Taking an Object-Centric View on Dynamic Information with Object Flow Analysis

Adrian Lienhard [a] Stéphane Ducasse [b] Tudor Gîrba [a]

[a] *Software Composition Group*
IAM — Universität Bern, Switzerland
[b] *INRIA-Lille Nord Europe – LIFL (CNRS UMR 8022) – University of Lille*

Abstract

A large body of research analyzes the runtime execution of a system to extract abstract behavioral views. Those approaches primarily analyze control flow by tracing method execution events or they analyze object graphs of heap memory snapshots. However, they do not capture how objects are passed through the system at runtime. We refer to the exchange of objects as the *object flow*, and we claim that it is necessary to analyze object flows if we are to understand the runtime of an object-oriented application. We propose and detail Object Flow Analysis, a novel dynamic analysis technique that takes this new information into account. To evaluate its usefulness, we present a visual approach that allows a developer to study classes and components in terms of how they exchange objects at runtime. We illustrate our approach on three case studies.

Key words: Dynamic Analysis, Object Flow Analysis

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

⋆ This work is based on an earlier work: "Object Flow Analysis – Taking an Object-Centric View on Dynamic Analysis", in Proceedings of the 2007 International Conference on Dynamic Languages (ESUG/ICDL 2007) http://doi.acm.org/10.1145/1352678.1352686 ©ACM, 2007.

Email addresses: lienhard@iam.unibe.ch (Adrian Lienhard), stephane.ducasse@inria.fr (Stéphane Ducasse), girba@iam.unibe.ch (Tudor Gîrba).

programming languages, statically analyzing the source code can be difficult. Dynamic binding, polymorphism, and behavioral and structural reflective capabilities pose limitations to static analysis.

Many approaches tackle these problems and complement static analysis by investigating the dynamic information collected from system runs [1,2,3,4]. 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 [4,5,6,7]. Such views analyze message passing to reveal the control flow in a system or the communication between objects or between classes [8,9]. Another category of approaches analyzes the interrelationships of objects on the heap [10,11]. None of those dynamic analysis approaches, however, capture the runtime *transfer* of object references. Those approaches lack information required to analyze, for instance, how objects created in one class or package propagate to others.

In this paper we present Object Flow Analysis, a novel dynamic analysis that tracks the transfer of object references in a program execution. In previous works, we demonstrated how we successfully applied Object Flow Analysis to reveal fine-grained dependencies between features [12] and to analyze side effects to extract blueprints for writing Unit tests [13]. While the concept of data flow has been widely studied in static analysis [14], it has attracted little attention so far in the field of dynamic analysis.

To illustrate the usefulness of our analysis, we propose an application for understanding the design of a legacy system. A difficulty with studying the control flow of a program, for instance using an UML sequence diagram, is that the propagation of objects is not apparent. Also, inspecting how objects refer to each other using debuggers or object inspectors does not reveal how the object reference graph evolved. Our goal is to analyze how the objects are passed at runtime to iteratively recover the design 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?
- Which continuous object flows span multiple classes?
- 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?

To address these questions, we present an experimental tool that implements Object Flow Analysis. It provides visualizations to explore 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

[1] See www.squeak.org

the reengineering platform Moose [15] and the visualization engine Mondrian [16].

This work is based on our previous work [17] and the main contributions are: (1) extended problem statement and overview of the state-of-the-art, (2) improved description and presentation of the approach, and (3) a new case study.

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 how classes interact with each other at runtime by exchanging objects. Section 5 evaluates our visual approach based on three case studies, Section 6 provides a discussion, and Section 7 concludes.

2 Approaches to Analyze Object-Oriented Runtime Behavior

The power of objects lies in the flexibility of their interconnecting structure. But this flexibility comes at a cost. Because an object can be modified via any object reference, and references can be transferred, object-oriented programs are hard to understand, maintain, and analyze. Although object aliasing introduces fundamental difficulties we accept its presence as it constitutes an essential feature of object-orientation [18].

A large body of research has been conducted into type systems for controlling object aliasing [19,20,21]. Those approaches extend the programming language with constructs that allow the developer to enforce object encapsulation. Important for program comprehension, however, are insights into the behavior of objects that are *not* encapsulated, that is, the objects that are aliased and that are passed around at runtime. Those are the objects that make understanding, maintaining and analyzing object-oriented programs difficult. Two examples of well known problems are representation exposure [22] and argument dependency [19]. Although difficulties caused by aliasing in object-oriented programs have long been recognized [18], the analysis of data flow has attracted little attention so far in the field of dynamic analysis.

In the literature we can distinguish two major categories of dynamic analysis approaches:

- Approaches that analyze message passing by recording method execution events of an instrumented system (Section 2.1).

- Approaches that analyze object reference graphs from program memory snapshots (Section 2.2).

2.1 Message passing analyses

Typically, dynamic analysis techniques focus on execution traces, which capture method execution events [1,5,6,8,23,24]. Most common tracing techniques are based on introducing sensors into methods that generate an event when they are encountered. Instrumentation is usually done by manipulating the bytecode of the target program [25,26,27,28]. Recent approaches have also made use of the Java VM Tool Interface [29] and aspects [30]. In Smalltalk, wrapping compiled method objects to intercept method execution is another often used technique [31].

Since execution traces cannot be directly analyzed by a human due to their typical large size, most approaches are concerned with filtering and building higher abstractions.

The following approaches focus on identifying (reoccurring) patterns in traces of method executions [23,32,33,34]. Lange and Yuichi built the Program Explorer to identify design patterns [2]. De Pauw et al. propose an approach to automatically identify reoccurring execution patterns to detect domain concepts that appear at different locations in the method trace [1]. Richner et al. recover collaborations and roles by applying pattern matching on execution traces [23]. Scenariographer is a tool, which computes groups of similar sequences of method executions to reveal class usage scenarios [34]. Zaidman and Demeyer propose to apply a heuristic clustering of method executions in a trace based on the execution frequency of methods [35], and in another work Zaidman et al. propose to apply web mining techniques to identifying important classes [8].

Various approaches make use of additional data that is captured for each method execution event, such as the identity of the receiver object, arguments, and return values. Some approaches also capture the creation of new instances [9,36,37,38]. For example, Gschwind et al. illustrate object interactions as UML sequence diagrams taking arguments into account [36]. De Pauw et al. exploited visualization techniques to present instance creation events and calls between classes [9,39].

Limitation of message passing analyses. Execution traces are usually represented as a method call-tree. Figure 1 illustrates a small excerpt of an execution trace, displaying the method executions as a tree (the notation follows the pattern *targetclass*≫*methodsignature*). The example is from a Smalltalk bytecode compiler.

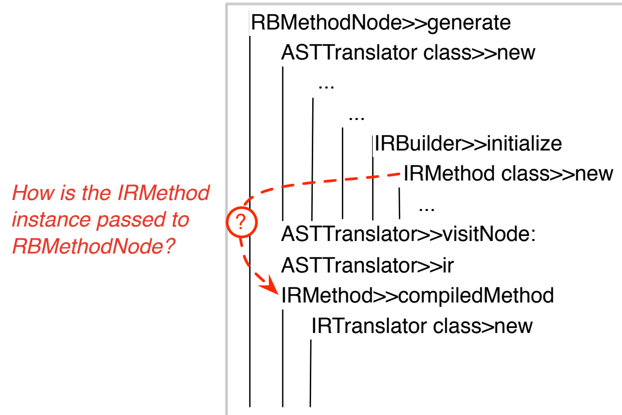

Fig. 1. Excerpt of an execution trace represented as a tree.

A limitation of execution traces is that they typically provide little information about the actual objects involved in the execution. By reading the trace in Figure 1 we see that an IRMethod instance is created in IRBuilder by a call of the new primitive. Later an IRMethod instance is sent the message compiledMethod in the method RBMethodNode≫generate. 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 object is the identical instance in both places of the trace.

However, from the information provided by the execution trace we cannot answer how this instance was passed from where it is instantiated to where it is used later on. 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 further hampered by the sheer size of execution traces. 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.

2.2 Object reference analyses

The second category of approaches analyze the *structure* of object relationships. Super-Jinsight visualizes object reference patterns to detect memory leaks [10], and the visualizations of ownership-trees proposed by Hill et al. show the encapsulation structure of objects [11]. To support debugging, tools like the GNU Data Display Debugger [40] visualize program memory. Tonella et al. extract the object diagram statically and dynamically by analyzing test runs [41].

Query based approaches let the programmer test relationships between objects and method execution trace data. Goldsmith proposes relational queries over program traces [38]. Testlog is a system to write tests using logic queries on execution traces [42]. Dynamic query-based debuggers offer programmers an effective tool that provides instant error alert by continuously checking inter-object relationships while the debugged program is running [43].

Dynamic data flow analysis is a method of analyzing the sequence of define, reference, and undefine actions on data fields at runtime. It has mainly been used for testing procedural programs, but has been extended to object-oriented programming languages as well [44,45]. The goal of those approaches is to detect improper sequences on data access.

Limitation of object reference analyses. Object graphs can be visualized by an UML object diagram [46] as illustrated by Figure 2.

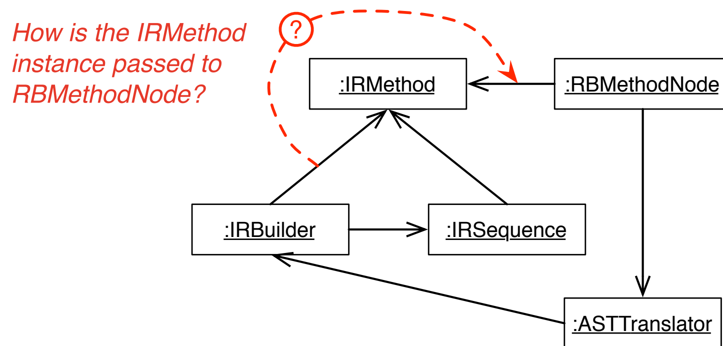

Fig. 2. UML object diagram representing an excerpt of the object graph.

This shows a snapshot of the objects and their reference relationships from the same example discussed in the previous section. The snapshot is taken just before IRMethod≫compiledMethod is executed. In the diagram we can see that IRBuilder references the IRMethod instance (from the trace we know that the IRMethod is instantiated in IRBuilder). But like with the method execution trace perspective, this perspective does not reveal how the IRMethod instance is passed from the IRBuilder to the MethodNode. The missing link is how the object references is transferred in conjunction with the methods executed.

2.3 Towards a deeper understanding of object-oriented program behavior

The two perspectives taken by the message passing and object reference analysis approaches seem like a natural choice as objects and messages are the cornerstones of object-orientation. In object-oriented systems, objects, which are connected through object references, collaborate together to accomplish a

complex task. This collaboration is expressed through the exchange of messages.

Program execution continuously transforms the object graph. Conceptually, behavior can be characterized as the process of transforming a graph of objects, where edges represent field references between objects. Through message passing, references can be transferred from one object to another.

The approaches focusing on method executions capture details of message passing. They reveal the sequence and nesting of messages. On the other hand, the approaches focusing on the object references, capture the shape of the object graph at *a particular point in time*. Neither category of approaches, however, captures how object references are transferred. Even a combination of both perspectives cannot close this gap as it is not possible to reveal the continuous path along which an object has been passed through the system.

We argue that bridging this gap opens a new perspective on dynamic analysis. We formulate the following thesis:

> *To gain an exhaustive understanding of object-oriented runtime behavior, we need to complement the analysis of message passing with the analysis of how object references are transferred.*

Our solution is to explicitly represent object references and capture how they propagate. The flow of an object through the system is then given by all its references and how they are transferred.

Figure 3 illustrates an execution trace (marked as gray nodes and edges) and the flow of one object (marked as red edges) as extracted from one of our case studies. The jumps from one branch of the execution trace to another branch shows that the object is stored into and later accessed from field references (A) as this is the only way how an object reference can be transferred between two distant method executions. (B) and (C) are cases where the object is transferred along the path of control. First, the object is passed multiple times as argument downwards the execution tree (B), then it is passed upwards as a method return value (C).

The flow of an object takes place within method executions (through field references and local variables) or between two method executions (arguments and return values). On the other hand, the effect of a message send depends on the referenced target object. Therefore, the object and control flow are tightly interrelated.

On the other hand, the object graph of the program at a particular point in time can be reconstructed from the object flow information as we capture each time an object is stored into a field. Compared to the object reference analyses,

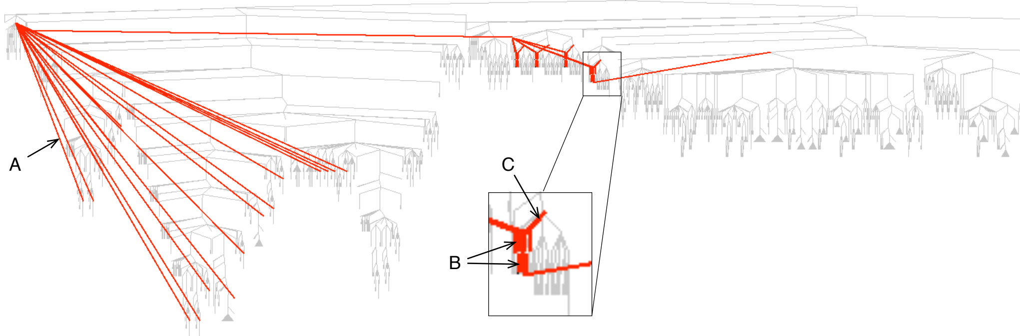

Fig. 3. Example of how an object can jump from one branch to another in an execution trace. The gray edges represent the method executions, and the red ones show the flow of an object.

which examine object graphs, our model is more powerful as it additionally captures the history of each reference in the graph. This information adds a new dimension to traditional execution traces. With our model, dynamic information can be navigated not only along the flow of control but additionally along the flow of objects.

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 reference is made visible in a method execution. We represent in our meta-model each such situation by a so-called *Alias* (see Figure 4).

In our program execution representation, an object alias is created when an object is (1) instantiated, (2) stored in a field (including indexable fields) (3) read from a field, (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 in which the alias makes the object visible. By definition, arguments, return values, and local variables are only visible in one method activation.

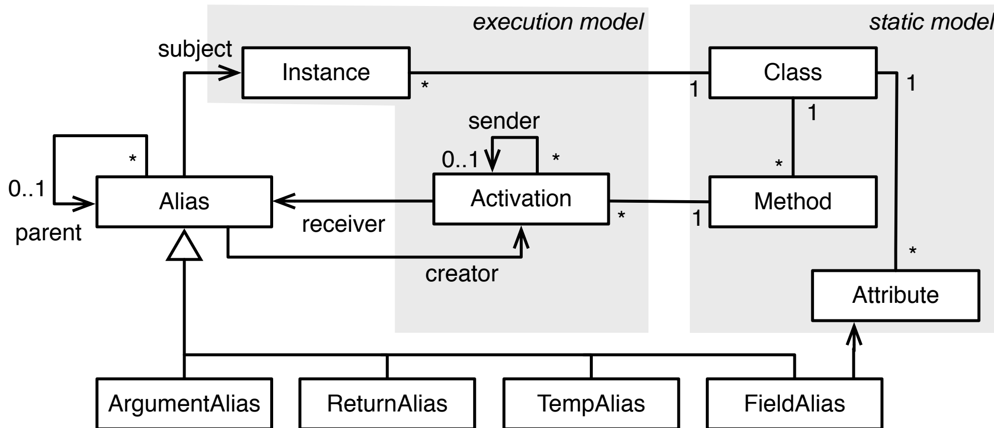

Fig. 4. The Object Flow meta-model extends the static and execution meta-models with the notion of Alias.

In contrast, objects that are stored in fields (*i.e.*, instance variables), can be accessed in other activations as well. Therefore, we distinguish between *read* and *write* access of fields.

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, that is, 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. The compiler has four phases: (1) scanning and parsing of source code, (2) verifying and annotating the abstract syntax tree (AST), (3) translating AST to the intermediate representation (IR), and (4) translating IR to bytecode.

The example used in this section shows the interplay of important classes of the last two compiling phases (translating the AST to the IR, and translating the IR to bytecode). We focus on an instance of the class IRMethod which represents

a method in the IR. 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 5 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.

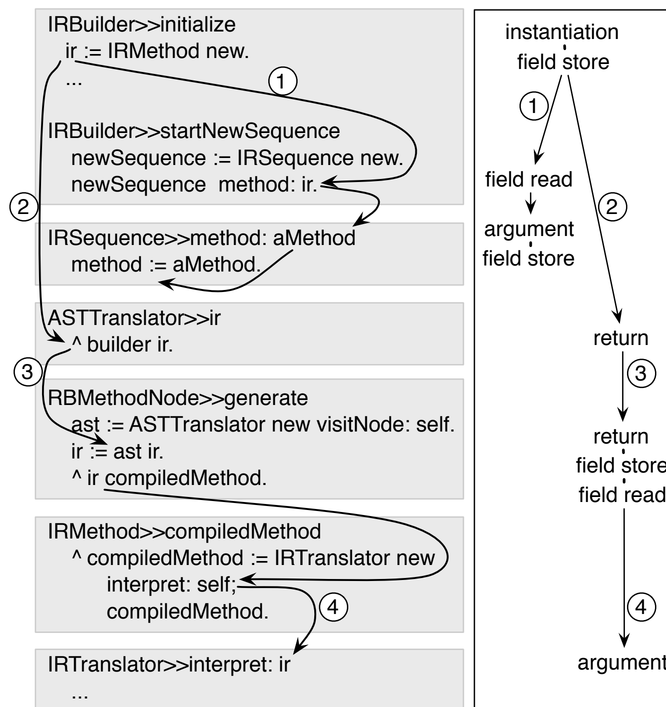

Fig. 5. Object flow of an IRMethod.

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. The object flow perspective focuses on the runtime flow of the IRMethod instance. The methods on the left side of Figure 5 are ordered by when the instance is passed through them (rather than by the order of the control flow). The right side of Figure 5 illustrates the corresponding flow as a tree. Nodes represent aliases and edges represent parent-child relationships 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 5 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 5). 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 analyze the transfer of object references between classes to answer questions such as:

(1) Which classes exchange objects?
(2) Which classes act as object hubs?
(3) Which continuous object flows span multiple classes?
(4) Given a class, which objects are passed to or from it?

(5) Which objects get stored in a class, and which objects just pass through it?

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 business logic or the user interface). This view answers the questions (1-3) stated above.
- 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). This view answers the questions (4-5).

4.1 Inter-unit Flow View

Figure 6 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.

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 experimental tool, units are stated by the developer using a declarative mapping language (similar to the approach of Walker et al. [37]). 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'
```

The following two rules gather all classes with names ending in scope or var.

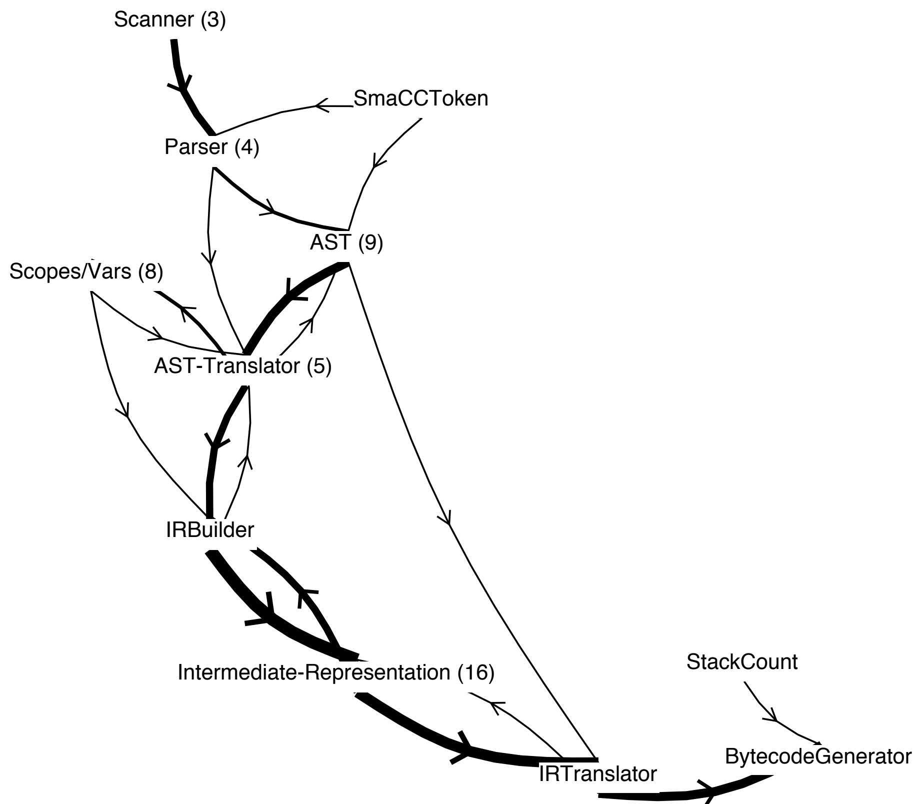

Fig. 6. Inter-unit Flow View of the bytecode compiler.

```
classes matchingName: '*scope' mapTo: 'Scopes/Vars'
classes matchingName: '*var' mapTo: 'Scopes/Vars'
```

The mapping is ordered. That is, each class is mapped to at most one unit, the first one for which a rule matches. If no rule matches, the class is displayed as a single node in the visualization. For convenience, there exist two additional rules:

```
classes mapAllToPackages
classes mapAllTo: 'Rest'
```

The first one maps each remaining class to the package it is contained in, that is, for each package a unit is created. The second rule maps all remaining classes to a single unit (it is syntactic sugar for matching the names with '*').

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 class to the other 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 6, 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, 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 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.

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 different 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 tool, the user can scope the visualized object flow information to a specific time period 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 help investigate how objects are propagated during a program execution.

Figure 7 illustrates three snapshots in the evolution of the compiler execution (compare with Figure 6). 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 BytecodeGenerator.

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.

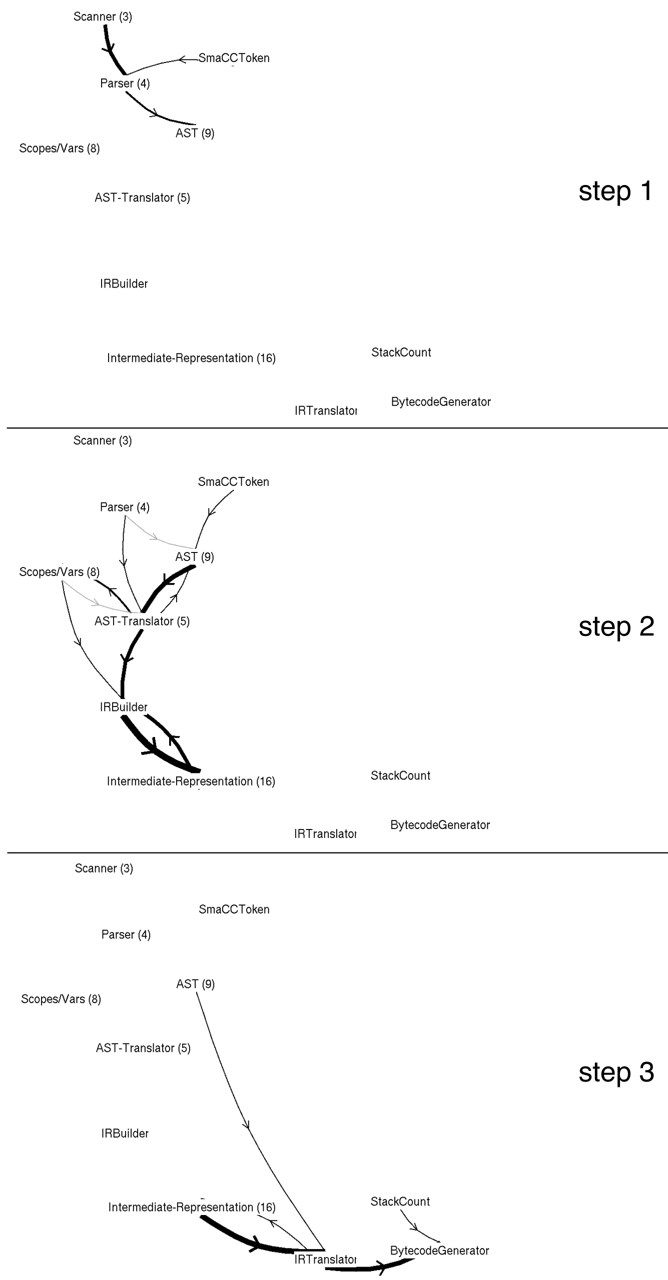

Fig. 7. Chronological propagation of flows in the compiler.

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 tool 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 [2].

Figure 8 shows twice the same visualization (compare with Figure 6) but with different classes selected. In Figure 8.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 8.B IRBuilder is selected. We see that it obtains objects from most above units and forwards objects to almost all units below.

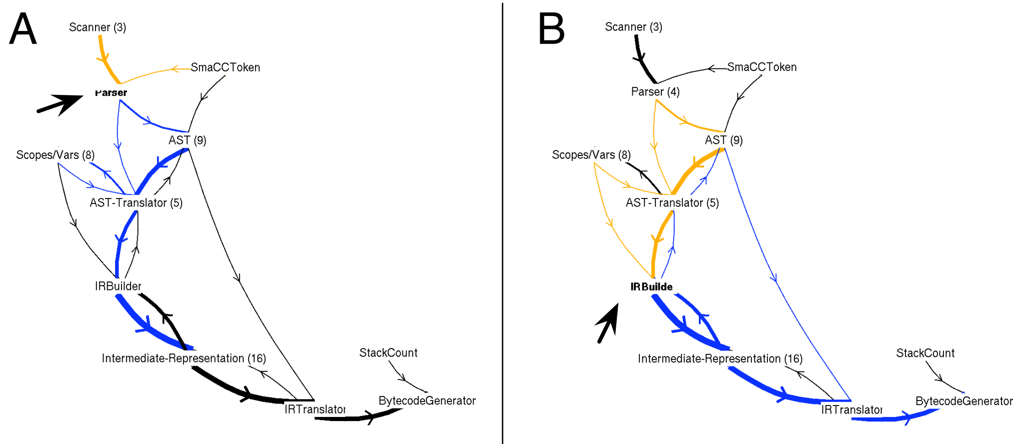

Fig. 8. Orange and blue arcs indicate flows leading to and coming from selected unit Parser (A), resp. unit IRBuilder (B).

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 help investigate this information, our tool allows the user to drill down to access detailed information about the objects transiting a unit.

Figure 9 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

[2] On a B/W print, orange corresponds to light gray and blue to dark gray.

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).

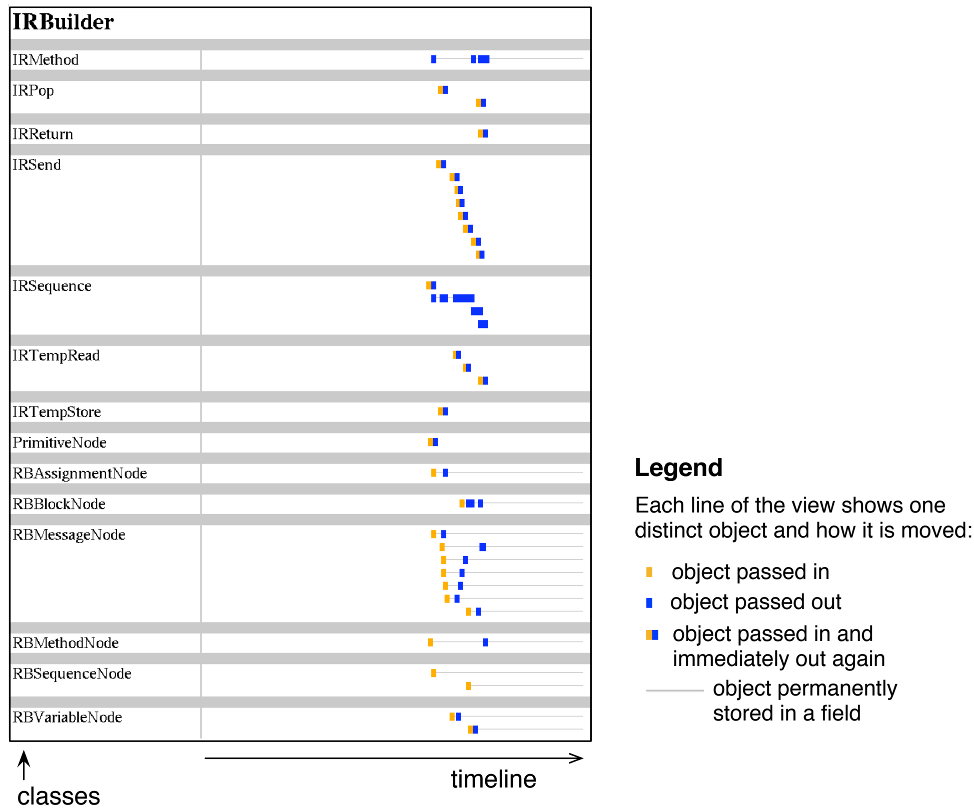

Fig. 9. IRBuilder Transit Flow View.

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 9, 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 three case studies: a Smalltalk bytecode compiler, a health insurance web application and an IRC chat client. All three applications are implemented in Squeak, an open-source Smalltalk dialect. Our choice of those case studies was motivated by the following reasons: (1) they are non-trivial and model very different domains, (2) we have access to the source code, and (3) for the compiler and health insurance application we have direct access to developer knowledge to verify our findings.

The table below shows the static dimensions of the code:

	Compiler	Insurance App.	IRC Chat Client
Classes	127	308	39
Methods	1912	4432	1063
Lines of code	11208	40917	7652

The objective of these preliminary investigations is to evaluate the usefulness of the two visualizations of our approach 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 6 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 three case studies in the next section.

Using the Inter-unit Flow View we could extract the key phases of the compiler. This was straight forward from studying the chronological propagation of the object flows. The activity starts on top (Scanner and Parser) and then

shifts downwards to center around AST-Translator and IRBuilder and eventually shifts to IRTranslator and BytecodeGenerator (see Figure 7). This observation is in line with the documentation, which describes the following main phases: (i) scanning and parsing, (ii) translating AST to the intermediate representation, and (iii) 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 9) we studied detailed interrelationships between the units. For example, in the transition from AST to IR (phase 2) we see that the unit AST-Translator passes AST nodes (classes with the RB prefix) to IRBuilder. In Figure 9 we see that AST nodes are passed into IRBuilder and from the Figure 6 we see that they come from AST-Translator. In the Figure 9 we also see that IRBuilder creates three sequence objects which are passed outside multiple times.

Surprisingly, also the AST nodes are forwarded to the Intermediate Representation package (we expected that after IRBuilder created the IR from the AST, the AST nodes are discarded). Following the flow of the AST nodes we reach the Intermediate Representation package. Figure 10 shows the Transit Flow View of the IR package. We see here that the AST nodes are passed in and are then stored in the class but are never passed out again. This points to the fact that IR objects hold a reference to the AST node from which they originate. Also interesting in Figure 10 is that one can distinguish two phases of activity. The first phase is when the IR is built, where we see IR and AST nodes being passed into the IR package (marked in Figure 10). The second phase is when the bytecode is generated, where IR objects are passed outside (but not AST nodes). In this phase we can also see that the single instance of IRTranslator is transferred many times.

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 [8,9].

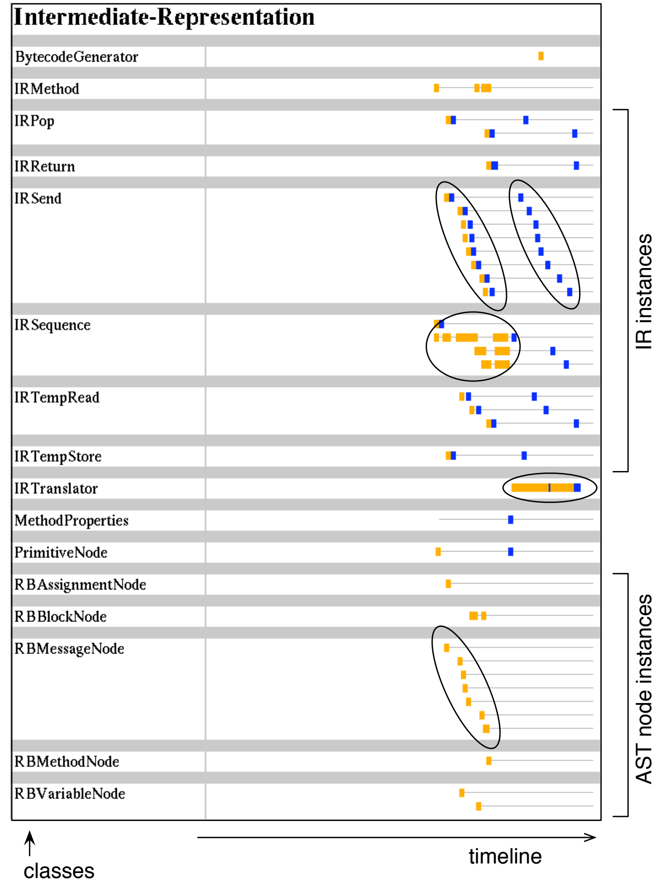

Fig. 10. Transit Flow View of the Intermediate Representation package.

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 5.

This industrial application was put into production six years ago and since that time has undergone various adaptations and extensions. The analyzed 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.

With this case study we focus our discuss on the exploration process, rather than on the details of the actual findings.

Step 1: Creating coarse-grained units. We started by investigating the Inter-unit Flow View with units corresponding to packages. However, the view was hard to work with because it was cluttered with many small packages that were part of the GUI layer (see Figure 11).

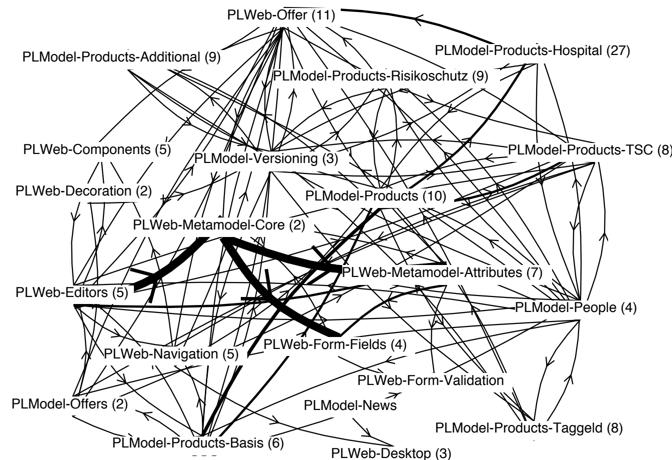

Fig. 11. Inter-unit Flow View of the insurance application case study *before* refining the mapping. In this state it is too cluttered for comprehension.

Step 2: Re-grouping to appropriate units. As a first refinement of the mapping of classes to units we put all classes of web GUI related packages into one unit, representing the presentation layer.

```
self containedInPackage: 'PLWeb*' mapTo: 'Web app UI layer'
```

The resulting view was already more concise. Now focusing on the business logic, 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 single products differ.

self hierarchyRootedIn: 'Product' mapTo: 'Products'
self hierarchyRootedIn: 'CalculationModel' mapTo: 'Models'

This change dramatically improved the view. We obtained only nine units and we could identify interesting flows between them (see Figure 12). 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).

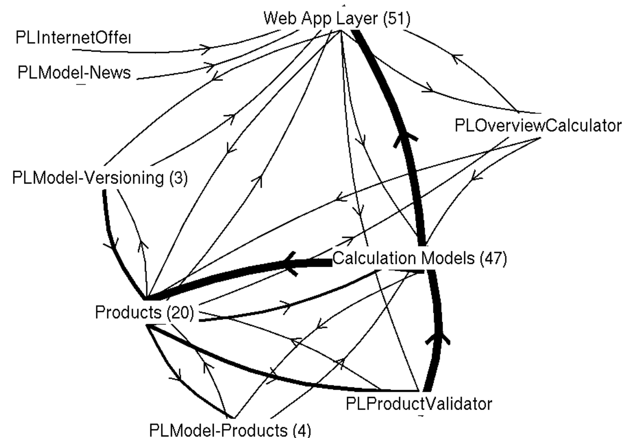

Fig. 12. Inter-unit Flow View of the insurance application case study.

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

To obtain more details of 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 this class passes persons and dates to products, which eventually return price objects.

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 structure units differently compared to packages. For instance, classes representing products and classes representing calculation models were organized together in the same packages. However, we wanted to distinguish products and their calculation models and hence created two units, one for all classes inheriting from Product, and the other for all classes inheriting from CalculationModel.

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.

5.3 IRC Chat Client

As the last case study we chose an IRC Chat Client [47]. A total of six developers contributed to this open source project which underwent various refactorings and enhancements over nine years. We analyzed nine features: open, setup, connect to server, request MOTD, join channel, send and receive message, opening new console, and disconnect.

In contrast to the other two case studies, we enhanced the Inter-unit Flow View with information about features. We wanted to study if two classes exchange objects in only one feature or in several features.

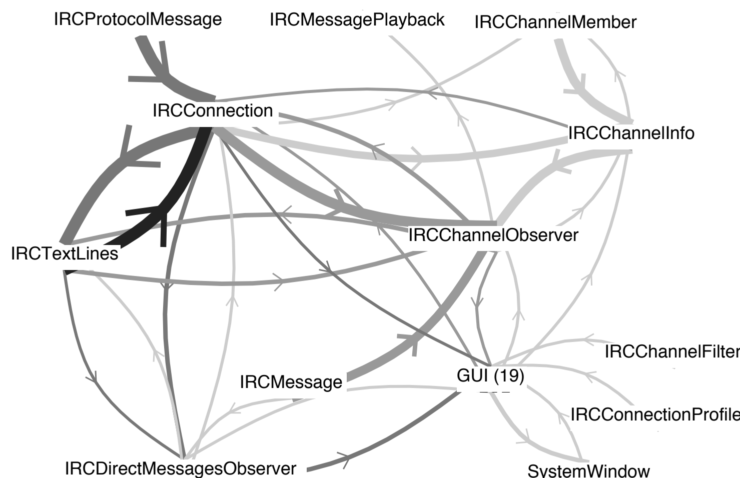

Fig. 13. Inter-unit Flow View of the IRC chat client case study with gray toning of edges indicating the number of participating features.

Figure 13 illustrates the Inter-unit Flow View after grouping all GUI classes into one unit. The largest flows are coming into or going out of IRCConnection. Those flows take place in several features (the darkest gray corresponds to 5 features, and the flow between IRCConnection and IRCChannelObserver takes place in 2 features). On the other hand, many flows around the GUI unit

are specific to one feature only. For example IRCConnectionProfile only passes objects to the GUI classes during the *setup* feature. Another example is the class IRCChannelInfo, where flows either take place in the feature *join channel* or in the feature *disconnect* – even though the edges are relatively thick (20 objects being transferred along the thickest edge).

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 design level, therefore, we focused on classes.

Other applications of Object Flow Analysis. We believe that Object Flow Analysis can be exploited in many other 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 analyze the flow of objects between *features* to detect feature runtime dependencies [12] and to analyze side effects in parts of an execution trace to extract blueprints for writing Unit tests [13]. 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 analysis and experimental tool 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.

Summarizing, Object Flow Analysis, which gathers both object aliasing and method executions, consumes roughly 2.5 times the space of conventional execution trace approaches that capture all method execution events including

	Compiler	Insurance App.	IRC Chat Client
Method executions	11910	120569	71183
Aliases	16033	197499	136485
Ratio	1.3	1.6	1.9

the identification of the message receiver and the return value. Our approach presented in this paper deals with the potential large number of events by providing abstract views. 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 [48]. 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 individual 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.

Dynamic analysis requirements. To cover all object flows in a program execution, the tracing technique has to be implemented carefully. Our Object Flow tracer not only tracks objects of application classes but also instances of system classes and primitive type values. For example, collections and arrays have to be taken into account as they preserve permanent object references between the holder of the collection and its contained elements. Furthermore, since all method executions and state modifications have to be captured, also behavioral reflection has to be dealt with appropriately.

We chose Smalltalk to implement a dynamic analysis prototype because of its openness and reflective capabilities, which allowed us to evaluate different alias tracking techniques. In our current implementation, each regular object reference is substituted by an alias object as presented in the meta-model. Every time a reference is transferred, we construct the corresponding alias. To track read and write of variables our current implementation uses bytecode instrumentation [49], and to capture argument passing and return values it uses method wrappers [31].

We are currently also investigating how to implement Object Flow Analysis in Java. In the current state of the Java implementation [50], which is based on Javassist [26], we can detect all aliases but tracking their transfer is still missing. A possible direction is to investigate how precise the object flow information can be reconstructed by complementing the dynamic analysis with a static points-to analysis (for instance, using a pointer assignment graph as proposed by Lhoták [51]).

7 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 called 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 illustrated 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 three 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 various 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] W. De Pauw, D. Lorenz, J. Vlissides, M. Wegman, Execution patterns in object-oriented visualization, in: Proceedings Conference on Object-Oriented

Technologies and Systems (COOTS'98), USENIX, 1998, pp. 219–234.

[2] D. Lange, Y. 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), ACM Press, New York NY, 1995, pp. 342–357.

[3] T. Richner, S. Ducasse, Recovering high-level views of object-oriented applications from static and dynamic information, in: H. Yang, L. White (Eds.), Proceedings of 15th IEEE International Conference on Software Maintenance (ICSM'99), IEEE Computer Society Press, Los Alamitos CA, 1999, pp. 13–22.

[4] G. Antoniol, Y.-G. Guéhéneuc, Feature identification: a novel approach and a case study, in: Proceedings IEEE International Conference on Software Maintenance (ICSM'05), IEEE Computer Society Press, Los Alamitos CA, 2005, pp. 357–366.

[5] M. F. Kleyn, P. 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), Vol. 23, ACM Press, 1988, pp. 191–205.

[6] O. Greevy, S. 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), IEEE Computer Society, Los Alamitos CA, 2005, pp. 314–323.

[7] M. El-Ramly, E. Stroulia, P. Sorenson, Recovering software requirements from system-user interaction traces, in: Proceedings ACM International Conference on Software Engineering and Knowledge Engineering, ACM Press, New York NY, 2002, pp. 447–454.

[8] A. Zaidman, T. Calders, S. Demeyer, 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), IEEE Computer Society Press, Los Alamitos CA, 2005, pp. 134–142.

[9] W. De Pauw, D. Kimelman, J. Vlissides, Modeling object-oriented program execution, in: M. Tokoro, R. Pareschi (Eds.), Proceedings of the European Conference on Object-Oriented Programming (ECOOP'94), LNCS 821, Springer-Verlag, Bologna, Italy, 1994, pp. 163–182.

[10] W. De Pauw, G. Sevitsky, Visualizing reference patterns for solving memory leaks in Java, in: R. Guerraoui (Ed.), Proceedings of the European Conference on Object-Oriented Programming (ECOOP'99), Vol. 1628 of LNCS, Springer-Verlag, Lisbon, Portugal, 1999, pp. 116–134.

[11] T. Hill, J. Noble, J. Potter, Scalable visualisations with ownership trees, in: Proceedings 37th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'00), 2000, pp. 202–213.

[12] A. Lienhard, O. Greevy, O. Nierstrasz, Tracking objects to detect feature dependencies, in: Proceedings International Conference on Program Comprehension (ICPC'07), IEEE Computer Society, Washington, DC, USA, 2007, pp. 59–68.

[13] A. Lienhard, T. Gîrba, O. Greevy, O. Nierstrasz, Test blueprints – exposing side effects in execution traces to support writing unit tests, in: 12th European Conference on Software Maintenance and Reengineering (CSMR'08), IEEE Computer Society Press, 2008, pp. 83–92.

[14] 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), ACM, New York, NY, USA, 2001, pp. 54–61.

[15] O. Nierstrasz, S. Ducasse, T. Gîrba, The story of Moose: an agile reengineering environment, in: Proceedings of the European Software Engineering Conference (ESEC/FSE'05), ACM Press, New York NY, 2005, pp. 1–10, invited paper.

[16] M. Meyer, T. Gîrba, M. Lungu, Mondrian: An agile visualization framework, in: ACM Symposium on Software Visualization (SoftVis'06), ACM Press, New York, NY, USA, 2006, pp. 135–144.

[17] A. Lienhard, S. Ducasse, T. Gîrba, Object flow analysis — taking an object-centric view on dynamic analysis, in: Proceedings of the 2007 International Conference on Dynamic Languages (ICDL'07), ACM Digital Library, New York, NY, USA, 2007, pp. 121–140.

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

[19] J. Noble, J. Potter, J. Vitek, Flexible alias protection, in: E. Jul (Ed.), Proceedings of the 12th European Conference on Object-Oriented Programming (ECOOP'98), Vol. 1445 of LNCS, Springer-Verlag, Brussels, Belgium, 1998, pp. 158–185.

[20] D. G. Clarke, J. Noble, J. M. Potter, Simple ownership types for object containment, in: Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'91), LNCS, Springer Verlag, London, UK, 2001, pp. 53–76.

[21] C. Boyapati, B. Liskov, L. Shrira, Ownership types for object encapsulation, in: Principles of Programming Languages (POPL'03), ACM Press, 2003, pp. 213–223.

[22] B. Liskov, J. Guttag, Abstraction and Specification in Program Development, MIT Press/McGraw-Hill, Cambridge, Mass., USA, 1986.

[23] T. Richner, Recovering behavioral design views: a query-based approach, Ph.D. thesis, University of Berne (May 2002).

[24] A. Hamou-Lhadj, 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'06), IEEE Computer Society, Washington, DC, USA, 2006, pp. 181–190.

[25] M. Dahm, Byte code engineering, in: Proceedings of Java-Informations-Tage (JIT'99), Düsseldorf, Deutschland, 1999, pp. 267–277.

[26] S. Chiba, M. Nishizawa, An easy-to-use toolkit for efficient Java bytecode translators, in: In Proceedings of the second International Conference on Generative Programming and Component Engineering (GPCE'03), Vol. 2830 of LNCS, 2003, pp. 364–376.

[27] E. Bruneton, R. Lenglet, T. Coupaye, ASM: A code manipulation tool to implement adaptable systems, in: Proceedings of Adaptable and Extensible Component Systems, Grenoble, France, 2002.

[28] M. Denker, S. Ducasse, É. Tanter, Runtime bytecode transformation for Smalltalk, Journal of Computer Languages, Systems and Structures 32 (2-3) (2006) 125–139.

[29] Sun microsystems, inc. JVM tool interface (JVMTI), http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/.

[30] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin, Aspect-Oriented Programming, in: M. Aksit, S. Matsuoka (Eds.), Proceedings ECOOP '97, Vol. 1241 of LNCS, Springer-Verlag, Jyvaskyla, Finland, 1997, pp. 220–242.

[31] J. Brant, B. Foote, R. Johnson, D. Roberts, Wrappers to the rescue, in: Proceedings European Conference on Object Oriented Programming (ECOOP'98), Vol. 1445 of LNCS, Springer-Verlag, 1998, pp. 396–417.

[32] D. J. Jerding, J. T. Stasko, T. Ball, Visualizing interactions in program executions, in: Proceedings of International Conference on Software Engineering (ICSE'97), 1997, pp. 360–370.

[33] T. Systä, K. Koskimies, H. Müller, Shimba — an environment for reverse engineering Java software systems, Software — Practice and Experience 31 (4) (2001) 371–394.

[34] M. Salah, T. Denton, S. Mancoridis, A. Shokoufandeh, F. 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), IEEE Computer Society Press, 2005, pp. 155–164.

[35] A. Zaidman, 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'04), IEEE Computer Society Press, Los Alamitos CA, 2004, pp. 329–338.

[36] T. Gschwind, J. Oberleitner, Improving dynamic data analysis with aspect-oriented programming, in: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering (CSMR'03), IEEE Computer Society, Washington, DC, USA, 2003, p. 259.

[37] R. J. Walker, G. C. Murphy, B. Freeman-Benson, D. Wright, D. Swanson, J. 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), ACM, 1998, pp. 271–283.

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

[39] W. De Pauw, R. Helm, D. Kimelman, J. Vlissides, Visualizing the behavior of object-oriented systems, in: Proceedings of International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93), 1993, pp. 326–337.

[40] A. Zeller, D. Lütkehaus, DDD — a free graphical front-end for Unix debuggers, SIGPLAN Not. 31 (1) (1996) 22–27.

[41] P. Tonella, A. Potrich, Static and dynamic C++ code analysis for the recovery of the object diagram, in: Proceedings of 18th IEEE International Conference on Software Maintenance (ICSM'02), IEEE Computer Society, Los Alamitos, CA, USA, 2002, p. 54.

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

[43] R. Lencevicius, U. Hölzle, A. K. Singh, Dynamic query-based debugging, in: R. Guerraoui (Ed.), Proceedings of European Conference on Object-Oriented Programming (ECOOP'99), Vol. 1628 of LNCS, Springer-Verlag, Lisbon, Portugal, 1999, pp. 135–160.

[44] T. Y. Chen, C. K. Low, Dynamic data flow analysis for C++, in: Proceedings of the Second Asia Pacific Software Engineering Conference (APSEC'95), IEEE Computer Society, Washington, DC, USA, 1995, p. 22.

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

[46] M. Fowler, UML Distilled, Addison Wesley, 2003.

[47] Squeak IRC client, http://www.preeminent.org/squeak/irc-help/irc-help.html.

[48] T. 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), no. 1687 in LNCS, Springer Verlag, Heidelberg, 1999, pp. 216–234.

[49] M. Denker, S. Ducasse, A. Lienhard, P. Marschall, Sub-method reflection, Journal of Object Technology 6 (9) (2007) 231–251.

[50] J. Fierz, Java Wiretap — extracting feature execution models for reverse engineering, Informatikprojekt, University of Bern, http://scg.iam.unibe.ch/Archive/Projects/Fier07a.pdf (Jun. 2007).

[51] O. Lhoták, L. Hendren, Scaling Java points-to analysis using Spark, in: G. Hedin (Ed.), Compiler Construction, 12th International Conference, Vol. 2622 of LNCS, Springer, Warsaw, Poland, 2003, pp. 153–169.