Object-Oriented Legacy System Trace-based Logic Testing

In Proceedings of European Conference on Software Maintenance and Reengineering (CSMR 2006)

Stéphane Ducasse
LISTIC,
Université de Savoie
France

Tudor Gîrba
SCG,
University of Bern
Switzerland

Roel Wuyts
deComp,
Université Libre de Bruxelles
Belgium

Abstract

When reengineering legacy systems, it is crucial to assess if the legacy behavior has been preserved or how it changed due to the reengineering effort. Ideally if a legacy system is covered by tests, running the tests on the new version can identify potential differences or discrepancies. However, writing tests for an unknown and large system is difficult due to the lack of internal knowledge. It is especially difficult to bring the system to an appropriate state. Our solution is based on the acknowledgment that one of the few trustable piece of information available when approaching a legacy system is the running system itself. Our approach reifies the execution traces and uses logic programming to express tests on them. Thereby it eliminates the need to programatically bring the system in a particular state, and handles the test-writer a high-level abstraction mechanism to query the trace. The resulting system, called TESTLOG*, was used on several real-world case studies to validate our claims.*

Keywords: legacy systems, testing, dynamic information, logic programming

1 Introduction

During reengineering, it is crucial to write tests to make sure that the changes made do not introduce bugs in the system. Furthermore, writing tests is one pattern to support software understanding: encode the assumption in a test and check whether the test fails or succeeds [6].

But even though it is known that tests are important during reengineering, it is quite difficult to use them in practice. Testing a system requires to bring the system into a particular state, to then trigger the functionality to be tested, and finally to assert a condition. But here arises a catch-22: bringing a *badly-understood* system into a particular state requires detailed knowledge about the system. Moreover, due to poor design, it is often difficult to bring the system into the wanted state [6].

Object-oriented legacy system behavior is distributed over many interacting objects, making it necessary to test for complex collaboration scenarios. For example, to test the correct application of the Observer design pattern, the collaborations between a subject and its *registered* observers need to be verified. Only registered objects should receive an update message when a subject receives a change message, so that they can execute an appropriate action. A test of this collaboration requires an analysis of messages exchanged between objects. Note that approaches that reason on a *structural* reification of a program cannot fully capture these behavioural collaborations (see the example in Section 3.1, or for example [13]).

In summary, writing tests for legacy systems is complicated because (1) the system needs to be brought in a certain state, and (2) complicated scenarios need to be expressed.

This paper tackles these two problems by using a logic programming language to query execution traces. Indeed, in the context of legacy systems, one of the few trustable sources of information is the execution of the application itself [6]. Our approach consists of (1) reifying the execution traces, including the evolution of the state of the objects, and (2) testing assumptions on the reified traces by writing logic queries. Examples of assumptions are: test whether a particular message send involves another message send, test the pre and postconditions, or test how the state of an object changed before and after a particular message send.

There are three advantages to our solution. First of all, there is no need to find out how the system needs to be set up: that information is contained in the execution trace, and can immediately be used. Secondly the trace can be queried, so the developer can learn about the system by expressing assumptions on the execution behavior. Thirdly, the queries have access to the state of the object before and after each message send.

To validate our claims, we extended the logic programming language SOUL [23, 24] with a library to reason about execution traces. This allowed us to experiment on a num-

ber of case studies. While doing so we encountered frequently recurring queries, and assembled those in a trace-based dedicated logic library.

The rest of the article is structured as follows. Section 2 lists the requirements for a language to express tests. Section 3 presents our approach of logic trace-based testing and the logic programming language we used. Section 4 presents a library of frequently occurring testing queries that we assembled. Section 5 presents the application of the approach on one case study. Section 6 presents the implementation details of the approach. Section 7 discusses the applicability of our approach and its pros and cons.

2 Run-Time Information Requirements

This paper proposes to use the information contained in execution traces to express and check assumptions on the code. The advantage of this approach is that there is no need to bring the system in a desired state, since this information is implicitly contained in the trace. It is however not a trivial task to extract this test information from the execution trace. One reason is that the amount of information in an execution trace is typically huge [26, 11]. Secondly, the behavior that needs to be tested is typically dispersed over many steps of execution, interlaced with other functionality.

Manually analyzing an execution trace is therefore impractical, and there is a need for a language and a tool support. Such a testing language should support queries on objects that are *not in the scope of each other* - that is, objects that are not necessary in direct relation, but that can be found in the execution stack [18]. 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.

The next section presents our testing language, that meets these requirements.

3 Trace-Based Logic Testing

Our approach of trace-based logic testing lets reengineers test assumptions about the dynamic behaviour of a system by expressing logic queries on a reified execution trace. It is a two-step approach, illustrated in Figure 1.

1. First the source code is instrumented in order to obtain an execution trace composed of events and *object states*. The trace is represented as *logic facts*. The instrumentation technique depends on the language at hand. For example, our experiments in Smalltalk used *method wrappers* [3] .

2. Tests are then expressed as logic queries over the trace, using a library of logic rules that facilitate the manipulation of the trace. We use logic programming because of its expressiveness in detecting patterns, where unification and back-tracking are needed.

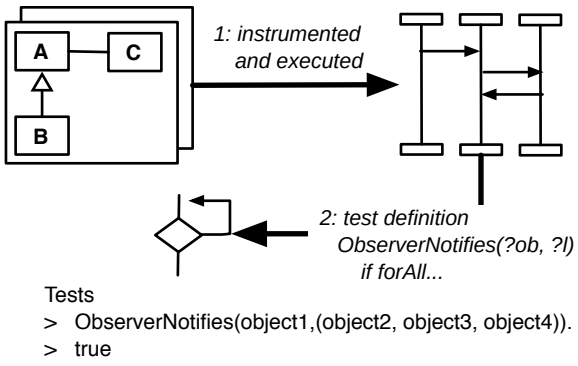

Figure 1. The principle of logic trace-based testing.

Other approaches already used execution traces of programs, but in the context of debugging procedural languages [8], or for exploring and reverse engineering object-oriented applications [5, 16, 21]. In object-oriented program debugging, query-based debugging combines conditional breakpoints with logic queries, evaluating a query-like expression is evaluated each time a conditional breakpoint is reached [17, 18].

To validate the approach we implemented it under form of a library for the logic language SOUL, and then by testing it on several small examples and on two larger systems: the MOOSE reengineering environment and a meta-interpreter. While experimenting we noticed that similar situations occur frequently while testing. We captured this information in the form of patterns and we explicitly supported them

with predicates in our library of logic predicates. Section 4 describes the patterns we found. However, before we discuss them we first need to introduce SOUL, the logic language that we used.

3.1 Logic Programming in SOUL

In this section, we give a short introduction on SOUL [23, 24]. SOUL is a full Prolog implementation in Smalltalk, but makes some syntactical and semantical enhancements that allow for a tight integration with an object-oriented language. Like in Prolog, the facts and the rules specify data, while the logic queries express the reasoning on this data. Unique in SOUL is that its atoms are objects (meaning that logic variables can be unified with objects), and that object-oriented code can be executed during the logic inference.

With SOUL it is therefore possible to perform logic queries directly on objects. Later on we use this feature to access the trace information. We also take advantage of the fact that everything in Smalltalk is an object, hence also classes and methods. Therefore SOUL can be used to directly query Smalltalk source code.

The approach is not limited to Smalltalk: Irish, for example, uses SOUL to reason about Java code [9]. The same approach is possible using a regular Prolog, but all the execution trace elements and objects have to be represented as logic facts. SOUL allows us to avoid this extra step.

The following query, for example, finds all methods in the class hierarchy of the class Collection that are named add: (denoted as #add: in the following example) [1]:

```
if classInHierarchyOf(?c, [Collection]),
    methodWithNameInClass(?m, [#add:], ?c)
```

As in Prolog, logic rules express relations between variables. For example, the following rule expresses the basic structure for the Visitor design pattern. It expresses the main structural relations between the visitor class, the abstract and concrete elements that are visited, and the name of the method that does the visiting. It accomplishes this by identifying the different inheritance relationships between the classes involved, and by expressing the form of the accept method, that sends back the visit method, passing along itself as an argument [24]:

```
visitor(?visitor, ?abstractElement, ?concreteElement, ?visitSelector) if
    classInHierarchyOf(?concreteElement, ?abstractElement),
    methodInClass(?acceptMethod, ?concreteElement),
    argumentsOfMethod(<?arg>, ?acceptMethod),
    parseTreeInMethod(<return(send(?arg, ?visitSelector, ?visitArgs))>,
                      ?acceptMethod),
    member(variable([#self]), ?visitArgs),
    selectorOfClassInProtocol(?visitSelector, ?visitor, ?)
```

When this rule is defined, we can check whether the class TreeNodeEnumerator is a visitor, and if so for which abstract and concrete elements:

```
if visitor(?visitor, [TreeNodeEnumerator], ?concrete, ?selector)
```

The result of this query are the ten concrete parse tree nodes, the four visitors used to visit these parse tree nodes, as well as the selectors that are present in the VisualWorks Smalltalk environment in which SOUL is implemented.

3.2 Logic Reification of the Trace

The execution traces obtained from running an application are represented as logic facts [16, 21]. A trace is composed of events, *i.e.*, messages sent to objects as shown in Section 6. Logic rules exists for querying the execution traces, such as querying for the existence of a given event, for accessing the event attributes (receiver, selector, arguments), or for querying and comparing the state of objects. We list some of these basic predicates that we use in the rest of the paper — for the complete description see [10].

- event(?e) returns all the events of a trace.
- event(?e, ?p) returns all the events satisfying a predicate.
- eventForSelector(?e, ?s) expresses that the event e is a message send with selector (*i.e.*, method name) s. For example eventForSelector(?e, [#add:]) holds for all events that have as selector #add:.
- receiver(?e, ?o) expresses that the event e has the object o as receiver.
- eventForSelectorContaining(?e, ?s, contains(?eventList)) expresses that the event e has a selector s and that it results in events from ?eventList being sent.
- resurrectReceiverBeforeEvent(?e, ?receiver) and ressurectReceiverAfterEvent(?e, ?receiver) are used to access the state of a particular object at a particular point in time, namely right before and right after the message associated with the event e is being sent to receiver.

4 Trace-Based Logic Testing Patterns

While the primitive predicates from Section 3.2 can be used to express tests over traces, we developed rules for patterns that we used repeatedly during our validation. We present two types of testing patterns: interaction and sequence testing patterns (Section 4.1) and state testing patterns (Section 4.2). We developed other patterns such as

[1] In SOUL, the keyword `if` separates the body from the head of a rule, which can be empty when we do not define a new query. # denotes symbols *i.e.*, unique strings in Smalltalk. Logic variables start with question marks (?); a comma denotes logical conjunction; lists are delimited with < and > , and terms between square brackets [ ] represent Smalltalk expressions that may use logic variables.

recursion, double dispatch and visitors testing but for space reason we do not present them here[2].

Before discussing the testing patterns, we first introduce the execution trace example we use to illustrate the patterns. The example uses the *classifications model* [25]. The major functionality of the classification model is to group *items* in *classifications*, and then to apply actions on these items by means of *services*. The service classes implement the Visitor design pattern and traverse classification items.

The execution trace we use is generated by adding an item to an existing classification, and then enumerating all the items in the classification to collect their textual representations. The resulting trace of this scenario is small, and contains 230 events, involving 30 objects (the root classification, some services, the objects contained as items and a wrapper class for each of these items). Figure 2 shows a subset of the trace that adds an item to the root classification. Note that all queries that we show query the complete execution trace, not just the subset shown in Figure 2.

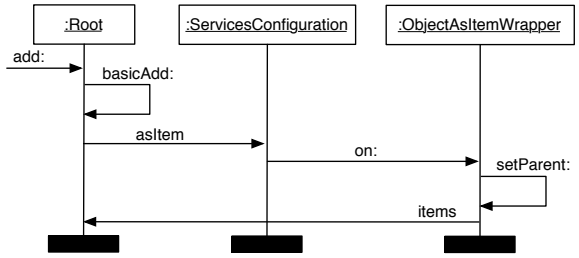

Figure 2. Part of the execution trace showing the addition of ServicesConfiguration to the root classification.

4.1 Testing Message Sequence Patterns

In this section we present two types of predicates to match message sequences in execution traces: message implications and scenario testing.

Testing Message Implications. By *testing message implication* we can express that a particular message send implies another particular message send.

For example, when adding an element to a classification, we see that 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 eventForSelectorContains predicate:

```
if eventForSelectorContaining(?addEvent, [#add:],
    < eventForSelectorContaining(?basicAddEvent,  [#basicAdd:],
                < eventForSelector(?setParentEvent, [#setParent:]) >) >)
```

[2]Please refer to [10] for a more detailed discussion

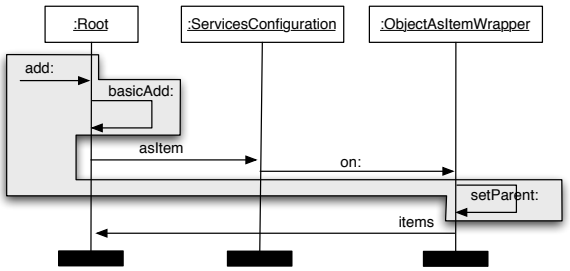

Figure 3. Result highlighting the result of the query expressing message implication.

The results of applying the above query on the trace displayed in Figure 2 is shown in Figure 3. Note how the pattern that we expressed did not include *all* the messages that were being sent: it just expressed those messages that were of interest for us. In this way, we can concentrate on only some parts of the trace, without needing to deal with the noise consisting in the messages we do not necessarily care about.

This feature is particularly important in the context of object-oriented programming since subclasses can extend super class behaviour.

Testing Scenario Trees. Testing message implication concentrates on expressing conditions over a sequence of messages. With *testing scenario trees* we concentrate on expressing predicates over trees of message sends. As the argument of the eventForSelectorContains predicate is a list, we express n-ary trees, and match these against the expression.

As a concrete example, we refine the message sequence defined above and express that sending basicAdd: in fact results in two messages being sent: a message asItem and a message items. In the course of evaluating asItem, a message on: is being sent. Hence we do not have a linear sequence of messages that is being sent, but a tree of message sends: basicAdd: has two branches, with one branch containing a single child. We can express this tree of messages as follows:

```
if eventForSelectorContaining(
        ?basicAdd,
        [#basicAdd:],
        < eventForSelectorContaining(
                    ?asItem,
                    [#asItem],
                    < eventForSelector(?on, [#on:]) >),
          eventForSelector(?items, [#items]) > )
```

4.2 Testing the State of Object Patterns

The previous section dealt with testing message sequences. This section concentrates on tests that take the

state of object state into account. We present four patterns related to state: pre and post condition testing, object inclusion, encapsulated state and object references

Testing Pre- and Postconditions. Pre- and postconditions encode generic assertions about the code. We use our approach to validate pre- and postconditions when examining execution traces.

To validate postconditions we identify the method execution that is the postcondition validation target, ensure the precondition and validate the postcondition, actions that are reflected in the head of the rule shown here:

```
if validatePostcondition(?eventQuery, ?precondQuery, ?postcondQuery)
```

Three queries are passed as arguments (?eventQuery, ?precondQuery and ?postcondQuery). The first query identifies the event (?e) for which pre- and postconditions should be ensured. The second query ensures the event's precondition and the third query ensures the event's postcondition.

Let's take once again the example of adding an object to a classification. The precondition asserts that the classification where the object is added can accept items (some classifications are closed, and do not allow object addition). The postcondition asserts that a classification that accepts the addition of items indeed has one more element than before the adding. The query looks as follows:

```
if validatePostcondition(
    eventForSelector(?addEvent, [#add:]),
    precondition(?addEvent,
                 ?pre,
                 [?pre acceptsAdding]),
    postcondition(?addEvent,
                  ?pre,
                  ?post,
                  equals([?pre size + 1], [?post size])))
```

The example above resembles a regular unit test method. One difference is that the unit test method requires a setup to set the original state of the system which, as we discussed in the beginning of the paper, is difficult to do. The other difference with a unit test is that this query can be reused on any trace,whilea unit test would only express the assumption for one specific classification.

Testing Objects Inclusion. Another valuable test that can be expressed is when one object contains another object in its state. For example, when adding an item to a classification, we want to test that the item object is present in the classification object after the addition.

The following query accomplishes this, by searching the add: message and recording its argument (?addedObject), then capturing the state of the classification after this message has been sent (?rootObject), and finally testing whether the added object is indeed included in the transitive state of the classification:

```
if eventForSelectorAndArguments(?e, [#add:], <?addedObject>),
  resurrectReceiverAfterEvent(?e, ?rootObject),
  includesObject(?rootObject, ?addedObject)
```

Note the usage of the predicate resurrectReceiverAfterEvent:. During the execution of the scenario the classification is in different states (once empty, then containing the element we added, etc.). Using this predicate we capture the state after the sending of an event. There exists a predicate resurrectReceiverBeforeEvent: that captures the state before the event is sent. Combining these two we can test that the added object did not yet exist before it was added, and that it exists afterwards.

Testing Encapsulated States. When testing states there are various situations where we need to break up the encapsulation as the internal state of an object may not be 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[3]. We call the sequence of instance variables used to access a nested state an *access path* from a root object to a nested object.

For example, given the state of the classification after the object was added to it we can get the internal lastIndex value of the collection used to store items in a classification using as access path items lastIndex, as shown in the following query:

```
if eventForSelector(?addEvent, [#add:]),
  stateAfterEvent(?addEvent, ?state),
  nestedObjectAt(<['items'], ['lastIndex']>, ?state, ?index)
```

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 access path. For example, when we add an element to a classification and we want to know where or whether an object passed as argument to the method add: has been added, we write the following query:

```
if eventForSelector(?e, [#add:]),
  argument(?e, [1], ?addedObject),
  resurrectReceiverAfterEvent(?e, ?rootObject),
  includesObject(?rootObject, ?addedObject, ?accessPath)
```

Note that the use of the resurrectReceiverAfterEvent query is necessary because we have multiple states for the ?rootObject (before and sending messages). With this predicate we specify that we are interested in the state after the message add: has been sent.

Testing Object References. An object has a reference 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 references between objects. To test the existence of a reference or a reference-path between a first and a second object accessible from a root we test whether the

[3]This access uses introspection as found in Smalltalk or Java.

first object is included in the recursive state of the root and if the first object includes the second object in its recursive state.

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

This rule gives us the basis to create two new rules for testing whether a reference is established or detached by a single operation defined as follows:

```
establishesReference(?event, ?fromObject, ?toObject) if
  resurrectReceiverBeforeEvent(?event, ?r1),
  resurrectReceiverAfterEvent(?event, ?r2),
  not(existsReference(?fromObject, ?toObject, ?r1)),
  existsReference(?fromObject, ?toObject, ?r2)

detachesReference(?event, ?fromObject, ?toObject) if
  resurrectReceiverBeforeEvent(?event, ?r1),
  resurrectReceiverAfterEvent(?event, ?r2),
  existsReference(?fromObject, ?toObject, ?r1),
  not(existsReference(?fromObject, ?toObject, ?r2))
```

This example terminates this short description of the most common patterns we used and identified while performing our case studies. It is worth to notice that our approach is open in the sense that new predicates can be implemented with relative ease using the abstractions provided by the *basic event queries* layer of our implementation (see Section 6).

5 Case Study: Verifying MOOSE Models

We claim that trace-based logic testing effectively tackles two problems that make writing tests for legacy systems complicated: (1) the system needs to be brought in a certain state in order to start the test, and (2) complicated scenarios need to be expressed. To validate these claims we used TESTLOG to experiment on several concrete cases [10].

We have already seen some tests on the classification model (although for our experiments we used bigger traces, as discussed later on) [25]. This section shows some more complicated tests that a reengineer could use to gain better understanding of the MOOSE environment [19].

The MOOSE reengineering environment has at its core the FAMIX meta-model (a UML-like meta-model with method invocations and attribute accesses). To extract models from source code, MOOSE has importers for various programming languages such as C++, Java or Smalltalk, and supports various tool interchange formats such as CDIF or XMI. The task of an importer is to parse source code entities and to reified them in a MOOSE model.

We experimented with the MOOSE importing from Smalltalk because it exposes interesting and complex behaviour: parse tree traversal, Smalltalk meta-model access, object creation, model entity reification and the establishment of references between objects. The complete behaviour of an import of a small model produces more than 10.000 message sends.

We stress that while two of the authors are the developers for MOOSE, the experiments were performed by a student that did not know MOOSE before starting the experiments. Thus, the student played the role of the reengineer that wants to understand an unknown system.

Testing 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, from browsing the code and from interviews the reengineer learned 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),
  eventForSelectorAndArguments(?e, [#ensureClassEntityFor:], < ?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))

if 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.

Testing 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.

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

Testing Class Composite Entity Representation. Representing a class implies representing its instance variables and its methods. Here we show how we test that a reference 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 reference 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
  eventForSelectorAndArguments(?e, [#ensureClassEntityFor:], < ?c >),
  eventForSelectorAndArguments(?e1, [#ensureInstVarFor:], < ?c >),
  equals(?cEntity, [?e return]),
  equals(?ivEntity, [?e1 return])
```

We now check whether for every pair ?cEntity and ?ivEntity a reference 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.

```
existsReferenceBetweenClassAndInstanceVariables(?c) if
  nonvar(?c),
  classAndInstanceVarEntity(?c, ?cEntity, ?ivEntity),
  existsReference(?cEntity, ?ivEntity, [MSEModel currentModel])

if testEveryClassInPackage([ReferenceModel],
  existsReferenceBetweenClassAndInstanceVariables(?c))
```

These examples illustrate how complex tests are assembled by composing advanced logic queries. They show how the approach supports an abstraction over the details of the execution trace [4, 26]. It can also be seen that there is no need to first bring the system in a certain state: the execution trace contains this information, and is simply harvested from the point in time which is interested to express tests for a certain behaviour,

6 TESTLOG: Implementation and Trace Representation

Figure 4 shows the layered architecture of TESTLOG. The bottom layer comprises an object-oriented model that

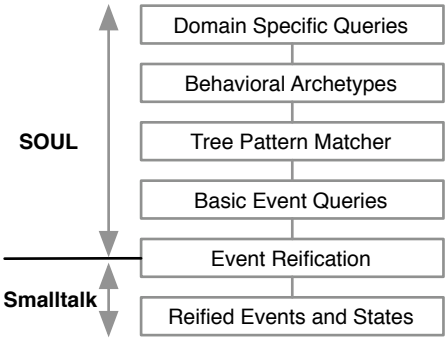

Figure 4. The architecture of TESTLOG

represents the execution trace. Each trace is stored as an object in the Smalltalk image. At the next abstraction level TESTLOG provides queries to access single events and states. This layer serves two purposes: (1) it reifies the object-oriented model into to the logic environment of SOUL by binding objects to logic variables, (2) it provides basic queries on the execution 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 implements a pattern matching query on event trees *i.e.*, matching a subtree in another tree.

Events. Based on the recorded execution trace we reify an event model with ordering and containment relations between events. As shown in the Figure 5, 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 able to reason about state changes.

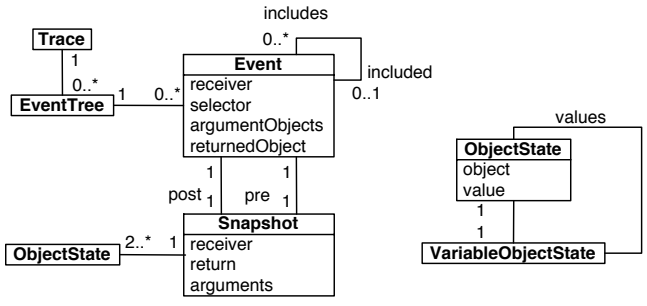

Figure 5. The event and trace model and the object state.

Reification of Object States. Every event refers the events it includes at the next level of the call tree. To make a state-

ment about object states that occurred during an execution a snapshot of the receiver recursive state before and after the event is made (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. We can then 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.

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 object collaborations. However, because a 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, we use the *left order embedding algorithm* described in [15] 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 left order embedding algorithm finds the first instance of the pattern if the tree is traversed in post-order.

7 Discussion

On the usefulness of the approach. Using execution traces as basis for our approach has the advantage that the testing context is already embodied within the trace. Moreover, traces capture the system as it is used and as such are easier to generate then writing unit tests one by one by hand. In addition after a query is writted, we can apply it to any traces we collect, hence reusing the abstraction and the test. Being able to manipulate tests in an abstract and declarative manner is powerful [16, 21].

While our approach supports a fine-grained and rich vocabulary to manipulate trace information (see Section 2), and while the addition of new abstractions is relatively easy, the syntax we use to define the queries is not really suited for end-users. We see this approach being used by expert reengineers in their first dealings with an unknown system. We acknowledge that our approach requires more knowledge of the application that we would like, which may seems to contradict our claim. However, using the system traces has the advantage that the engineer can directly start expressing tests on existing scenarios and does not have to struggle to bring the system in the wanted state. Our experience proved that this is a serious advantage when dealing with unknown systems. Nevertheless, we plan to assess how regular software engineers can use the approach without having a deep knowledge of the system.

On the scalability of the approach. The following table shows the sizes (number of classes and methods) of the three cases we studied. Note that we give this merely for completeness: since we only used information from the execution traces, the size of the source code does not really matter for the scalability of the approach.

System	#Classes	#Methods
Moose	215	3881
StarBrowser	80	569
MetaInterpreter	45	622

The next table shows the size of the execution traces that was used for querying. As shown the traces are not trivial (*i.e.*, around 200000 events), but our infrastructure managed to respond within seconds on even the most complicated examples from Section 5.

Traces	#Events	#Classes	#Methods
StarBrowser	200394	16	237
MetaInterpreter	27890	35	123
MooseImport	150876	83	553

Since our approach uses execution traces, it has the usual advantages and disadvantages related with using large amount of data. Large traces may pose scalability problems. Since TESTLOG stores the traces into object models expressed in Smalltalk, we can manipulate as large traces as can be stored in the Smalltalk image. This proved enough for our case studies. If not, we could resort to storing the trace information in a (preferably object-oriented) database, like *Gemstone*. The use of the logic language did not impose any limitation. However assessing the scalability of the approach with a larger case studies is one of our future work. We plan also to see how we can cut traces into coherent scenario or use cases based on the collaborating classes.

On the decision of recording the state. Contrary to most other approaches [16, 21, 11], our approach records the state of the objects before and after each message send. As such, we have fine-grained control over state changes, and we can for example express side-effects or references between collaborating objects. The natural question that arises is whether this approach scales in terms of space when the number of events grow. Our current implementation is quite naive and does not optimise memory usage. The following table shows the data we collected with the classification example.

Case	#Events	Size
StarBrowser	886	200k
	1 768	700k
	42 504	26.73MB
	539 096	372.62MB
Moose	42 350	12MB
	333 838	83MB
	514 422	144MB

The memory consumed per number of events in our naive implementation was around 48 k per 100 events in

the case of the StarBrowser experiments, and around 27 k per 100 events for the Moose case study. These measurements can change from one application to the other since the recursive state depends on the complexity of the interacting objects. For example, more memory is needed for the StarBrowser experiments because classification objects reference complex objects from the VisualWorks Smalltalk GUI application framework.

On the problem of sampling the traces. Another problem usually posed by dynamic analysis is the fact that a trace embodies only a specific object interaction, and that the code coverage is only based on a specific scenario [21]. Relying on traces does not prevent us to have incomplete or uninteresting contexts for the tests. In the case of badly understood systems, it is even more of a problem to know how to choose the traces. Yet, the literature present us with a number of solutions to this problem.

One reengineering pattern is to start the reengineering from running the system, by emphasizing that it is important to have the knowledge of the domain when understanding the code [6]. For example, the experiments with the StarBrowser consisted of opening the browser and adding items to classifications, resulting in the traces used in Section 4. Observing these traces and expressing assumptions on them introduced the main concepts for that application (classifications, services, visitor patterns, etc.).

Zaidman and Demeyer propose an approach to locate the most important places regarding a given feature [26]. They detect where one should start the implementation of a given feature by starting from the most important places from other similar features. In another complementary approach, Greevy and Ducasse offer means to relate features with structural entities (*e.g.*, classes), and thus providing the link towards combining the dynamic with static analysis [11]. With these approaches one can analyze the traces to check whether they are or not relevant for the wanted test.

Nevertheless, after obtaining the traces we can run the tests against all of them. Furthermore, we can obtain several versions of the same scenarios, and we can run the original assumptions against all the versions. Thus, we support the reengineering process by making sure that important assumptions are kept after the reengineering effort.

8 Related Work

In a pioneering paper [14] the authors argue that testing object-oriented software should not focus on units but on the message exchange between them in a scenario. However, the infrastructure to actually do this was not shown.

Auguston [2, 1] also uses a trace composed of event models and test programs. However it is based on procedural programming languages and does not take into account the specific behavioural aspects of object-oriented languages such object creation and the state of objects. Furthermore the author does not reason about the kinds of behaviour that can occur in a program, nor how to test for them.

Query-based debugging [17, 18] use logic programming to express complex queries over a large number of object. Some queries are triggered at run-time while the program is running. However contrary to our approach they cannot express temporal relationship or refer to previous state of an object.

Caffeine [12] is a Java-based tool that uses the Java debugging API to capture execution events and uses 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, and hence it is not possible to reason about nested events. Caffeine is also missing state reification. The reason is that its main context is not testing but reasoning about dynamic properties.

OPIUM [8] is a tool that allows a user to debug Prolog program using a set of debugging queries on event traces. 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 programs based on events. Opium and Coca are mainly used to show the values of variables. In addition, both Opium and Coca do not support object-oriented programming and object state analysis. They do not have been used to specify large application tests.

While not exactly related to testing object-oriented applications, enhancements of traditional debuggers use dynamic information to display traces. Visualising debuggers can work directly via instrumentation on the program being executed, or are based on post-mortem traces [5, 16]. Visualisation of dynamic information is also related to our work in the sense that it is based on a program trace. DePauw et al. [20] and Walker et al. [22] use program events traces to visualise program execution patterns and event-based object relationships such as method invocations and object creation.

9 Conclusion

For a reengineering effort to be successful it is crucial to have a reliable automated test suite because we want to ensure that the functionality remains the same after the change. Furthermore, automated testing can be used as an understanding tool by encoding the assumptions as tests and then running these tests to check the assumptions.

Testing requires bringing the system into a testable state. The problem with writing tests for legacy object-oriented systems is that it is difficult to bring the system into the

wanted state due to multiple reasons: lack of knowledge of the system, poor design, lack of time etc.

In this paper, we showed how to use the execution traces as a basis for expressing tests. We represent these traces in the form of logic facts and express the tests in the form of logic queries. We implemented our approach in the form of a query library using SOUL, a logic engine implemented in Smalltalk. Although our implementation is in Smalltalk, the approach itself is not specific for Smalltalk but can be applied to other languages as well – *e.g.*, Java.

We applied our approach on several Smalltalk case studies and we reported the experience by detailing several testing patterns. The tests patterns expressed are generic and can be applied to any traces of a given application. Furthermore, the definition of new abstractions to manipulate the trace information is simple.

We paid special attention to the performance issue and we showed that even our naive implementation scales for medium sized applications. Another problem posed by our approach is the coverage problem. Although it is not in the scope of this paper, we briefly pointed out complementary approaches from the literature.

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] M. Auguston. Program behavior model based on event grammar and its application for debugging automation. In *2nd International Workshop on Automated and Algorithmic Debugging, France*, May 1995.

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

[3] 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.

[4] A. Chan, R. Holmes, G. Murphy, and A. Ying. Scaling an object-oriented system execution visualizer through sampling. In *International Workshop on Program Comprehension*, 2003.

[5] M. P. Consens and A. O. Mendelzon. Hy+: A hygraph-based query and visualisation system. In *Proceeding ACM SIGMOD International Conference on Management Data*, pages 511–516, 1993.

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

[7] M. Ducassé. Coca: An automated debugger for C. In *International Conference on Software Engineering*, pages 154–168, 1999.

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

[9] J. Fabry and T. Mens. Language-independent detection of object-oriented design patterns. *Elsevier Computer Languages, Systems and Structures*, 30(1-2):21–33, 2004.

[10] M. Freidig. Trace based object-oriented application testing. Diploma thesis, University of Bern, Jan. 2004.

[11] O. Greevy and S. Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *Proceedings CSMR 2005*, pages 314–323, 2005.

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

[13] Y.-G. Guéhéneuc and H. Albin-Amiot. Using design patterns and constraints to automate the detection and correction of inter-class design defects. In *Proceedings TOOLS 2001*, pages 296–305, 2001.

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

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

[16] D. B. Lange and Y. Nakamura. Interactive Visualization of Design Patterns can help in Framework Understanding. In *Proceedings of OOPSLA '95*, pages 342–357, 1995.

[17] 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.

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

[19] O. Nierstrasz, S. Ducasse, and T. Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings ESEC/FSE 2005*, pages 1–10, 2005. Invited paper.

[20] W. D. Pauw, D. Lorenz, J. Vlissides, and M. Wegman. Execution patterns in object-oriented visualization. In *Proceedings COOTS '98*, pages 219–234. USENIX, 1998.

[21] T. Richner and S. Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In *Proceedings ICSM '99*, pages 13–22, Sept. 1999.

[22] 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, Oct. 1998.

[23] R. Wuyts. Declarative reasoning about the structure object-oriented systems. In *Proceedings TOOLS USA '98 Conference*, pages 112–124, 1998.

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

[25] R. Wuyts and S. Ducasse. Unanticipated integration of development tools using the classification model. *Journal of Computer Languages, Systems and Structures*, 30(1-2):63–77, 2004.

[26] A. Zaidman and S. Demeyer. Managing trace data volume through a heuristical clustering process based on event execution frequency. In *Proceedings CSMR 2004*, pages 329–338, Mar. 2004.