Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets, © JOT 2011
Online at http://www.jot.fm.

Modeling Domain-Specific Profilers

Jorge Ressia[b] Alexandre Bergel[a] Oscar Nierstrasz[b]
Lukas Renggli

a. PLEIAD Lab, Department of Computer Science (DCC), University of Chile
pleiad.dcc.uchile.cl

b. Software Composition Group, University of Bern, Switzerland
scg.unibe.ch

Abstract Domain-specific languages and models are increasingly used within general-purpose host languages. While traditional profiling tools perform well on host language code itself, they often fail to provide meaningful results if the developers start to build and use abstractions on top of the host language. In this paper we motivate the need for dedicated profiling tools with three different case studies. Furthermore, we present an infrastructure that enables developers to quickly prototype new profilers for their domain-specific languages and models.

Keywords Programming Languages, Reflection, Domain-specific Languages, Profilers.

1 Introduction

Recent advances in domain-specific languages and models reveal a drastic change in the way software is being built. The software engineering community has seen a rapid emergence of domain-specific tools, ranging from tools to easily build domain-specific languages [Vis04], to transform models [TCJ10], to check source code [RDGN10a], and to integrate development tools [RGN10].

While research on domain-specific languages has made consistent progress in language specification [DKV00], implementation [CM09], evolution [FP06] and verification [KR08], little has been done to support profiling. We consider profiling to be the activity of recording and analyzing program execution. Profiling is essential for analyzing transient run-time data that otherwise would be difficult to harvest and compare. Code profilers commonly employ execution sampling as the way to obtain dynamic run-time information. Unfortunately, information extracted by regularly sampling the call stack cannot be meaningfully used to profile a high-level domain built on top of the standard language infrastructure. Specialized domains need specialized profilers.

Jorge Ressia, Alexandre Bergel, Oscar Nierstrasz, Lukas Renggli. *Modeling Domain-Specific Profilers*. In Journal of Object Technology, vol. N, 2011, pages M:1–0. doi:10.5381/jot.201Y.VV.N.aN

Let us consider the example of the Mondrian visualization engine (details follow in Section 2.1). Mondrian models visualizations as graphs, *i.e.,* in terms of nodes and edges. One of the important performance issues we recently faced is the refresh frequency: nodes and edges were unnecessarily refreshed too often. Standard code profilers did not help us to localize the source of the problem since they are just able to report the share of time the CPU spends in the method `displayOn:` of the classes `MONode` and `MOEdge`. The problem was finally resolved by developing a custom profiler that could identify which nodes and edges were indeed refreshed too often. This domain-specific profiler was able to exploit knowledge of Mondrian's domain concepts to gather and present the needed information.

We argue that there is a need for a general approach to easily develop specialized profilers for domain-specific languages and tools. A general approach must offer means to (i) *specify* the domain concepts of interest, (ii) *capture* the relevant information from the run-time execution, and (iii) *present* the results to the developer.

In this paper we detail MetaSpy, an event-based approach for domain-specific profiling. With MetaSpy, a developer specifies the events of interest for a given domain. A profiler captures domain information either by subscribing to existing application events, or by using a reflective layer to transparently inject event emitters into the domain code. The collected events are presented using graph-based visualizations.

The contributions of this paper are: (1) the identification of the need for domain-specific profilers, (2) the presentation of three real-world case-studies where domain-specific profilers helped to significantly improve performance and correctness of domain-specific code, (3) the presentation of an infrastructure for prototyping domain-specific profilers, and (4) a simple but expressive mechanism to express causality between events. MetaSpy was introduced in an earlier publication [BNRR11]. This article extends our previous work by considering the causality between low-level events.

Outline. The remainder of this paper is structured as follows: Section 2 illustrates the problems of using a general-purpose profiler on code that is built on top of a domain-specific language. Section 3 introduces our approach to domain-specific profiling. Section 4 demonstrates how our approach solves the requirements of domain-specific profilers with three use cases. Section 5 demonstrates how our approach deals with event causality. Section 6 presents our infrastructure to implement domain-specific profilers. Section 7 presents an analysis on the performance impact of MetaSpy. Section 8 summarizes the paper and discusses future work.

2 Shortcomings of Standard Profilers

Current application profilers are useful to gather runtime data (*e.g.,* method invocations, method coverage, call trees, code coverage, memory consumption) from the static code model offered by the programming language (*e.g.,* packages, classes, methods, statements). This is an effective approach when the low-level source code has to be profiled.

However, traditional profilers are far less useful for a domain different than the code model. In modern software there is a significant gap between the model offered by the execution platform and the model of the actually running application. The proliferation of meta-models and domain-specific languages brings new abstractions that map to the underlying execution platform in non-trivial ways. Traditional profiling tools fail to display relevant information in the presence of such abstractions.

MetaSpy[1] and the examples presented in this paper are implemented in the Pharo Smalltalk[2] programming language, an open-source Smalltalk [GR83]. Readers unfamiliar with the syntax of Smalltalk might want to read the code examples aloud and interpret them as normal sentences: An invocation of a method named `method:with:`, using two arguments looks like: `receiver method: arg1 with: arg2`. A method with no arguments looks like `receiver method`. Other syntactic elements of Smalltalk are: the dot to separate statements: `statement1. statement2`; square brackets to denote code blocks or anonymous functions: `[ statements ]`; and single quotes to delimit strings: `'a string'`. The caret `^` returns the result of the following expression.

2.1 Difficulty of profiling a specific domain

This section illustrates three shortcomings of traditional profiling techniques when applied to a specific domain.

CPU time profiling

Mondrian [MGL06] is an open and agile visualization engine. Mondrian describes a visualization using a graph of (possibly nested) nodes and edges. In June 2010 a serious performance issue was raised[3]. Tracking down the cause of the poor performance was not trivial. We first used a standard sample-based profiler.

Execution sampling approximates the time spent in an application's methods by periodically stopping a program and recording the current set of methods under executions. Such a profiling technique is relatively accurate since it has little impact on the overall execution. This sampling technique is used by almost all mainstream profilers, such as JProfiler, YourKit, xprof [GH92], and hprof.

MessageTally, the standard sampling-based profiler in Pharo Smalltalk, textually describes the execution in terms of CPU consumption and invocation for each method of Mondrian:

```
54.8% {11501ms} MOCanvas>>drawOn:
  54.8% {11501ms} MORoot(MONode)>>displayOn:
   30.9% {6485ms} MONode>>displayOn:
      | 18.1% {3799ms} MOEdge>>displayOn:
         ...
      |  8.4% {1763ms} MOEdge>>displayOn:
      |    | 8.0% {1679ms} MOStraightLineShape>>display:on:
      |    |  2.6% {546ms} FormCanvas>>line:to:width:color:
        ...
   23.4% {4911ms} MOEdge>>displayOn:
        ...
```

We can observe that the virtual machine spent about 54% of its time in the method `displayOn:` defined in the class `MORoot`. A root is the unique non-nested node that contains all the nodes of the edges of the visualization. This general profiling information says that rendering nodes and edges consumes a great share of the CPU time, but it does not help in pinpointing which nodes and edges are responsible for the time spent. Not all graphical elements equally consume resources.

Traditional execution sampling profilers center their result on the frames of the execution stack and completely ignore the identity of the object that received the

[1] http://scg.unibe.ch/research/bifrost/metaspy/
[2] http://www.pharo-project.org/
[3] http://forum.world.st/Mondrian-is-slow-next-step-tc2257050.html#a2261116

method call and its arguments. As a consequence, it is hard to track down which objects cause the slowdown. For the example above, the traditional profiler says that we spent 30.9% in `MONode>>displayOn:` without saying which nodes were actually refreshed too often.

Coverage

PetitParser is a parsing framework combining ideas from scannerless parsing, parser combinators, parsing expression grammars and packrat parsers to model grammars and parsers as objects that can be reconfigured dynamically [RDGN10b]. A number of grammars have been implemented with PetitParser, including Java, Smalltalk, XML and SQL.

Let us consider a Java grammar in PetitParser which is defined in 210 host language methods. The `if` statement parsing rule is defined as follows:

```
PPJavaSyntax>>ifStatement
   ^ ('if' asParser token , conditionalExpression , statement) ,
     ('else' asParser token , statement) optional
```

These methods build a graph of objects describing the grammar. It would be useful to establish how much of the grammar is actually exercised by a set of test cases to identify untested productions.

Traditional coverage tools focus on the source code artifacts instead of domain-specific data. They assess the coverage of the application source code by listing the methods and source lines covered by an execution.

In our case all methods and all lines of code are covered to build the grammar, but some parts of the resulting graph are not exercised by the tests. This is why we are unable to analyze the parsing and production coverage of this grammar with traditional tools.

Causality

Traditional profilers report events based on the run-time structure of the application. A run-time profiling report is typically structured as a tree in which indentation indicates nested calls. The sequence of methods executed is reported in a linear fashion: A method `m1` that is executed before `m2` will be reported as `m1` above `m2`.

This hardcoded presentation is disconnected from the profiled model. When considering the Mondrian example, the sequence of `displayOn:` methods executed cannot be related to the order in which the nodes are rendered. In PetitParser the order does not represent the sequence in which the parsers are activated.

Understanding the sequence of a large number of events is challenging at best. Unfortunately, textual searching over a log file discards the structure of the model by solely operating on what the user decided to log. Textual search is a rather limited technique, even though it is commonly employed [Nag10].

2.2 Requirements for domain-specific profilers

The three examples given above are representative. They illustrate the gap between a particular domain and the source code model. We argue that to efficiently profile an arbitrary domain, the following requirements need to be fulfilled:

- *Specifying the domain.* Being able to effectively designate the objects relevant for the profiling is essential. In Mondrian we are interested in the different nodes and the invocation of the `displayOn:` methods, rather than focusing on the implementing classes. In PetitParser we are interested in how often and if at all production objects are activated by a given input.
- *Capturing domain-related events.* Relevant events generated by the domain have to be monitored and recorded to be analyzed during or after the execution. An event represents a particular change or action triggered by the domain being profiled. Whereas the class `MOGraphElement` and its subclasses total more than 263 methods, only fewer than 10 methods are related to displaying and computing shape dimensions.
- *Effectively and concisely presenting the necessary information.* The information collected by traditional profilers is textual and targets method invocation. A method that invokes another will be located below it and indented. Moreover, each method frame represented has a class name and a method name, which completely ignores the identity of the object and arguments that are part of the call. Collected information has to be presented in such a way as to bring the important metrics and domain object composition into the foreground.
- *Relation between events.* An important and recurrent task in profiling is to understand the meaning of a sequence of emitted events. This is necessary when a developer wants to understand the causes of a suboptimal execution. Captured events have to be causally related to each other to trace high level operations. Since such relation between events cannot be enforced by the domain, it has to be reconstructed upon reception. Captured events have to be presented in a sequence that reflects the meaning of the model operations.
- *Browsing events.* The number of events generated by a typical application execution may easily skyrocket. Diving into those events is often the only way to understand the reason for suboptimal execution. Navigating through and giving a meaning to such a large number of events requires adequate tools that are aware of the model used to generate the events.

Common code profilers employ execution sampling as the way to cheaply obtain dynamic information. Unfortunately, information extracted when regularly sampling the method call stack cannot be used to profile a domain other than the source code model.

3 MetaSpy in a Nutshell

In this section we will present MetaSpy, a framework that supports building domain-specific profilers. The key idea behind MetaSpy is to provide domain-specific events that can later be used by different profilers with different objectives.

Figure 1 shows a class diagram of MetaSpy. There are two main abstractions: the instrumentation strategies and the domain-specific profilers.

An instrumentation strategy is responsible for adapting a domain-specific model and triggering specific actions in the profiler when certain events occur. A profiler models a domain-specific profiling requirement by composing multiple instrumentation strategies.

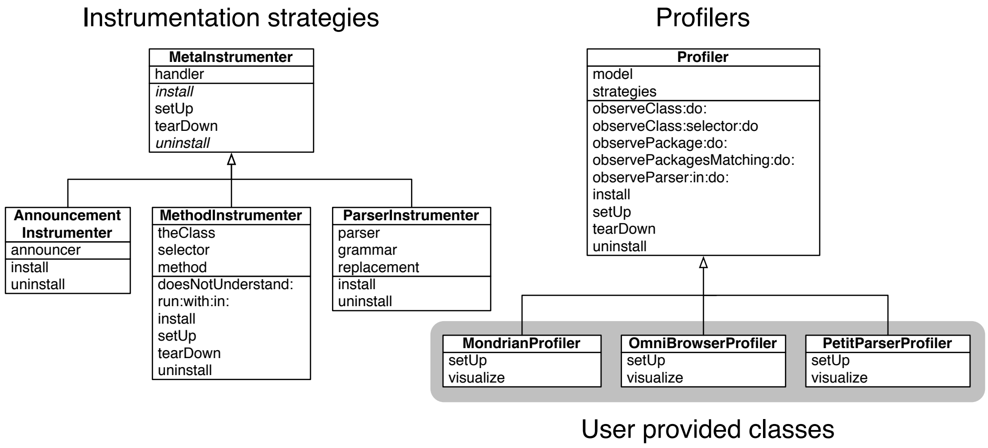

Figure 1 – The architecture of the MetaSpy profiler framework.

Some instrumentation strategies work by registering to existing events of the application domain. Other instrumentation strategies intercept the system by meta-programming, *i.e.,* conventional instrumentation. Installing an instrumentation strategy activates it and its associated events, while uninstalling deactivates them.

Some of the instrumentation strategies provided by MetaSpy are:

- *Announcement Instrumenter* dispatches events satisfying a particular condition from the announcer (subject) to the external profiler (observer).
- *Method Instrumenter* triggers an event whenever a specific method is invoked on any instance of a specified class.
- *Object Instrumenter* triggers an event whenever a specific method is invoked on a particular object. This is called object-specific profiling.
- *Parser Instrumenter* triggers an event whenever a specific grammar production is activated. This is a very specific instrumentation strategy only working with PetitParser productions.

Other dedicated instrumentation strategies can be implemented by adhering to the same interface.

Profilers are responsible for modeling the domain-specific behavior to profile the main abstractions in each domain. The abstract `Profiler` class models the behavior of a general profiler. Subclasses are instantiated with a domain-specific model and implement the set-up and tear-down of one or more instrumentation strategies into the model. Furthermore, they define how and what data is collected when the instrumented model is exercised. To actually instrument the model and start collecting events the method `install` is used. Similarly, to remove all instrumentation from the model, `uninstall` is used. Both methods dispatch the requests to the respective instrumentation strategies using the current model.

Each profiler is responsible for presenting the collected data in the method `visualize`. Depending on the nature of the data, this method typically contains a Mondrian [MGL06] or Glamour [Bun09] script, or a combination of both. Mondrian is a visualization engine to depict graphs of objects in configurable ways. Glamour is a browser framework to script user interfaces for exploratory data discovery.

Next, we will show real-world examples of domain-specific profilers.

4 Validation

In this section we will analyze three case studies from three different domains. We will show how MetaSpy is useful for expressing the different profiling requirements in terms of events. We will also demonstrate how MetaSpy fulfills the domain-specific profiling requirements, namely specifying, capturing, and presenting domain-specific information.

For each case study we show the complete code for specifying and capturing events. We do not show the code for visualizing the results, which typically consists of 20–50 lines of Mondrian or Glamour script code. We use the Mondrian visualization tool to visually and interactively report profiles. In Section 4.1 we also consider Mondrian as the profiling subject. We therefore visualize using Mondrian the profile of Mondrian itself.

4.1 Case Study: Displaying invocations

A Mondrian visualization may comprise a great number of graphical elements. A refresh of the visualization is triggered by the operating system, resulting from user actions such as a mouse movement or a keystroke. Refreshing the Mondrian canvas iterates over all the nodes and edges and triggers a new rendering. Elements that are outside the window or for which their nesting node has an active bitmap in the cache should not be rendered.

A graphical element is rendered when the method `display:on:` is invoked. Monitoring when these invocations occur is key to having a global view of what should be refreshed.

Capturing the events

The MetaSpy framework is instantiated to create the `MondrianProfiler` profiler.

```
Profiler subclass: #MondrianProfiler
  instanceVariableNames: 'actualCounter previousCounter'
```

`MondrianProfiler` defines two instance variables to monitor the evolution of the number of emitted events: `actualCounter` keeps track of the current number of triggered events per event type, and `previousCounter` stores the number of event types that were recorded before the previous visualization step.

```
MondrianProfiler>>initialize
   super initialize.
   actualCounter := IdentityDictionary new.
   previousCounter := IdentityDictionary new
```

The installation and instrumentation of Mondrian by MetaSpy is realized by the `setUp` method:

```
MondrianProfiler>>setUp
   self model root allNodes do: [ :node |
      self
```

```
        observeObject: node
        selector: #displayOn:
        do: [ :receiver :selector :arguments |
           actualCounter
              at: receiver
              put: ((actualCounter at: receiver ifAbsent: [ 0 ]) + 1) ] ]
```

All the nodes obtained from the root of the model object are "observed" by the framework. At each invocation of the `displayOn:` method, the block given as parameter to `do:` is executed with the object receiver on which `displayOn:` is invoked, the selector name and the argument. This block updates the number of displays for each node of the visualization.

Specifying the domain

The instrumentation described in the `setUp` method is only applied to the model specified in the profiler. This model is an object which models the domain to be profiled, in this case a Mondrian visualization. The instrumentation is only applied to all nodes in this visualization. Only when these nodes receive the message `displayOn:`, the actual counter is incremented. This object-specific behavior is possible due to the use of a reflection framework called Bifröst [RRGN10].

Presenting the results

The profiling of Mondrian is visualized using Mondrian itself. The `visualizeOn:` method generates the visualization given in Figure 2.

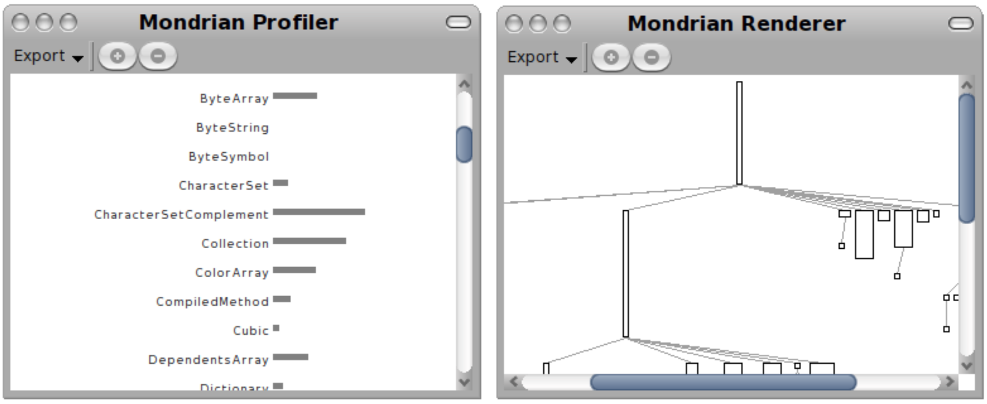

Figure 2 – Profiling (left) the System Complexity visualization (right).

One important point of `visualizeOn:` is to regularly update the visualization to be able to see the evolution of the domain events over time.

Figure 2 gives a screenshot of a visualization and the profiler. The right-hand side is an example of the *System Complexity* visualization [LD03] of the collection class hierarchy. System complexity is a typical usage of Mondrian, which exhibits the problem mentioned in Section 2.1.

The left-hand side shows the profiler applied to the visualization on the right-hand side. The profiler lists all the classes visualized in the system complexity. The profiler associates to each class a horizontal bar indicating the number of times the corresponding node in the system complexity has been displayed. This progress bar widens upon node refresh. The system complexity visualization remains interactive, even when being profiled. Selecting, dragging and dropping nodes refreshes the visualization, thus increasing the displayed progress of the corresponding nodes. This profile helps in identifying unnecessary rendering. Thanks to this profiler, we identified a situation in which nodes were refreshing without receiving user actions which caused the sluggish rendering. More precisely, edges were constantly refreshed, even when they were not visible. The profiler is uninstalled when the profiled Mondrian visualization is closed.

4.2 Case Study: Events in OmniBrowser

OmniBrowser [BDPW08] is a framework for defining and composing new browsers, *i.e.,* graphical list-oriented tools to navigate and edit elements from an arbitrary domain. In the OmniBrowser framework, a browser is described by a domain model specifying the domain elements that can be navigated and edited, and a metagraph specifying the navigation between these domain elements. Nodes in the metagraph describe states the browser is in, while edges express navigation possibilities between those states. The OmniBrowser framework then dynamically composes widgets such as list menus and text panes to build an interactive browser that follows the navigation described in the metagraph.

OmniBrowser uses announcements for modeling the interaction events of the user with the IDE. A very common problem is to have certain announcements be triggered too many times for certain scenarios. This behavior impacts negatively the performance of the IDE. Moreover, in some cases odd display problems are produced which are very hard to track down.

Capturing the events

To profile this domain-specific case we implemented the class `OmniBrowserProfiler`:

```
Profiler subclass: #OmniBrowserProfiler
   instanceVariableNames: 'actualCounter'
```

The instrumentation in the `setUp` method counts how many times each announcement was triggered.

```
OmniBrowserProfiler>>setUp
   self
      observeAnnouncer: self model announcer
      do: [ :ann |
         actualCounter
            at: ann class
            put: (actualCounter at: ann class ifAbsent: [ 0 ]) + 1 ]
```

Specifying the domain

We specify the entities we are interested in profiling by defining the model in the profiler. The model is an instance of the class `OBSystemBrowser`, the entry point of OmniBrowser. All OmniBrowser instances have an internal collaborator named *announcer* which is responsible for the signaling of announcements. This is the object used by the profiler to catch the announcement events.

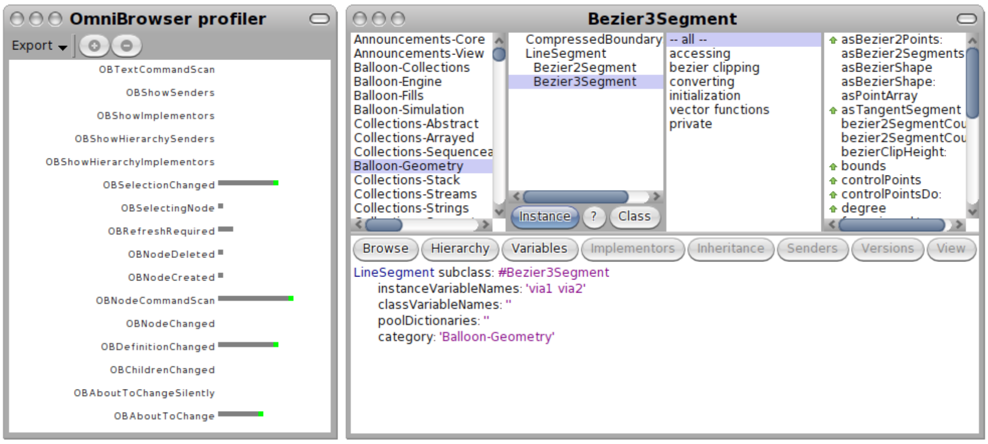

Figure 3 – Profiling (left) an OmniBrowser instance (right).

Presenting the results

A Mondrian visualization was implemented to list the type and the number of announcements triggered (cf. Figure 3).

4.3 Case Study: Parsing framework with PetitParser

Rigorous test suites try to ensure that each part of the grammar is covered by tests and is well-specified according to the respective language standards. Validating that each production of the grammar is covered by the tests is a difficult activity. As mentioned previously, traditional tools of the host language work at the method and statement level and thus cannot produce meaningful results in the context of PetitParser where the grammar is modeled as a graph of objects.

Capturing the events

With MetaSpy we can implement the grammar coverage with a few lines of code. The instrumentation happens at the level of the primitive parser objects. The method `observeParser:in:` wraps the parser object with a handler block that is called for each activation of the parser.

```
1 PetitParserProfiler>>setUp
2    self model allParsers do: [ :parser |
3       self observeParser: parser in: self grammar do: [
4          counter
5             at: parser
6             put: (counter at: parser ifAbsent: [ 0 ]) + 1 ] ]
```

Line 2 iterates over all primitive parser objects in the grammar. Line 3 attaches the event handler on Lines 4–6 to each parser in the model. The handler then counts the activations of each parser object when we run the test suite of the grammar.

Specifying the domain

The domain in this case is an instance of the grammar that we want to analyze. Such a grammar may be defined using hundreds of interconnected parser objects.

Presenting the results

This provides us with the necessary information to display the grammar coverage in a visualization such as that shown in Figure 4.

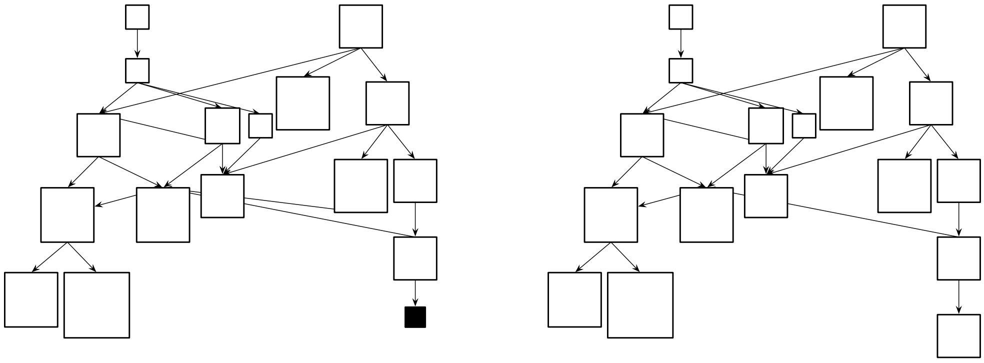

Figure 4 – Visualization of the production coverage of an XML grammar with uncovered productions highlighted in black (left); and the same XML grammar with updated test coverage and complete production coverage (right). The size of the nodes is proportional to the number of activations when running the test suite on the grammar.

5 Identifying Event Causality

Mondrian visualizes graphs of nodes and edges. Apart from the edges displayed in the visualization, nodes can support other relationships: nodes might be nested within each other, *i.e.,* when a parent is moved, its children have to be redrawn; nodes might have interaction dependencies, when one node is selected another one is updated; nodes might have caching dependencies, when one node changes dependent nodes need to invalidate their state; and so on.

The use of log files to identify such dependencies may indeed be successful [YD00]. However, producing an adequate log file that covers all the different situations requires a significant amount of work and good system knowledge.

We favor prototyping of lightweight tools to address the possible problems on the spot.

5.1 Expressing causality

According to the experience we gain by profiling multiple model executions, events generated by the model cannot be used to meaningfully structure an execution profile. This is not really a surprise since events are generated from a model to fulfill a need of the model itself, and not really for profiling purposes. No assumption can therefore be made on the information carried by those events.

A practical solution is to annotate events upon reception with information about the sequentiality and the timing. MetaSpy offers a generic event class, called `SpyEvent`. A spy event knows its creation time and the previously emitted event.

The class `SpyEvent` may be subclassed to capture domain relations. For example, `MondrianEvent` knows about siblings of the node that emitted the event. This is an important relation for tracing how the cache is activated.

5.2 Navigation between events

To analyze the event activation sequence in Mondrian we have the following spy:

```
Profiler subclass: #MSMondrianCacheActivationSequenceProfiler
   instanceVariableNames: 'lastEvent mapping announcer'
```

This profiler has three variables. The last event that has been emitted is kept in the variable `lastEvent`. Since all the events are kept in a linked list, it is sufficient to keep a reference of the last event to access previous events. The association between a Mondrian node and the events the node has emitted is kept in the variable `mapping`. The browser is updated via an announcer.

The profiler is installed with `setUp`:

```
MondrianCacheActivationSequenceSpy>>setUp
   super setUp.
   nodes do: [ :node |
      self
         observeObject: node
         selector: #displayOn:
         do: [ :receiver :selector :arguments |
            lastEvent := (MondrianEvent for: receiver next: lastEvent).
            (mapping at: receiver ifAbsentPut: [ OrderedCollection new ])
               add: lastEvent ] ]
```

`MondrianCacheActivationSequenceSpy` is responsible for adapting Mondrian nodes to find out the order in which the method `displayOn:` was executed. Each execution of the method `displayOn:` should create an instance of `MondrianEvent`. Each Mondrian node is instrumented so that every time that the message `displayOn:` is invoked a `MondrianEvent` is created and saved within the mappings indexed by node. Each `MondrianEvent` knows the node that generated it and the previous event. The `setUp` is invoked to install the instrumentation.

Mondrian events are first captured during the profile. The browsing tool described below is useful to navigate between them.

Glamour [Bun09] is an engine for scripting browsers. We use it to build navigation tools for the captured events. The Glamour-based tool is set up in the `visualize` method:

```
MondrianCacheActivationSequenceSpy>>visualize
   | browser |
   browser := Tabulator new.
   browser title: 'Mondrian event crawler'.
   browser
      column: #events;
      column: #model.
   browser transmit to: #events;
      andShow: [ :constructor | self eventsIn: constructor ].
   browser transmit from: #events; to: #model;
       andShow: [ :constructor | self modelIn: constructor ].
   browser updateOn: Announcement from: [ :v | announcer ].
   browser openOn: lastEvent.
```

Figure 5 shows the result of a Mondrian profiling using the Glamour script. The left-hand side gives the sequential list of the events we captured using emitted by Mondrian. The right-hand side gives the information associated to the event selection.

The method `modelIn:` is invoked when one selects an event. The method fills a glamour element with three tabs, two lists and a Mondrian visualization:

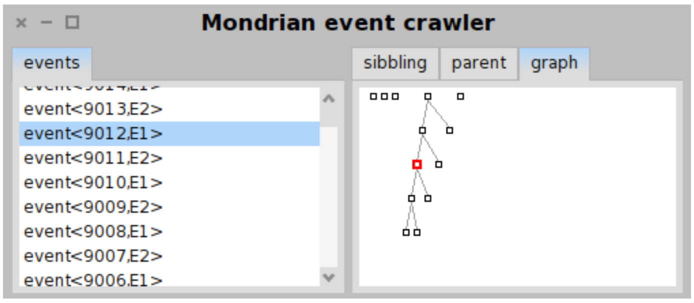

Figure 5 – Glamour-based event navigation tool.

```
MondrianCacheActivationSequenceSpy>>modelIn: constructor
modelIn: constructor
   constructor list
      title: 'sibling';
      display: [ :event | self siblingOf: event ].
   constructor list
      title: 'parent';
      display: [ :event | self parentOf: event ].
   constructor mondrian
      title: 'graph';
      painting: [ :view :event |
         view nodes: (self siblingOf: event).
         view edgesFrom: #owner.
         view treeLayout ].
```

The methods `parentOf:` and `siblingOf:` are used to retrieve the data from the Mondrian model and are not presented here.

The list of events are accessed using the helper method:

```
MondrianCacheActivationSequenceSpy>>eventsIn: constructor
   constructor list
      title: 'events';
      display: [ :event | event allPreviousEvents ];
      updateOn: Announcement from: [ :v | announcer ]
```

Events are linked to each other forming a list. The method `allPreviousEvents` returns the list of all previous events.

Using an adequate model, a browsing tool is easily implementable using Glamour. Presentations are constructed and combined to reflect the navigation flow of the extracted events.

The variable `mapping` plays an important role since it associates the events with the node who emitted them. A hash map effectively implements this relation.

6 Implementing Instrumentation Strategies

MetaSpy has two ways of implementing instrumentation strategies: listening to pre-existing event-based systems, or using the meta-level programming techniques of the host language to define a meta-event the strategy is interested in.

Let us consider the class `AnnouncementInstrumenter`, whose responsibility is to observe the generation of specific announcements.

```
AnnouncementInstrumenter>>install
   self announcer
      on: Announcement
      send: #value:
      to: self handler
```

The `install` method installs an instrumentation strategy object on the domain specified in the `install` method. In this snippet of code we can see that the strategy is hooked into the announcement system by evaluating the strategy's handler when an announcement is triggered.

However, not all profiling activities can rely on a pre-existing mechanism for registering to events. In some cases, a profiler may be hooked into the base code using an existing event mechanism, for example the OmniBrowser profiler. In other cases, extending the base code with an appropriate event mechanism is simply too expensive. Because of this, we need to rely on the meta-programming facilities of the host language. These facilities are not always uniform and require ad hoc code to hook in behavior. To avoid this drawback we decided to use a framework that provides uniform meta-programming abstractions.

6.1 Bifröst

Bifröst [RRGN10] offers a model of fined-grained unanticipated dynamic structural and behavioral adaptation. Instead of providing reflective capabilities as an external mechanism, Bifröst integrates them deeply into the environment. Bifröst is a reflective system based on explicit meta-objects to improve meta-level engineering.

Bifröst has been designed as an evolution of partial behavioral reflection for Smalltalk [RDT08], which in turn was conceived as an extension of the Reflex model [TNCC03]. Bifröst is a reflective architecture [Mae87] where meta-objects control the different aspects of reflection offered by the language. Bifröst's meta-objects provide a structural view and a behavioral view. In the context of MetaSpy we were mainly interested in behavioral reifications. A behavioral meta-object reifying message sends was used for the message send instrumenter. A Message Received event is also provided by the behavioral meta-object. State read and write are also supported thus MetaSpy can profile these dynamic events. Bifröst meta-objects when attached to a single object are object-specific in nature, thus fulfilling an important domain-specific profiler design requirement.

The Bifröst model solves the main problems of previous approaches while providing the main reflection requirements. Moreover, these requirements are key for having a domain-specific profiler which can adapt and evolve at runtime on top of the objects that define the domain. These requirements are:

Partial Reflection. Bifröst allows meta-objects to be bound to any object in the system thus reflecting selected parts of an application.

Selective Reification. When and where a particular reification should be reified is managed by the different meta-objects.

Unanticipated Changes. At any point in time a meta-object can be bound to any object thus supporting unanticipated changes.

Meta-level Composition. Composable meta-objects provide the means for combining different adaptations.

Runtime Integration. Bifröst's reflective model lives entirely in the language model, so there is no VM modification or low-level adaptation required.

Bifröst's adaptation mechanism is built on top of lower-level meta-objects. In the Smalltalk implementation of Bifröst we bind meta-objects to abstract syntax tree (AST) nodes. A meta-object can be associated to a single AST node or to multiple ones. The next time the method is compiled the system automatically generates new bytecodes that take the meta-object into account. This behavior allows Bifröst to adapt the predefined behavior of objects. AST meta-objects reify AST-related information depending on the AST node. For example, a message send node can reify the sender, the receiver and the arguments at runtime. The meta-level behavior specified in the meta-object can be executed before, after or instead of the behavior of the AST node the meta-object is adapting.

Let us consider the Message Received Instrumenter, whose responsibility is to instrument when a specific object receives a specific message.

```
MessageReceivedInstrumenter>>install
   self observerMetaObject bind: self object
```

```
MessageReceivedInstrumenter>>setUp
   profilingMetaObject := BehaviorMetaObject new
      when: self selector
      isReceivedDo: self handler
```

The method `install` binds a meta-object to the object to be observed. The method `setUp` initializes the profiling meta-object with a behavioral meta-object. This meta-object evaluates the handler when a specific message is received by the profiled object. This mechanism is termed *object-specific instrumentation.*

In our Smalltalk implementation of Bifröst, the profiled application, the profiler, and the visualization engine are all written in the same language, Pharo, and run on the same virtual machine. Nothing in our approach prevents these components from being decoupled and having them written in a different language or running remotely. This approach is often taken with profilers and debuggers running on the Java virtual machine (*e.g.,* Java debugging interface[4]).

6.2 Feasability of Domain-specific Profiling

Let us analyze the feasibility of implementing this approach in other contexts. Object-specific instrumentation is not trivial to achieve in class-based languages like Smalltalk and Java. Classes are deeply rooted in the language interpreter or virtual machine and performance is tweaked to rely heavily on these constructs. Moreover, most languages provide a good level of structural reflection to deal with structural elements like classes, method, statements, *etc.* Most languages, however, do not provide a standard mechanism to reflect on the dynamic abstractions of the language. There are typically no abstractions to intercept meta-events such as a message send, a message receive, a state read, *etc.* There has recently been extensive work on object-specific runtime adaptations and operation decomposition of the runtime system.

Aspect-Oriented Programming (AOP) [Kic96, KLM+97, KIL+97] is a technique which aims at increasing modularity by supporting the separation of cross-cutting concerns. Dynamic object-specific aspects have been introduced with an operational decomposition view of the system.

[4] `http://download.oracle.com/javase/1.5.0/docs/guide/jpda/jvmdi-spec.html`

Douence, Motelet and Südholt [DMS01] introduced a general operational model for crosscutting based on execution monitors called Event-based Aspect-Oriented Programming (EAOP). Douence and Südholt [DS02] later introduced constructor calls and constructor returns as events. The Execution Monitor in their implementation observes events emitted during execution. The execution of the base program is suspended when an event is emitted. The monitor matches this event against different event patterns. When a pattern is satisfied the associated actions are executed.

The JAsCo language [SVJ03] provides a way of separating *when* an aspect should be applied and *what* should be done. Hooks are defined with abstract pointcuts. Traps are introduced in the potential places where an event should be triggered. Connectors link these events to the hooks that dictate what should be done. Connectors can be loaded dynamically making this approach highly dynamic.

Stateful aspects or tracematches make it possible to restrain the application of an aspect to the occurrences of certain execution event patterns. AspectJ extension with tracemath [AAC$^+$05] events patterns are matched in all threads of the system.

7 Micro-benchmark

Profiling always impacts the performance of the application being analyzed. We have performed a micro-benchmark to assess the maximal performance impact of MetaSpy. We assume that the behavior required to fulfill the profiling requirements is constant to any instrumentation strategy.

We analyze the impact of MetaSpy on both profiling uses cases. All benchmarks were performed on an Apple MacBook Pro, 2.8 GHz Intel Core i7 in Pharo 1.1.1 with the jitted Cog VM.

Registering instrumentation strategies to a pre-existing event-based system depends heavily on the the system used and how it is used.

Using meta-level programming techniques on a runtime system can have a significant performance impact. Consider a benchmark in which a test method is being invoked one million times from within a loop. We measure the execution time of the benchmark with Bifröst reifying the 10^6 method activations of the test method. This shows that in the reflective case the code runs about 35 times slower than in the reified one. However, for a real-world application with only few reifications the performance impact is significantly lower. Bifröst's meta-objects provide a way of adapting selected objects thus allowing reflection to be applied within a fine-grained scope only. This provides a natural way of controlling the performance impact of reflective changes.

Let us consider the Mondrian use case presented in Section 2.1. The main source of performance degradation is from the execution of the method `displayOn:` and thus whenever a node gets redisplayed. We developed a benchmark where the user interaction with the Mondrian easel is simulated to avoid human delay pollution in the exercise. In this benchmark we redraw one thousand times the nodes in the Mondrian visualization. This implies that the method `displayOn:` is called extensively. The results showed that the profiler-oriented instrumentation produces on average a 20% performance impact. The user of this Mondrian visualization can hardly detect the delay in the drawing process. Note that our implementation has not been aggressively optimized. It has been shown [AR01] that combining instrumentation and sampling profiling leaded to accurate profiles (93–98% overlap with a perfect profile) with low overhead (3–6%). The profilers we presented in this paper are likely to benefit from such instrumentation sampling.

8 Conclusions and Future Work

Our contributions are the following:

1. We demonstrated the need for domain-specific profilers. We argued that traditional profilers are concerned with source code only and are inadequate for profiling domain-specific concerns. We demonstrated this drawback with two use cases.
2. We formulated the requirements domain-specific profilers must fulfill: specifying the domain, capturing domain related events and presenting the necessary information.
3. We presented MetaSpy, a framework for defining domain-specific profilers. We also presented three real-world case studies showing how MetaSpy fulfills the domain-specific profiler requirements.

As future work we plan to:

- Provide ready-made and pluggable visualizations that can be used by new domain-specific profilers. We plan to use Glamour to build these visualizations.
- Apply MetaSpy in the context of large meta-models, such as the FAMIX meta-model in Moose and the Magritte meta-model in Pier.
- Provide additional ready-made event types that enhance the expressibility of new profilers.
- Profiler scoping is of key importance to obtain tailored information. We plan to enhance the scoping mechanism to be able to dynamically attach events to groups of objects.

References

[AAC+05] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 345–364, New York, NY, USA, 2005. ACM. `doi:10.1145/1103845.1094839`.

[AR01] Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, pages 168–179, New York, NY, USA, 2001. ACM. Available from: `http://altair.snu.ac.kr/newhome/kr/course/system_software/2005/Arnold01Sampling.pdf`, `doi:10.1145/378795.378832`.

[BDPW08] Alexandre Bergel, Stéphane Ducasse, Colin Putney, and Roel Wuyts. Creating sophisticated development tools with OmniBrowser. *Journal of Computer Languages, Systems and Structures*, 34(2-3):109–129, 2008. `doi:10.1016/j.cl.2007.05.005`.

[BNRR11] Alexandre Bergel, Oscar Nierstrasz, Lukas Renggli, and Jorge Ressia. Domain-specific profiling. In *Proceedings of the 49th International Conference on Objects, Models, Components and Patterns (TOOLS'11)*, volume 6705 of *LNCS*, pages 68–82. Springer-Verlag, June 2011. Available from: `http://scg.unibe.ch/archive/papers/Berg11b-Profiling.pdf`, `doi:10.1007/978-3-642-21952-8_7`.

[Bun09] Philipp Bunge. Scripting browsers with Glamour. Master's thesis, University of Bern, April 2009. Available from: `http://scg.unibe.ch/archive/masters/Bung09a.pdf`.

[CM09] Jesús Sánchez Cuadrado and Jesús García Molina. A model-based approach to families of embedded domain specific languages. *IEEE Transactions on Software Engineering*, 99(1), 2009. `doi:10.1109/TSE.2009.14`.

[DKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000. Available from: `http://homepages.cwi.nl/~arie/papers/dslbib.pdf`, `doi:10.1145/352029.352035`.

[DMS01] Remi Douence, Olivier Motelet, and Mario Sudholt. A formal definition of crosscuts. In *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns (Reflection 2001)*, volume 2192 of *Lecture Notes in Computer Science*, pages 170–186, Berlin, Heidelberg, and New York, September 2001. Springer-Verlag. `doi:10.1007/3-540-45429-2_13`.

[DS02] Rémi Douence and Mario Südholt. A model and a tool for event-based aspect-oriented programming (EAOP). Technical report, Ecole des Mines de Nantes, December 2002.

[FP06] Steve Freeman and Nat Pryce. Evolving an embedded domain-specific language in Java. In *OOPSLA'06: Companion to the 21st Symposium on Object-Oriented Programming Systems, Languages, and Applications*, pages 855–865, Portland, OR, USA, 2006. ACM. `doi:10.1145/1176617.1176735`.

[GH92] Aloke Gupta and Wen-Mei W. Hwu. Xprof: profiling the execution of X Window programs. In *Proceedings of the 1992 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, SIGMETRICS '92/PERFORMANCE '92, pages 253–254, New York, NY, USA, 1992. ACM. Available from: `http://doi.acm.org/10.1145/133057.133152`, `doi:10.1145/133057.133152`.

[GR83] Adele Goldberg and David Robson. *Smalltalk 80: the Language and its Implementation.* Addison Wesley, Reading, Mass., May 1983. Available from: `http://stephane.ducasse.free.fr/FreeBooks/BlueBook/Bluebook.pdf`.

[Kic96] Gregor Kiczales. Aspect-oriented programming: A position paper from the Xerox PARC aspect-oriented programming project. In Max Muehlhauser, editor, *Special Issues in Object-Oriented Programming.* Dpunkt Verlag, 1996.

[KIL+97] Gregor Kiczales, John Irwin, John Lamping, Jean-Marc Loingtier, Cristina Videira Lopes, Chris Maeda, and Anurag Mendhekar. Aspect-oriented programming. Technical report, Xerox Palo Alto Research Center, 1997.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *ECOOP'97: Proceedings of the 11th European Conference on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242, Jyvaskyla, Finland, June 1997. Springer-Verlag. `doi:10.1007/BFb0053381`.

[KR08] Jevgeni Kabanov and Rein Raudjärv. Embedded typesafe domain specific languages for Java. In *PPPJ'08: Proceedings of the 6th International Symposium on Principles and Practice of Programming in Java*, pages 189–197, Modena, Italy, 2008. ACM. `doi:10.1145/1411732.1411758`.

[LD03] Michele Lanza and Stéphane Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE)*, 29(9):782–795, September 2003. Available from: `http://scg.unibe.ch/archive/papers/Lanz03dTSEPolymetric.pdf`, `doi:10.1109/TSE.2003.1232284`.

[Mae87] Pattie Maes. Concepts and experiments in computational reflection. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, volume 22, pages 147–155, December 1987. `doi:10.1145/38765.38821`.

[MGL06] Michael Meyer, Tudor Gîrba, and Mircea Lungu. Mondrian: An agile visualization framework. In *ACM Symposium on Software Visualization (SoftVis'06)*, pages 135–144, New York, NY, USA, 2006. ACM Press. Available from: `http://scg.unibe.ch/archive/papers/Meye06aMondrian.pdf`, `doi:10.1145/1148493.1148513`.

[Nag10] Meiyappan Nagappan. Analysis of execution log files. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 409–412, New York, NY, USA, 2010. ACM. Available from: `http://doi.acm.org/10.1145/1810295.1810405`, `doi:10.1145/1810295.1810405`.

[RDGN10a] Lukas Renggli, Stéphane Ducasse, Tudor Gîrba, and Oscar Nierstrasz. Domain-specific program checking. In Jan Vitek, editor, *Proceedings of the 48th International Conference on Objects, Models, Components and Patterns (TOOLS'10)*, volume 6141 of *LNCS*, pages 213–232. Springer-Verlag, 2010. Available from: `http://scg.unibe.ch/archive/papers/Reng10bDomainSpecificProgramChecking.pdf`, `doi:10.1007/978-3-642-13953-6_12`.

[RDGN10b] Lukas Renggli, Stéphane Ducasse, Tudor Gîrba, and Oscar Nierstrasz. Practical dynamic grammars for dynamic languages. In *4th Workshop on Dynamic Languages and Applications (DYLA 2010)*, Malaga, Spain, June 2010. Available from: `http://scg.unibe.ch/archive/papers/Reng10cDynamicGrammars.pdf`.

[RDT08] David Röthlisberger, Marcus Denker, and Éric Tanter. Unanticipated partial behavioral reflection: Adapting applications at runtime. *Journal of Computer Languages, Systems and Structures*, 34(2-3):46–65, July 2008. Available from: http://scg.unibe.ch/archive/papers/Roet08aUPBReflectionJournal.pdf, doi:10.1016/j.cl.2007.05.001.

[RGN10] Lukas Renggli, Tudor Gîrba, and Oscar Nierstrasz. Embedding languages without breaking tools. In Theo D'Hondt, editor, *ECOOP'10: Proceedings of the 24th European Conference on Object-Oriented Programming*, volume 6183 of *LNCS*, pages 380–404, Maribor, Slovenia, 2010. Springer-Verlag. Available from: http://scg.unibe.ch/archive/papers/Reng10aEmbeddingLanguages.pdf, doi:10.1007/978-3-642-14107-2_19.

[RRGN10] Jorge Ressia, Lukas Renggli, Tudor Gîrba, and Oscar Nierstrasz. Run-time evolution through explicit meta-objects. In *Proceedings of the 5th Workshop on Models@run.time at the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MODELS 2010)*, pages 37–48, October 2010. http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-641/. Available from: http://scg.unibe.ch/archive/papers/Ress10a-RuntimeEvolution.pdf.

[SVJ03] Davy Suvée, Wim Vanderperren, and Viviane Jonckers. Jasco: an aspect-oriented approach tailored for component based software development. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 21–29, New York, NY, USA, 2003. ACM. doi:10.1145/643603.643606.

[TCJ10] Massimo Tisi, Jordi Cabot, and Frédéric Jouault. Improving higher-order transformations support in atl. In *Proceedings of the Third international conference on Theory and practice of model transformations*, ICMT'10, pages 215–229, Berlin, Heidelberg, 2010. Springer-Verlag. doi:10.1007/978-3-642-13688-7_15.

[TNCC03] Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In *Proceedings of OOPSLA '03, ACM SIGPLAN Notices*, pages 27–46, nov 2003. Available from: http://www.dcc.uchile.cl/~etanter/research/publi/2003/tanter-oopsla03.pdf, doi:10.1145/949305.949309.

[Vis04] Eelco Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In C. Lengauer et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag, June 2004. Available from: http://archive.cs.uu.nl/pub/RUU/CS/techreps/CS-2004/2004-011.pdfhttp://www.stratego-language.org/Stratego/ProgramTransformationWithStrategoXT, doi:10.1007/b98156.

[YD00] Karim Yaghmour and Michel R. Dagenais. Measuring and characterizing system behavior using kernel-level event logging. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '00, pages 2–2, Berkeley, CA, USA, 2000. USENIX Association. Available from: http://www.usenix.org/event/usenix2000/general/full_papers/yaghmour/yaghmour.pdf.

About the authors

Jorge Ressia is a PhD candidate at the Software Composition Group of the University of Bern, Switzerland. His research interests include programming languages, object-oriented programming, meta-programming and reflection. Contact him at `ressia@iam.unibe.ch`, or visit `jorgeressia.com`.

Alexandre Bergel is Assistant Professor at the University of Chile. He and his collaborators carry out research in diverse aspects of software engineering and software quality, including software profiling and visualization. Contact him at `abergel@dcc.uchile.cl`, or visit `bergel.eu`.

Oscar Nierstrasz is a Professor of Computer Science at the Institute of Computer Science (IAM) of the University of Bern, where he founded the Software Composition Group in 1994. Prof. Nierstrasz is co-author of over 200 publications and co-author of the books Object-Oriented Reengineering Patterns and Pharo by Example. Contact him at `oscar@iam.unibe.ch`, or visit `scg.unibe.ch/staff/oscar`.

Lukas Renggli is working as a Software Engineer at Google Zurich. He received a PhD in Computer Science from the Software Composition Group of the University of Bern, Switzerland. His research interests include domain-specific languages and tools. Contact him at `renggli@gmail.com`, or visit `lukas-renggli.ch`.

Acknowledgments We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Synchronizing Models and Code" (SNF Project No. 200020-131827, Oct. 2010 – Sept. 2012). This work has been partially funded by Program U-INICIA 11/06 VID 2011, grant U-INICIA 11/06, University of Chile and FONDECYT project 1120094. We also like to thank Mircea Lungu and Toon Verwaest for their feedback on earlier drafts of this paper.