Object Flow Analysis — Taking an Object-Centric View on Dynamic Analysis*

Adrian Lienhard[1], Stéphane Ducasse[2], Tudor Gîrba[1]

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

Abstract. To extract abstract views of the behavior of an object-oriented system for reverse engineering, a body of research exists that analyzes a system's runtime execution. Those approaches primarily analyze the control flow by tracing method execution events. However, they do not capture information flows. We address this problem by proposing a novel dynamic analysis technique named Object Flow Analysis, which complements method execution tracing with an accurate analysis of the runtime flow of objects. To exemplify the usefulness of our analysis we present a visual approach that allows a system engineer to study classes and components in terms of how they exchange objects at runtime. We illustrate and validate our approach on two case studies.

1 Introduction

A large body of research exists for supporting the reverse engineering process of legacy systems. However, especially in the case of dynamic object-oriented programming languages, statically analyzing the source code can be difficult. Dynamic binding, polymorphism, and especially behavioral and structural reflective capabilities pose limitations to static analysis.

Many approaches tackle these problems by investigating the dynamic information collected from system runs [6,16,21,1]. Most proposed approaches are based on execution traces, which typically are viewed as UML sequence diagrams or as a tree structure representing the sequence and nesting of method executions [15,10,1,9]. Such views analyse message passing to reveal the control flow in a system or the communication between objects or between classes [25,5].

Various approaches extend method execution tracing with object information to improve object-oriented program understanding. For example, they trace instantiation events to analyze where objects are created [5,24,1,8]. Other approaches analyze object reference graphs to make object encapsulation explicit [12] or to find memory leaks [7].

While data flows have been widely studied in static analysis [13], none of the above mentioned dynamic analysis approaches captures the runtime *transfer* of object references. In this paper we present *Object Flow Analysis*, a novel

* Proceedings of International Conference on Dynamic Languages (ICDL'07), ACM Digital Library, 2007, pp. 121–140, DOI 10.1145/1352678.1352686

dynamic analysis approach, which captures the complete and continuous path of objects. We propose a meta-model that explicitly captures the transfer of object references.

To exemplify the usefulness of our analysis, we propose the following application for facilitating program comprehension of object-oriented legacy systems. A difficulty with studying the control flow of a program, *e.g.*, using an UML sequence diagram, is that the propagation of objects is hard to understand. For example, it is often not obvious how objects created in one class or package propagate to others. Also, inspecting how objects refer to each other, *e.g.*, using object inspectors, does not reveal how the object reference graph evolved. Our goal is to analyze how the objects are passed at runtime to facilitate understanding the architecture of a system. We want to address the following explicit questions that arose from our experience maintaining large industrial applications written in dynamic languages:

- Which classes exchange objects?
- Which classes act as object hubs?
- Given a class, which objects are passed to or from it?
- Which objects get stored in a class, and which objects just pass through it?
- Which continuous object flows spanning multiple classes exist?

To address the questions, we present a prototype tool that is based on Object Flow Analysis. It provides visualizations to facilitate exploring the results of our analysis. We implemented the tracing infrastructure in Squeak[1], an open source Smalltalk dialect, and the meta-model in Visual Works Smalltalk using Moose/Mondrian [19,18].

Outline. In the next section we emphasize the need for complementing execution traces with object flow information to provide a more exhaustive fundament for object-oriented program analysis. Section 3 is dedicated to discuss Object Flow Analysis, our novel dynamic analysis technique that tracks how objects are passed through the system. Section 4 presents our approach to exploit object flows for studying the behavior of object-oriented programs at the architectural level. Section 5 evaluates our visual approach based on two case studies, and Section 6 provides a discussion. Section 7 reports on the state-of-the-art and Section 8 presents the conclusions.

2 Challenges Understanding Object Propagation

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. However, also with today's dynamic analysis approaches, understanding object flows is hard.

[1] See http://www.squeak.org/

The predominantly captured data of dynamic object-oriented program behavior are execution traces. An execution trace can be represented as a method call-tree. Figure 1 illustrates a small excerpt of such a tree from one of our case studies (a Smalltalk bytecode compiler). It shows as nodes the class name of the receiver and the method that was executed.

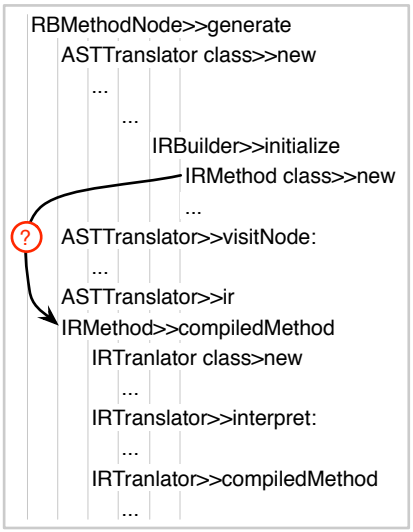

Fig. 1. Excerpt of an execution tree.

A limitation of execution traces is that they typically provide little information about the actual objects involved. In Figure 1 we see that an IRMethod instance is created in IRBuilder. Later an IRMethod instance is sent the message compiledMethod in RBMethodNode. Some of the execution trace approaches record the identity of the receiver object of a method execution. With this information we can reveal that the IRMethod instance created is the identical one in both places of the trace.

However, the execution trace cannot answer how this instance was passed there. The instance could be passed from IRBuilder to RBMethodNode via a sequence of method return values through other classes, but it could as well be stored in a field and then be accessed directly later on.

Speculating about the answer is hampered even more because execution traces are usually very large. Figure 1 only shows the first five levels and ten method executions. In our case, the area of the tree hidden by the dots, though, is 46 levels deep and comprises 4793 method executions.

Gschwind *et al.* propose a dynamic analysis that captures also the objects passed as arguments [11]. They argue that this information was essential to gain

a deeper insight into the program execution. Walker *et al.* visualize the operation of a system at the architectural level and note in their discussion that "it may be useful to capture the migration of objects" [24].

Object passing and sharing complicates program comprehension because it can introduce complex object interrelationships. Nevertheless, object passing, *i.e.*, the transfer of object references, is an essential feature of object-orientation. A large body of research has been conducted into controlling object aliasing at the type system level to provide a strong notion of object encapsulation [14,20]. However, such advanced typed systems are still in the realm of advanced research.

Our goal is to analyze how the objects are passed at runtime to support program comprehension of object-oriented legacy systems. This analysis is especially interesting for the objects that are not encapsulated, *i.e.*, objects that are aliased and that are passed around. At the heart of our analysis are the following two questions:

- In which method executions was an object made visible through a reference?
- Where did a specific object reference come from?

The first question reveals all locations, *i.e.*, arguments, return values, instance and global variables, the object is passed through (including those in which it does not receive messages). The second question reveals the path of the object, that is, how it is passed from one to another location, starting from where it is instantiated.

In the next section we introduce the concept of Object Flow Analysis and present its meta-model. Based on the captured information we can then precisely answer how the IRMethod instance in Figure 1 is passed through the system.

3 Object Flow Analysis

In this section we present Object Flow Analysis, our approach to track how objects are passed through the system at runtime. This technique is based on an explicit model of object reference and method execution.

3.1 Representing Object References

The key idea we propose to extract such runtime information is to record each situation in which an object is made visible in a method execution through an object reference. We represent in our meta-model each such situation by a so-called *Alias* (see Figure 2).

In our program execution representation, an object alias is created when an object is (1) instantiated, (2) stored in a field (*i.e.*, instance variable) or global, (3) read from a field or global, (4) stored in a local variable, (5) passed as argument, or (6) returned from a method execution.

The rationale is that each object alias is bound to exactly one method execution (referred to as *Activation* in our meta-model), namely the method execution

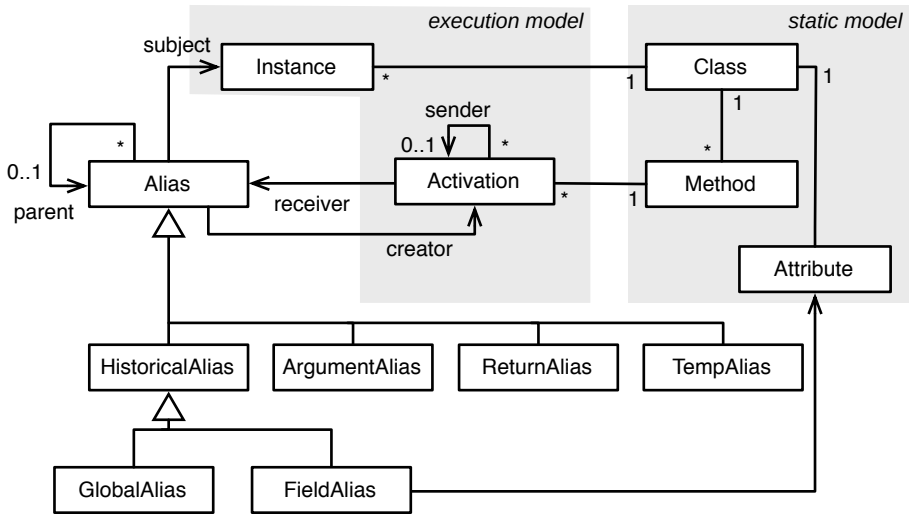

Fig. 2. The Object Flow meta-model extends the static and execution meta-models with the notion of Alias.

in which the alias makes the object visible. By definition, arguments, return values, and local variables are only visible in one method activation. In contrast, objects that are stored in fields (*i.e.*, instance variables) or globals, can be accessed in other activations as well. Therefore, we distinguish between *read* and *write* access of fields and globals.

With the record of all aliases of an object and their relationships to method activations, we can now determine where the object was visible during program execution. The other key information from which we can extract the object flow resides in the relationships among the aliases.

Apart from the very first alias, which stems from the object instantiation primitive, all aliases are created from a previously existing one. This gives rise to a *parent-child* relationship between aliases originating from the same object. With this relationship we can organize the aliases of an object as a tree where the root is the alias created by the instantiation primitive. This tree represents the object flow, *i.e.*, it tells us how the object is passed through the system.

3.2 Control Flow vs. Object Flow Perspective

To illustrate and discuss details of the object flow construction we introduce a concrete example taken from one of our case study applications, namely the Smalltalk bytecode compiler (see Section 5 for more details). The example used in this section shows the interplay of important classes of the last two compiling phases (translating the abstract syntax tree (AST) to the intermediate representation (IR), and translating the IR to bytecode).

The center of interest is the instance of the class IRMethod. It acts as a container of IRSequence instances, which group instructions and form a control graph. We now take two complementary perspectives to study the program behavior in which the IRMethod instance is used. The first perspective emphasizes the control flow, that is, the sequence and nesting of the executed methods. The second perspective, illustrates the one we gain from studying object flows.

Control flow perspective. The execution trace in Figure 1 illustrates the order and nesting of the executed methods through which the IRMethod instance is passed. RBMethodNode≫generate is the first executed method. On the left side of Figure 3 we see its source code. It creates an instance of ASTTranslator, which in turn instantiates and stores an instance of IRBuilder in a field. IRBuilder in its constructor method initialize instantiates an IRMethod and stores it in the field named ir.

After this, the control is returned to RBMethodNode which then sends to ASTTranslator the message visitNode:. ASTTranslator is a visitor which traverses the AST and delegates the building of the IR to the IRBuilder instance. In the process IRBuilder creates several new sequences.

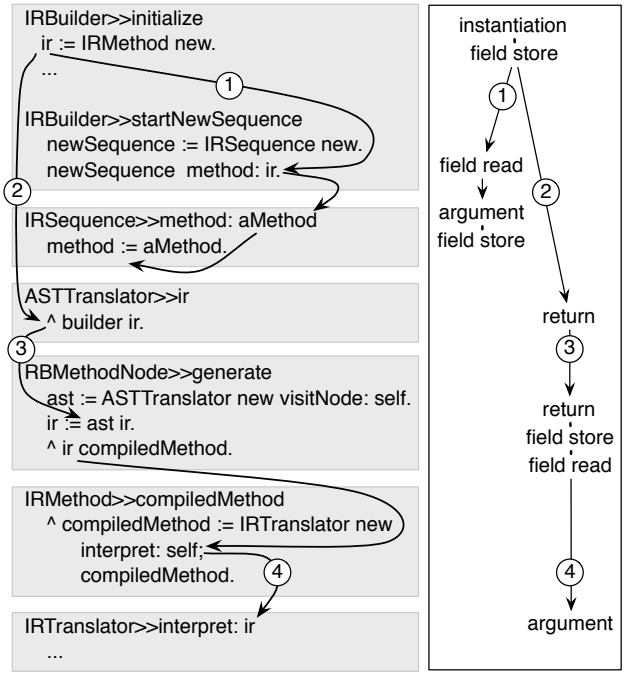

Fig. 3. Object flow of an IRMethod.

When the IR is built, the IRMethod object is obtained by sending ir to the ASTTranslator which indirectly gets it from the IRBuilder instance. The execution now continues by sending compiledMethod to the IRMethod instance which eventually generates bytecode.

Object flow perspective. In this perspective we take the point of view of how the IRMethod instance flows through the system. The methods on the left side of Figure 3 are ordered by when the instance is passed through them (rather than by the order of the control flow). The right side of Figure 3 illustrates the corresponding flow as a tree. Nodes represent aliases and edges are created depending on the parent-child relationship of the aliases.

This tree represents the object flow of the IRMethod instance. The flow starts with the root alias *instantiation* in the method IRBuilder≫initialize where the IRMethod instance is created. The object is then directly assigned to a field named ir (represented as a field store alias).

During the lifetime of IRBuilder the object is read from the field (1) and then passed as argument to IRSequence objects where it is stored in a field called method. Notice that in the actual execution the branch starting with (1) happens multiple times, but Figure 3 only shows one for conciseness.

When RBMethodNode≫generate requests the IRMethod instance, the object is first returned from the IRBuilder to the ASTTranslator (2) (this happens through a getter not shown here). Only then it is returned to RBMethodNode (3). This last return alias directly gets stored into the field ir of RBMethodNode.

A special case of aliasing is when an object passes itself as argument. In our code example this happens when the method compiledMethod of IRMethod is executed through the field read alias in RBMethodNode. The object instantiates an IRTranslator and passes itself to it as argument (see bottom of Figure 3). The argument alias which is created in IRTranslator has as parent alias the field read alias, that is, the alias which was used to activate the object that passed itself. This property of our model assures that the object flows are continuous.

In the next section, to illustrate the usefulness of Object Flow Analysis, we present a visual approach to answer the reverse engineering questions formulated in Section 1.

4 Visualizing Object Flows between Classes

Based on our meta-model we can analyse the transfer of object references between classes to answer questions such as:

- Which classes exchange objects?
- Which classes act as object hubs?
- Given a class, which objects are passed to or from it?
- Which objects get stored in a class, and which objects just pass through it?
- Which continuous object flows spanning multiple classes exist?

We propose two explorative and complementary views to address the above questions:

- The *Inter-unit Flow View* depicts units connected by directed arcs subsuming all objects transferred between two units (Section 4.1). By unit we understand a class, or a group of classes that a software engineer knows they conceptually belong together (*e.g.*, all classes in a package, in a component, or in an application layer like the domain model or the user interface).
- The *Transit Flow View* allows a user to drill down into a unit to identify details of the actual objects and of the sequence of their passage (Section 4.2).

4.1 Inter-unit Flow View

Figure 4 shows an Inter-unit Flow View produced on our compiler case study. The nodes represent units (*i.e.* either individual classes or groups of classes), and the directed arcs represent the flows between them. The thickness of an arc is proportional to the number of unique objects passed along it.

A force based layout algorithm is applied (nevertheless, the user can drag nodes as she wishes). This layout results in a spatial proximity of classes and units that exchange objects. This supports a software engineer in building a mental model and systematically exploring the software architecture.

Constructing the visualization. The Object Flow model shows how objects are passed between other objects. As the goal of our visualization is to show how objects are passed through *classes*, we aggregate the flow at the level of classes and groups of classes (units). In our prototype units are stated by the system engineer using a declarative mapping language (similar to the approach of Walker *et al.* [24]). Rules are provided to map classes to units based on different properties such as the package they are contained in, their inheritance relationship, or a pattern matching their names. For instance, the first rule below maps all classes in the AST-Nodes package to the unit AST. The second rule maps IRInstruction and all classes inheriting from it to the unit IR.

```
classes containedInPackage: 'AST-Nodes' mapTo: 'AST'
classes hierarchyRootedIn: 'IRInstruction' mapTo: 'IR'
```

For the proposed visualization we do not take into account (i) through which instances of a class objects are passed, and (ii) the flow of objects that are only used within one class. Another important property is that we treat the flows through collections transparently. This means that when an object is passed from one class to a collection, and later from the collection to another class, the intermediate step through the collection is omitted in the visualization. The flow directly goes from one to the other class and there is no node created for the collection class. This abstraction makes the visualization much more concise and emphasises the conceptual flows between application classes.

Example. Let's consider again Figure 4, which shows the Inter-unit Flow View of the Smalltalk bytecode compiler case study. Various classes are aggregated to units, displayed with the number of contained classes in brackets. For instance,

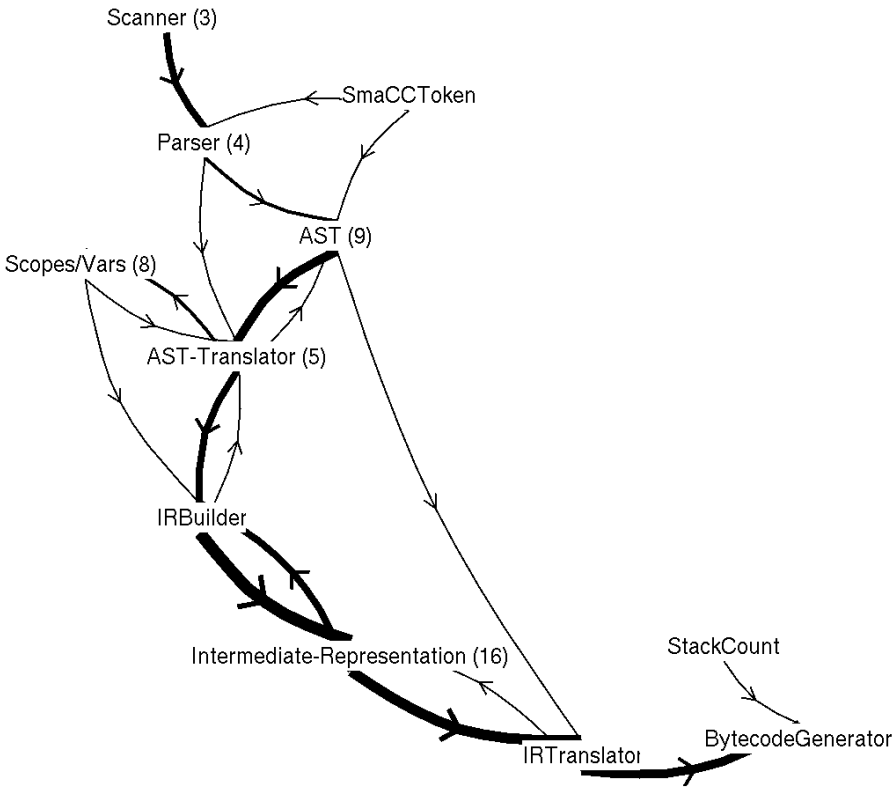

Fig. 4. Inter-unit Flow View of the bytecode compiler.

the group AST (9) contains the nine classes representing the abstract syntax tree.

The visualization shows which classes exchange objects. For example, there are many objects passed from the Scanner to the Parser or from Intermediate-Representation to IRTranslator. On the other hand, we also see which classes are distant in that objects can only flow between them via several other classes.

Considering the thick arcs, we can detect a propagation of objects from Scanner (top) to BytecodeGenerator (bottom-right) traversing the Parser (top). This corresponds to the conceptual steps of a compiler.

An interesting exceptional flow is the one from AST to IRTranslator. It contains exactly one object, the IRMethod instance we encountered in the previous examples. As we can find out with the features introduced below, the object flow starts at IRBuilder and passes via AST-Translator. This corresponds to the object flow illustrated in Figure 3.

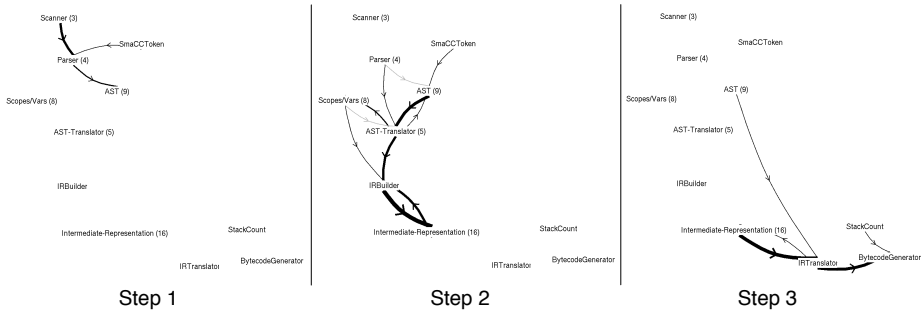

Fig. 5. Chronological propagation of flows in the compiler.

The chronological propagation of objects. The Inter-unit Flow View shows an overview of the entire execution. However, as not all objects are passed around at the same time, we are also interested in the chronological order to identify the phases of a system's execution. For example, in a program with an UI the phases may be related directly to the exercised features.

With our prototype implementation the user can scope the visualized object flow information to a specific time periode by using a slider representing the timeline. The position of the slider defines up until which point in time object flows are taken into account. A recently active arc is displayed in dark gray which then fades and eventually becomes invisible. The goal of this feature is to facilitate investigation of how objects are propagated during a program execution.

Figure 5 illustrates three snapshots in the evolution of the compiler execution (compare with Figure 4). In the first step we see that objects are passed from Scanner to Parser and from Parser to AST. In the second step, many objects are passed between AST and AST-Translator, IRBuilder and Intermediate-Representation. In the third step, many objects pass from Intermediate-Representation to Bytecode-Generator.

Highlighting spanning flows. With the aforementioned features we can see which units directly exchange objects and when. However, we cannot see if there exist objects that are passed from one unit to another *indirectly*, *i.e.*, spanning intermediate units.

This information is useful to understand which units act as steps in object flows leading to a unit. The same holds for the objects passed outside a unit where it is interesting to know to which other units the objects are forwarded and which paths are taken.

In our prototype the user can select a unit. Thereafter, all arcs that contain objects being passed to the selected unit are highlighted in orange and all arcs with objects passed from the selected entity are highlighted in blue.

Figure 6 shows twice the same visualization (compare with Figure 4) but with different classes selected. In Figure 6.A Parser is selected. We see that objects are passed to it directly from Scanner (orange arc). On the other hand, the objects it passes outside reach many different units, the longest path reaches the Intermediate-Representation unit (blue arcs). In Figure 6.B IRBuilder is selected. We see that it obtains objects from most above units and forwards objects to almost all units below.

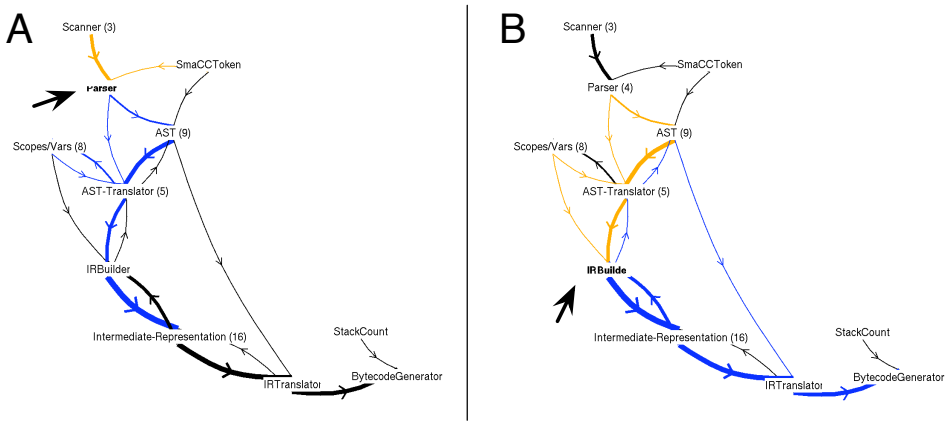

Fig. 6. Orange and blue arcs indicate flows leading to and coming from selected unit Parser (A), resp. unit IRBuilder (B).

This view highlights from where objects are passed to a unit and which routes are taken. This tells us, for example, how dependent a unit is on other units, *e.g.*, IRBuilder depends on objects created by or passed through all upper classes except for Scanner and SmaCCToken. The highlighted outgoing flows, on the other hand, tell us how influential a class is.

4.2 Transit Flow View

The aforementioned visualization lacks information about the actual objects being passed through a unit. To facilitate investigating this information our prototype allows the user to drill down to access detailed information about the objects transiting a unit.

Figure 7 illustrates the Transit Flow View for the class IRBuilder. It lists from top to bottom all instances that transit IRBuilder grouped by their class. The objects inside a class are grouped by their arrival time. For each instance the point in time when it was passed into or out of the class is indicated with a rectangle. An orange rectangle shows that the object is passed *in*; a blue

rectangle that it is passed *out*. A line is displayed during the time when the object is stored in a field (or contained in a collection that is stored in a field).

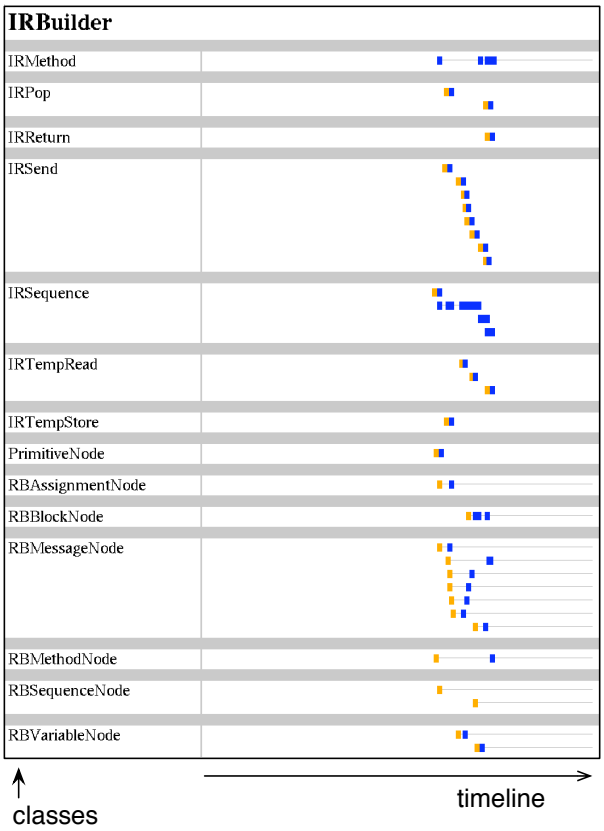

Fig. 7. IRBuilder Transit Flow View.

The Transit Flow View shows when flows take place and how many instances of which class are involved. Further exploration reveals: (1) objects passed through directly (orange/blue pairs without line), (2) objects stored in fields or collections (line), (3) objects created (the first rectangle is not orange, therefore, the object is created in the class), and (4) objects passed in or out multiple times (several rectangles for the same object).

For example, in Figure 7, the IRMethod instance is created in IRBuilder, it is stored in a field, and it is passed out multiple times. Intermediate representation instances are created in IRBuilder but are not stored in it, whereas the instances at the bottom (AST nodes) are passed from outside and are stored.

5 Case Studies

In this section we provide an overview of the results we obtained from applying the visualizations on two case studies: a Smalltalk bytecode compiler and a health insurance web application. Both are implemented in Squeak, an open-source Smalltalk dialect. Our choice of those two case studies was motivated by the following reasons: (1) they are both non-trivial and model a very different domain, (2) we have access to the source code, and (3) we have direct access to developer knowledge to verify our findings.

The table below shows the static dimensions of the code :

	Compiler	Insurance App.
Classes	127	308
Methods	1'912	4'432
Lines of code	11'208	40'917

The objective of these preliminary investigations is to evaluate the usefulness of the two visualizations and to learn about a practical exploration process using our tool.

5.1 Bytecode Compiler

We chose the Smalltalk bytecode compiler as a case study because we wanted to understand its underlying mechanism to use it as basis for the future implementation of our Object Flow Analysis infrastructure. The compiler is a complex program, yet, its domain is well known.

To generate experimental data we run the compiler on a typical method source code which includes class instantiations, local variable usage, a conditional and a return statement.

The Inter-unit Flow View illustrated by Figure 4 shows the final state of the view after several iterations of exploring and refining the mappings of units. We describe the exploration process of our tool which crystallized from the two case studies in the next section.

Using the Inter-unit Flow View (see Figure 5) we could correctly, *i.e.*, in line with the documentation, extract the key phases of the compiler: (1) scanning and parsing, (2) translating AST to the intermediate representation, and (3) translating the intermediate representation to bytecode.

With the help of the highlighting feature we obtained more detailed knowledge about the system. For example, IRBuilder plays a key role as it is a hub through which objects from the upper units in the view are passed to the lower ones. Using the Transit Flow View (see Figure 7) we studied detailed inter-relationships between the units. For example, we could see where exactly the

transitions between the three different representations happen. For the transition from AST to IR (phase 2) we see that the unit AST-Translator passes all AST nodes to IRBuilder, which in turn creates the intermediate representation objects.

In the the remaining part of this section we want to shed light on an interesting aspect of our approach we noticed in this case study.

Inversion of execution flow. The object flows do not necessarily evolve in the same direction as the execution flow. For instance, the Parser creates the Scanner and then regularly accesses it to get the next token. An analysis of the execution trace shows the call relation Parser → Scanner. The object flow view, on the other hand, shows the conceptually more meaningful order Scanner → Parser. The reason is that with Object Flow Analysis we can provide object-centric views, which abstract implementation details, *e.g.*, the distinction of sender and receiver of a message. This trait also distinguishes our approach from the ones that are based call graphs in which edges point from the sender to the receiver class of a method execution [25,5].

There are two ways how objects are passed to an instance: (i) objects are pushed to an object, *i.e.*, passed to the instance as method arguments, or (ii) objects are pulled by the instance, *i.e.*, received as return value in response to a message send. In the latter case (ii) the objects flow in the opposite direction compared to the message sends.

To further illustrate this point, let's consider again the introductory example of the IRMethod instance. In the execution trace excerpt, shown in Figure 1, the first executed method in which the IRMethod instance occurs is RBMethodNode≫generate. Studying this method first, however, leaves us with the question of how the object is set up and how it is passed there – something which is only visible later (or deeper) in the execution tree. In contrast, Object Flow Analysis is capable of providing a more meaningful viewpoint for studying the life cycle of the instance as illustrated by Figure 3.

5.2 Insurance Web Application

This industrial application was put into production six years ago and since that time has undergone various adaptations and extensions. The analysed scenario comprises the oldest and most valuable part for the customer, the process of creating a new offer. It is composed of ten features, including adding persons, specifying entry dates, selecting and configuring products, computing prices, and generating PDFs.

We first report on the exploration process, which we refined in this second case study and then discuss our investigations.

Step 1: Creating coarse-grained units. We started by investigating the Inter-unit Flow View with units corresponding to packages. Although we pictured first coherences among the packages, the view was hard to work with because there were many web-related packages that seemed less interesting for now. Therefore, we put all classes of the web packages into one unit, representing the UI layer.

Step 2: Re-grouping to appropriate units. The resulting view was already more concise. Now focusing on the domain model, we saw many packages corresponding to individual products, each package containing the product classes and associated calculation model classes. We re-grouped the classes into a Products unit and a Calculation Models unit because we wanted to learn about the higher-level concepts rather than how products differ.

This change dramatically improved the view. We obtained only nine units and we could identify interesting flows between them (see Figure 8). For instance, with the help of the Transit Flow View, we could understand how versioning works and how products and calculation models relate to each other. Products pass dates to the package responsible for versioning and in turn calculation models are passed to the products (we used the Transit Flow View to access this information).

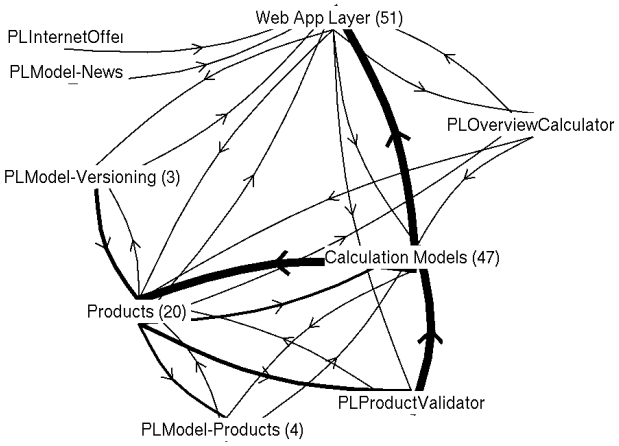

Fig. 8. Inter-unit Flow View of the insurance application case study.

Step 3: Extract interesting candidate classes. Once we gained an overview, we started to dig deeper. We split off packages to show individual classes, *e.g.*, ProductValidator which is packaged in PLModel-Products, but from its name does not seem be a product but rather provides specific behavior. Another similar candidate class is OveviewCalculator.

To obtain more details about those classes we used the highlighting feature to show which other units are involved in passing objects with respect to the selected class. In the case of OverviewCalculator we see that it passes a person and a date to products which forwards date and eventually returns price objects to the calculator.

As soon as we had gained an overview of the domain model, we further explored the UI layer by extracting classes responsible for generating web views.

Discussion. The exploration process we took, which proved useful, was to first gain a coarse-grained view (step 1), find appropriate units (step 2), and only then get into more detail (step 3). Our internal declarative mapping language was helpful to create conceptual groups of classes with varying level of detail. It was essential to be able to specify units that crosscut the package structure.

From the Inter-unit Flow View, conceptual relationships between units were intuitively understandable. The presented information is high-level and thus appropriate for studying an unfamiliar system. Yet, means are provided to drill down to gain more detailed knowledge where appropriate.

In contrast to the compiler case study, the feature for investigating the chronological propagation of objects was not particularly useful. A plausible explanation is that the compiler has a much stronger notion of sequentially transforming one representation to another. The exercised features of the health insurance application, on the other hand, do not exhibit this characteristic.

6 Discussion

The application of Object Flow Analysis proposed in this paper focuses on studying classes or intentional groups of classes referred to as units. Hence, in our information space, classes are the basic parts which represent fixed points on which the object flows are mapped.

As a point of variation, the basic entities could be instances. This would allow one to study the object flows at a much more detailed level ("which objects pass through a particular instance?"). Whether this information is valuable depends on the task at hand. The objective of the approach presented in this paper is to study a system at the architectural level, therefore, we focused on classes.

Other applications of Object Flow Analysis. We believe that Object Flow Analysis can be successfully exploited in many more ways, which yet have to be discovered. Our approach maps object flows to structural entities. For instance, a very different way of looking at object flows is to consider dynamic boundaries. In previous work we described an approach to analyse the flow of objects between *features* to detect feature runtime dependencies [17]. Another promising application of our object flow analysis technique could be to analyze object flows between *threads*, which would reveal how objects are shared between them and how they are transferred.

Scalability. Naturally, the additional information about object flows do not come for free. Namely, a larger amount of data has to be dealt with. We adopt an offline approach, that is, at runtime the tracer gathers aliasing and method execution events. After execution, the data is then fed into the analysis framework on top of which our prototype is implemented.

In a typical program, the number of alias events is higher compared to method execution events. However, as the figures in the table below show, the relative increase is moderate.

	Compiler	Insurance App.
Method executions	11'910	120'569
Aliases	16'033	197'499
Ratio	1.3	1.6

Summarizing, Object Flow Analysis, which gathers both object aliasing and method executions, consumes about 2.5 times the space of conventional execution trace approaches. Our approach deals with the potential large number of events by providing an abstract view at the architectural level. Detailed information about the objects is only shown on demand.

While a factor of 2.5 is not negligible, we believe that the complementary information about the flow of objects justifies the overhead.

Limitations. Considering recall, a noteworthy limitation of our approach is the well-known fact that dynamic analysis is not exhaustive, as not all possible paths of execution are exercised [2]. Therefore, a dynamic analysis always has to be understood in the context of the actual execution.

Like with most other dynamic analysis approaches, scalability may be a limiting factor. The Transit Flow View is most vulnerable because it shows single objects. While in our case studies this view scaled well (the largest one displaying about 100 instances that were passed between two units), it may be cumbersome to study when containing thousands of instances. A solution to this problem may be to even further compact the representation, to apply filters to sort out less interesting objects, or to make selected application classes transparent like collections.

Language independence. To perform object-flow analysis, we need to capture details in the trace that reveal the path of objects through the system. We chose Smalltalk to implement our Object Flow Tracer 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. Particular difficulties are the instrumentation of system classes and the capturing of object reference transfers. On the other hand, the meta-model (both the static and dynamic part) and the visualizations we describe are language independent.

7 Related Work

Dynamic analysis covers a number of techniques for analyzing a program's runtime behavior [2,23,10]. Many techniques focus on analyzing execution traces [15,25,6,22].

Many different approaches exist to make execution traces accessible. For instance, Lange and Yuichi built the Program Explorer to identify design patterns [16]. Pauw *et al.* propose a tool to visually present execution traces to the user [6]. They automatically identify reoccurring execution patterns to detect domain concepts that appear at different locations in the method trace. Scenariographer is a tool which computes groups of similar sequences of method executions to reveal class usage scenarios [22]. While those approaches target understanding a system through the analysis of method executions our approach provides a complementary view based on the the flow of objects. As discussed, an advantage of Object Flow Analysis is that it can reveal more meaningful architectural views because object flows do not depend on implementation details like caller-callee relationships.

Most approaches, including the ones mentioned above, primarily analyze the program's execution *behavior*. Other approaches analyze the *structure* of object relationships. Super-Jinsight visualizes object reference patterns to detect memory leaks [7], and the visualizations of ownership-trees proposed by Hill *et al.* show the encapsulation structure of objects [12]. Those two approaches are based on heap snapshots whereas our approach has an explicit notion of the *evolution* of object references in the form of object flows. With our meta-model we can accurately track continuous flows of the objects which is key to understanding flows spanning multiple classes.

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 [4,3]. 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.

A large body of research has also been conducted into facilitating program comprehension through static analysis. The static analysis of data flows, *e.g.*, based on a points-to analysis, is an active research area. Challenges are precision and cost of the algorithms. The visualizations we present in this paper could potentially also be based on a static data flow analysis instead of a runtime analysis of object flows. The drawback of a static analysis is that it provides a conservative view (which in some cases may even include infeasible execution paths of the program). Our analysis, on the other hand, produces a precise under-approximation. A key advantage is that it allows the user to constrict analysis to the features of interest and thus directly relate them to the obtained knowledge. The fact that the results depend on the program execution is an advantage in this case. Our approach trades off precision for completeness, and therefore we have to anticipate that the results do not apply to all possible program executions.

8 Conclusions

The hallmark of object-oriented applications is the deep collaboration of objects to accomplish a complex task. Understanding such applications is then difficult since reading the classes only reveals the static aspects of the computation. While dynamic analysis approaches offer solutions, they often focus only on the execution of a program from a message passing point of view.

In this paper we identified a missing aspect of dynamic object-oriented program analysis, namely the tracking of how objects are passed through the system. We introduce our approach we named Object Flow Analysis, in which we treat object references as first class entities to track the flows of objects. This approach complements the view on method executions with the view on objects.

To show the usefulness of our approach, we exemplified it with an application in the form of two visualizations: Inter-unit Flow View and Transit Flow View. We used these visualizations to explore the object flows between classes and we applied them on two case studies. These initial experiments showed promising benefits of this new perspective.

We strongly believe that our approach opens a new perspective on dynamic analysis and we intend to further pursue different applications based on it.

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 Cook ANR project "COOK (JC05 42872): Réarchitecturisation des applications industrielles objets".

References

1. Giuliano Antoniol and Yann-Gaël Guéhéneuc. Feature identification: a novel approach and a case study. In *Proceedings IEEE International Conference on Software Maintenance (ICSM'05)*, pages 357–366, Los Alamitos CA, September 2005. IEEE Computer Society Press.
2. Thomas Ball. The concept of dynamic analysis. In *Proceedings European Software Engineering Conference and ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSC 1999)*, number 1687 in LNCS, pages 216–234, Heidelberg, sep 1999. Springer Verlag.
3. Abdulazeez S. Boujarwah, Kassem Saleh, and Jehad Al-Dallal. Dynamic data flow analysis for Java programs. *Information & Software Technology*, 42(11):765–775, 2000.
4. T. Y. Chen and C. K. Low. Dynamic data flow analysis for C++. In *Proceedings of the Second Asia Pacific Software Engineering Conference (APSEC'95)*, page 22, Washington, DC, USA, 1995. IEEE Computer Society.
5. Wim De Pauw, Doug Kimelman, and John Vlissides. Modeling object-oriented program execution. In M. Tokoro and R. Pareschi, editors, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'94)*, LNCS 821, pages 163–182, Bologna, Italy, July 1994. Springer-Verlag.

6. Wim De Pauw, David Lorenz, John Vlissides, and Mark Wegman. Execution patterns in object-oriented visualization. In *Proceedings Conference on Object-Oriented Technologies and Systems (COOTS'98)*, pages 219–234. USENIX, 1998.
7. Wim De Pauw and Gary Sevitsky. Visualizing reference patterns for solving memory leaks in Java. In R. Guerraoui, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'99)*, volume 1628 of *LNCS*, pages 116–134, Lisbon, Portugal, June 1999. Springer-Verlag.
8. Stéphane Ducasse, Michele Lanza, and Roland Bertuli. High-level polymetric views of condensed run-time information. In *Proceedings of 8th European Conference on Software Maintenance and Reengineering (CSMR'04)*, pages 309–318, Los Alamitos CA, 2004. IEEE Computer Society Press.
9. Mohammad El-Ramly, Eleni Stroulia, and Paul Sorenson. Recovering software requirements from system-user interaction traces. In *Proceedings ACM International Conference on Software Engineering and Knowledge Engineering*, pages 447–454, New York NY, 2002. ACM Press.
10. Orla Greevy and Stéphane Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *Proceedings of 9th European Conference on Software Maintenance and Reengineering (CSMR'05)*, pages 314–323, Los Alamitos CA, 2005. IEEE Computer Society.
11. Thomas Gschwind and Johann Oberleitner. Improving dynamic data analysis with aspect-oriented programming. In *Proceedings of the Seventh European Conference on Software Maintenance and Reengineering (CSMR'03)*, page 259, Washington, DC, USA, 2003. IEEE Computer Society.
12. T. Hill, J. Noble, and J. Potter. Scalable visualisations with ownership trees. In *Proceedings 37th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'00)*, pages 202–213, June 2000.
13. Michael 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)*, pages 54–61, New York, NY, USA, 2001. ACM.
14. John Hogg, Doug Lea, Alan Wills, Dennis deChampeaux, and Richard Holt. The Geneva convention on the treatment of object aliasing. *SIGPLAN OOPS Mess.*, 3(2):11–16, 1992.
15. Michael F. Kleyn and Paul C. Gingrich. GraphTrace — understanding object-oriented systems using concurrently animated views. In *Proceedings of International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'88)*, volume 23, pages 191–205. ACM Press, November 1988.
16. Danny Lange and Yuichi Nakamura. Interactive visualization of design patterns can help in framework understanding. In *Proceedings ACM International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'95)*, pages 342–357, New York NY, 1995. ACM Press.
17. Adrian Lienhard, Orla Greevy, and Oscar Nierstrasz. Tracking objects to detect feature dependencies. In *Proceedings International Conference on Program Comprehension (ICPC'07)*, pages 59–68, Washington, DC, USA, June 2007. IEEE Computer Society.
18. Michael Meyer, Tudor Gîrba, and Mircea Lungu. Mondrian: An agile visualization framework. In *ACM Symposium on Software Visualization (SoftVis'06)*, pages 135–144, New York, NY, USA, 2006. ACM Press.
19. Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*, pages 1–10, New York NY, 2005. ACM Press. Invited paper.

20. James Noble, John Potter, and Jan Vitek. Flexible alias protection. In Eric Jul, editor, *Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP'98)*, volume 1445 of *LNCS*, pages 158–185, Brussels, Belgium, July 1998. Springer-Verlag.
21. Tamar Richner and Stéphane Ducasse. Recovering high-level views of object-oriented applications from static and dynamic information. In Hongji Yang and Lee White, editors, *Proceedings of 15th IEEE International Conference on Software Maintenance (ICSM'99)*, pages 13–22, Los Alamitos CA, September 1999. IEEE Computer Society Press.
22. Maher Salah, Trip Denton, Spiros Mancoridis, Ali Shokoufandeh, and Filippos I. Vokolos. Scenariographer: A tool for reverse engineering class usage scenarios from method invocation sequences. In *Proceedings of 21th International Conference on Software Maintenance (ICSM'05)*, pages 155–164. IEEE Computer Society Press, September 2005.
23. Tarja Systä, Kai Koskimies, and Hausi Müller. Shimba — an environment for reverse engineering Java software systems. *Software — Practice and Experience*, 31(4):371–394, January 2001.
24. Robert J. Walker, Gail C. Murphy, Bjorn Freeman-Benson, Darin Wright, Darin Swanson, and Jeremy Isaak. Visualizing dynamic software system information through high-level models. In *Proceedings of International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'98)*, pages 271–283. ACM, October 1998.
25. 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'05)*, pages 134–142, Los Alamitos CA, 2005. IEEE Computer Society Press.