Logic and Trace-based Object-Oriented Application Testing

Stéphane Ducasse Michael Freidig [*] Roel Wuyts[†]

Abstract

Due to the size and the extreme complexity of legacy systems, it is nearly impossible to write from scratch tests before refactoring them. In addition object-oriented legacy *systems present specific requirements to test them. Indeed late-binding allow subclasses to change fundamental aspects of the superclass code and in particular call flows. Moreover Object-oriented programming promotes a distribution of the responsibilities to multiple entities leading to complex scenario to be tested.*

In such a context one of the few trustable source of information is the execution of the application itself. Traditional forward engineering approaches such as unit testing do not really provide adequate solution to this problem. Therefore there is a need for a more expressive way of testing the execution of object-oriented applications. We propose to represent the trace of object-oriented applications as logic facts and express tests over the trace. This way complex sequences of message exchanges, sequence matching, or expression of negative information are expressed in compact form. We validated our approach by implementing TestLog a prototype tool and testing the Moose reengineering environment and a meta-interpreter.

Keywords: Legacy System, Testing, Dynamic Information, Logic Programming

During reengineering, it is crucial to write tests to make sure that the changes made do not affect the overall behavior of the system. Since reengineering is typically applied to large, complex and badly understood systems, it is extremely hard to write such tests from scratch. This is especially true since reengineers most of the time do not know the system they are working on, and face a lot of time pressure [6]. The situation is even worse in the case of object-oriented *legacy* systems. Indeed late-binding allow subclasses to change fundamental aspects of the superclass code, in particular control flow. Moreover object-oriented programming promotes a distribution of the responsibilities to multiple entities leading to complex scenario to be tested.

In the specific context of reengineering legacy object-oriented applications, one of the few trustable sources of information is the execution of the application itself. The approach presented in this article is based on the logic representation of program execution and the specification of tests as logic queries. This work is related to the use of logic queries for advanced debugging [15].

This article is structured as follows. Section 1 stresses the specific problems encountered when testing legacy object-oriented applications. Section 2 presents our approach. Sections 3, 4 and **??** presents frequently occuring patterns to test in object-oriented applications. Sections 6 and 7 present the validation of the approach on two bigger case studies. Section 8 presents the implementation details of the approach.

1. Testing Object-Oriented Programs

Object-oriented programming promotes a distribution of responsibilities among multiple interacting objects. In addition to this spatial repartition a temporal repartition occurs because of the presence of late-binding. The fact that the message receiver is late-bound and always represents the receiver of the current method is extremely powerful and is the basis for building frameworks. As such it introduces another dimensions as it opens the door to the introduction of new behavior or modifications of behavior defined in superclasses [22]. A subclass can easily customize locally fundamental aspects of its superclasses by changing the sequence of messages and calling relationships between the methods, leading to the fragile class problem [19].

Unit testing, incarnated recently by the flurry of *Unit frameworks, supports object-oriented testing very well [1]. A context is built, within this context methods are invoked and their result or changes on the tested objects are checked. Even if unit testing is powerful and has proven to be valuable, there are some situations that are difficult to express with such an approach.

Let us consider that we want to test the application of an observer pattern. In such a case one must verify the collaborations between a subject and its *registered* observers. Only registered objects should receive an update message, when a subject receives a change message, it responses with an

[*] Software Composition Group, University of Bern, Switzerland
[†] Decomp Laboratory, Université Libre de Bruxelles, Belgium

updated message to every registered observers that in turn takes an appropriate action. A test of this collaboration requires an analysis of messages exchanged between objects that is not easily feasible with a unit test.

Requirements for Testing Run-Time Information. Because it is painful to manually analyze a behavior that spawns over many steps of an execution, there is a need to create a language that enables a tester to formally specify a test and automatically evaluate whether it passes or fails. Moreover, the solution should support the expressions of query on objects that are *not in the scope of each other* [14]. Here is a non-exhaustive list of actions that a tester would like to be able to use to express tests:

Because it is painful to manually analyze a behavior that spawns over many steps of an execution, there is a need to create a language that enables a tester to formally specify a test and automatically evaluate whether it passes or fails. Here is a non-exhaustive list of actions that a tester would like to be able to use to express tests:

- Identification of a message based on its name, its sender, its receiver or its arguments,
- identification of object creation,
- identification of specific message sequences within complex interactions,
- identification of messages inclusion, i.e., that a sequence of messages is included in another one,
- identification that certain messages are not sent or not received by an object,
- access to the state of an object at a given point in time such as before and after an invocation,
- detection of state changes,
- observation of the history of an object as it is created, passed around as argument or serves as sender or receiver,
- access the state of an object in the recursive state of another object.
- access whether a reference between two objects exists, is established or detached.

Note that message sequences identification is particularly important in presence of late-binding as a subclass may redefine a method and introduce new method invocations.

2. Logic and Trace-Based Testing

Reengineering object-oriented legacy applications is a challenging task as legacy applications traditionally lack tests, documentation and are generally very complex.

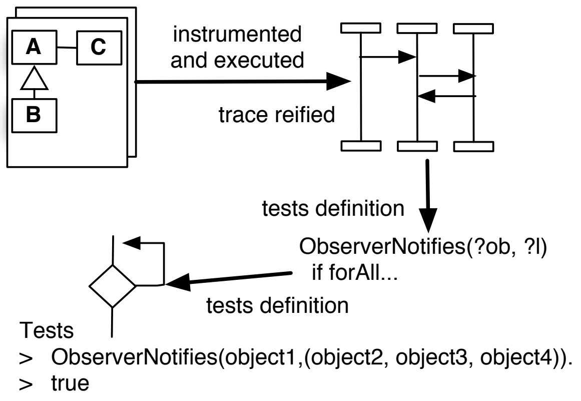

A
C
B
instrumented and executed
trace reified
tests definition
ObserverNotifies(?ob, ?l)
if forAll...
tests definition
Tests
> ObserverNotifies(object1,(object2, object3, object4)).
> true

Figure 1. The principle of logic trace-based testing.

Within this context, program execution is a trustable source of information that can be used to develop tests. The approach proposed in the paper is to specify tests based on the execution trace of the application and to express tests as logic predicates over the trace. Figure 1 describes our two-step approach.

1. First the source code is instrumented using some libraries [4]. This phase creates a program trace composed of events and *object states*. The trace is represented under the form of logic facts.
2. Tests are then expressed as logic queries over the trace, using a library of logic rules that facilitate the manipulation of the trace.

Figure 1 describes our approach: first the source code is instrumented using some libraries [4]. This phase creates a specific program trace composed of events and *object states*. This trace is represented as logic facts. Using a logic library that manipulates the trace tests are then expressed as logic queries over the trace. As explain below, the logic facts are represented as objects that represent logically the code of the applications under study. Note that other approaches used the trace of program but in the context of debugging procedural languages [8], or exploring and reverse engineering object-oriented applications [5],[12], [18]. In the context of debugging object-oriented program, query-based debugging has been proposed which combine conditional breakpoints with query-based [13][14]. A query-like expression is evaluated each time a conditional breakpoint is reached.

We validated our approach on several smaller examples, and on two complex systems: the MOOSE reengineering environment and the meta-interpreter [20]. While performing these experiments we noticed that similar situations occur frequently while testing. We captured this information under the form of patterns and explicitly supported them with predicates in our library of logic predicates. Sections 3, 4 and 5 describe the patterns we found in detail, categorized in three kinds. However, before we discuss the patterns in detail we first explain the logic language Soul that we used throughout the experiments to reprent and query the traces, and then the example that we will use to illustrate the patterns.

2.1. Logic Programming in SOUL

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, we specify rules and perform logic queries. The extension over standard prolog is that logic variables can be unified with objects. This means that there exists a binding of a logic variable to an object as it exist in the virtual machine.

The following query binds the class Object the root of the Smalltalk class hierarchy to the variable ?c. The term in angle brackets is evaluated by the Smalltalk compiler and the result is passed as argument to the query.

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

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

The implementation of the rule classWithName shows the syntax of rules where the head is separated from the body by 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 also used. This mechanism allows a programmer to pass variables from a logic environment to the Smalltalk environment.

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:]]
```

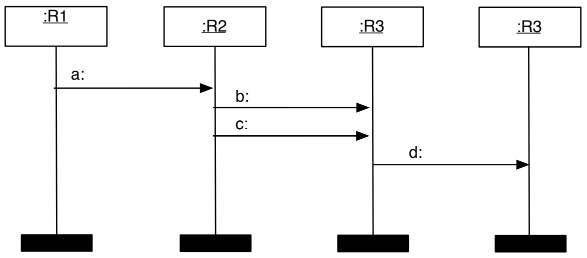


Figure 2. A simple scenario

2.2. Example

This example 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 process when performing trace-based test.

Scenarios. 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. Scenarios are behavioral archetypes that occur frequently in object-oriented systems because OOP distributes the implementation of a service between different classes, each with its own responsibility. Numerous design patterns are based on scenarios.

3. Interaction Testing

The first test pattterns provide a mechanism to match message sequences in the interaction diagram. This is frequently done to express patterns of object interactions, as will be illustrated on the example described in Section 2.2.

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

For example, when adding an element to a classification, a message add: is sent, resulting in a send of basicAdd: and, later on, in a send of setParent:. This chain of messages implying each other can be expressed by nesting the events (the sends of messages) that occur by using the predication contains. We express this chain as:

```
if event(?addEvent,
        selector(?addEvent, [#add:]),
        contains(<event(?basicAddEvent,
                    selector(?basicAddEvent, [#basicAdd:]),
                    contains(<event(?setParentEvent,
                                selector(? setParentEvent, [#setParent:]))
                              >))
                  >))
```

The query finds our desired message sequence. Note how the pattern that we expressed did not include *all* the messages that were being sent: it just expressed those messages that interested us. This feature allows us to express the important parts of the message sequence that we want to express, while not cluttering it with noise about messages we do not necessarily care about.

Scenario The first example showed how to express a message sequence, where one message implies the sending of another message (and so on). This can be extended to express trees of message sends instead of simply sequences. Because the argument of the contains predication is a list, we can express n-ary trees, and match these against the expression.

As a concrete example we see a recurring pattern in the trace when we ask the current service for a particular registered name. For example, while adding an object, the classification where we add the object needs to know its children (so that it can insert the element to be added). Because the root needs to know its children, it sends the message serviceFor: to the class ServicesConfiguration to retrieve the service that it will use to get its children. This class acts as a singleton, and get its sole instance (by sending itself the message current) for the registered service (using the message currentServiceFor:, which results in some internal messages to be sent, which does not interest us. Once the root classification has the service it needs, doExtentionalClassification will get called. We can express this tree of messages as follows:

```
if event( ?serviceForEvent,
              selector(?serviceForEvent, [#serviceFor:]),
               contains(< event( ?currentEvent,
                                    selector(?currentEvent, [#current])),
                           event( ?currentServiceForEvent,
                                     selector(?currentServiceForEvent, [#cur-
rentServiceFor:]),
                                    contains(< event(?basicServiceEvent,
                                                                            selec-
tor(?basicServiceEvent, [#basicServiceInformationFor:do:]))
                                            >))
             >))
```

4. State Testing

The second category of patterns deals with the change of object state during the execution of a program.

Context. 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.

The errors that state testing covers are: violations of a pre-postcondition, incorrect initialization of objects after creation, return of a wrong object from a method.

Pre- and Postconditions. Trace-based testing is also effecient to validate that a postcondition holds when examining legacy code, because with a single logic query it shows that a postcondition holds for every method execution that satisfies a precondition. To validate postconditions we need to accomplish the following steps: Identify the execution of an operation that is a target for postcondition validation. ensure the precondition and validate the postcondition, actions that are reflected in the following code:

```
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.

Let us 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])))
```

On a hypothetical bank application, testing that a debit account is debited if the balance is greater than the amount to be transferred, is expressed as follow:

```
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 as the internal state of an object may be encapsulated, *i.e.*, not accessible through a public interface.

To check the state of an encapsulated object we specify an expression that is similar to an OCL navigation expression. The navigation expression consists of a sequence of instance variable names that is used to stepwise access objects[1]. We call the sequence of instance variables used to access a nested state an *accesspath* from a root object to a nested object. Here is the definition of the query that given an accesspath accesses 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 write the following query

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

If we know an access path we can query an object at the specified position, but sometimes we would like to know whether an object is included in the recursive state of another object and retrieve its accesspath. For example if we want to know where or wether an object passed as argument to the method addObject: has been added, we write the following query:

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

Note that the use of the ressurectReceiverAfterEvent query is necessary because we have multiple states for the ?rootObject but are only interested in his state after

Now we can test whether the object passed as an argument is the same object accessible by a given 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 Objects. An object has a link to another object whenever it is possible to access an object through navigation along a path of instance variables. Frequent tests are based on the existence, the creation or the destruction of a link between objects.

To test the existence of a link or a linkpath between a first and a second object accessible from a root when can test whether the first object is included in the recursive state of the root and if 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 we use for testing whether a link is established or detached by a single operation defined as follows:

```
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))
```

5. Recursion Testing

The third category of patterns that we identified during our experiences is linked to the recursive traversal of structure as examplified by the Visitor pattern. It contains one pattern that expresses queries to test the traversal of a composite object structure with a visitor. Identify and test single or composite events that occur during recursion.

Context. Recursion in object-oriented systems is often concerned with traversing hierarchical object structures often realized by a composite pattern and a visitor that visits every element. 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.

Basically the following errors can be identified. 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 method in the visitor may 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.

Visitor testing. 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. The task of testing a recursive traversal is composed of the following tasks: ensuring a complete tree traversal, ensuring correct operation on nodes and specifying behavioral assertions for operations that are performed

[1]This access use the reflective API of Smalltalk but can be find in other languages such as Java.

on the composite structure. First we must make sure that every node that have to be traversed by a visitor, is really traversed. Then we require that the correct node type dependent operations are performed by a visitor. Finally, 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 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, *e.g.*, 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 such as 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 such 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,  ?recSelector,  ?compositeClasses) if
  objectsInRecursiveState( ?object,
    member([?object class], ?compositeClasses),
    ?root),
  event(?e, selectorAndReceiver(?e, ?recSelector, ?object)),
  ressurectReceiverBeforeEvent(?e, ?receiver),
  forall(objectsInState(?component,
    member([?object class], ?compositeClasses),
    ?receiver)),
  event(?e,
    contains(event(?ei,
      selectorAndReceiver(?ei, ?recSelector, ?component)
    )))
```

Now that we can verified that the object structure is properly traversed, we test method executions during the traversal. We classify methods whether they are executed on a single node and or wether they depend on the previous traversal of sub nodes. As a visitor traverses objects a type dependent method is called on the visitor by performing a double dispatch between the node and the visitor. In implementations, the method that is associated with a type is often based on a method naming convention.

Together with a query that checks if a double dispatch between an object and a visitor is done, we can check whether ever an object in a composite structure performs a double dispatch with a visitor.

```
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 such 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))
```

Let us 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.

```
arithmeticRecursesComposite 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.

6. Case Study: Verifying MOOSE Models

The MOOSE reengineering environment supports code representation and analysis. This code representation is based on the FAMIX model is similar to the UML but with extensions representing method invocations and attribute accesses [16]. To fill up models, MOOSE has importers for various source code languages such as C++, Java or Smalltalk. The task of an importer is to parse source code entities and to reified them in a MOOSE model. A MOOSE importer exposes interesting and complex behavior that is produced after calling the importModel method on the importer facade. It defines method parse tree traversal, Smalltalk meta-model access, object creation, model entity reification and the establishment of links between them. The complete behavior of an import of a small model produces more than ten thousand message sends.

In VisualWorks Smalltalk classes are organized in packages. Therefore, MOOSE imports a model from the set of classes that are located in a package.

Checking Class Import. The first fact that we would like to verify is that when a class is imported it is effectively created and added into the current model and if other dependent entities such as instance variables and methods are also imported.

We have to identify a location in the trace where FAMIX model entities of a certain type are created. For example, we learned browsing the code that a FAMIX class is created by the method #ensureClassEntityFor: which takes a Smalltalk class and returns a FAMIX entity.

Then we query the trace to observe the creation of a new entities and finally check if the set of classes is properly imported. Finally we check if other entities that are dependent of classes are properly imported.

Below are two queries that test whether every class in a package 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 exists in the Smalltalk image. We find 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. The expression [#MSEModel currentModel] refers to the current model in which imported entities will be added.

```
testEveryClassInPackage(?packageName, ?test(?c)) if
  forall(classInPackage(?packageName, ?c),
  ?test(?c))

testEveryClassInPackage([ReferenceModel],
      classEntityReifiedAndInModel(?c))
```

In the test testEveryClassInPackage, we check whether the query classEntityReifiedAndInModel(?) succeeds for every class in a package. The query classInPackage(?c) unifies ?c with every class belonging to ?packageName. The query forall checks whether a predicate passed as a second argument is true for every solution passed by the first query. 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.

Checking Metaclass Import. The importer should make sure that a class and its metaclass are imported. The following rule specifies this constraint by simply composing twice the previous query #classEntityReifiedAndInModel(?x) once for the class and a second time for its meta-class.

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

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.

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

Checking Class Composite Entity Representation. Representing a class implies representing its instance variables and its methods. Here we show how we test that a 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.

In MOOSE instance variables are reified by calling the method [#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:].

```
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])
```

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.

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

testEveryClassInPackage([ReferenceModel],
  existsLinkBetweenClassAndInstanceVariables(?c))
```

7. Case Study: Testing a Meta Interpreter

A meta interpreter for a language is an interpreter written in the same language that the language it interprets. A meta interpreter for Smalltalk is an interpreter written in Smalltalk that interprets Smalltalk code. Such an interpreter is useful to develop coverage tools and fine grained dynamic analysis [20]. But to be able to use a meta interpreter we should be sure that it is correctly implemented, *i.e.*, if a code fragment is correctly executed when interpreted. This case study show how our approach allows one to test complex recursive behavior. Basically the meta interpreter parses the source code and creates an abstract syntax tree. Then each node recursively receives the message eval: with evaluation context as argument.

Let us take as example the following Smalltalk method. It stores in the instance variable lastIndex the result of the invocation of the method makeRoom: sent to the instance variable space with the result of the message extraSpace as argument.

```
makeRoom

  lastIndex := space makeRoom: self extraSpace.
```

We then instrument the meta interpreter and generate a trace for the evaluation of the method makeRoom. Then we define a query that

Get references to the node objects in the parse tree:

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

Passing argument. The following test verifies that the result of the argument expression self extraSpace evaluation is passed as argument to the expression space makeRoom::

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

The following test checks whether after the method execution the instance variable ?lastIndex has the result returned by the expression x doSomething evaluation.

```
testAssignement if
  getParseTreeNodes(?assignment, ?messageExpression, ?argument),
  event(?evalMessageExpression,
        selectorAndReceiver(?evalMessageExpression, [#eval:], ?messageExpression)),
  event(?makeRoom, selector(?makeRoom, [#makeRoom])),
  ressurectReceiverAfterEvent(?makeRoom, ?receiver),
  instVarValue(?receiver, ['lastIndex'], ?lastIndex),
  equals(?lastIndex, [?evalMessageExpression return])
```

Test if the receiver of the message makeRoom: is the value of the instance variable space.

```
testReceiverEvaluation if
  getParseTreeNodes(?assignment, ?messageExpression, ?argument),
  event(?foo, selector(?makeRoom, [#makeRoom])),
  ressurectReceiverBeforeEvent(?makeRoom, ?receiver),
  event (?perform, selectorAndReceiver(?perform,
                          [#perform:receiver:arguments:class)),
  argument(?perform, [2], ?performOnReceiver),
  instVarValue(?receiver, ['space'], ?space),
  equals(?space, ?performOnReceiver)
```

8. Implementation and Trace Representation

TESTLOG. The approach presented has been fully implemented in a tool called TESTLOG whose 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 semantics of a logic query expresses the pass or fail semantics of a test: If a logic query fails then a test fails and if a logic query produces at least one result then the test succeeds.

The terms and logic rules are expressed in SOUL (Smalltalk Open Unification Language) [23]. SOUL is an extended prolog-engine written in Smalltalk which allows a programmer to integrate Smalltalk expressions within the logic rules themselves and use logic variables in Smalltalk expressions, creating a symbiotic relationship between the logic engine and the object-oriented language. TESTLOG is implemented on top of SOUL in a layered architecture that spawns different levels of describing behavior from single events to high level behavior of a whole application system.

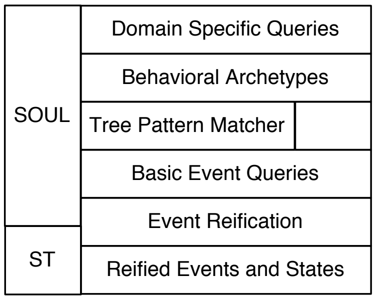
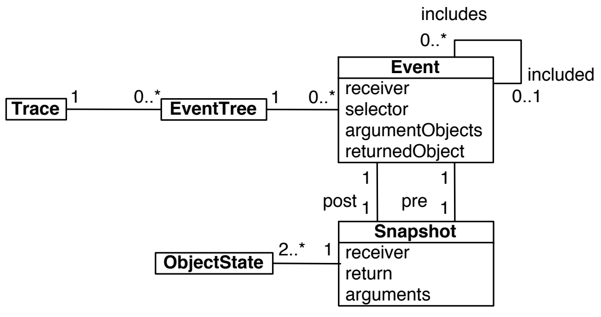


Figure 3. The Architecture of TESTLOG

The bottom layer comprises an object-oriented model that represents the event trace. A trace is stored as an object in the Smalltalk image and accessible via a singleton pattern. At the next abstraction level TESTLOG provides queries to access single events and states. This layer serves two purposes: First it reifies layer the object-oriented model into to the logic environment of SOUL by binding objects to logic variables. Second it provides basic queries on the event trace for querying events according to their attributes. This layer also defines queries based on state properties 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 expected collaboration patterns at different abstraction levels.

Events. Based on the recorded execution trace we reify an event model that allows us to express ordering and containment relations between events. As shown in the Figure 4, an event contains the following information: the sender object of a message, the receiver object of a message, the message name, a list of arguments that are passed and snapshot of the complete recursive state of the receiver before and after a method execution so that we are capable of reasoning on state changes.



What is the difference between return and returnObject

Figure 4. The TESTLOG event and trace model

On the set of events two basic relations exist 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.

$e < e1$ expresses that an event e precedes another event $e1$, while $e1\ in\ e$ expresses the fact that an event $e1$ is included within another event e. Here are the general axioms that are satisfied by any events a, b, c in the set of events: Mutual exclusion of relations ($a < b \rightarrow not(a\ in\ b)$), Non commutativity ($a < b \rightarrow not(b < a)\ a\,in\,b \rightarrow not(b\,in\,a)$), Transitivity: ($(a < b)\ and\ (b < c) \rightarrow (a < c)$), and Distributivity: ($(a\ in\ b)\ and\ (b < c) \rightarrow (a < c)$ $(a < b)\ and\ (c\,in\,b) \rightarrow (a < c)$).

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. As experienced by trying out on realistic instrumentation scenario the number of event trees is one order of magnitude smaller than the set of messages recorded.

Reification of Object States. In addition to the attributes we described earlier, every event refers the events it includes at the next level of the call tree. 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 (instance of the class ObjectState). Object states and object identity are completely separated in the model. An object identity remains the same during the whole lifetime of an object, however its object state may change.

For each event we reify the complete recursive state of the receiver before and after a method execution as a new 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 supports the possibility to identify an object in the recursive state of another. However the recursive state of an object can change during an execution, therefore we need to preserve it.

One strategy to preserve object states is to create a deep copy of an object with the same structure but with new objects in it. However we loose the ability to make a statement about objects in different events and states. For example we can no longer express that this is the *same* object that is passed as an argument that is added to the recursive state of another object by comparing the two object identities. Therefore we chose the following strategy: we build a graph that is isomorphic to the recursive state of the receiver and at each node of this graph a reference to the original object is maintained.

We define now two operations. The first operation is the 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, such that messages 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 shows the class model for reified object states.

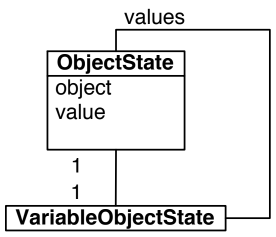


Figure 5. Reified State Model

Event Tree Pattern Matching. The requirement for having a tree pattern matching facility emerges from the fact that in object-oriented systems the message structure is deeply nested because of complex collaborations between objects. However, to test whether 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 as the hierarchy of events is represented as a tree, a form of tree pattern matching is used to locate event patterns that take into account the event hierarchy.

Because the general tree pattern matching problem with variables is NP-complete and would no be usable for pattern matching an execution trace consisting of several thousand messages the *left order embedding algorithm* described in [11] is used to pattern match an execution trace. The left-order 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.

9. Related Work

There is few work on dynamic analysis focusing on testing of object-oriented systems. In a pioneering paper [10] 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 [9] 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 with TESTLOG is that Caffeine has a linear representation of a trace such that it is not possible reason about nested events. Caffeine is also missing state reification such 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 [8] 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. Coca [7] supports the debugging of C program based on events. Lewis and Ducassé in [15] proposes to merge the approach of omniscient debuggers which collect all the run-time information and supports the exploration of the history and event-based tools that monitors program execution and allow the expression of sophisticated queries. However, the approach supports the exploration of the trace history but is not intended to express tests.

Other work that is based on event models and computations over an event trace to test program behavior can be found in [3] [2]. However it is based on procedural programming languages and does not take into account the specific behavioral aspects of object-oriented languages such object creation and the state of objects. Furthermore the author does not reason about the kind of behavior can occur in a program and how to test them.

While not exactly related to testing object-oriented applications, enhancements of traditional debuggers uses dynamic information to display traces. Visualizing debuggers can work directly via instrumentation on the program been executed or based on post-mortem traces [5], [12]. Visualization of dynamic information is also related to our work in the sense that it is based on a program trace. DePauw et al. [17] and Walker et al. [21] use program events traces to visualize program execution patterns and event-based object relationships such as method invocations and object creation.

Hart et al. use Pavane for query-based visualization of distributed applications. However Pavane only displays selected attributes of different processes and the does not allow complex queries.

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project Recast: Evolution of Object-Oriented Applications (SNF 2000-061655.00/1).

References

[1] D. Astels. *Test-Driven Development - A Practical Guide*. Prentice Hall, 2003.

[2] 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.

[3] M. Auguston. Building program behavior models. In *European Conference on Artificial Intelligence ECAI-98, Workshop on Spatial and Temporal Reasoning, Brighton, England*, Aug. 1998.

[4] J. Brant, B. Foote, R. Johnson, and D. Roberts. Wrappers to the Rescue. In *Proceedings ECOOP '98*, volume 1445 of *LNCS*, pages 396–417. Springer-Verlag, 1998.

[5] M. Consens, M. Z. Hazan, and A. Mendelzon. Debugging distributed programs by visualizing and querying event traces. In *Proceedings 1st. International Conference on Applications of Databases, LNCS 819*, 1994.

[6] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[7] M. Ducassé. Coca: An automated debugger for c. 1999.

[8] M. Ducassé. Opium: An extendable trace analyser for prolog. *The Journal of Logic programming*, 1999.

[9] Y.-G. Guéhéneuc. *Un cadre pour la traçabilité des motifs de conception*. PhD thesis, cole des Mines de Nantes, juin 2003.

[10] P. C. Jorgenson and C. Erickson. Object-oriented integration testing. *CACM*, 37(9):30–38, Sept. 1994.

[11] P. Kilpelinen. *Tree Matching Problems with Applications to Structured Text Databases*. PhD thesis, University of Helsinki, Departement of Computer Science, Nov. 1992.

[12] D. B. Lange and Y. Nakamura. Interactive visualization of design patterns can help in framework understanding. In *Proceedings of OOPSLA '95*, pages 342–357. ACM Press, 1995.

[13] R. Lencevicius, U. Hölzle, and A. K. Singh. Query-based debugging of object-oriented programs. In *Proceedings OOPSLA '97*, ACM SIGPLAN, pages 304–317, Oct. 1997.

[14] R. Lencevicius, U. Hölzle, and A. K. Singh. Dynamic query-based debugging. In R. Guerraoui, editor, *Proceedings ECOOP '99*, volume 1628 of *LNCS*, pages 135–160, Lisbon, Portugal, June 1999. Springer-Verlag.

[15] B. Lewis and M. Ducassé. Using events to debug java programs backwards in time. In *OOPSLA 03 but in something strange to check*, 2003.

[16] Object Management Group. Unified Modeling Language (version 1.3). Technical report, Object Management Group, June 1999.

[17] W. D. Pauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution patterns in object-oriented visualization. In *Proceedings Conference on Object-Oriented Technologies and Systems (COOTS '98)*, pages 219–234. USENIX, 1998.

[18] T. Richner. Using recovered views to track architectural evolution. In *ECOOP '99 Workshop Reader*, number 1743 in LNCS. Springer-Verlag, June 1999.

[19] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt. Reuse contracts: Managing the evolution of reusable assets. In *Proceedings of OOPSLA '96 Conference*, pages 268–285. ACM Press, 1996.

[20] M. Tilman. Building run-time analysis tools by means of pluggable interpreters. *ESUG 2000 Summer School*, 2000.

[21] R. J. Walker, G. C. Murphy, B. Freeman-Benson, D. Wright, D. Swanson, and J. Isaak. Visualizing dynamic software system information through high-level models. In *Proceedings OOPSLA '98*, pages 271–283. ACM, Oct. 1998.

[22] N. Wilde and R. Huitt. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, SE-18(12):1038–1044, Dec. 1992.

[23] R. Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, 2001.