Test Blueprints — Exposing Side Effects in Execution Traces to Support Writing Unit Tests*

Adrian Lienhard, Tudor Gîrba, Orla Greevy and Oscar Nierstrasz
Software Composition Group, University of Bern, Switzerland
{lienhard, girba, greevy, oscar}@iam.unibe.ch

Abstract

Writing unit tests for legacy systems is a key maintenance task. When writing tests for object-oriented programs, objects need to be set up and the expected effects of executing the unit under test need to be verified. If developers lack internal knowledge of a system, the task of writing tests is non-trivial. To address this problem, we propose an approach that exposes side effects detected in example runs of the system and uses these side effects to guide the developer when writing tests. We introduce a visualization called Test Blueprint, through which we identify what the required fixture is and what assertions are needed to verify the correct behavior of a unit under test. The dynamic analysis technique that underlies our approach is based on both tracing method executions and on tracking the flow of objects at runtime. To demonstrate the usefulness of our approach we present results from two case studies.

Keywords: Dynamic Analysis, Object Flow Analysis, Software Maintenance, Unit Testing

1 Introduction

Creating automated tests for legacy systems is a key maintenance task [9]. Tests are used to assess if legacy behavior has been preserved after performing modifications or extensions to the code. Unit testing (*i.e.*, tests based on the XUnit frameworks [1]) is an established and widely used testing technique. It is now generally recognized as an essential phase in the software development life cycle to ensure software quality, as it can lead to early detection of defects, even if they are subtle and well hidden [2].

The task of writing a unit test involves (i) choosing an appropriate program unit, (ii) creating a *fixture*, (iii) executing the unit under test within the context of the fixture, and (iv) verifying the expected behavior of the unit using *assertions* [1]. All these actions require detailed knowledge of the system. Therefore, the task of writing unit tests may prove difficult as developers are often faced with unfamiliar legacy systems.

Implementing a fixture and all the relevant assertions required can be challenging if the code is the only source of information. One reason is that the gap between static structure and runtime behavior is particularly large with object-oriented programs. Side effects[1] make program behavior more difficult to predict. Often, encapsulation and complex chains of method executions hide where side effects are produced [2]. Developers usually resort to using debuggers to obtain detailed information about the side effects, but this implies low level manual analysis that is tedious and time consuming [25].

Thus, the underlying research question of the work we present in this paper is: *how can we support developers faced with the task of writing unit tests for unfamiliar legacy code?* The approach we propose is based on analyzing runtime executions of a program. Parts of a program execution, selected by the developer, serve as examples for new unit tests. Rather than manually stepping through the execution with a debugger, we perform dynamic analysis to derive information to support the task of writing tests without requiring a detailed understanding of the source code.

In our experimental tool, we present a visual representation of the dynamic information in a diagram similar to the UML object diagram [11]. We call this diagram a *Test Blueprint* as it serves as a plan for implementing a test. It reveals the minimal required fixture and the side effects that are produced during the execution of a particular program unit. Thus, the Test Blueprint reveals the exact information that should be verified with a corresponding test.

To generate a Test Blueprint, we need to accurately analyze object usage, object reference transfers, and the side effects that are produced as a result of a program execution. To do so, we perform a dynamic Object Flow Analysis in conjunction with conventional execution tracing [17].

Object Flow Analysis is a novel dynamic analysis which tracks the transfer of object references in a program execution. In previous work, we demonstrated how we success-

*Proceedings of European Conference on Software Maintenance and Reengineering (CSMR'08), 2008, pp. 83–92

[1] We refer to *side effect* as the program state modifications produced by a behavior. We consider the term *program state* to be limited to the scope of the application under analysis (*i.e.*, excluding socket or display updates).

fully applied Object Flow Analysis to reveal fine-grained dependencies between features [18]. While the concept of data flow has been widely studied in static analysis , it has attracted little attention so far in the field of dynamic analysis. Typically, dynamic analysis approaches focus on traces of method execution events [14, 24] or they analyze the interrelationships of objects on the heap [8, 15].

We have implemented a prototype tool to support our approach and applied it to two case studies to assess its usefulness. The main contributions of this paper are to show (i) a way to visually expose side effects in execution traces, (ii) how to use this visualization, the Test Blueprint, as a plan to create new unit tests, and (iii) a detection strategy based on our Object Flow meta-model.

Outline. This paper is organized as follows. Section 2 discusses difficulties of writing unit tests in the reengineering context. Section 3 introduces the Test Blueprint, which is fundamental to our approach. Section 4 presents our approach and Section 5 presents the case studies. The remainder of the paper (Section 6 – Section 9) includes a presentation of our implementation, a discussion of our approach, an outline of related work and a conclusion.

2 The Challenge of Testing Legacy Code

To illustrate the task of writing unit tests for unfamiliar code, we take as an example system a Smalltalk bytecode compiler. The compiler consists of three main phases: (1) scanning and parsing, (2) translating the Abstract Syntax Tree (AST) to the intermediate representation (IR), and (3) translating the IR to bytecode.

The following code (written in Smalltalk) illustrates a test we would like to generate for the addTemp: method of the FunctionScope class of our Compiler example. During the AST to IR transformation phase of compilation, variables are captured in a scope (method, block closure, instance, or global scope). The temporary variables are captured by the class FunctionScope, which represent method scopes.

```
function := FunctionScope new.
name := 'x'.

var := function addTemp: name.

self assert: var class = TempVar.
self assert: var name = name.
self assert: var scope = function.
self assert: (function tempVars includes: var).
```

Without prior knowledge of a system, a test writer needs to accomplish the following steps to write a new test:

1. **Selecting program unit to test.** When writing tests for a legacy system, the developer needs to locate appropriate units of functionality, *i.e.* a unit which is currently not already covered by a test and is not too large a unit for which to write a test.

2. **Creating a fixture.** To create a fixture, the developer has to find out which objects need to be set up as a prerequisite to execute the behavior under test. In this example, we need to create a FunctionScope instance, which is used as the receiver, and a string, which is used as the argument of the message send addTemp:. Creating this fixture is straightforward, however, if more objects need to be set up, it may be difficult to understand how they are expected to reference each other and how to bring them into the desired state. Incorrectly set up objects may break or inadvertently alter the behavior of the unit under test.

3. **Executing the unit under test.** Once we have the fixture, this step just involves executing the method, addTemp: in our example, using the appropriate receiver and arguments from the fixture. The execution of the program unit stimulates the fixture, that is, the execution returns a value and produces side effects.

4. **Verifying expected behavior.** We need to know what the expected return value and the side effects are. In our example, the returned object is expected to be a new TempVar instance with the same name 'x'. Furthermore, the returned TempVar should reference the FunctionScope object that we used as receiver. And finally, the FunctionScope should include the returned object in its tempVars collection. It is difficult to detect which side effects have been produced as a result of a program execution, as this information may be obscured in complex chains of method executions. Furthermore, by browsing the source code, it is difficult to ascertain this information. And although a debugging session reveals the required information, this may be a tedious approach in a large and complex system.

3 The Test Blueprint

In this section we introduce the Test Blueprint, which is fundamental to our approach. In Section 4 we then present how it supports writing tests to revealing the information required to establish the fixtures and identify the required assertions to verify the correct behavior.

The Test Blueprint exposes the object usage and the object state modifications, that is, side effects, in an execution trace. It is scoped to provide the required information about only a part of the program execution. We refer to such a part of the execution trace as an *execution unit.*

The left of Figure 1 illustrates an excerpt of an execution trace, displaying the method executions as a tree (the no-

tation follows the pattern *targetclass≫methodsignature*). The sub-tree with the root FunctionScope≫addTemp: represents an execution unit. The Test Blueprint of this execution unit is displayed on the right side of Figure 1.

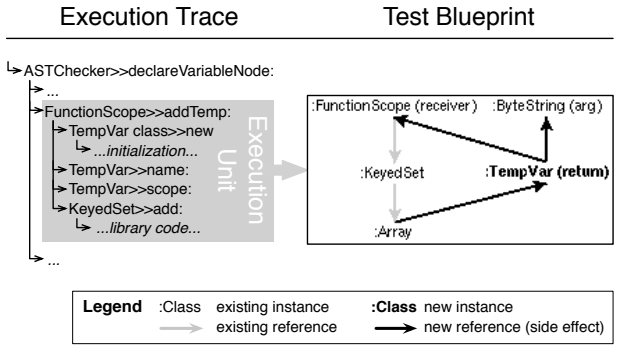

Figure 1. An execution unit and the Test Blueprint produced from it.

The Test Blueprint is similar to a UML object diagram [11] in that it shows objects and how they refer to each other. The key difference is that the Test Blueprint is scoped to the behavior of an execution unit and that it also shows (i) which objects were used by the execution unit, (ii) which references between the objects have been accessed, (iii) what objects have been instantiated, and (iv) what side effects were produced.

This information is encoded as follows in the Test Blueprint. We use regular typeface to indicate objects that existed before the start of the execution unit and bold typeface to indicate objects that are instantiated in the execution unit. The receiver object, the arguments, and the return value are annotated. The visualization shows only objects that have actually been accessed (but not necessarily received messages).

An arrow between two objects indicates that one object holds a field reference to another object. Like with objects, only references are displayed that have actually been accessed. Gray arrows indicate references that already existed before the execution unit was run.

A gray arrow displayed as a dashed line means that the corresponding reference is deleted during the execution unit. Black arrows indicate references that are established during the execution unit. Thus, the black and dashed arrows represent the side effects produced by the execution unit.

Let us consider again the highlighted execution unit in Figure 1, which contains all methods in the sub-tree rooted in FunctionScope≫addTemp:. In the execution trace this method is called in the following code.

```
ASTChecker≫declareVariableNode: aVarNode
  | name var |
  name := aVarNode name.
  var := scope rawVar: name.
  var ifNotNil: [ ... ] ifNil: [ var := scope addTemp: name ].
  aVarNode binding: var.
  ↑ var
```

As the Test Blueprint in Figure 1 shows, the receiver of the addTemp: message is an instance of the class FunctionScope and the single argument is a string. Furthermore, the returned object is a newly created instance of the class TempVar. The Test Blueprint in Figure 1 also shows the state of the returned object and what side effect the method execution addTemp: produced. Let us compare it to the addTemp: method printed below.

```
FunctionScope≫addTemp: name
1   | temp |
2   temp := TempVar new.
3   temp name: name.
4   temp scope: self.
5   tempVars add: temp.
6   ↑ temp
```

In the Test Blueprint we see that a new TempVar instance is created (compare to the code at line 2). The string passed as argument is stored in a field of the TempVar instance (3). Another side effect is that the new object is assigned a back reference to the receiver (4) and that it is stored in a keyed set of the receiver (5). Eventually, the new instance is returned (6).

In the case of the above example, most information contained in the Test Blueprint could also be obtained manually from the source code without too much effort (although, the successively called methods like name:, scope: and add: need to be studied as well). However, this task would not be so trivial in the case of more complex execution units, which may contain many executed methods and complex state modifications.

4 Approach: Supporting Test Writing

In this section we present our approach and experimental tool to support a developer to overcome the problems stated in Section 2 when writing tests. The underlying idea of our approach is to analyze the execution of the program to find examples for new unit tests.

Figure 2 provides an overview of our approach. From instrumented program and test executions we obtain data about the object flows, method execution events, and the existing test coverage. An analysis of this data then generates the interactive views for the user (the analysis is described in more detail in Section 6).

From the developer's perspective, the approach works as follows. In a first step, the developer interacts with our prototype tool to select an appropriate execution unit serving

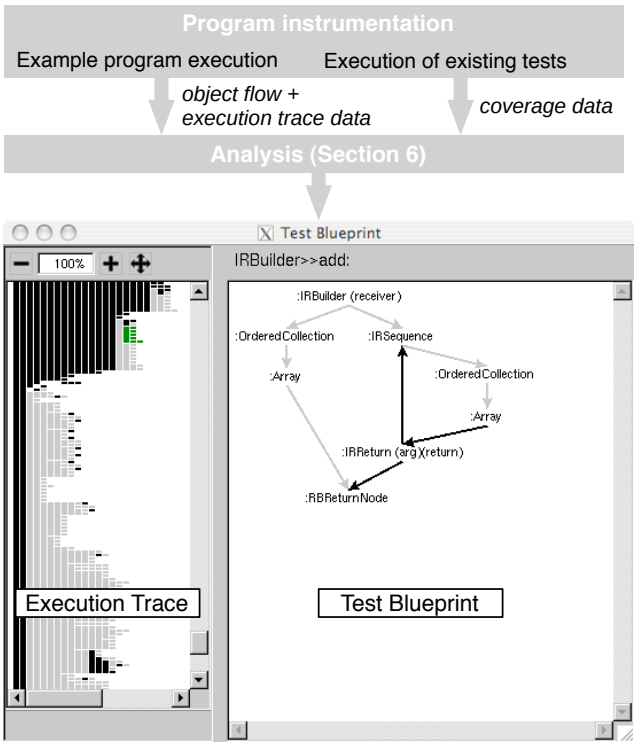

Figure 2. Overview of the Approach.

as an example for writing a new test in the execution trace. This is described in detail in Section 4.1. The selection causes the Test Blueprint view to be updated. The developer subsequently uses the view as a reference or a plan, providing him with the information necessary to implement the fixture (Section 4.2), to execute the unit under test (Section 4.3) and to write the assertions (Section 4.4).

4.1 Selecting a program unit to test

The left view of our tool illustrated in Figure 2 shows a filtered execution trace of the program, which was exercised by the developer. The trace is shown as a tree where the nodes (vertical rectangles) represent method executions. The layout emphasizes the progression of time; messages that were executed later in time appear further to the right on the same line or further down than earlier ones. This view is an adaptation of a view proposed by De Pauw *et al.* [7], which was later used in the Jinsight tool [8].

The goal of this view is to provide the developer a visual guide to search for appropriate example execution units in the trace that need to be tested. The tool provides options for filtering the amount of information in the trace based on different criteria, for example to show only the execution of methods of a particular package or a class of the system for which tests should be created.

Additionally, the execution trace is annotated with test coverage information. We compute this information by determining if, for each method in the trace, there exists a test that covers that method. Methods that are not covered are shown in black, whereas methods that are already exercised by a test are shown in gray. With the help of these visual annotations, a developer can more easily locate execution units of untested code on which he needs to focus.

4.2 Creating a fixture

When the developer selects an execution unit, the corresponding Test Blueprint is generated and displayed in the right view of our tool (see Figure 2). The selected execution unit (highlighted in green) now serves as an example to create a new unit test. To create the fixture of the new test, the Test Blueprint can be interpreted as follows.

First, the Test Blueprint reveals which objects need to be created, namely the ones that are not displayed in bold. Second, the gray references show the object graph, that is, how the objects are expected to refer to each other via field references. (In our tool, the name of the field can be accessed with a tooltip). The created object graph represents a minimal fixture as the Test Blueprint shows only those objects and references that have been accessed in the execution unit.

Unfortunately, it can be difficult to implement the fixture as proposed by the Test Blueprint. The problem is that it not always obvious how to create and initialize the objects. Often, not only the constructor has to be called with appropriate parameters but also further messages have to be sent to bring the object into the desired state. In some cases, the order in which those methods are executed may also be relevant.

To address this problem, the Test Blueprint provides a means to query for more detailed information about the creation of any of the objects it displays. We exploit the information we captured with our Object Flow Analysis of the system behavior. For each object, we backtrack its flow starting from the location where it is first used by the execution unit. Figure 3 shows the popup window for the FunctionScope instance.

This view reveals (i) the path of methods through which an object was passed into the execution unit (the top method indicates where the object under investigation was instantiated), and (ii) all messages sent to the object along this path. The number in parentheses indicates how many state modifications were produced, including transitive state.

In Figure 3 we see that the FunctionScope object of our running example is instantiated in the method newFunctionScope and that its constructor, initialize, produced seven side effects. In the same method, the execution of outerScope: produces one side effect. No other method execution except for addTemp: modified the object.

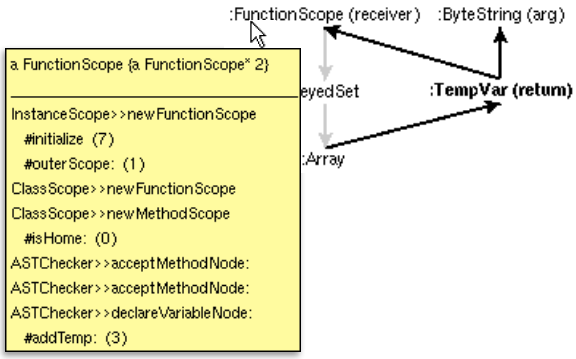

Figure 3. Backtracking object setup

Using the backtracking view, we can also find out that the KeyedSet is instantiated in the constructor of FunctionScope. Since no other object state is needed in the fixture apart from the string used as the argument, the following fixture is sufficient.

```
function := FunctionScope new.
name := 'x'.
```

4.3 Executing the unit under test

With the fixture created in the previous step, executing the unit under test is straightforward. The Test Blueprint shows which object from the fixture is the receiver and which objects are used as arguments:

```
var := function addTemp: name.
```

4.4 Verifying expected behavior

In this last step, the test writer needs to verify the expected behavior of the unit under test using assertions. The Test Blueprint reveals which assertions should be implemented:

- The objects shown in bold typeface are the instances that are expected to be created. Thus, assertions should be written to ensure their existence.
- The expected side effects have to be verified: black and dashed arrows between objects denote newly created or deleted field references.
- The Test Blueprint reveals which is the expected return value.

Once again, we illustrate this with our running example. Here the assertions derived step by step from the Test Blueprint are the following.

```
self assert: var class = TempVar.
self assert: var name = name.
self assert: var scope = function.
self assert: (function tempVars includes: var).
```

The assertions verify that the FunctionScope includes in its tempVars set the returned TempVar instance and that the back pointer from the TempVar to the FunctionScope exists. Furthermore, the new TempVar is expected to store the string passed as argument.

5 Initial Case Studies

In this section we present the results of two preliminary case studies. The first case study provides anecdotal evidence of the applicability of our approach for an industrial system, which supports the daily business of an insurance company. Our main focus with this study was to investigate how well our approach performs in the context of a real world legacy application and to gain experience for future controlled experiments.

In our second case study we applied our approach to a web content management system to evaluate how tests written supported by our approach differ from the tests already present, written by an expert.

5.1 Insurance broker application

In this case study we wanted to investigate questions regarding the applicability of our approach in a real world scenario, such as: *How straightforward is it to use our tool? How large do tests written using our approach get? Does the Test Blueprint concisely communicate the required information?*

Context. The Insurance Broker system is a web based application used both in-house by the insurance company employees as well as remotely by the external insurance brokers. The system has been in production for six years. While the system has been constantly extended over time, its core, which implements the insurance products and their associated calculation models, has not changed much. For the near future, however, a major change affecting core functionality is planned.

One problem associated with this project is that two of the three original developers of this application have left the team and the new members lack detailed knowledge about older parts of the system. At the time we carried out this experiment, the system consisted of 520 classes and 6679 methods. The overall test coverage amounted to 18% (note that we consider only method coverage).

Study setup. To investigate the usefulness of our approach in this context, we had access to a developer to write tests for the application core, which comprises 89 classes and 1146 methods. This developer has only ever worked

on newer parts of the system. This meant that he had basic knowledge of the system but was lacking internal knowledge about the core of the system.

As we wanted to ensure that the developer did not have to test any of the code he himself had implemented, we selected a version of the system dating from the time before he joined the team. In the first part of our study we trained him in our experimental tool and demonstrated how to use it to implement a new test. During the following two hours he used our tool to write new tests for the selected part of the system.

Results. The developer quickly understood the principle of the Test Blueprint and how to use it. Within these two hours, the developer created 12 unit tests. With the new tests, the coverage of the core increased from 37% to 50%.

Table 1 shows figures from the analysis of the developer's work. The first column labels the tests from 1 to 12. The second column indicates the total time the developer spent to find a new execution unit, to study the Test Blueprint and to implement the test. In the remaining columns we show the following measurements:

- The number of method executions in the selected execution unit.
- The size of the fixture in the Test Blueprint (number of pre-existing objects plus number of gray references). This number is about the same as the number of statements required to set up the fixture.
- The number of side effects in the Test Blueprint (number of new objects plus number of black and dashed references). This number corresponds to the number of assertions.

test #	time	exec. unit	fixture	side eff.
1	13	7	2	5
2	6	13	1	3
3	12	7	4	3
4	6	66	2	2
5	5	1	3	2
6	4	1	2	1
7	4	3	3	5
8	5	35	7	2
9	32	194	21	13
10	5	3	1	2
11	10	201	3	4
12	15	349	10	1
average	10	73	5	4

Table 1. Measurements of 12 tests (time in minutes, size of execution unit, fixture size, number of side effects).

The most complex test the developer created was #9, which tests critical functionality of the system (the calculation of discounts and subsidies). Surprisingly, this functionality was not covered by existing tests. A part of the Test Blueprint of this test is shown in Figure 4.

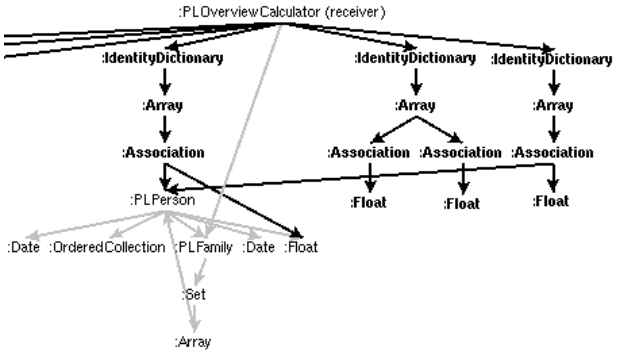

Figure 4. Detail of Test Blueprint from test #9

As Table 1 shows, this is an exceptional test with respect to the size of the Test Blueprint (the size of the Test Blueprint is the sum of the last two columns). Most tests were created from rather small Test Blueprints. Roughly, the size of the Test Blueprint corresponds to the number of minutes spent implementing the test.

On the other hand, the size of the execution unit (number of executed methods) does not seem to have a direct relationship to the complexity of writing a test. For instance, test #4 has an execution unit of size 66 but only a size of the Test Blueprint of 4. This test exercises the functionality of querying for available products. This involves iterating over all product models and verifying their availability, which caused the 66 method executions.

The largest execution unit is test #12 with 349 method executions. This behavior verifies the validity of a set of products, which involves complex business logic. This test, however, only required one assertion, which is to verify the returned boolean value.

Observations. One problem we observed was that selecting appropriate execution units in the execution trace is not supported well enough. Although the filtering of relevant methods and the highlighting of uncovered method proved very useful, the developer spent unnecessary time to find execution units that were not too trivial (for example, accessor methods) or not too complex to test.

On the other hand, the Test Blueprint worked very well and as intended. The developer used it as the primary source of information to implement the tests. Yet, sporadically he resorted to consulting the source code, for instance to study how to set up an object. Although, the backtracking of object setup helped to indicate what methods to look at, it did not completely replace the activity of consulting the code.

In summary, the developer successfully applied our tool to write tests for a system he only had basic knowledge of. Most of the chosen execution units had rather small Test Blueprints, so that it was generally not a problem to keep track of the objects and references to write the fixture and assertions. Large execution units in the trace did not necessarily indicate large Test Blueprints.

In a future case study, it would be interesting to measure how productive developers using our tool are compared to developers without tool support. Also, it would be interesting to see how and which program units are chosen and how the quality of the tests differ. The study we present in the following section, provides initial insights into the differences of the tests.

5.2 Web content management system

In this case study we wanted to investigate the following question: *How do tests written using our approach differ compared to conventional tests written by an expert?*

For this study, we selected the Pier web content management system [20]. Its core comprises 377 classes. We chose this application because we have access to the source code and we have direct access to developer knowledge to verify our findings.

Setup. To be able to directly compare tests written using the Test Blueprint with tests written by an expert of the system, we performed the following study. We randomly selected 14 non-trivial unit tests that are shipped with Pier. First we removed all assertions from the source code of the tests (84 in total), leaving only the code for the setup of the fixture and the execution of the unit under test. In the next step we used our approach to analyze the execution of each stripped down test case. Using the guidance of our Test Blueprint, we then systematically rewrote assertions for the tests as demonstrated in Section 4.

Results. In summary, the difference of the recreated assertions compared to the original 84 assertions is: (a) 72 of the recreated assertions are identical to the original ones, (b) 12 original assertions had no corresponding assertion in our test, and (c) our tests had 5 additional assertions not existing in the original code.

In 85% of the cases, the assertions we derived from the Test Blueprint were exactly the same as the ones implemented by the main author of the system. But with our approach some assertions were also missed (result b). A closer investigation revealed that most of the missing 12 assertions verify that special objects are left unmodified by the unit under test. The focus of our approach is the side effects, that is, on the modified state. Assertions to verify that special program state was left unchanged require in depth knowledge of the implementation. Our approach does not provide hints which unmodified objects would be worthwhile to verify.

The last result (c), shows that we found additional assertions not existing in the original tests. For instance, one of those assertions tests whether the state of the object passed as argument is correctly modified. The developer of Pier confirmed that, indeed, those relevant assertions were missed (and that he plans to integrate them).

6 Implementation

To generate the Test Blueprint, we employ our Object Flow Analysis technique. This dynamic analysis captures object reference transfers (*i.e.*, a dynamic pointer analysis) as well as method execution events as captured by traditional tracing techniques. Currently, our tracing infrastructure is implemented in Smalltalk, and is based on bytecode manipulation. The analysis, as described in the remainder of this section, has been performed postmortem in the Moose reengineering environment [19].

With the object flow data we can detect for any part of an execution trace the side effect its method executions produced on the program state. We consider the term *program state* to be limited to the scope of the system (for instance, arrays, streams, etc.) and the application under analysis. We do not take changes outside this scope into consideration, for instance, writing to a network socket or updating the display. Therefore, we refer to the *side effect* of some program behavior as the set of all heap modifications it produces.

The strategy we adopt for detecting the object flows described above is based on the concept of Object Flow Analysis [17]. The core of this analysis is the notion of object *aliases* (*i.e.*, object references) as first class entities, as shown in the Object Flow meta-model in Figure 5.

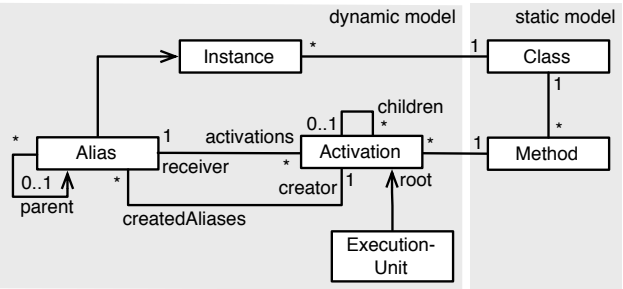

Figure 5. Core Object Flow meta-model.

The Object Flow meta-model explicitly captures object references, represented as *alias* entities, which are created in a method execution, represented by the *activation* entity. An alias is created when an object is (1) instantiated, (2) stored in a field or array, (3) read from a field or array,

(4) stored in a local variable, (5) passed as argument, or (6) returned from a method execution.

Each alias is bound to exactly one creator activation, the activation in which the alias makes the object visible. The transfer of object references is modeled by the *parent-child* relationship between aliases of the same object.

Once we have established our Object Flow meta-model, we can detect the import and export sets of an execution unit. We refer to imports and exports as the set of aliases through which an object is passed into respectively out of an execution unit. The imported flows show us how references to previously existing objects have been obtained, whereas the exported aliases represent the side effect of the execution unit. In the Test Blueprint, the imported aliases are represented as gray arrows and the exported aliases as black arrows.

We define the imported and exported aliases of an execution unit in OCL as shown in Figure 6 (according to the model in Figure 5).

```
context Activation::allChrildren : Set(Activation)
derive: children->union(children.allChildren)

context Alias::allParents : Sequence(Alias)
derive: parent->append(parent.allParents)

context ExecutionUnit::activations: Set(Activation)
derive: root->union(root.allChildren)

context ExecutionUnit::exported : Set(Alias)
derive: activations.createdAliases.isWriteAlias

context ExecutionUnit::imported : Set(Alias)
derive: activations.createdAliases.isReadAlias
  ->reject(a | self.exported->includes(a.parent))

context Alias::path : Sequence(Activation)
derive: creator->append(allParents.creator)
```

Figure 6. OCL specification of an execution unit's derived properties.

The constraint `exported` derives all aliases that are written to a field or array, that is, the side effects produced in the execution unit. Similarly, the constraint `imported` derives all aliases created in the execution unit that are reading from a field or array that was written outside the execution unit. To check for the latter restriction, we test whether the parent of the read alias (which always is a write alias) is in the `exported` set. This ensures that the situation where a value is read from a field, but the field was defined in the execution unit, is not considered an import.

The constraint `path` is used for backtracking object setup (Figure 3). It returns the method activations that created an alias and its parents. To backtrack the path of an object, we select the first alias through which an object is imported into an execution unit. The messages sent through an alias are modeled by the 1:n relationship between aliases and activations.

7 Discussion

In this section we discuss the limitations and the scalability of our approach, and we lay out the required capabilities of the dynamic analysis technique.

Limitations. As the goal of our approach is to support test writing for unfamiliar systems, it is of less value for developers with in depth knowledge of the system. With our approach, the developer is limited to writing tests for only the program units that have been exercised in the example runs of the program. The tests created with our approach only verify the behavior for one particular fixture, whereas an expert may know how to vary the the fixture. Furthermore, our approach does not support finding existing defects in the system. In spite of those limitations, we believe that our approach is valuable in the context of legacy system maintenance and reengineering as expert knowledge is typically missing in such projects.

The Test Blueprint has limitations with respect to the amount of data it can display. As the insurance broker case study showed, Test Blueprints of the size of 30 objects can still be understood, but the view does not scale for the analysis of truly large parts of the execution. For our approach, this is not directly a problem since unit testing practice suggests to choose small units [1]. Therefore, if the Test Blueprint contains too many objects, this is a sign that the unit covers too much functionality and should be split into several unit tests. Yet, we believe that the Test Blueprint can be enhanced to be more concise.

Another limitation of the Test Blueprint is that it does not indicate exceptions thrown but not caught in the execution unit. In some cases, tests are written to verify that an exception is thrown. Since the behavior of normal program runs usually does not rely on the exceptions, this is not a severe limitation but it should still be added for completeness.

Scalability. As with most other dynamic analysis approaches, scalability is a potential problem. Object Flow Analysis gathers both object reference transfers and method execution events. It consumes about 2.5 times the space of conventional execution trace approaches. Our approach does not require tracing programs over a long period of time but rather captures single user sessions. Therefore, making our implementation performant enough to be practically usable was not a big problem. For instance, the trace used in the insurance broker case study is 54MB on disk and it takes about 2 minutes to load it, generate the model and analyze it (on a MacBook Pro, 2GHz Intel Core Duo).

Dynamic analysis requirements. To cover all object flows in a concrete program execution, the tracing technique has to be implemented carefully. Our Object Flow tracer not

only tracks objects of application classes but also instances of system classes and primitive type values. For example, collections and arrays have to be taken into account as they preserve permanent object references between the holder of the collection and its contained elements. Furthermore, since all method executions and state modifications have to be captured, also behavioral reflection has to be dealt with appropriately. The execution of multiple concurrent threads are captured as different traces to make them distinguishable in the user interface.

We chose Smalltalk to implement a dynamic analysis prototype because of its openness and reflective capabilities, which allowed us to evaluate different alias tracking techniques. We are currently implementing an Object Flow Tracer for Java. In its current state, it allows us to detect side effects, but lacks support for tracking the transfer of object references, which is required for precisely detecting imported aliases and the backtracking of object setup.

8 Related work

Specification based testing. There is a large body of research on automatically generating tests or test input from specifications and models [22]. In the work of Boyapati *et al.*, they present the Korat framework which generates Java test cases using method preconditions [4]. Compared to our approach, these test generation tools require *a priori* specifications, which often do not exist for legacy systems. For our approach, the code and the running system are the only required sources.

Automatic testing. Fully automated testing tools exist such as DART [12]. DART performs random testing and employs dynamic analysis to optimize coverage. In contrast, our approach analyses real program execution that has been initiated by the developer to create example scenarios for which to write tests. The result of applying our approach are conventional unit tests. In contrast to the automated testing approaches, the intent of our approach is not to achieve a full branch coverage as required by McCabes structured testing criterion [23].

Trace based testing. Testlog is a system to write tests using logic queries on execution traces [10]. Testlog tackles the same problem as our approach, however, it does so in a very different way. The problem of creating a fixture is eliminated by expressing tests directly on a trace. With our approach, the developer creates conventional unit tests without the need to permanently integrate a tracing infrastructure into the system to be able to run the tests.

Other query based approaches primarily targeted for debugging can also be used for testing [16, 13]. In contrast to our approach, the query based approaches are tailored towards finding inconsistencies rather than to support writing new tests because these techniques require *a priori* knowledge to write queries.

Object reference analyses. Related to the Test Blueprint and our underlying Object Flow Analysis is the dynamic analysis research concerned with the runtime structure of object relationships. For instance, Super-Jinsight visualizes object reference patterns to detect memory leaks [8], and the visualizations of ownership-trees proposed by Hill *et al.* show the encapsulation structure of objects [15]. The key difference of our Object Flow Analysis is that it has an explicit notion of the *transfer* of object references. This is crucial as it allows us to analyze side effects of arbitrary parts of a program execution. Also, the object flow information is required for backtracking object setup.

Tonella *et al.* extract object diagrams statically and dynamically from test runs [21]. As discussed earlier in this paper, the Test Blueprint is similar to an object diagram. The difference is that the Test Blueprint provides additional information such as side effects. Our approach is based on a dynamic analysis because dynamic analysis produces a precise under-approximation whereas static analysis provides a conservative view. This property of dynamic analysis is welcome we are interested in concrete and minimal examples, since unit testing else is rendered unpractical.

Dynamic data flow analysis is a method of analyzing the sequence of actions (define, reference, and undefine) on data at runtime. It has mainly been used for testing procedural programs, but has also been extended to object-oriented languages [3, 6]. Since the goal of those approaches is to detect improper sequences on data access, they do not capture how read and write accesses relate to method executions. The output of the dynamic analysis is used directly as test input, whereas our analysis is used to instruct the developer.

9 Conclusions

In this paper we present an approach to provide support to developers faced with the task of writing unit tests for unfamiliar legacy code. The underlying idea of our approach is to use example runs of the program to expose side effects in execution traces. The paper illustrates how the proposed visualization, the Test Blueprint, serves as a plan to write new unit tests without requiring a detailed understanding of the source code.

A preliminary evaluation of our approach indicates its usefulness for real world reengineering projects. From the insights of our case studies, we plan to improve our approach in the following two ways. First, we want to make the Test Blueprint more concise to achieve a better guidance of the developer during test writing. A potentially interesting way would be to collapse and summarize parts of the Test Blueprint. For instance, this would allow us to hide implementation details such as the internal structure of collections. Second, we want to evaluate existing execution trace analysis approaches to ease the identification of interesting

execution units. For instance, the clustering of event traces to identify related functionality [24] or the use of metrics to predict testability [5] would be interesting starting points.

Eventually, further empirical evaluations are needed to obtain more evidence of the applicability of our approach in large object-oriented systems.

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Analyzing, capturing and taming software change" (SNF Project No. 200020-113342, Oct. 2006 - Sept. 2008) and the financial support of CHOOSE and ESUG.

References

[1] K. Beck and E. Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):51–56, 1998.

[2] A. Bertolino. Software testing research: Achievements, challenges, dreams. In *FOSE '07: 2007 Future of Software Engineering*, pages 85–103, Washington, DC, USA, 2007. IEEE Computer Society.

[3] A. S. Boujarwah, K. Saleh, and J. Al-Dallal. Dynamic data flow analysis for Java programs. *Information & Software Technology*, 42(11):765–775, 2000.

[4] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *ISSTA'02*, pages 123–133, Roma, Italy, 2002. ACM.

[5] M. Bruntink and A. van Deursen. Predicting class testability using object-oriented metrics. In *Proceedings of the Fourth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*. IEEE Computer Society Press, Sept. 2004.

[6] T. Y. Chen and C. K. Low. Dynamic data flow analysis for C++. In *APSEC '95: Proceedings of the Second Asia Pacific Software Engineering Conference*, page 22, Washington, DC, USA, 1995. IEEE Computer Society.

[7] W. De 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.

[8] W. De Pauw and G. Sevitsky. Visualizing reference patterns for solving memory leaks in Java. In R. Guerraoui, editor, *Proceedings ECOOP '99*, volume 1628 of *LNCS*, pages 116–134, Lisbon, Portugal, June 1999. Springer-Verlag.

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

[10] S. Ducasse, T. Gîrba, and R. Wuyts. Object-oriented legacy system trace-based logic testing. In *Proceedings of 10th European Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 35–44. IEEE Computer Society Press, 2006.

[11] M. Fowler. *UML Distilled*. Addison Wesley, 2003.

[12] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on programming language design and implementation*, pages 213–223, New York, NY, USA, 2005. ACM.

[13] S. Goldsmith, R. O'Callahan, and A. Aiken. Relational queries over program traces. In *Proceedings of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, pages 385–402, New York, NY, USA, 2005. ACM Press.

[14] A. Hamou-Lhadj and T. Lethbridge. A survey of trace exploration tools and techniques. In *Proceedings IBM Centers for Advanced Studies Conferences (CASON 2004)*, pages 42–55, Indianapolis IN, 2004. IBM Press.

[15] T. Hill, J. Noble, and J. Potter. Scalable visualisations with ownership trees. In *Proceedings of TOOLS '00*, June 2000.

[16] 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.

[17] A. Lienhard, S. Ducasse, T. Gîrba, and O. Nierstrasz. Capturing how objects flow at runtime. In *Proceedings International Workshop on Program Comprehension through Dynamic Analysis (PCODA 2006)*, pages 39–43, 2006.

[18] A. Lienhard, O. Greevy, and O. Nierstrasz. Tracking objects to detect feature dependencies. In *Proceedings International Conference on Program Comprehension (ICPC 2007)*, pages 59–68, Washington, DC, USA, June 2007. IEEE Computer Society.

[19] O. Nierstrasz, S. Ducasse, and T. Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE 2005)*, pages 1–10, New York NY, 2005. ACM Press. Invited paper.

[20] L. Renggli. Magritte — meta-described web application development. Master's thesis, University of Bern, June 2006.

[21] P. Tonella and A. Potrich. Static and dynamic c++ code analysis for the recovery of the object diagram. In *Processings of 18th IEEE International Conference on Software Maintenance (ICSM 2002)*, Los Alamitos, CA, USA, 2002. IEEE Computer Society.

[22] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan-Kaufmann, 2006.

[23] A. Watson and T. McCabe. Structured testing: A testing methodology using the cyclomatic complexity metric. Technical report, National Institute of Standards and Technology, Washington, D.C., 1996.

[24] A. Zaidman and S. Demeyer. Managing trace data volume through a heuristical clustering process based on event execution frequency. In *Proceedings IEEE European Conference on Software Maintenance and Reengineering (CSMR 2004)*, pages 329–338, Los Alamitos CA, Mar. 2004. IEEE Computer Society Press.

[25] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, Oct. 2005.