Capturing How Objects Flow at Runtime*

Adrian Lienhard [1], Stéphane Ducasse[2], Tudor Gîrba[1] and Oscar Nierstrasz[1]

[1] Software Composition Group, University of Bern, Switzerland
[2] LISTIC, Université de Savoie, France

Abstract

Most of today's dynamic analysis approaches are based on method traces. However, in the case of object-orientation understanding program execution by analyzing method traces is complicated because the behavior of a program depends on the sharing and the transfer of object references (aliasing). We argue that trace-based dynamic analysis is at a too low level of abstraction for object-oriented systems. We propose a new approach that captures the life cycle of objects by explicitly taking into account object aliasing and how aliases propagate during the execution of the program. In this paper, we present in detail our new meta-model and discuss future tracks opened by it.

1 Introduction

Understanding an object-oriented system is not easy if one relies only on static source code inspection [19]. Inheritance, and late-binding in particular, make a system hard to understand. The dynamic semantics of *self* (or *this*) produces yo-yo effects when following sequences of method calls [18]. Moreover, the method lookup depends on the receiver which in turn varies depending on the transfer of object references at runtime.

Dynamic analysis covers a number of techniques for analyzing information gathered while running the program [2, 17]. Dynamic analysis was first used for procedural programs for applications such as debuggers [5] or program analysis tools [15].

As object-oriented technology became more widespread, it was only natural that procedural analysis techniques were adapted to object-oriented languages. In this context many dynamic analysis techniques focus on only the execution trace as a sequence of message sends [11, 20, 1]. However, such approaches do not treat the characteristics of object-oriented models explicitly.

Although dynamic analysis has the potential to overcome limitations of static source code inspection, it is not without its own limits. We identify the characteristic of *nonlocal effects* in object-orientation which renders program comprehension difficult and motivates a need for a dynamic analysis at the level of objects.

Nonlocal effects are possible due to *object aliasing*, which occurs when more than one reference to an object exists at the same time [10]. Objects often are not short-lived but are passed as arguments or return values and hence get aliased (or referenced) by multiple other objects. In this way, object aliasing introduces nonlocal effects: an object can be visible from different locations of the program at the same time and hence, through side-effects, a message sent to the object in one context can influence the behavior of the object in another context.

These effects are hard to understand from method traces alone because object aliasing and the transfer of aliases are not explicit. Another area in which object aliasing complicates the understanding of program execution is debugging. Although the debugger shows the current execution stack in which the error occurred, it is often hard to find the actual cause of the error because object references may have been incorrectly set at some distant time in the past.

The main contribution of this paper is a novel meta-model of object-oriented program execution which explicitly represents object aliasing, the transfer of aliases and the evolution of object state. We believe that such a model opens new ways of understanding the dynamics of object-oriented systems. To illustrate our approach we present three works in progress that are applications of our model

Outline. Section 2 discusses the object flow meta-model. Next we describe in Section 3 three example applications of our model. In Section 4 we describe the implementation of our prototype for obtaining the object flow information. Section 5 reports on the state-of-the-art and Section 6 presents the conclusions.

2 Representing the runtime space

We propose a novel model to capture how objects circulate or flow through a software system. Our model is intended to express the fact that an instance of a class can be

*Proceedings of the 2nd International Workshop on Program Comprehension through Dynamic Analysis (PCODA 2006), pp. 39–43

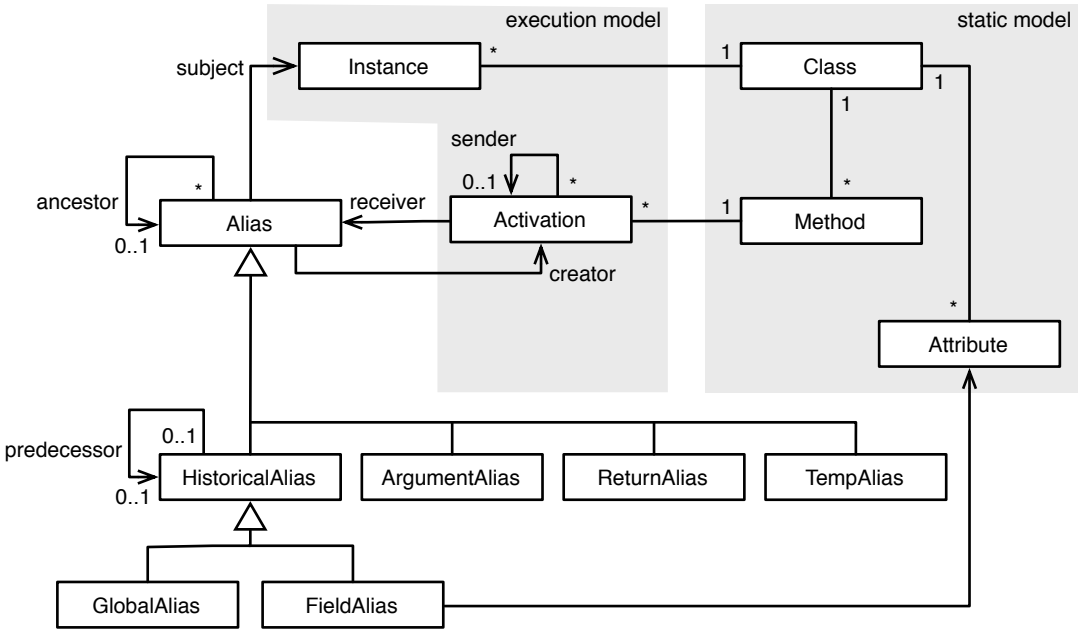

Figure 1. The object flow meta-model

referenced from several places at once, and that those references can be passed around to create other references. To capture this, we put at the center of our model the *alias*, as an explicit reification of an object reference.

Figure 1 shows the class diagram of our model. To get a complete picture of the system, we model the static part (*e.g.*, classes, methods, attributes), the execution part (*i.e.*, instances and method activations) and we complement these parts with aliases. In the following we detail the execution model and its integration with aliases.

Activation. An activation in our model represents a method execution. It holds the *sender* activation and the *method* it executes. This is very similar to commonly-used dynamic analysis approaches. Depending on the usage, approaches additionally identify the receiver instance for which a method is executed.

In our model, however, the *receiver* of an activation is the *alias* through which the message is sent. The fact that an activation does not directly reference an instance but rather an alias of this instance is an important property of our model.

Alias. We define an *alias* to be a first-class entity identifying a specific reference to an object in the analyzed program. An alias is created when:

- an object is instantiated,
- an object is stored in a field,
- an object is stored in a local variable,
- an object is passed as argument, and
- an object is returned from a method execution.

Except for the very first alias which stems from the object instantiation primitive, an alias can only be created from a previously existing alias, its *ancestor*. Based on this relationship we can construct the flow of objects. The flow shows where the instance is created and how it is then passed through the system. Since several new aliases can be created for each alias, the aliases of an instance form a tree.

Each alias is bound to its *creator*, a method activation. By *creator* we understand the activation in which the alias is first visible. For example, when passing an object as argument, the argument alias is created in the activation which handles the message received (rather than the activation in which the message was sent). The same holds for return values: the alias of a return value is created in the activation to which the object is returned.

The rationale is that aliases should belong to the activation in which an object becomes visible. Aliases of arguments, return values and temporary variables are only visible in the activation where they are created whereas field aliases may be accessed in other activations as well.

Special kinds of aliases include field and global aliases, as they additionally carry information about the evolution of the program's state. Field and global aliases point via their *predecessor* to the alias which was stored in the field before the assignment. The impact on our model is that it is capable of capturing the full history of the state evolution of objects. The predecessor reference enables backtracking of the state of objects to any point in the past.

3 Applications

We envision that our model has an impact both on reverse engineering and on forward engineering. In this section we describe three application examples that make use of the data captured in our model.

3.1 Relating Object Flows to Static Structure

The most straightforward application is relating the dynamic information to the static information. Figure 2 shows the hierarchy of the Smalltalk Squeak bytecode compiler [16], and on top of it we show how the aliases have traveled through the system at runtime. We emphasize in red the aliases of the particular instance that is the focus of our attention.

We envision several usages of such views. For example with such a view:

- We can check whether the path of the objects is what is expected.
- We can identify which classes play a primary role in the runtime object flow.

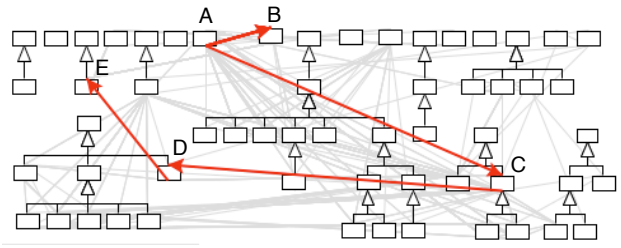

Figure 2. Example of several object flows mapped to a class hierarchy.

3.2 Characterizing Object Flows

Another application is to reason how a certain instance is aliased within the system. We are working on a simple visualization that captures the flow of an object by displaying a tree of aliases.

Figure 3 shows the same instance as in Figure 2, only now we emphasize the different kinds of aliases of this object using distinct colors.

Thus, the initial alias (1) is assigned to the field (2). The following six sequences of yellow and blue aliases (3-4) show that the instance is passed six times as argument to other objects in which it is then stored in a field. If we want to see in which class and method an alias is created,

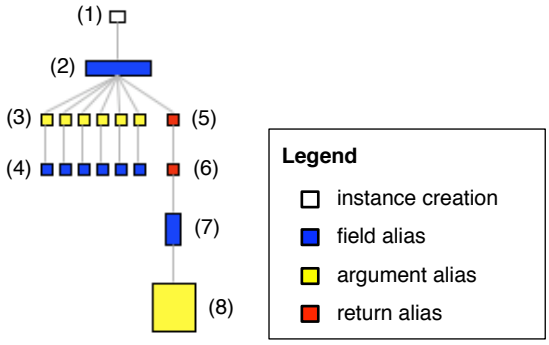

Figure 3. Example of object flow visualization of an instance.

our visualization tool [14] allows us to interactively get this information by moving the mouse over an alias.

The rightmost path shows that the object is passed as return value through (5) and (6) and is then stored in a field (7). Finally the object is passed by argument (8).

We also map metrics to the shape of the boxes in the diagrams [12]. We map the number of messages sent to the alias to the width of a box, and we map the number of messages sent from this alias to other objects to the height. In our example, the field alias (2) is wide which means that many messages have been sent to that alias. On the other hand, the alias (7) has a rather tall shape which means it sent more messages to other object than it received.

This visualization offers useful information regarding how the instance interacted with other objects during its life cycle. From the visualization of Figure 3 we can for example understand the following usage scenario of the instance: in a first stage it is set up, then it is passed around but is never used, and in the last stage, the object is used and itself interacts with other objects.

3.3 Object-centric Debugging

Another application area is debugging. In object-oriented programming, the understanding of problems is often complicated due to the *temporal and spatial gap* between the root cause and the effect of errors.

Figure 4 illustrates an example execution trace of a program. While the cause of the bug is introduced at the beginning of the execution, the effect occurs later (temporal gap). Figure 4 also shows the execution stack at the point when the error occurred. This is the typical view of a debugger showing the method activation in which the bug is manifested. The location of the cause of the bug, however, is hidden because objects have been passed around during execution (spatial gap).

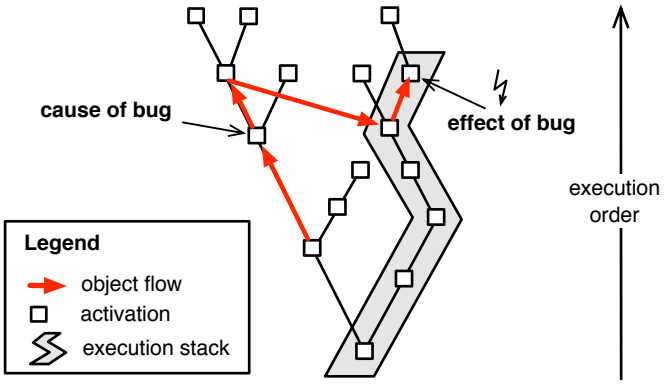

Figure 4. Example of how the cause of a bug can be outside the current stack.

By means of fields the flow of an object can bridge the linear sequence of method executions. With red we illustrate the flow of an object relative to the same program execution. While the object is first passed along with the execution trace, its path later diverges and jumps to distant branches of the tree.

This example illustrates how changes to the software system which modify the behavior of objects may have unexpected effects at distant locations in the program execution. Therefore, to connect the cause and the effect of errors we need to trace the flow of objects. This will support the developer in finding errors by allowing him to follow incorrectly behaving objects back along their path. We call this approach object-centric debugging.

4 Implementation

We have implemented the model extractor in Squeak, a Smalltalk dialect. Since instrumenting method activation and field access would not allow us to trace the flow of objects precisely enough, we also keep track of aliases at runtime. That is, during the program execution we actually instantiate for each reference an alias object. The alias objects then act as proxies which trap message sends and forward them to the object.

The instrumentation of the target program happens in two phases. In the first phase the relevant part of the class hierarchy is replicated to facilitate scoping the effects of the instrumentation (*i.e.*, the classes are copied and the class hierarchy is reconstructed). This is necessary because we also want to instrument core libraries such as the class *Array* which is used by other parts of the system.

In the second phase the classes are instrumented by annotating the abstract syntax tree (AST) of the methods. This means that our approach does not rely on source code modification but rather operates on a more abstract level. The instrumentations modify for example field assignment. In this case the assignment is replaced with bytecode which instantiates a new field alias, sets its ancestor (and predecessor reference if appropriate) and eventually stores it in the actual field.

The performance overhead of the current prototype implementation is around a factor of 10. However, we have not yet done any performance optimization, and we expect to improve the performance in the future. We plan to push aliases down one level into the VM. The responsibility of aliases (capturing a specific reference to an object) can be implemented at this level much more efficiently. Instead of instantiating new alias objects, the indirection can be achieved by using a table which maps object pointers.

5 Related work

Dynamic analysis covers a number of techniques for analyzing information gathered while running the program [2, 6]. Many techniques focus on analyzing the program as a sequence of method executions [11, 20].

To better understand object-oriented program behavior various approaches have extended method traces. As an example, Gschwind *et al.* illustrate object interactions taking arguments into account [7] and De Pauw *et al.* exploit visualization techniques to present instance creation events [3].

These approaches extend method traces with some additional information. In contrast, our approach is much more radical as it proposes a new model which is centered around objects, capturing object aliasing, a key characteristic of object-orientation.

Most approaches of dynamic analysis in the context of object-orientation primarily analyze the program's execution *behavior* rather than the *structure* of its object relationships. An exception is Super-Jinsight, which visualizes object reference patterns to detect memory leaks [4], and the visualizations of ownership-trees proposed by Hill *et al.* to show the encapsulation structure of objects [8].

Those two approaches are based on snapshots whereas our model has an explicit notion of the *evolution* of objects: on one hand the *object flow*, and on the other the *object history*. Another practical advantage and difference to the two approaches mentioned above is that we do not only show how objects are referenced and how references evolve, but that we also combine this information with method traces.

Backward-in-time debuggers [9, 13] allow one to navigate back in the history program execution. Those debuggers capture the full execution and object history and like this allow the user to inspect any intermediate state of the program. Although some navigation facilities are provided,

the notion of how objects flow through the system is missing because the event-based tracing approaches do not provide complete information about the flows.

6 Conclusions

Dynamic information contains valuable information about how the systems works at runtime. Most of the approaches to analyze dynamic information use a trace-based view. However, in the case of object-oriented programs, the trace needs to be complemented with a view of how objects are referenced and passed around in the system.

In this paper, we present a novel approach in which we capture object references and explicitly model them as first class entities (*i.e.*, aliases). In our model we distinguish between several types of aliases and we build a meta-model that incorporates static information, trace information, and object flow information.

We have chosen to build our prototype implementation in Squeak because of the flexible nature of Squeak's environment. Until now, we have performed several case studies to test the scalability. Even though we witness a factor of 10 of slowdown, we are optimistic to improve the performance by adding support for aliases to the VM.

We foresee that this model opens new research tracks both from a reverse engineering perspective and from a forward engineering perspective. We have listed three examples of our work in progress, namely: (1) relating object flow to static structure, (2) characterizing objects based on the objects flows, and (3) object-centric debugging.

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] G. Antoniol and Y.-G. Guéhéneuc. Feature identification: a novel approach and a case study. In *Proceedings of ICSM 2005 (21th International Conference on Software Maintenance 2005*. IEEE Computer Society Press, Sept. 2005.

[2] T. Ball. The Concept of Dynamic Analysis. In *Proceedings of ESEC/FSE '99 (7th European Software Engineering Conference and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, number 1687 in LNCS, pages 216–234, sep 1999.

[3] W. De Pauw, D. Kimelman, and J. Vlissides. Modeling object-oriented program execution. In M. Tokoro and R. Pareschi, editors, *Proceedings ECOOP '94*, LNCS 821, pages 163–182, Bologna, Italy, July 1994. Springer-Verlag.

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

[6] O. Greevy and S. Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *Proceedings of CSMR 2005 (9th European Conference on Software Maintenance and Reengineering*, pages 314–323. IEEE Computer Society Press, 2005.

[7] T. Gschwind and J. Oberleitner. Improving dynamic data analysis with aspect-oriented programming. In *Proceedings of CSMR 2003*. IEEE Press, 2003.

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

[9] C. Hofer, M. Denker, and S. Ducasse. Design and implementation of a backward-in-time debugger. In *Proceedings of NODE'06*, 2006.

[10] J. Hogg, D. Lea, A. Wills, D. deChampeaux, and R. Holt. The Geneva convention on the treatment of object aliasing. *SIGPLAN OOPS Mess.*, 3(2):11–16, 1992.

[11] M. F. Kleyn and P. C. Gingrich. GraphTrace — understanding object-oriented systems using concurrently animated views. In *Proceedings OOPSLA '88 (International Conference on Object-Oriented Programming Systems, Languages, and Applications*, volume 23, pages 191–205. ACM Press, Nov. 1988.

[12] M. Lanza and S. Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, Sept. 2003.

[13] B. Lewis. Debugging backwards in time. In *Proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG 2003)*, Oct. 2003.

[14] M. Meyer, T. Gîrba, and M. Lungu. Mondrian: An agile visualization framework. In *ACM Symposium on Software Visualization (SoftVis 2006)*, 2006. To appear.

[15] H. Ritch and H. M. Sneed. Reverse engineering programs via dynamic analysis. In *Proceedings of WCRE '93*, pages 192–201. IEEE, May 1993.

[16] Squeak home page. http://www.squeak.org/.

[17] T. Systä. Understanding the behavior of Java programs. In *Proceedings WCRE '00, (International Working Conference in Reverse Engineering)*, pages 214–223. IEEE Computer Society Press, Nov. 2000.

[18] D. Taenzer, M. Ganti, and S. Podar. Problems in object-oriented software reuse. In S. Cook, editor, *Proceedings ECOOP '89*, pages 25–38, Nottingham, July 1989. Cambridge University Press.

[19] N. Wilde and R. Huitt. Maintenance Support for Object-Oriented Programs. *IEEE Transactions on Software Engineering*, SE-18(12):1038–1044, Dec. 1992.

[20] A. Zaidman, T. Calders, S. Demeyer, and J. Paredaens. Applying webmining techniques to execution traces to support the program comprehension process. In *Proceedings of CSMR 2005 (9th European Conference on Software Maintenance and Reengineering)*. IEEE Computer Society Press, 2005.