

Exploiting Runtime Information in the IDE*

David Röthlisberger, Orla Greevy and Oscar Nierstrasz
Software Composition Group, University of Bern, Switzerland
{roethlis,greevy,oscar}@iam.unibe.ch

Abstract

Developers rely on the mechanisms provided by their IDE to browse and navigate a large software system. These mechanisms are usually based purely on a system's static source code. The static perspective, however, is not enough to understand an object-oriented program's behavior, in particular if implemented in a dynamic language. We propose to enhance IDEs with a program's runtime information (e.g., message sends and type information) to support program comprehension through precise navigation and informative browsing. To precisely specify the type and amount of runtime data to gather about a system under development, dynamically and on demand, we adopt a technique known as partial behavioral reflection. We implemented navigation and browsing enhancements to an IDE that exploit this runtime information in a prototype called Hermion. We present preliminary validation of our experimental enhanced IDE by asking developers to assess its usefulness to understand an unfamiliar software system.

Keywords: dynamic analysis, development environments, partial behavioral reflection, reverse engineering, program comprehension

1 Introduction

Gaining an understanding of large object-oriented systems by navigating the source code in a development environment (IDE) is an inherently difficult and time-consuming task. Object-oriented language characteristics such as inheritance and polymorphism make it difficult to understand how an application is implemented purely by navigating and browsing source code [5, 11, 26]. Often conceptually related code is scattered over many different source artifacts, e.g., classes and methods. The task of program understanding is more acute with dynamically-typed languages such as Smalltalk or Ruby, as developers typically require access to runtime type information to gain a

complete understanding.

Program comprehension is a prerequisite when faced with the task of extending and maintaining a system. Exploration of a system is constrained by the mechanisms provided by the IDE to browse and navigate a large software space. Today's popular IDEs base browsing and navigation mechanisms on a system's static source code. They do not provide an integrated view of the dynamic and static structures of the system. Hence they offer little to understand the runtime behavior of a system. Researchers in program comprehension have recognized the value of combined static and dynamic views for program comprehension [6, 11, 15, 23, 26]. Despite this, integrating dynamic information to enhance static views of source code directly in an IDE, the tool probably most frequently used by software developers, has attracted little attention.

We focus on the challenge of integrating runtime information directly in the IDE. We claim the following advantages: (i) a developer is not faced with a context switch from an IDE to a reverse engineering tool, (ii) the dynamic information is available immediately, (iii) and it is seamlessly integrated in the workflows of the IDE.

We identify different kinds of dynamic information and illustrate how each contributes to a developer's system understanding. We claim that direct access within an IDE to runtime information such as message sends, class references and runtime types of variables enhances program comprehension through informative and efficient browsing and navigation. The key research questions we address in this paper are:

- *How do we integrate dynamic information into an IDE's browsing and navigation mechanisms?*
- *How can we efficiently collect dynamic data in a running IDE?*

We address these questions and present our working prototype, an IDE called Hermion with the capability to capture runtime information from an application under development and to exploit this information by enhancing navigation and browsing of the source code. We validate the usefulness of Hermion by using it to understand two

*In: ICPC 2008 (International Conference on Program Comprehension), 2008

medium-sized applications. As validation of our work we perform a preliminary experiment where we ask five developers, unfamiliar with the applications, to report on how Hermion supports their understanding of the systems.

We cover and address the typical dynamic analysis issues such as efficiency, coverage and completeness and outline future improvements.

The key contributions of this paper are: (i) we identify which kinds of dynamic information are useful for enhancing the navigation and understanding of systems in the IDE, (ii) we describe our dynamic information enhancements to the IDE, and (iii) we apply a partial behavioral reflection technique to selectively gather runtime information within the IDE.

Outline. In the next section we identify shortcomings of purely static IDEs and elaborate with example scenarios on what kinds of dynamic information can improve and optimize the navigation of a source space. Section 3 presents our technique to dynamically collect runtime information. We present the validation of our work in Section 4 with two medium-sized object-oriented systems and a preliminary empirical evaluation. In Section 5 we discuss our work by highlighting efficiency, coverage and completeness issues. Section 6 presents related work, while in Section 7 we draw our conclusions and outline future work.

2 Dynamic Information in the IDE

This section answers the research question: *How do we integrate dynamic information into an IDE’s browsing and navigation mechanisms?*

We motivate our work by highlighting the restrictions a developer faces in IDEs when trying to understand systems implemented in object-oriented and dynamically-typed languages. The general problem is that the IDE’s view focuses purely on static source code. It provides little support for understanding the dynamic behavior of a software system, in particular of systems written in dynamic object-oriented languages that make widespread use of inheritance and polymorphism. This makes it difficult for a developer to determine how classes interact at runtime, *e.g.*, to which receiver a message is sent at runtime, or what kind of objects are stored in variables. In dynamically-typed languages such as Smalltalk or Ruby, but also to a lesser degree in statically-typed languages such as Java, the IDE generally provides no means to support the developer browsing the source code to understand the runtime behavior of the system under study. For instance, the Eclipse IDE [12] does not provide an integrated view showing the precise types that are dynamically assigned to a variable or to seamlessly navigate directly in the source code to the message sends actually occurring at runtime.

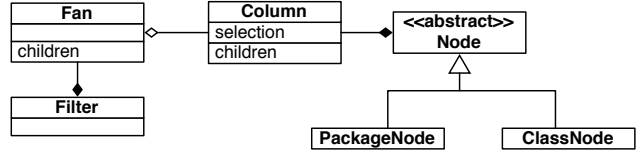


Figure 1. UML Class Diagram of the Omni-Browser Kernel Classes

2.1 Scenario: Understanding a Complex System

In the following we illustrate with a scenario some concrete problems a developer faces when trying to understand a complex and generic application written in the dynamic language Smalltalk. Subsequently, we present our experimental “dynamic IDE” called Hermion which encompasses solutions to the problems we are going to identify. We emphasize that the issues we address in this discussion are generally applicable in the context of IDEs and object-oriented and dynamically-typed languages.

We center our discussion around an example software, the OmniBrowser framework. OmniBrowser is a highly customizable, generically implemented framework, rendering it well-suited for the implementation of a range of source code browsers (*e.g.*, package browsers or class browsers). It makes extensive use of method dispatching and polymorphic message sends which may make it difficult to understand its source code purely by static browsing.

We briefly introduce the OmniBrowser framework, providing a structural overview with a UML class diagram in Figure 1. It provides a class Column to represent a vertical list of selectable elements, *e.g.*, a list of packages. Every element in the column is an instance of class Node. Concrete types of nodes are represented by subclasses of Node, for example ClassNode or PackageNode. The elements of a column may be filtered. This feature is realized using a dedicated Filter class.

We describe a realistic scenario of how a developer, new to the OmniBrowser framework, might gain an understanding of the code. Initially, the developer wants to understand how elements are loaded into a column for display. Subsequently, she tries to discover how a selected element is represented in a column. Finally, she wonders how different columns are embedded in a browser.

Message Send Navigation. To find out how OmniBrowser loads elements in a column for display, the developer may start by looking at the class Column. This class defines a method children which reads as follows:

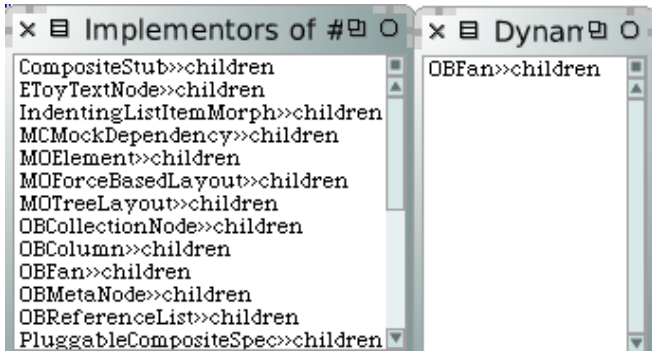


Figure 2. Static search (1) vs. precise dynamic search (2) for implementors of *children* in Hermion

```
children
↑fan children
```

Browsing this method reveals that `Column` delegates the message send `children` to an object called `fan` which then apparently answers all elements of a column. Static browsing does not reveal where the elements actually come from. By just looking at the source code of `Column`»`children` it is not obvious which `children` method will eventually return the elements. Furthermore, the class of the variable `fan` cannot be determined statically. Due to polymorphism, different `children` methods may be invoked at runtime, depending on what kind of object is stored in `fan`. Often modern IDEs, such as the Squeak IDE, provide an *implementors view* which displays a list of all methods named `children` that exist in the current application. However, this mechanism populates the list of methods by performing a static search of the entire application for `children` methods. This results in a long list of possible methods (see Figure 2). Exploring this list is time-consuming and inefficient.

When we analyze the runtime behavior of an `OmniBrowser` application, we discover that only one single `children` method, the `Fan`»`children` method, is invoked from the `children` method of the `Column` class. This implies that if the IDE provided a mechanism to consult the runtime information, the search space for the developer could be greatly reduced, thus resulting in a more precise and efficient navigation of the source code. Figure 2 compares the different lists of implementors, on the left is the list generated using the existing *implementors view* mechanism (1) and on the right is a list that was generated by `Hermion` which takes dynamic behavior (*i.e.*, message sends) of the `OmniBrowser` application into account (2).

The IDE should even include the navigation of mes-

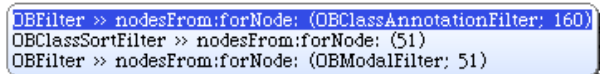


Figure 3. List of methods invoked for message send *nodesFrom:forNode:* in Hermion

sage sends directly in the source code view by enriching this static view with dynamic information gathered while a method is being executed. In the `Column`»`children` method for instance we can enrich the message send `#children` to the `fan` object with an icon. Clicking on the icon while reading the method’s source code navigates the developer directly to the implementation of the `children` method in the class `Fan`, which is the only method invoked at runtime at this location of the code. In the case of a polymorphic message send which has different receivers, there are often several possible methods that may be executed. In such a case, clicking on the icon near the written message send results in the display of a list of all methods that have been invoked at runtime. This list also contains information about the receiver type of the message send and which and how often a method was invoked. In Figure 3. we present an example of this method list in `Hermion` for the polymorphic message send `#nodesFrom:forNode:` which has different receivers.

Type Information. The developer wants to know how a column stores the selection of a specific element. As the `OmniBrowser` framework is implemented in `Smalltalk`, type information is not available in the source code. Thus browsing the source of method `selection` does not reveal the type of information the selection instance variable contains at runtime (see Figure 4). It may contain the selected element itself (*i.e.*, a node) or the index in the list of elements (*i.e.*, an integer). This information is essential to understand how the application generates the list of new elements for the subsequent column.

By analyzing the `selection` method of `Column` dynamically, an IDE can provide the developer with precise information about how the current selection of the column is stored, information that is otherwise hard to determine, when variables can virtually store any kind of objects.

Similar to the message send icon previously described, `Hermion` also enhances the static view of variable accesses in the source code with an icon. Clicking on this icon reveals a list of object types that have been stored in this variable at runtime at this position in source code, along with quantitative information, for example the number of times this variable has assumed a specific type. We refer to this mechanism of revealing the type of variable as *type view*. To promote deeper understanding of the code under investigation, a developer may wish to use the type information

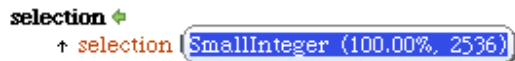


Figure 4. List of types of instance variable *selection* extracted from dynamic information in Hermion

of variables as starting points for further navigation to the corresponding classes behind the types.

Figure 4 depicts the *type view* for the `selection` instance variables in the method `selection`. It reveals that the `selection` variable always stores an integer, which in turn reveals the intent of the `Column` class to store the position of the selected element in the list, not the element itself.

Reference Information. If a developer wants to add a new column to a browser based on the `OmniBrowser` framework, but is unsure how to do this, a static browsing of the code of class `Column` does not quickly reveal which class implements the container component for the columns, as this communication is very well hidden in a few methods of `Column`.

Using runtime information, we can identify all classes with which `Column` communicates. We refer to those classes as dynamic references of `Column`. Many current IDEs such as Eclipse are only capable of locating those references that are explicitly written in source code. As our example shows, the dynamic references are more interesting and useful for understanding. By integrating type inference mechanisms, the IDE could provide a mechanism to generate a more comprehensive list of referenced classes. For dynamic languages this list is still often not accurate or not even correct, because type inference in dynamic languages is not able to infer the types correctly in all situations [17].

If we analyze a software system dynamically we can provide a precise and correct list of references in a class. We propose a *reference view* that presents a list of referenced classes based on dynamic information. This view reveals to the developer which classes are referenced by the class under investigation, *e.g.*, in which methods and in which variables. Similar to our proposed *type view*, the reference view is enriched with quantitative information extracted from the dynamic information, *e.g.*, how often classes are referenced. When the developer selects a class or a specific method of this class, she immediately gets this reference view (see Figure 5 on the right) to learn to which other class the selected class or method communicates. In this reference view the developer also finds a class called `ColumnPanel` whose intention revealing name indicates that it contains the different columns the `OmniBrowser` holds. By looking at the places in `Column` where this class is referenced dynamically, we indeed learn that columns get added to an instance

of `ColumnPanel` when the browser is built up.

2.2 Integrating Dynamic Data in the IDE

As a proof-of-concept for our claims, we implemented Hermion as an enhancement to the Squeak Smalltalk IDE [22] to integrate the collection and presentation of dynamic information. Our solution is not restricted to Smalltalk or Squeak. The mechanisms described in this paper and the problem of gathering dynamic data in the IDE are applicable to any of the widely used IDEs such as Eclipse [12].

Our approach to seamless integration of dynamic information in IDEs is based on enriching the mechanisms and tools for navigating, browsing, viewing and editing source code currently existing in the IDE. The developer does not need to change her perspective or learn a new tool. She is just provided with new features in her familiar environment. In the case of the source code view where the developer browses, writes and edits source code, we enriched this view with additional visual information (based on the use of color-coding) that takes the runtime behavior of the application into account. In Squeak Smalltalk the developer browses source code in a class browser which currently displays a single method at a time. Figure 5 shows an example of a enriched method view for method `addCommandsToMenu:` of class `Column`. Variable accesses are shown in gray, method sends in blue. Each statement is also enriched with a clickable icon to access the gathered dynamic information.

In this enriched source code view we integrated the solutions to two of the mentioned problems in Section 2.1: message send (as well as sender) navigation and type view are integrated in the source code view while the reference view is placed to the right of the source code pane.

Such an enhanced IDE is not only useful to understand existing software systems, but shows its advantages also during development of new software. The developer can for instance permanently dynamically analyze her application under development and study in the IDE if a method *e.g.*, invoke the methods it is indented to invoke, or if classes communicate correctly to other classes at runtime. This gives the developer viable feedback about the dynamics of her program even during development.

3 Collecting Dynamic Data for the IDE

In this section we elaborate on the question: *How can we efficiently collect dynamic data in a running IDE?*

Traditional approaches to collecting dynamic data such as tracing are generally not efficient enough to provide dynamic data to be displayed at runtime in the IDE. Another drawback of these approaches is that they typically yield

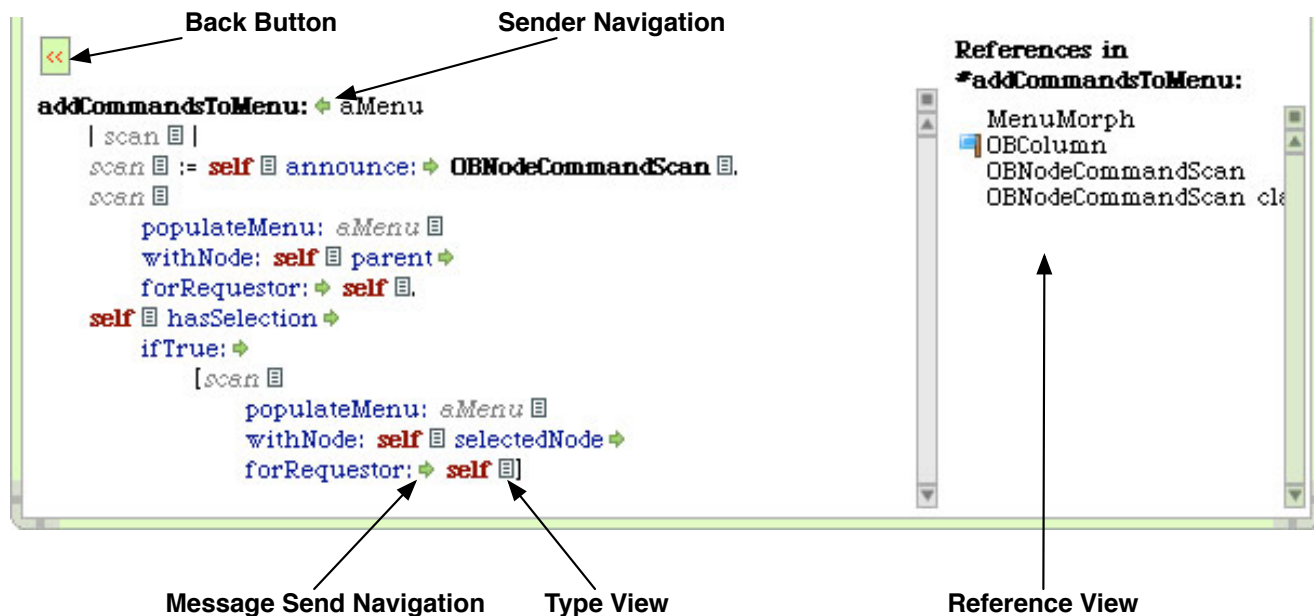


Figure 5. Enriched method source code view including a reference view in Hermion

large amounts of data, making it difficult to mine useful information [4]. Thus it is not really feasible to adopt these approaches as a basis for providing direct access to dynamic information in an IDE.

We exploit an approach known as *partial behavioral reflection* [24] that enables selective collection of a system’s runtime data. For instance, for many use cases it is not required to collect all method invocations occurring in a run of a system, because during the maintenance phase a developer is mainly interested in specific parts of the system, *e.g.*, in the classes implementing a specific feature. Furthermore, our approach does not generate a huge tracing file intended for offline postmortem analysis, but directly gathers and stores the data in the format in which it can be used without any offline analysis. This format is optimized for a fast lookup, *i.e.*, it contains precisely the information the IDE wants to present in the form of objects.

3.1 Partial Behavioral Reflection

With behavioral reflection we can intercept and modify the execution of programs, thus it is the basis of our dynamic analysis of applications in the IDE. A full behavioral reflection that reflects on every runtime event occurring in a program (*e.g.*, every message send), would be very inefficient. We thus need to narrow the focus of dynamic analysis and selectively reflect on specific parts of a program’s execution, *e.g.*, only on message sends occurring in classes we need to understand to solve a maintenance task [1].

The *Partial Behavioral Reflection* model of Reflex [24]

offers mechanisms to precisely select entities (*e.g.*, classes, methods, variables) and operations (*e.g.*, message sending, variable access). In particular, the approach of Reflex allows us to collect data at different levels of granularity, also at a sub-method level [7]. For instance, we can select what kind of operation occurrence in a method we want to reflect on (*e.g.*, only accesses to a specific variable) and can also precisely determine what information has to be reified about this operation (*e.g.*, only the value and name of a variable). Hence the Reflex model is well suited to efficiently gather runtime information about the program currently being developed [8]. For our work we use a variation of Reflex in Smalltalk, called Geppetto [7, 20].

The crucial entity in the Reflex model is the *link*. A link *causally connects* the base level of a system with its metalevel. Links are introduced in the binary of an application and upon occurrences of base-level operations they invoke methods on the corresponding metaobjects. During the installation of reflective behavior, the developer precisely selects the operations where links are to be installed and specifies the metaobject that the link will trigger. In our scenario, this metaobject implements a tracing mechanism which stores dynamic data captured at runtime in a format that is accessible by the IDE. This concept of base level, link and tracing metaobject is illustrated in Figure 6. The “x”s in the application code are message sends to be traced at runtime. For every message send we install a link that is triggered at runtime. This link reifies the runtime information about the occurring message send and passes this

information to the tracing metaobject which can then reason about this information, *e.g.*, store it in a database. For a more comprehensive treatment of partial behavioral reflection we refer the reader to the work of Tanter *et al.* [24].

To gather dynamic data required to realize the features presented in Section 2, we install three links for every method to be analyzed: The first link reifies receiver and arguments of every message send, the second link reasons about variable accesses by reifying name and value of a variable while the third link does the same, but for assignments to variables where also the new value of the variable gets reified. All these links pass their information to a metaobject where it is processed and then stored in a database and optimized for efficient retrieval so that it can readily display the information in the IDE at various places. We eliminate the need to gather large traces, thus minimizing the amount of dynamic data necessary for runtime analysis of the system.

We sum up by emphasizing the advantages of using partial behavioral reflection over trace-based approaches:

- *Precise specification of the kind and amount of dynamic information.* The developer controls when dynamic data is gathered (*i.e.*, where to install links and which metaobjects), even while the application under study is running. The selection is driven by a developer’s current focus, *e.g.*, comprehending a specific class or a certain feature of a system. The developer’s selections directly influence the performance and amount of collected data.
- *Instantaneous access to runtime information within the IDE.* Runtime information is processed while it is being gathered (*i.e.*, while the application is running). The selected source entities are instrumented to introduce the reflective behavior necessary for gathering dynamic information. Instrumenting a medium-sized class typically takes less than a second to perform. On subsequent execution of these entities, dynamic information is collected and displayed in the IDE. For instantaneous integration of runtime information in the IDE it is crucial to apply a fast data collecting technique such as partial behavioral reflection so that dynamic information is readily available in the IDE.

4 Validation

To validate that our approach is usable and scalable in real-world scenarios, we applied it to two case studies. As a first case study we choose Pier, a web-based content management system written in Smalltalk [19]. As a second case study, we selected the OmniBrowser framework introduced in Section 2, which forms part of the traditional IDE

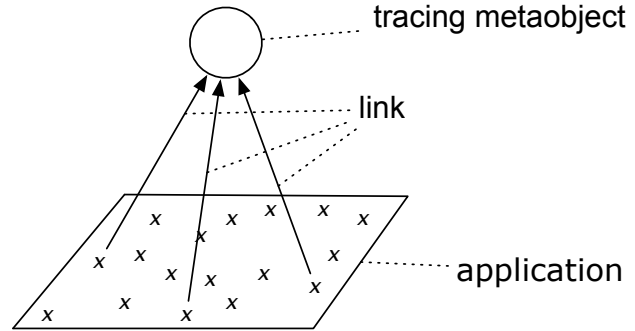


Figure 6. The link is invoking the metaobject upon occurrence of selected base-level operations

of Squeak Smalltalk 3.9 [22]. We performed a benchmark using these two applications to report on efficiency issues, *i.e.*, the factor by which the applications slow down the IDE when gathering dynamic data and when presenting dynamic information. Subsequently, we present results of a preliminary empirical evaluation where we asked developers unfamiliar with Pier and OmniBrowser to test our environment while working with these systems.

4.1 Case Studies: Pier and OmniBrowser

Pier. Pier consists of 104 classes, most of which are implemented generically. This makes it quite hard for a developer to understand for example how these generic classes interact, *i.e.*, which methods invoke which other methods of other classes at runtime. For that reason, we consider Pier as an appropriate case study to evaluate the usefulness of our enriched IDE. We asked developers unfamiliar with Pier if our enriched IDE supported them in gaining an understanding of the application and if it eases navigation of the application classes. Before we present the results of this empirical evaluation in Section 4.3, we first verified that it is technically possible to instrument Pier and to integrate the resulting dynamic data in our IDE.

In our initial experiment, we only instrumented the seven core classes of the Pier model package. While the dynamic data gathering was active in these core classes, we subjectively noticed just a negligible slowdown of the running Pier application.

In a second step, we instrumented the entire Pier application. This instrumentation only takes a few seconds to perform. Even though the application slowed down while dynamic data collection was active, we were instantaneously able to make use of the collected information in the IDE. We consider this as a good indication that it is possible to access and use dynamic information in an IDE while a sys-

	Not instrumented (ms)	Message sends only (ms)	Message sends and variables (ms)
Pier, core classes	42	141 (+236%)	297 (+607%)
OmniBrowser, core classes	625	2343 (+274%)	4146 (+564%)

Figure 7. Comparison of execution times for different levels of instrumentation for OmniBrowser and Pier.

tem under study is running.

OmniBrowser. The OmniBrowser framework currently consists of approximately 170 classes. We also used this framework to implement Hermion. With this experiment, we thus analyze the OmniBrowser framework dynamically within itself. By displaying the results of the dynamic analysis, we trigger the generation of new dynamic information which is instantly available in the IDE.

We instrumented all eleven classes in the model package of OmniBrowser. These classes form the core of OmniBrowser and are heavily used. We were able to successfully instrument all these classes to gather information about message sending and variable accesses occurring at runtime. Despite partial behavioral reflection instrumentation, the active browser in the IDE was still usable even during the collection of the dynamic information. We were able to immediately look at the dynamic information collected, *e.g.*, we could navigate the message sends or view the types of variables as explained in Section 2.2. We verified that our technique did not introduce flaws in the functionality of OmniBrowser, as we could still successfully run the comprehensive test suite of OmniBrowser. We were able to accurately collect all the required dynamic information even for methods heavily used at runtime.

4.2 Efficiency

As a performance test we compare the execution speed of the two applications, Pier and OmniBrowser, with and without dynamic data collection. Furthermore, we vary the extent of data collection: In a first scenario, we reify information about message sends, while in the second scenario we also reify variable accessing. As a reference, we measure the execution time for the same usage scenario while no data collection is active.

In Pier, we measure the time to display the standard start page of the web application. In the case of OmniBrowser, the usage scenario of the application is opening a new system browser which includes our integration of dynamic information. Figure 7 contains the results of this performance test. The slowdown introduced due to the collection of dynamic information is perceivable, in case of the Pier framework we get a slowdown of factor 7. It is noticeable that it makes a difference of more than factor 2 if only message sending or also variable accessing is reified. This rep-

resents clear evidence that the amount of information reified and collected at runtime has a high impact on performance. Limiting the amount of reifications, *e.g.*, only one kind of operation, is a good strategy for improving performance. The same is true for the coverage of static source artifacts: It is much more expensive to cover all classes, *i.e.*, to gather dynamic information about the usage of all classes in a package, than to select precisely the classes for which dynamic information is actually of interest.

In Section 5 we discuss the impact of these performance results on the usefulness of Hermion in a practical situation where it is used to comprehend or maintain a software system.

4.3 Preliminary Empirical Evaluation

We conducted an empirical evaluation with five Smalltalk developers to learn how useful and practical they consider Hermion. The subjects used Hermion to learn how the Pier application is structured. Concretely, we gave them the task to understand how a page is stored and displayed in Pier. None of the subjects had ever worked with Pier before, but were familiar with the Squeak IDE. We present qualitative feedback the developers gave us as well as our observations made while watching the subjects working with Hermion.

We presented the subjects with a list of concrete questions about how certain aspects of Pier (*e.g.*, displaying, adding a new or moving an existing page) are implemented or how certain classes interact with each other. Some of the questions they had to answer using dynamic information integrated in Hermion, the other questions they had to solve in the original Squeak IDE. As a start, we advised the subjects to look at and to dynamically analyze three core classes of Pier. From there on, they could start to understand how specific features of Pier are implemented and how those classes interact with the rest of the system.

Our own observations reveal that the answers of the subjects were significantly more precise when they were working with the dynamic information in the IDE. Concerning efficiency, our measures report that subjects could answer all of the questions in approximately 25 percent less time.

Student Feedback. We gave the subjects a questionnaire covering various aspects of Hermion. In this questionnaire subjects could rate the contribution of Hermion on execu-

tion overview, type information, navigational aid and program comprehension. We used a five-level Likert Scale for the ratings ('1' means strongly disagree, '5' strongly agree that there is an effect of *Hermion*). Furthermore, we allowed the subjects to also write free text on the questionnaire to give us qualitative feedback that cannot be expressed with a Likert Scale. In the following we present the results from the questionnaire:

Execution overview. Subjects considered it as very useful (average rating value above 4) to see precisely what parts of the code were executed when *Pier* is displaying a page. Hence they could focus on the relevant classes and methods necessary to understand how pages are stored and displayed, namely those that get executed at runtime.

Type Information. Subjects stressed that readily available type information makes it much easier to read the source code of a dynamically-typed language (average rating was 3.5).

Navigational aid. Students reported that they could more efficiently locate the methods that were executed by a specific method under investigation when dynamic information is available (average rating was close to 4).

Program comprehension. Consequently, they found that dynamic information accelerated their understanding of the application, as they no longer had to guess what methods would be executed at runtime. In general, all subjects considered the total impact on program comprehension as high. They reported that it took them much more time to comprehend the system when they had to use the original *Squeak* IDE not providing dynamic information (average rating was above 3).

We recognize the need to verify these results with a controlled experiment in which more subjects participate to obtain more evidence. As the results of this preliminary empirical evaluation are promising, we plan to perform a more complete validation with a wider range (*i.e.*, more subjects) and scope (*i.e.*, more tasks for the developer to perform). For instance, we give the same task of *e.g.*, understanding a specific feature of a software system once to subjects working with *Hermion* and once to subjects working without. Then we compare the completeness, efficiency and correctness of the results the subjects came up with to see if using a dynamic IDE is indeed beneficial.

5 Discussion

We discuss some specific restrictions and limitations of our current approach and possible solutions to overcome these limitations. First, we discuss the existing efficiency issues. Subsequently, we elaborate on the coverage and completeness problem of dynamic information in depth.

Efficiency Issues. In Section 4.2 we spotted certain efficiency issues when analyzing applications dynamically from within the IDE. In the following, we discuss the impact of these results to the applicability of our approach in real-world scenarios and elaborate on optimizations.

During development and maintenance, the execution performance of an application is not crucial. A debugging session in step mode for instance, takes much more time to perform and the developer loses all information visible in the debugging session when she reverts to the source code browsing in the IDE. Moreover, a debugger only shows one distinct run of a software system and neglects information of all other runs. To understand a software system means to effectively navigate the system. For this reason, the dynamic data collection, although not yet fully optimized, still contributes to program comprehension. We consider that the performance of dynamic data gathering compared to the gained benefit does not constitute an obstacle to applying this approach for the task of program comprehension.

Nonetheless, we identified a need to work on these performance issues in the future. Based on our experience with the experiments, we propose a possible *best practice* for efficiently collecting dynamic information is to only instrument classes which we want to understand to the level of detail required for *e.g.*, a given maintenance task. The key advantage of our approach is to provide the developer with the flexibility to choose and define trade-offs of information over performance as she sees necessary.

Coverage and Completeness. An open issue is the coverage aspect of dynamic information. The completeness of dynamic information always depends on what is actually executed at runtime. Normally, a full coverage of all behavior of a system is not achievable [1], instead we only cover parts of the system by *e.g.*, exercising some specific features. This means that we can only highlight the type information for concrete executions of a system, we cannot show all types that can be theoretically stored in a variable. But this need not be viewed as a shortcoming, but rather an advantage. In particular, when maintaining software, *e.g.*, correcting defects, the developer is often interested in understanding specific executions of the application, namely those that are broken and need to be revised.

The difference concerning completeness to analyzing the system in a particular debugging session is that in *Hermion* we permanently present all so far gathered dynamic information, this information is the result of various executions of a system and is not volatile, *i.e.*, it does not disappear as in the debugger. Furthermore, debuggers included in IDEs such as Eclipse [12] or *Squeak* [22] indeed help developers analyzing the runtime of a system, but provide little to support navigation of a source space. In a debugging session, a programmer focuses on a very specific run of the system, observing and analyzing a slice of the program to discover

the cause for a specific defect [25, 27]. Our work aims at easing the understanding and navigation of a whole software space in general, the dynamic information thereby presented in the IDE is not restricted to a specific run of a program. Instead we merge dynamic information in the static perspective on source code presented in the IDE. Löwe *et al.* [15] followed a similar approach by merging information from static analysis with information from dynamic analysis to generate visualizations. However, their work is not integrated into the development environment.

6 Related Work

There is a large body of research in the area of dynamic analysis as well as in the area of development environments, but only a few publications connect these two topics. We present some important work on static as well as dynamic analysis. Subsequently we report on the state of the art in development environment research.

Static Type Inference. With static analysis, especially with static type inference [2, 14, 16], it is possible to gain insights into the types that variables assume at runtime. However, static type inference is a computationally expensive task and cannot in all cases give a precise and concise result in the context of dynamic object-oriented languages [17]. Rapi-cault *et al.* [17] propose dynamic type inference based on message-passing control, but they adapt the Smalltalk compiler in order to gather the information. This makes their approach less portable.

Dynamic Analysis. Many of today’s tracing tools such as those based on method wrappers [3] implement techniques allowing selective instrumentation of the source code based on criteria such as package boundary or on individual selection of methods. Unlike partial behavioral reflection, only a few tracing techniques such as that described in the work of Ducasse *et al.* [10] consider sub-method elements such as variable assignments. Mechanisms based on bytecode modification or virtual machine instrumentation support reification of sub-method elements. However, these mechanisms are often low level and language specific. The intention of partial behavioral reflection is to introduce a layer of abstraction between the low level details of the implementation language and the concept of capturing and reifying high level runtime events such as message sends and variable accesses [7].

Dynamic analyses based on tracing mechanisms focus traditionally on capturing a call tree of message sends but many approaches do not bridge the gap between dynamic behavior and the static structure of a program [11, 13, 26]. Our work aims at incorporating the information obtained through dynamic analyses into the IDE and thus connecting the static structure with the dynamic behavior of the application.

Dynamic Information in the IDE. To the best of our knowledge, there is not much research in the area of integrating results of dynamic analyses into development environments. However, the work of Reiss [18] visualizes the dynamics of Java programs in real time, *e.g.*, the number of message sends received by a class. These visualizations are not tightly integrated in an IDE though, but are provided by a separated tool. Therefore, it is not directly possible to use these analyses while working with source code. We consider it as crucial to incorporate knowledge about the dynamics of programs into the development environment to ease navigating within the source space. Löwe *et al.* [15] followed a similar approach by merging information from static analysis with information from dynamic analysis to generate visualizations. However, their work is not integrated into the development environment.

Other approaches ease and support the navigation of large software systems by other means than program analysis. For instance, NavTracks [21] keeps track of the navigation history of software developers. Using this history, NavTracks forms associations between related source files (*e.g.*, class files) and can hence present related entities to the developers.

Our approach also focuses on hyper-linking code elements that communicate to each other at runtime, *e.g.*, methods that send messages to other objects or classes referencing other classes. These hyper-links ease and accelerate the navigation, because they reflect how static source artifacts actually integrate which each other at runtime [9].

7 Conclusions

In this paper we addressed the restrictions and limitations current IDEs impose on developers faced with the task of understanding an object-oriented system, in particular if implemented in a dynamically-typed language. We motivated our work by addressing the questions:

- *How do we integrate dynamic information into an IDE’s browsing and navigating mechanisms?* We identified the types of dynamic information and how this can be integrated into the workflows and views of an IDE to support developers to gain an understanding of a large software space through enriched views and precise navigation.
- *How can we efficiently collect dynamic data of a running application in the IDE?* We introduced an approach based on partial behavioral reflection to efficiently and selectively collect dynamic data at different levels of granularity (including sub-method data about variable accesses) at runtime.

We implemented an experimental IDE called *Hermion* which integrates dynamic information and validated its approach by applying it to two real-world applications and measured the efficiency of the dynamic data gathering using these applications. Moreover, we let developers experiment with *Hermion* to assess its benefit on navigation and understanding of a software system. This initial empirical evaluation revealed promising results which motivates us to conduct a more extensive empirical evaluation to accumulate more evidence for the usefulness of *Hermion*.

In the future, we plan to enhance the Eclipse IDE with dynamic information so as to achieve a wide-reaching practical relevance for our work.

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

References

- [1] 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)*, number 1687 in LNCS, pages 216–234, Heidelberg, 1999. Springer Verlag.
- [2] A. H. Borning and D. H. Ingalls. A type declaration and inference system for Smalltalk. In *Proceedings POPL '82*, Albuquerque, NM, 1982.
- [3] J. Brant, B. Foote, R. Johnson, and D. Roberts. Wrappers to the rescue. In *Proceedings European Conference on Object Oriented Programming (ECOOP 1998)*, volume 1445 of LNCS, pages 396–417. Springer-Verlag, 1998.
- [4] B. Cornelissen. Dynamic analysis techniques for the reconstruction of architectural views. In *Proceedings of the 14th Working Conference on Reverse Engineering (WCRE)*. IEEE, 2007. To appear.
- [5] S. Demeyer, S. Ducasse, K. Mens, A. Trifu, and R. Vasa. Report of the ECOOP'03 workshop on object-oriented reengineering, 2003.
- [6] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *Proceedings of 15th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*, pages 166–178, New York NY, 2000. ACM Press. Also appeared in ACM SIGPLAN Notices 35 (10).
- [7] M. Denker, S. Ducasse, A. Lienhard, and P. Marschall. Submethod reflection. *Journal of Object Technology*, 6(9):231–251, Oct. 2007.
- [8] M. Denker, O. Greevy, and M. Lanza. Higher abstractions for dynamic analysis. In *2nd International Workshop on Program Comprehension through Dynamic Analysis (PCODA 2006)*, pages 32–38, 2006.
- [9] M. Desmond, M.-A. Storey, and C. Exton. Fluid source code views. In *ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 260–263, Washington, DC, USA, 2006. IEEE Computer Society.
- [10] S. Ducasse, T. Gîrba, and R. Wuyts. Object-oriented legacy system trace-based logic testing. In *Proceedings of 10th European Conference on Software Maintenance and Reengineering (CSMR'06)*. IEEE Computer Society Press, 2006.
- [11] A. Dunsmore, M. Roper, and M. Wood. Object-oriented inspection in the face of delocalisation. In *Proceedings of ICSE '00 (22nd International Conference on Software Engineering)*, pages 467–476. ACM Press, 2000.
- [12] Eclipse Platform: Technical Overview, 2003. <http://www.eclipse.org/whitepapers/eclipse-overview.pdf>.
- [13] A. Hamou-Lhadj, E. Braun, D. Amyot, and T. Lethbridge. Recovering behavioral design models from execution traces. In *Proceedings IEEE European Conference on Software Maintenance and Reengineering (CSMR 2005)*, pages 112–121, Los Alamitos CA, 2005. IEEE Computer Society Press.
- [14] R. E. Johnson. Type-checking Smalltalk. In *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, volume 21, pages 315–321, Nov. 1986.
- [15] W. Löwe, A. Ludwig, and A. Schwind. Understanding software - static and dynamic aspects. In *17th International Conference on Advanced Science and Technology*, 2001.
- [16] J. Pleviak and A. A. Chien. Precise concrete type inference for object-oriented languages. In *Proceedings of OOPSLA '94*, pages 324–340, 1994.
- [17] P. Rapicault, M. Blay-Fornarino, S. Ducasse, and A.-M. Dery. Dynamic type inference to support object-oriented reengineering in smalltalk, 1998. Proceedings of the ECOOP '98 International Workshop Experiences in Object-Oriented Reengineering, abstract in Object-Oriented Technology (ECOOP '98 Workshop Reader forthcoming LNCS).
- [18] S. P. Reiss. Visualizing Java in action. In *Proceedings of SoftVis 2003 (ACM Symposium on Software Visualization)*, pages 57–66, 2003.
- [19] L. Renggli. Pier — the meta-described content management system. European Smalltalk User Group Innovation Technology Award, Aug. 2007.
- [20] D. Röthlisberger, M. Denker, and É. Tanter. Unanticipated partial behavioral reflection. In *Advances in Smalltalk — Proceedings of 14th International Smalltalk Conference (ISC 2006)*, volume 4406 of LNCS. Springer, 2007.
- [21] J. Singer, R. Elves, and M.-A. Storey. Navtracks: Supporting navigation in software maintenance. In *International Conference on Software Maintenance (ICSM'05)*, pages 325–335, sep 2005.
- [22] Squeak home page. <http://www.squeak.org/>.
- [23] T. Systä. On the relationships between static and dynamic models in reverse engineering java software. In *Working Conference on Reverse Engineering (WCRE99)*, Oct. 1999.
- [24] É. Tanter, J. Noyé, D. Caromel, and P. Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In *Proceedings of OOPSLA '03, ACM SIGPLAN Notices*, pages 27–46, nov 2003.
- [25] M. Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.
- [26] N. Wilde and R. Huitt. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, SE-18(12):1038–1044, Dec. 1992.
- [27] A. Zeller. Program analysis: A hierarchy. In *Proceedings of the ICSE 2003 Workshop on Dynamic Analysis*, 2003.