Exposing Side Effects in Execution Traces*

Adrian Lienhard, Tudor Gîrba, Orla Greevy and Oscar Nierstrasz
Software Composition Group, University of Bern, Switzerland

Abstract

We need to understand the impact of side effects whenever changing complex object-oriented software systems. This can be difficult as side effects are at best implicit in static views of the software, and typically execution traces do not capture data flow between parts of the system. To solve this problem, we complement execution traces with dynamic object flow information. In our previous work we analyzed object flows between features and classes. In this paper, we use object flow information to analyze side effects in execution traces and to detect how future behavior in the trace is affected by it. Using a visualization, the developer can study how a selected part of the program accessed program state and what side effect its execution produced. Like this, the developer can investigate how a particular part of the program works without needing to understand the source code in detail. To illustrate our approach, we use a running example of writing unit tests for a legacy system.

1 Introduction

With object-oriented programs, the gap between static structure and runtime behavior is particularly large. Unlike pure functional languages where the entire flow of data is explicit, in object-oriented systems the flow of objects is not apparent from the source code. Through reference fields, objects may outlive the execution scope in which they are created and thus may influence behavior of another part of a system at a later point in time. This characteristic of object-oriented systems represents *side effects* on the program state. As this is a key characteristic of object-orientation, it is crucial when analyzing a program functionality, to take side effects into consideration.

While the concept of data flow has been widely studied in static analysis [11], it has attracted little interest in the field of dynamic analysis. Most approaches either analyze traces of method execution events [8, 22] or they analyze the interrelationships of objects on the heap [4, 10]. However, to detect side effects and how they affect future behavior in the trace, we need to also capture fine-grained information about the transfer of object references.

Side effects are difficult to understand, not only because of implicit information flow, but also because complex chains of method executions can hide where they are produced [20]. In this paper we explore how exposing side effects in execution traces can support a developer to better understand and to maintain an object-oriented system. Before making a change to complex object-oriented legacy system, a developer needs to identify and understand side effects produced by the behavior he intends to change, and the parts of the system are potentially affected by it.

To facilitate the detection of side effects we adopt our Object Flow Analysis technique [14]. We demonstrated in previous work the usefulness of this technique to identify dependencies between features [15] and to discover relationships between classes by analyzing how they exchange objects [13]. The use of Object Flow Analysis as presented in this paper takes a different angle. Our focus here is to relate object flows to method execution events to reason about side effects in object-oriented systems.

Motivating example. As a motivation for side effect analysis, we present an example use case where knowledge of side effects supports the production of tests. Writing regression tests for legacy systems is a crucial maintenance task [5]. Tests are used to assess if legacy behavior has been preserved after modifying the code. They also document reengineering efforts. However, the task of writing tests is nontrivial when there is a lack of internal knowledge of a legacy system [6].

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

- *Creation of a fixture.* This involves determining which objects should be initialized so that the behavior to be tested can be successfully executed.
- *Execution of a method under test.* Once the fixture is established, this just involves executing the method under test using the appropriate receiver and arguments.
- *Verification of the expected results.* We need to know which conditions to test, *i.e.*, what the expected side effect is and what the return value of the method is as a result of execution.

*Proceedings of the 3rd International Workshop on Program Comprehension through Dynamic Analysis (PCODA 2007), pp. 11-17

We propose a visualization to expose side effects, which serves as a blueprint to set up a minimal fixture and to verify the expected test results.

Outline. In Section 2 we introduce our approach and subsequently in Section 3 we illustrate how it can be applied to facilitate the generation of unit tests. We outline related work in Section 4 and we conclude in Section 5.

2 Approach

To analyze side effects, we complement execution traces with dynamic object flow information by tracing the transfer of object references (*i.e.*, a dynamic pointer analysis). With this additional behavioral information, we can detect for a selected part of an execution trace, the precise effect it had on the program state and which future behavior in the trace was affected by the resulting heap modifications. We consider the term *program state* to be limited to the scope of the application under analysis and the system classes it uses. We do not take changes outside this scope into consideration, *e.g.*, 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.

We structure the discussion as follows. First we present our analysis of information flows in execution traces and how we detect side effects. Then, in Section 2.2 we describe how to expose the side effects. A visualization shows how program execution used and affected the program state. In Section 2.3 we present how side effects were propagated so as to influence behavior in the trace at a later point in time.

2.1 Detecting side effects

Typically, UML sequence diagrams are used to visualize execution traces (or parts of them) [7]. We base our analysis of side effects on an adaptation of a more scalable view introduced by De Pauw *et al.* [3], which was later also implemented in the Jinsight tool [4].

Figure 1 (left) illustrates a small portion of an execution trace represented by the experimental tool we built for side effect analysis. The trace is presented as a tree where the nodes 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. For a comparison with sequence diagrams we refer the reader to work by de Pauw *et al.* [3].

This visual representation emphasizes the underlying model of our approach — namely, to consider a method execution as including the transitively executed methods. We refer to a partial execution trace as a *sub-trace*. This corresponds to the call-return procedure abstraction of most programming languages. Figure 1 illustrates a selected sub-trace indicated by a rectangle in the execution trace.

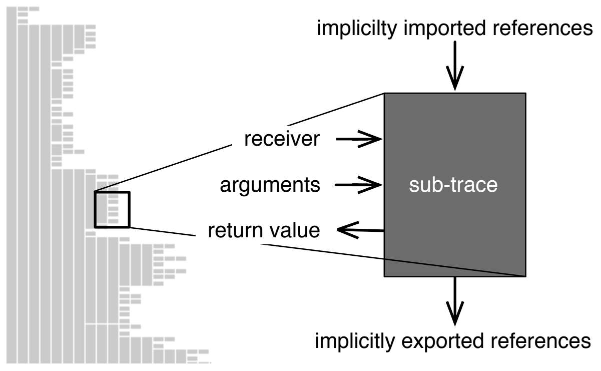
implicilty imported references
receiver
arguments
return value
sub-trace
implicitly exported references

Figure 1. Flow of objects into and out of a sub-trace.

To contribute to the computation of a program, the method executions of a sub-trace must have some effect on information flow. A sub-trace defines an encapsulation boundary with respect to object references being passed in and out of it. The out going flows are represented by the returned value (of the first method) and all objects stored in fields. We refer to those flows as *exports*.

During execution, the methods of the sub-trace also use existing program state. Accessible objects are the receiver and the arguments (of the first method execution) and further objects obtained from fields and global variables. We refer to those in going flows as *imports* (see Table 1).

	import	export
explicit	receiver and arguments	return value
implicit	field/global read	field/global write

Table 1. Flows at the sub-trace boundary

The receiver, arguments and the return value are *explicitly* passed at the sub-trace boundary. However, the flows out of or in to fields or global variables are hidden in the methods of the sub-trace. Therefore, it is often complex to grasp those *implicit* flows when studying an object-oriented system.

The implicitly exported flows represent the *side effects* of the execution of the sub-trace on the program state. The implicitly imported flows show us which objects have been used to produce those effects.

The strategy we adopt for detecting the object flows described above is based on the concept of Object Flow Analysis. The core of this analysis is the notion of object *aliases* (*i.e.*, object references) as first class entities [14, 13], as shown in the Object Flow meta-model in Figure 2.

The Object Flow meta-model explicitly captures the fol-

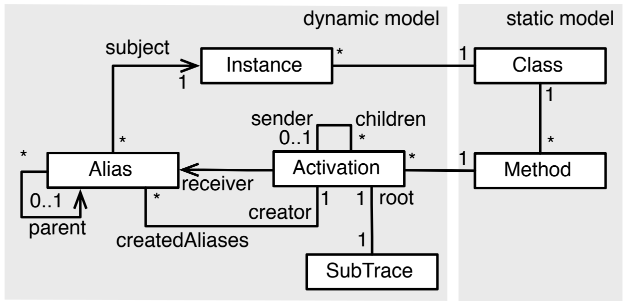

Figure 2. Core Object Flow meta-model.

lowing object references (represented as alias entities in the meta-model) created in a method execution (represented by the *activation* entity in the meta-model). An alias is created when an object is (1) instantiated, (2) stored in a field or global variable (including indexable fields), (3) read from a field or global variable, (4) stored in a local variable, (5) passed as argument, or (6) returned from a method execution. 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 a sub-trace. The implicit object flows are defined as follows:

- The implicitly exported references are exactly those that are represented by the field and global *write* aliases that are created in activations of the sub-trace.
- The implicitly imported references are exactly those that are represented by the field and global *read* aliases that (i) are created in activations of the sub-trace and that (ii) do not have a parent write alias in the set of exported references.

The constraint (ii) makes use of the object flow information. That is, for each field read alias the corresponding field write alias is known (parent relationship). This constraint ensures that field references that are defined in the same sub-trace where they are used, are not considered as imports.

2.2 Exposing side effects

In the previous section we discussed how exported and imported object references are detected. The implicitly exported object references represent the side effects produced by a sub-trace, while the imported object references indicate which previously existing program state is used during its execution.

In this section we present our approach to expose side effects. Our experimental tool provides two interactive views, which are both illustrated in Figure 3. On the left side of Figure 3 the view with the execution trace is shown (as discussed previously). When clicking on a sub-trace in this view, a new window is opened (see Figure 3 right). We refer to the view it shows as the *side effect view*.

This view is similar to a UML object diagram [7] in that it shows objects and how they refer to each other. The key differences are: (1) it is scoped to the behavior of a selected sub-trace, and (2) it provides additional information based on the side effect analysis.

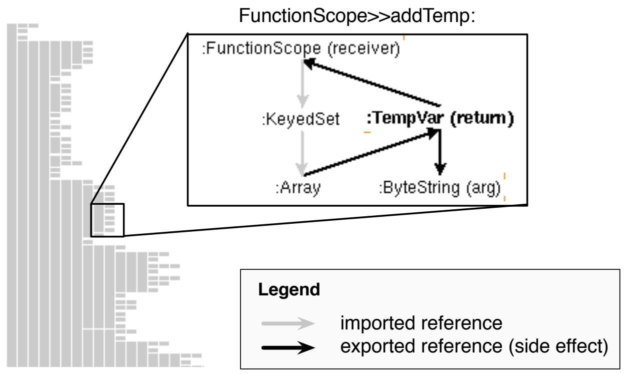

Figure 3. The side effect view of a selected part of the execution trace.

Figure 3 shows all objects being accessed (but not necessarily receiving messages) during the execution of a selected sub-trace. We annotate the class name of objects that have been explicitly passed into a sub-trace, *i.e.*, the receiver, the arguments, and the return value. We use regular typeface to indicate objects that existed before the execution of the sub-trace, whereas we use bold typeface to indicate objects that are instantiated during its execution.

An edge between objects indicates that one object has a field reference to another object. Gray edges indicate references that already existed before the sub-trace was run, *i.e.*, they refer to imported object references. If a gray edge is dashed this means that the reference is deleted during the execution of the sub-trace. Black edges indicate the references that are established during the execution of the sub-trace, *i.e.*, the ones that are exported. Thus, the black edges represent references that are the side effects of the sub-trace.

The main goal of this view is to (i) show which objects are used by the sub-trace, (ii) what side effects are produced on those objects, and (iii) how the objects refer to each other, *i.e.*, to make the reference paths between objects visible.

In Figure 3 we see, for example, the execution of a method addTemp: by an instance of FunctionScope (receiver). A ByteString is passed as argument. This execution produces a side effect: a new TempVar instance is created (it is displayed with bold text), and the instance is not only returned but also stored in an already existing array. Another side effect is that the returned object is assigned a back reference to the receiver. Also, the object passed as argument, a ByteString, is stored in a field of the TempVar instance.

In our prototype, the name of a field (object reference) is displayed as a tooltip when moving the mouse over it.

2.3 Exposing the impact of side effects

In the previous section we discussed how we expose the side effects produced by the execution of a sub-trace. In this section we show which future behavior in the trace is affected by the side effect.

The side effects of a sub-trace are essentially the implicitly exported references (field or global stores). Other sub-traces, which occur later in the trace, may then import these references. The importing sub-traces may then proceed to further export the references. Therefore, to detect also method executions indirectly affected by the side effects of a sub-trace, we need to track how imported references are further propagated in the trace.

In the execution trace under analysis we highlight methods that are affected by a side effect. When a sub-trace is selected, we highlight all method executions that contain references originating from its exported references.

An execution trace with affected method activations is illustrated in the subsequent section (see highlighted methods in Figure 4), which exemplifies how the detection of side effects can be used to support the task of writing tests.

3 Case Study: Using the Side Effect View as a Test Blueprint

With our approach we make use of dynamic information captured from instrumented example runs of the system. The side effect view serves as a blueprint for writing tests by making explicit:

- The minimal fixture: only the gray objects and gray references are expected to exist before executing the method to be tested.
- What results to verify: the black objects and black references, which are produced as side effects of executing the method.

We motivate our work by presenting an example to illustrate how knowledge of side effects supports writing unit tests. The example is taken from an open-source Smalltalk bytecode compiler. The compiler works in three phases: (1) scanning and parsing, (2) translating the Abstract Syntax Tree (AST) representation to the intermediate representation (IR), and (3) translating the IR to bytecode.

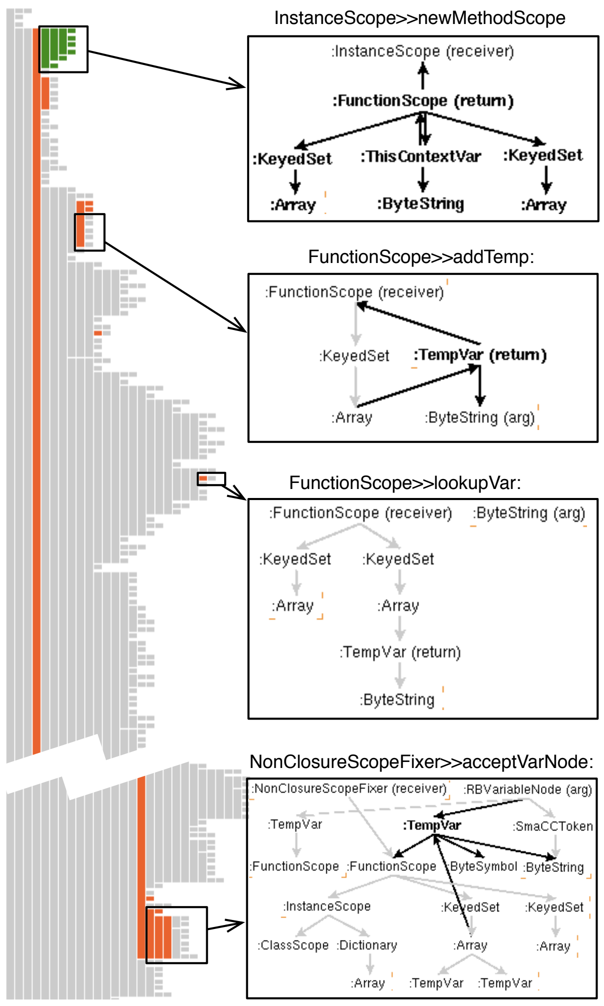

Figure 4. Side effect views serving as blueprints for writing tests.

Let us assume we want to test the implementation of how variables are captured in the AST to IR transformation phase. Since variables are always defined in a specific scope (method, block closure, instance, or global scope), classes like InstanceScope or FunctionScope look like interesting classes to test. However, they are complex to understand without studying the source code in detail.

First, we identify in the source code the method InstanceScope≫newMethodScope, which looks promising as a starting point. Thus, we first query the trace for one of the executions of the method. Figure 4 on the top right shows the side effect view of an execution. On the top left the corresponding sub-trace is highlighted in green.

For the test we want to write, setting up the fixture only requires the creation of an InstanceScope (this is the only object that is used but not created in the sub-trace and there are no gray references). Then we can execute the method we want to test.

```
instance := InstanceScope new.
function := instance newMethodScope.
```

Next, we investigate the side effect view to determine what conditions we need to verify. First we want to assert that the return value actually is a function scope object. Then we check whether the function scope correctly references the instance scope as its outerScope (this is the name of the field, which in our prototype is obtained by a tooltip and hence is not shown in Figure 4). Both keyed sets, tempVars and capturedVars, are assumed to be empty. Also a new instance of ThisContextVar is created, which is stored in the function scope and has a back reference.

```
self assert: function class = FunctionScope.
self assert: function outerScope = instance.
self assert: function tempVars isEmpty.
self assert: function capturedVars isEmpty.

var := function thisContextVar.
self assert: var class = ThisContextVar.
self assert: var scope = function.
self assert: var name = 'thisContext'.
```

With the assertions above, we have tested all side effects that the method, together with the 9 methods it indirectly executes, is expected to produce. Now, what further tests can we write for this part of the system? We can answer this question by investigating the methods that are affected by the side effects of the method under test. In the execution trace, the affected methods are marked with orange. In our example (see Figure 4), we identify five locations in the trace where methods are affected, the last one being much later in time than the others. These method executions are examples of which other behavior uses the state that is changed as a result of running the method we are testing.

For instance, addTemp: is called on the function scope we created. We can now take its side effect view (see Figure 4) to write the next test. It shows that for the fixture the function scope as it is created in the previous test is sufficient. Additionally, we need the string 'x' as an argument. We can now test the expected side effects, for instance, that the function scope includes the new instance TempVar and that this instance correctly references the name of the temporary variable we passed as an argument.

```
...
var := function addTemp: 'x'.

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

Along the same lines we can write tests for the remaining usage examples. For instance, to write a fixture for FunctionScope≫lookupVar:, we see that it depends on the TempVar produced as a side effect of the previous test. Therefore, we only need to add the following lines to it.

```
result := function lookupVar: 'x'.
self assert: result = var.
```

The method lookupVar: is special in that it produces no side effects (there are no bold instance names nor black references).

The last side effect view shown in Figure 4 is more complex than the previous ones. It is also an example for the deletion of a reference. The reference from the RBVariableNode to the TempVar instance is deleted (dashed arrow) and replaced by a reference to a newly instantiated TempVar object.

```
function := InstanceScope new newMethodScope.
block := function newFunctionScope.
var := block addTemp: 'x'.
node := (RBVariableNode named: 'x') binding: var.
fixer := NonClosureScopeFixer new scope: method.

fixer acceptVariableNode: node.

newVar := method lookupVar: 'x'.
self assert: newVar class = TempVar.
self deny: newVar = var.
self assert: node binding = newVar.
self assert: newVar scope = method.
```

4 Related work

Typically, dynamic analysis techniques focus on execution traces, which capture method execution events [3, 9, 21]. As dynamic analysis implies large amounts of data, much research effort has been concerned with the accessibility of large traces using filtering or summarization techniques [3, 9], or by identifying recurring execution patterns [12]. While those approaches mainly analyze method execution sequences, our approach additionally takes into account the object flow, and hence is capable to relate program execution and its effect on the program state.

Another dynamic analysis research area is concerned with the structure of object relationships. For instance, Tonella *et al.* extract the object diagram from test runs [19], Super-Jinsight visualizes object reference patterns to detect memory leaks [4], and the visualizations of ownership-trees proposed by Hill *et al.* show the encapsulation structure of objects [10]. To support debugging, tools like the GNU Data Display Debugger [23] visualize program state. The key difference to our approach is that we not only extract the object reference relationships, but in addition we detect how reference relationships are modified by a specific part of the program execution.

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 been extended to object-oriented programming languages as well [1, 2]. Since the goal of those approaches is to detect improper sequences on data access, they do not capture how objects are passed through the system, nor how read and write accesses relate to method executions. To the best of our knowledge, Object Flow Analysis is the only dynamic analysis approach that explicitly models object reference transfers.

In the area of static analysis, there is a large body of research on interprocedural side effect analysis. More recent research addresses the precision problem of static side effect analysis (or the analysis of pure methods) in object-oriented programs [16, 17, 18]. As static analysis does not take a concrete execution scenario into account, it provides a conservative view (which may even include infeasible execution paths of the program). Dynamic analysis on the other hand produces a precise under-approximation. Our approach makes use of this property by accurately detecting the side effects as they are produced during example runs of the program. This allows for directly relating side effects to where they occur in an execution trace. Another advantage of our approach is that it handles reflection, multi-threading, or dynamic code updates, which typically pose problems in static analysis.

5 Conclusions

In this paper we propose to expose side effects in execution traces through Object Flow Analysis. We use the dynamic object flows to define the side effect view, and we exemplify how we can use this information to guide the developer when writing tests, in particular in the case of legacy object-oriented systems.

As with most other dynamic analysis approaches, scalability is a potential limiting factor. Object Flow Analysis gathers both object references and method executions, thus it consumes about 2.5 times the space of conventional execution trace approaches [13].

Apart from the amount of data gathered, the side effect view is most vulnerable to large amount of data because it shows single objects and references between them. Our initial case studies indicate that also sub-traces of the size of several hundred method executions can be analyzed. However, the view does not scale for the analysis of truly large parts of the execution in which hundreds of objects are modified. We plan to tackle this problem by collapsing and summarizing parts of the object reference graph shown in the view. In this way, implementation details such as the internal structure of collections can be hidden to yield a more concise side effect view.

In our current studies we limit the analysis of program state to the application and the system library classes. However, side effects outside this scope, for instance, writing on a network socket or updating the display, are not captured. For future work we plan to extend the analysis to take also side effects into account that are outside this scope (*e.g.*, to capture how data in a RDBMS is affected).

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).

References

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

[2] 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.

[3] 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.

[4] 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.

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

[6] 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.

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

[8] 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.

[9] A. Hamou-Lhadj and T. Lethbridge. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In *Proceedings of International Conference on Program Comprehension (ICPC 2006)*, pages 181–190, Washington, DC, USA, 2006. IEEE Computer Society.

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

[11] M. Hind. Pointer analysis: Haven’t we solved this problem yet? In *2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE’01)*, Snowbird, UT, 2001.

[12] D. J. Jerding, J. T. Stasko, and T. Ball. Visualizing interactions in program executions. In *Proceedings of ICSE ’97*, pages 360–370, 1997.

[13] A. Lienhard, S. Ducasse, and T. Gîrba. Object flow analysis — taking an object-centric view on dynamic analysis. In *International Conference on Dynamic Languages (2007)*, 2007. To appear.

[14] 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.

[15] 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.

[16] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 1–11, New York, NY, USA, 2002. ACM Press.

[17] A. Rountev. Precise identification of side-effect-free methods in Java. In *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 82–91, Washington, DC, USA, 2004. IEEE Computer Society.

[18] A. Salcianu and M. C. Rinard. Purity and side effect analysis for Java programs. In *VMCAI*, pages 199–215, 2005.

[19] 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.

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

[21] A. Zaidman, T. Calders, S. Demeyer, and J. Paredaens. Applying webmining techniques to execution traces to support the program comprehension process. In *Proceedings IEEE European Conference on Software Maintenance and Reengineering (CSMR 2005)*, pages 134–142, Los Alamitos CA, 2005. IEEE Computer Society Press.

[22] 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.

[23] A. Zeller and D. Lütkehaus. DDD – a free graphical front-end for unix debuggers. *SIGPLAN Not.*, 31(1):22–27, 1996.