain their semantics in the context of a simple bank example.

5.3.1 Bank Example

The sole purpose of this example application is to have a basis for expressing example queries in order to illustrate their usage and their semantics. The bank example simulates a simple bank business where customers can open accounts and transfer money. Below is a class model an a short specification about its behavior is given.

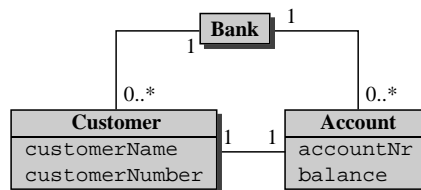


Figure 5.4: Bank Model

A hypothetical application that implements the bank business supports two operations one for adding a customer and another for transferring money. Below are the implementations and the message sequences for these two operations.

```

newCustomerWithName: aString
|customer account|
customer := Customer withName: aString
self addCustomer: customer.
account := Account new.
self addAccount: account.
customer addAccount: account.

transferAmount: amount from: debitAccount to: creditAccount
(debitAccount getBalance > amount) ifTrue: [
debitAccount setBalance: debitAccount getBalance - amount.
creditAccount setBalance: creditAccount getBalance + amount
]
  
```

Imagine now a scenario where first two new customers are created and money is transferred from one account to another. This scenario would produce the following trace. It will serve us as a basis for our queries. Every line of the trace description represents an event. The first term in a line is an object then follows the selector and the argument objects.

```

1 aBank newCustomerWithName: 'Bill'
2 Customer withName: 'Bill'.
  
```

```
3 Customer new.
4 aCust1 name: 'Bill'.
5 aBank addCustomer: aCust1.
6 Account new.
7 aBank addAccount: account1.
8 aCust1 addAccount: account1.
9 aBank newCustomerWithName: 'George'
10 Customer withName: 'George'.
11 Customer new.
12 aCust2 name: 'George'.
13 aBank addCustomer: aCust2.
14 Account new.
15 aBank addAccount: account2.
16 aCust2 addAccount: account2.
17 aBank transferAmount: 100 from: account1 to: account2
18 account1 getBalance
19 account1 getBalance
20 account1 setBalance
21 account2 getBalance
22 account2 setBalance
```

The first two fragments of the trace show how new customers with name Bill and George are added to the bank. The third fragment then shows the trace produced by executing the method `#transferAmount:from:to` with the two newly created accounts.

5.4 Basic Logic Queries over the Event Trace

In this section we give an overview of the basic rules, how they are applied in the context of the bank example and how they can be composed in order to express complex queries. The basic rules are classified as follows:

- Event querying deals with accessing single events and their attributes
- Event pattern matching shows how to pattern match the event hierarchy
- State related queries show how to access properties of state at an arbitrary point of the execution
- Combination of queries describes which of the queries can be combined or passed as higher order queries to form complex expression

The documentation of a query starts with an informal description of its purpose. Then the implementation of the corresponding rule is given, followed by an informal description of its semantics. Finally a simple example in the context of the bank example shows the usage.

5.4.1 Event Querying

This section introduces queries for querying single events or querying sets of events by pattern matching their attributes.

All Events

Intent

Query every event in the trace.

Implementation

```
event(?ev) if
  member(?ev, [DYNTTestLog current trace messages]).
```

Semantics

Unifies a variable with every event in the trace. The most recently generated trace is accessed through the singleton `DYNTTestLog`. The embedded Smalltalk expression returns an ordered collection of event objects. The query `member` unifies a variable with every object in the collection.

Examples

To query every event in the trace we would write the following query. It unifies the variable `?e` with every event object from the trace.

```
event(?e)
```

Events that satisfy a predicate

Intent

Query the trace for events that satisfy a predicate. Pass this predicate a higher order query

Implementation

```
event(?ev, ?q) if
  event(?ev),
  ?q.
```

Semantics

For every event the higher order predicate ?q that is passed as argument is evaluated. Every event object that satisfies this predicated is then a solution to the query.

Examples

```
event(?e, selector(?e, [ #getBalance]))
```

Query every event with the selector #getBalance.

```
event(?e, [?e class == Account])
```

Query every event which is representing a message to the class Account. In contrast to the first example the query that is passed as argument is represented as a Smalltalk term and not as a SOUL query. Applied to the example the variable ?e would unify with the two events representing new messages to class Account.

Event attribute matching

Intent

For each attribute provide a rule to compare an events attribute with an expected value.

Implementation

```
selector(?e, ?s)
receiver(?e, ?r)
sender(?e, ?s)
arguments(?e, ?expectedArgumentsList)
```

Semantics

Every rule takes an event as argument. This argument may not be a variable term, because otherwise no Smalltalk message must be sent to it. The predicate `nonvar(?e)` tests therefore whether the term `?e` is not a variable. The comparison between a current and an expected event attribute is performed by writing a Smalltalk expression that queries an attribute, and using it together with an expected value as argument to the predicate `equals`. Rules for event attribute matching are typically used for querying events that match a certain predicate.

Examples

```
event(?e, receiver(?e, [Account]))
```

Every event with the class `Account` as receiver.

```
event(?e, and(selector(?e, [ #withName:]), arguments(?e, <'George'>)))
```

The event with selector `#withName:` and arguments `'George'`

Event containment

Intent

Test whether one event contains some other event in the event tree.

Implementation

```

event(?e, contains(event(?ec, ?q))) if
  nonvar(?e),
  member(?ec, [?e allRecursivelyContainedEvents]),
  ?q.

```

Semantics

The event `?e` includes another event `?ec` that satisfies a predicate `?q`

Examples

```

event(?e, selector(?e, [#newCustomerWithName:]),
event(?e, contains(?e1, selector(?e1, [#addAccount])))

```

For every event with selector `#newCustomerWithName` find events with the selector `#addAccount` in the example trace.

Event Tree Pattern Matching

Pattern matching queries deal with pattern matching events and take into account event inclusion. By pattern matching we understand the process of locating a substructure, the pattern, in a larger structure, the target. Because the hierarchy of events is represented as a tree, a form of tree pattern matching is used to locate event patterns.

The requirement for having a tree pattern matcher emerges from the fact that in object-oriented systems the message structure is deeply nested because of complex collaborations between objects. However, in order to test if an expected collaboration pattern occurs, there is a need for having a formalism that allows a specification of an expected pattern and an algorithm to perform a tree pattern matching.

Because the general tree pattern matching problem with variables is NP-complete and would not be usable for pattern matching an execution trace consisting of several thousand messages the *left order embedding algorithm* described in [Kil92] is used. The leftorder embedding algorithm has a time complexity of $O(mn)$ where m is the number of pattern nodes and n is the number of tree nodes. Informally the leftorder embedding algorithm finds the first instance of the pattern if the tree is traversed in postorder.

Intent

Specify a formalism to express a tree pattern on the event trace and match the event trace using the leftorder tree pattern matching algorithm.

Implementation

```

event(?ev, ?q, contains(?evlist)) if
  createPNodeTree(event(?ev, ?q, contains(?evlist)), ?pNode),
  patternMatchTree(?pNode).

```

The variable `?ev` is representing the root of the pattern tree and is unified with an event object if a match occurs. The term `?q` is a higher order query that is called to match event attributes. Finally the variable `?evlist` denotes the list of child pattern nodes. The rule `patternMatchTree(?pNode)` then executes the pattern matching algorithm.

Semantics

The term `event(?ev, ?q, contains(?evlist))` declares a pattern tree. The pattern tree is matched with the event tree using the leftorder embedding algorithm. If a match occurs the variables in the pattern term that are representing events are unified with event objects from the trace. The comparison of nodes is performed by calling a query that is specified for each pattern node.

Examples

Test whether a customer with the name 'Bill' is created through the bank interface.

```

event(?e, selector(?e, #newCustomerWithName),
  contains(<event(?e1, selector(?e1, #withName:),
  contains(<?e2, selectorAndArguments(?e2, #name, <['Bill']>>
  )>>
  )>>

```

State Related Queries

Beside message passing, state changes in the states of objects are other behavioral artifacts that are important in testing. Some examples of state changes are the establishments and detachments of links between object instances, the addition of an object that is passed as argument in the recursive state of the receiver, incrementally building a composite structure by creating new objects and adding them in a structure. Furthermore state changes are the behavioral artifacts that are most frequently targeted by unit testers, so that there is enough justification to provide some queries about state.

Intent

Recreate the original recursive state of the receiver object at an arbitrary point in the trace.

Implementation

```

ressurrectReceiverBeforeEvent(?e, ?receiver) if
  stateBeforeEvent(?e, ?s),
  dereifyState(?s, ?receiver)
ressurrectReceiverAfterEvent(?e, ?receiver) if
  stateBeforeEvent(?e, ?s),
  dereifyState(?s, ?receiver)

```

Semantics

The term `?e` is an event object from the trace. For this event the receiver object with its recursive state is reconstructed with the same objects as it was during the execution and bound to the variable `?receiver`. There are two rules, one reconstructs the receiver as it was before the event, the second one as it was after the event. This rule can be used to simulate a unit test by identifying an event that is representing a stimulus to a unit, reconstructing the state of the receiver before and after the event and specifying a predicate that test a state change.

Examples

Test whether a customer with a certain name is added to a bank.

```

event(?e, selector(?e, #newCustomerWithName:)),
argument(?e, [1], ?customerName),
ressurrectReceiverBeforeEvent(?e, ?receiverBefore)
ressurrectReceiverAfterEvent(?e, ?receiverAfter)
equals([false], [?receiverBefore hasCustomerWithName: ?customerName])
equals([true], [?receiverAfter hasCustomerWithName: ?customerName])

```

Object Inclusion

Intent

Test if an object is included in the recursive state of a root object.

Implementation

```

includesObject(?rootObject, ?includedObject) if
  [DYNDDeepCopy new exists: ?includedObject inRecursiveStateOf: ?rootObject]

```

Semantics

The query succeeds whenever the object bound to the variable `?includedObject` is found in the recursive state of the object bound to the variable `?rootObject`.

Examples

Test if a customer create with a certain name is included in the bank object afterwards.

```
event(?e, selectorAndArguments(?e, [#newCustomerWithName], <['Bill']>),
  contains(<event(?e1, selectorAndClass(?e1, [#new], [Customer])>))
  return(?e1, ?newCustomer),
  ressurectReceiverBeforeEvent(?e, ?bank),
  includesObject(?bank, ?newCustomer)
```

Links between Objects**Intent**

Test if links between objects in the recursive state of a root object exist, are established or detached by a single operation.

Implementation

```
existsLink(?fromObject, ?toObject, ?rootObject) if
  includesObject(?rootObject, ?fromObject),
  includesObject(?fromObject, ?toObject)

detachesLink(?event, ?fromObject, ?toObject) if
  ressurectReceiverBeforeEvent(?event, ?r1),
  ressurectReceiverAfterEvent(?event, ?r2),
  existsLink(?fromObject, ?toObject, ?r1),
  not(existsLink(?fromObject, ?toObject, ?r2))

establishesLink(?event, ?fromObject, ?toObject) if
  ressurectReceiverBeforeEvent(?event, ?r1),
  ressurectReceiverAfterEvent(?event, ?r2),
  not(existsLink(?fromObject, ?toObject, ?r1)),
  existsLink(?fromObject, ?toObject, ?r2)
```

Semantics

The rule `existsLink(?fromObject, ?toObject, ?rootObject)` checks whether there is a reference

path from the object bound to `?fromObject` to the object bound to `?toObject` somewhere in the recursive state of the object bound to `?rootObject`.

The query `detachesLink(?event, ?fromObject, ?toObject)` finds a solution whenever there exists a link between `?fromObject` and `?toObject` in the receiver before the event and this link is no longer existent after the event.

The query `establishesLink(?event, ?fromObject, ?toObject)` finds a solution whenever there is no link between `?fromObject` and `?toObject` in the recursive state of the receiver before the and there exists a link after the event.

Examples

Test if a link between a new customer and a new account is established.

```
event(?createCustomer, selectorAndArguments([#newCustomerWithName:],
<['Bill']>),
  contains(<event(?custNew, selectorAndClass(?custNew, [#new]),
  event(?accNew, selectorAndClass(?accNew, [#new], [Account])>)),
  receiver(?createCustomer, ?bank),
  return(?custNew, ?newCustomer),
  return(?accNew, ?newAccount),
  establishesLink(?createCustomer, ?newCustomer, ?newAccount)
```

Chapter 6

Conclusion

6.1 Summary

In this thesis a new concept called *trace-based object-oriented testing* was developed. It is based on creation of a trace by executing a program and test a behavioral property by specifying an expression over the trace. Trace-based object-oriented testing supports test for any kind of program behavior that can be found in object-oriented programs. The behavior of single operations can be tested by identifying them in a trace and validating postconditions. More complex behavior can be tested by specifying a predicate over a set of events.

Trace-based object-oriented testing was applied to different types of behaviors that frequently occur in object-oriented systems such as scenarios and the visitor pattern. A small case study of the MOOSE importer component showed the usability for a complex example.

A prototype tool TESTLOG has been developed to serve as an infrastructure to support trace-based testing. It is based on the logic meta language SOUL that extends the prolog language with facilities to unify logic variables with objects and embed Smalltalk expressions in the logic code. The general purpose rules can be classified in such that are used for event querying, and such that deal with accessing object states and such that are use to specify a pattern matching expression over an event tree.

6.2 Further Work

6.2.1 Test Libraries

The current implementation supports a base infrastructure for trace-based object-oriented testing. However testing is often concerned with complex domain specific behavior or similar behavior that frequently can be found in different programs, such as behavior of design patterns. A tester could therefore profit from a repository with a set of rules that can be parametrized and composed

according to his needs.

Appendix A

Logic Code For Tree Pattern Matcher

This appendix shows the logic code of the left embedding tree pattern matching algorithm [Kil92] implemented in SOUL.

A.1 Event Tree Pattern Matcher

```
event(?ev,?q,contains(?evlist)) if
  createPNodeTree(event(?ev,?q,contains(?evlist)),?pNode),
  patternMatchTree(?pNode)

patternMatchTree(?root) if
  getPatternMatcher(?patternMatcher),
  enumPNodes(?root,[0],?numPNodes),
  [?patternMatcher setup: ?numPNodes. true],
  matchTree(?root),
  hasResult,
  evalResultForPTree(?root)

matchTree(?pNode) if
  children(?pNode,?children),
  matchChildNodes(?children),
  collectPostOrderNumberList(?children,?numberList),
  do(matchTreeNodesForPNode(?pNode,?numberList)),
  getPatternMatcher(?patternMatcher),
  getPostOrderNumber(?pNode,?V),
  [?patternMatcher hasMatchForNode:?V]

matchTreeNodesForPNode(pNode(query(?currentNode,?matchPredicate),
  pNum(?V,?x),
  children(?children)),
  ?cPostorderNumbers) if
  getPatternMatcher(?patternMatcher),
  equals(?N,[?patternMatcher numberOfNodes]),
  member(?treeNode,[?patternMatcher forestNodeList]),
```

```
equals(?currentNode,[?treeNode message]),
equals(?W,[?treeNode postOrderNumber]),
?matchPredicate,
[|i p k q |
 p:= (?patternMatcher minDesc: ?treeNode N: ?N).
 i:= 0.
 k:= (?cPostorderNumbers size).
 [(i < k) and: [p < ?W]] whileTrue:[
  p:= ?patternMatcher result row: (?cPostorderNumbers at: (i+1)) col:p.
  (?patternMatcher is: p descendentOf: ?treeNode) ifTrue: [i:= i+1].
 ].
 (i= k) ifTrue:[
  q:= ?patternMatcher result matchValueAt: ?V.
  [?patternMatcher is: q leftRelativeOf: ?treeNode] whileTrue:[
   ?patternMatcher result row:?V col:q value: ?W.
   q:= q+1.].
  ?patternMatcher result matchValueAt: ?V put: q.
 ]. true
 ],
 [false]
```

Bibliography

- [ABW98] Sherman R. Alpert, Kyle Brown, and Bobby Woolf. *The Design Patterns Smalltalk Companion*. Addison Wesley, 1998.
- [Aug95] M. Auguston. Program behavior model based on event grammar and its application for debugging automation. In *2nd International Workshop on Automated and Algorithmic Debugging, Saint-Malo, France, May 1995*.
- [Aug98] M. Auguston. Building program behavior models. In *European Conference on Artificial Intelligence ECAI-98, Workshop on Spatial and Temporal Reasoning, Brighton, England, August 1998*.
- [Bar97] S. Barbey. *Test selection for specification-based unit testing*. PhD thesis, Swiss Federal Institute of Technology, 1997.
- [Bin99] Robert V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Object Technology Series. Addison Wesley, 1999.
- [BM00] A. Bertolino and M. Marre. Automatic generation of path covers based on the control flow analysis of computer programs. *IEEE Trans. Soft. Eng.*, December 2000.
- [Bro75] Frederick P. Brooks. *The Mythical Man-Month*. Addison Wesley, Reading, Mass., 1975.
- [DLT00] Stéphane Ducasse, Michele Lanza, and Sander Tichelaar. Moose: an extensible language-independent environment for reengineering object-oriented systems. In *Proceedings of the Second International Symposium on Constructing Software Engineering Tools (CoSET 2000)*, June 2000.
- [Duc99] M. Ducasse. Opium: An extendable trace analyser for prolog. *The Journal of Logic programming*, 1999.
- [Gué03] Yann-Gaël Guéhéneuc. *Un cadre pour la traçabilité des motifs de conception*. PhD thesis, École des Mines de Nantes, juin 2003.
- [Har00] Mary Jean Harrold. Testing: a roadmap. In *ICSE - Future of SE Track*, pages 61–72, 2000.

- [JE94] Paul C. Jorgenson and Carl Erickson. Object-oriented integration testing. *CACM*, 37(9):30–38, September 1994.
- [KGH⁺95] David Kung, Jerry Gao, Pei Hsia, Yasufumi Toyoshima, Chris Chen, Young-Si Kim, and Young-Kee Song. Developing and object-oriented software testing and maintenance environment. *Communications of the ACM*, 38(10):75–86, October 1995.
- [Kil92] P. Kilpeläinen. *Tree Matching Problems with Applications to Structured Text Databases*. PhD thesis, University of Helsinki, Department of Computer Science, November 1992.
- [MP00] D. J. Murray and D. E. Parson. Automated debugging in Java using OCL and JDI. In Mireille Ducassé, editor, *4th Workshop on Automated Debugging*, August 2000.
- [OMG99] Object Management Group. Unified Modeling Language (version 1.3). Technical report, Object Management Group, June 1999.
- [RHD00] Gregg Rothermel, Mary Jean Harrold, and Jeinay Dedhia. Regression test selection for C++ software. *Software Testing, Verification and Reliability*, 10(2):77–109, 2000.
- [Sil00] M. Silverstein. Automated testing of object oriented-components using intelligent test artifacts. In *Proceedings of the Thirteenth International Software and Internet Quality Week*, June 2000.
- [Til00] Michel Tilman. Building run-time analysis tools by means of pluggable interpreters. *ESUG 2000 Summer School*, 2000.
- [Wuy01] Roel Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, 2001.