Selected Papers of the International Workshop on Smalltalk Technologies (IWST'10)

Michael Haupt, Robert Hirschfeld (eds.)

Technische Berichte Nr. 40

des Hasso-Plattner-Instituts für
Softwaresystemtechnik
an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Softwaresystemtechnik an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Softwaresystemtechnik an der Universität Potsdam | 40

Michael Haupt | Robert Hirschfeld (eds.)

Selected Papers of the International Workshop on Smalltalk Technologies (IWST'10)

Barcelona, Spain, September 14, 2010

Universitätsverlag Potsdam

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Universitätsverlag Potsdam 2010
http://info.ub.uni-potsdam.de/verlag.htm

Am Neuen Palais 10, 14469 Potsdam
Tel.: +49 (0)331 977 4623 / Fax: 3474
E-Mail: verlag@uni-potsdam.de

Die Schriftenreihe **Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam** wird herausgegeben von den Professoren des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam.

ISSN (print) 1613-5652
ISSN (online) 2191-1665

Online veröffentlicht auf dem Publikationsserver der Universität Potsdam
URL http://pub.ub.uni-potsdam.de/volltexte/2010/4855/
URN urn:nbn:de:kobv:517-opus-48553
http://nbn-resolving.org/urn:nbn:de:kobv:517-opus-48553

Zugleich gedruckt erschienen im Universitätsverlag Potsdam:
ISBN 978-3-86956-106-6

Program Committee

Co-Chairs

- Michael Haupt (Hasso-Plattner-Institut, University of Potsdam, Germany)
- Robert Hirschfeld (Hasso-Plattner-Institut, University of Potsdam, Germany)

Committee Members

- Gabriela Arevalo (Universidad Nacional de La Plata, Argentina)
- Johannes Brauer (Nordakademie, Germany)
- Johan Brichau (Université catholique de Louvain, Belgium)
- Marcus Denker (INRIA Lille – Nord Europe, France)
- Theo D'Hondt (Vrije Universiteit Brussel, Belgium)
- James Foster (GemStone Systems, Inc., USA)
- Andy Kellens (Vrije Universiteit Brussel, Belgium)
- Andreas Raab (Teleplace Inc., USA)
- Lukas Renggli (Software Composition Group, University of Bern, Switzerland)
- Randal L. Schwartz (Stonehenge Consulting Services, Inc., USA)
- Dave Ungar (IBM Research, USA)

Workshop Program

Selected Papers of the Int. Workshop on Smalltalk Technologies, Barcelona, Spain, September 14, 2010

Web Pontoon: A Method for Reflective Web Applications

Reza Razavi

AAS (Ambient Activity Systems) SARL
Ecostart - centre d'entreprise et d'innovation
Bâtiment 1, Rue de l'industrie
L-3895 Foetz - Luxembourg
razavi@acm.org

Abstract

Network-based provisioning of custom-made and adaptive services offers unlimited opportunities for service development. Examples include ICT-based information, assistance, coordination, and remote monitoring services for senior citizens. Addressing diversity and unpredictable changeability requirements of such service platforms entails novel design solutions. I present *Web Pontoon*, a method tailored specifically for handling these requirements by a combination of web content management, client-side end-user programming, closed-loop management of object lifecycles, and domain-driven design. Opportunities for massive deployment of relevant applications are being studied.

Categories and Subject Descriptors D.2.6 [**Programming Environments**]; D.2.2 [**Design Tools and Techniques**]; D.1.5 [**Programming Languages**]: Object-oriented Programming; D.2.10 [**Software Engineering**]: Design

General Terms Design, Languages, Human Factors.

Keywords User-Generated Service, Client Side Web Application Programming, Content Management System, Application Wiki, Object-Oriented Programming, Adaptive Object-Model, Business Application Development, Domain-Driven Design, End-User Programming, Meta-Data, Smalltalk, Seaside, Pier CMS, Dart.

1. Introduction

The Ambient Assisted Living Joint Programme (AAL JP) is a R&D funding programme by 20 European Member States and 3 Associated States to provide innovative ICT-based solutions to elderly persons (products, services, and systems). The primary goal of this program is to improve the quality of life of older people, their autonomy, and participation in social life, skills, and employability. Its secondary target is to improve service delivery and to reduce care costs. Examples include network-based socialization and education, emotional and cognitive stimulation, coordination of services, and sharing of personal information with family and care givers. The third Call for Proposals within this program was launched on April 2010, with the topic "ICT based solutions for Advancement of Older Persons' Independence and Participation in the *Self-Serve Society*".

A close analysis of the Call document [1] demonstrates a particular emphasis on the following quality attributes: accessibility, acceptance and usability; wide applicability and affordability across Europe; privacy, control of personal data and information, confidentiality, transparency, autonomy and dignity; as well as flexibility and *adaptability*. The latter is required to address (1) the diversity of the European population and their needs, cultures, capabilities, attitudes, values, economies, legislations, technologies and infrastructures, and (2) the unpredictable changeability of end-users' conditions and needs throughout the phases of ageing.

Current network-based development solutions do not address these requirements.

Managing diversity and unpredictable changeability of conditions and needs in network-based applications in a usable, affordable and widely applicable way would offer limitless opportunities. It would make network-based services acceptable and usable not only for older persons, but for all citizens (provided appropriate ethical, legal and licensing policies are established in parallel). AAL JP provides a particularly interesting framework for both interdisciplinary research and innovation in these areas. For the last two years it has motivated our investment in developing a novel method, *Web Pontoon*, for reflective web applications based on a combination of (1) client-side content management and end-user programming, (2) closed-loop management of object lifecycles, and (3) domain-driven design. By *reflective web application* I mean network-based software systems with integrated *introspection* and *intercession* capabilities. This means that they can consult and alter at runtime a representation of their own organisation, content and functionality, to adapt to diverse and changing user requirements.

The remainder of the paper is organized as follows. Section 2 gives a short overview of the state of the art on relevant web technologies. Section 3 describes the concepts and ideas behind Web Pontoon, while Section 4 discusses how it may be implemented by combining and extending several existing solutions. Section 5 illustrates how it works in practice. Section 6 discusses the related work. Section 7 summarizes the contributions of this work, and Section 8 gives an outlook. This is an overview paper specifically targeting Smalltalk Web experts. It will be followed by complementary research papers and technical reports.

2. Navigator-based End-User Programming

This work builds on the state-of-the-art solutions for developing flexible web applications, namely Seaside and Pier CMS. I specifically explain how to extend the design of a typical content management system to support client side end-user programming and user-generated services (UGS). UGS is perceived as an evolution of network-based user-generated content (UGC), e.g. through virtual social networks, with potentially much more significant impact than UGC on our living style and quality of life.

2.1 User-centric Approaches to Web Programming

The first International Workshop on User-generated Services was held in Stockholm, Sweden, in November 2009, and several reported studies attest the trend towards UGS [2].

Florian Gilles *et al.* from SAP Research (St. Gallen, Switzerland) and University of St. Gallen report a "huge demand for ad-hoc and situational enterprise-class applications" and propose a context-aware concept for enterprise mashups that targets end-users from the business units with no programming skills. Their goal is to enable these users to rapidly combine and reuse existing company internal and external resources to create new value added applications [3]. Tobias Nestler *et al*. from SAP Research (Dresden, Germany) focus on empowering users with limited IT skills to get easy access to web services, and propose a user-centric design approach to model and create simple service-based applications in a graphical way without being necessary to write any code [4]. Eduardo Silva *et al.* from University of Twente, (The Netherlands) attest that "supporting end-users in the creation process, at runtime, is a difficult undertaking", and propose automatic service discovery, selection and composition support [5].

The emergence of client side web development as a new field of software engineering is confirmed by other studies. Robert Krahn *et al.* from University of Potsdam (Germany) and Sun Microsystems (USA) present Lively Wiki, a development and collaboration environment based on Lively Kernel [6], which enables users to create rich and interactive Web pages and applications, without leaving the Web [7]. Dirk Riehle, when at SAP Research (Palo Alto, USA), reported on the need to better support end-users in their collaborative processes and the integration of their work into the overall IT infrastructure, and described Application Wikis as an enhancement to wiki engines with lightweight programming features [8]. The 13.65 million Euro SOA4All project (Service Oriented Architectures for All) co-funded by the European Seventh Framework Programme (FP7), that started in March 2008, pursues the goal of offering easy to use and flexible service composition authoring tools that simplify the composition process [9]. FAST (Fast and Advanced Storyboard Tools), another ongoing FP7 project with an overall budget of 5.5 M€, targets a novel approach to application composition and business process definition from a top-down user-centric perspective [10]. Furthermore, several major technology and economic actors already provide web-based lightweight services to dynamically combine content from several web sites and integrate web services as widgets and gadgets. Examples include Facebook [12], iGoogle [13], myYahoo [14], Yahoo!Pipes [15], and IBM Mashup Center [16]. One of the starting points of this work is our Ambiance tool for navigator-based end-user programming [29].

2.2 State-of-the-Art Web Application Development Frameworks: Seaside and Pier CMS

At the implementation technologies side, the most advanced web application framework that I am aware of is Seaside [17], which, compared to many current integration-oriented practices [19], greatly eases the development of dynamic object-oriented web applications in Smalltalk [18]. It additionally comes with online programming tools for professional programmers.

Pier CMS is an extendable content management system that uses Seaside as implementation backbone [20]. It offers several interesting runtime adaptation features, including adaptation of the application's page structure, content and presentation [21]. Pier CMS even allows adding fully functional Seaside applications as content. However, it does not allow changing the flow of control among those components (as they are "hardcoded" in Seaside).

2.3 The Problem Statement

End-user web programmability is currently not systematically approached as a mechanism for addressing adaptability and unpredictable changeability in domain-specific applications. User-centric approaches to web programming like UGS, lack the facilities provided by advanced frameworks like Seaside and Pier CMS for developing industrial-scale, flow-independent web applications. While the latter, although providing advanced adaptability features, lack solutions for client side end-user programmability like those researched by SOA4ALL and FAST.

There are significant achievements towards a much simpler approach to end-user programming web applications, which go far beyond the standard, professional integration-based practices. However, current solutions for end-user programmable web applications are either designed for limited use cases (e.g. mashup solutions) or are still too general for most non-professional programmers and too software development methodology and application domain neutral for them to be effective and productive (e.g. Lively Wiki and its extension Lively Fabrik).

To the best of my knowledge, no method has been yet published for a disciplined development of web applications by object-oriented programmers with support for adaptability and unpredictable changeability of needs and conditions as primary considerations. Closed-loop management of object lifecycles is also missing, although necessary for implementing many key features like smart decision support in real-life applications (Section 5.4, also optimization opportunities in [30]).

3. Programming your Business Applications

Web Pontoon is a method for developing a novel class of highly adaptable object-oriented networked applications, hereafter called *pontoon application*s for short. It shows how to enable object-oriented programmers developing networked applications that behave out-of-the-box as a content management system seamlessly extended with client side end-user programming and closed-loop lifecycle management capabilities, all supporting flexibility, online adaptability and smartness.

Developing pontoon applications requires appropriate tool support and software development methodology. The reminder of this Section describes the major design principles that apply to such tooling and how they influence the whole process of developing networked object-oriented applications.

3.1 Explicit Representation of Structure

The first design principle to get a web application behaving as a content management system consists in reifying its structure. Pontoon applications are uniformly structured as a tree of nodes, and maintain an explicit representation of that structure.

The primary unit of structuring for *both* content and behaviour in pontoon applications are nodes of that structure. This structure guarantees the unique identification of each node by a URL.

3.2 Explicit Representation of Content

The second design principle consists in reifying the "type" of nodes that are handled as content by the above structure. Each type of node *exposes as content* a specific data type managed by a pontoon application. Recurrent node types include *page*, for exposing text and the specification of node interconnections (primarily internal and external links), and *file*, for exposing different types of external files like text, image, and video. The exact types included in a pontoon application depend on business needs.

This design principle is key to the *management of behaviour as content* in pontoon applications as follows.

3.3 Explicit Representation of Control Flows

The third major design principle is flow-independency as described for example by Dragos Manolescu [22]. There should be a clear separation between the implementation of the internal logic of the application components, and that of the logic that governs their (external) interactions. The internal logic of components is "hard-coded" using the implementation language. But, the flow of data and control among those components is abstracted away and "composed" at runtime (next paragraph). Since runtime definition of behaviour is based on composition, application components shall be designed as *atomic services*.

Given the above principles and my previous work on Dart (Section 4.3), adding client side end-user programming capability to pontoon applications becomes as "simple" as adding several new types of nodes: activity, action, operation, contract, and concept nodes, which parallel the behaviour representation elements of Dart (Subsection 4.3.2).

A pontoon *activity node* is a page node extended with the capability to aggregate an unordered collection of action nodes. A pontoon *action node* is also designed as a page node extended with the capability to aggregate an unordered collection of operation nodes. Finally, *operation nodes* are designed as page nodes extended with two additional capabilities: (1) referring to a *contract node*, which basically determines the code that is executed upon the invocation of that operation, and (2) referring to an ordered collection of arguments, whose order and formal types are also defined by the contract node defined in (1). The effective arguments to operations nodes are either operation nodes or *action nodes*. The capability of operation nodes to accept action nodes as arguments is key to the expressiveness of this behaviour representation model. An operation node may further refer to itself as an argument (Subsection 5.2.1). Finally, operations may be procedural or functional. Functional operations instantiate business concepts, exposed as *concept nodes*.

3.4 Explicit Representation of Activity Enactments

To address the requirement of closed-loop lifecycle management, i.e., systematically tracing and situating objects (instances of domain concepts) in the context of the domain processes that produce and/or consume them, pontoon applications should maintain an explicit representation of their activity enactments that relates objects to their execution context. This may be achieved by adding four other types of node: *activity trace*, *action trace*, *operation trace* nodes, related respectively to the execution of an activity, action, and operation, as well as an *object* node type. These may all be designed as an extension to the page node, which allows end-users attaching them extra documentation.

When an operation node is functional, then its corresponding operation trace node points additionally to an *object node* that exposes as content the object returned as the execution result.

3.5 Pontoon Languages

Pontoon applications are deployed not only with the necessary end-user development tools (Subsection 3.9), but also serve as a platform for executing user-defined activities. The term *pontoon language* refers to the combination of these two elements.

The rationale for deploying pontoon applications with an embedded behavioural meta-data interpreter is typical to many environments for end-user programming, i.e., as enumerated by Lively Wiki [6], (1) to avoid the latency related to the traditional compile-run cycles, (2) to support the immediate effect of changes, and (3) to ease the programming process through the elimination of the deployment step.

3.6 Architecture Overview

Figure 1 shows the architectural components and their functions for a deployed pontoon application, organised following the adaptive object-models architectural style [11]. Elements that instantiate the above explicit representations are meta-data producers, while others are meta-data consumers, specifically the pontoon language runtime engine (Section 4.3.2).

In Lively Wiki most components reside at the client side, including the application logic (downloaded from sever at start time) and the end-user programming runtime environment. Figure 1 wants specifically to illustrate that Web Pontoon components are rather server-based. The business model, atomic services, programmer and end-user defined application logics, security measures like authentication and access rights, as well the runtime for the embedded end-user programming language, all remain at the server side mostly for scalability reasons, specifically in presence of lifecycle management and eventually online capturing and processing of field data. At the client side, in addition to interfaces for ordinary end-users, graphical interfaces are provided to both professional and non-professional programmers, respectively, for online administration and end-user programming.

3.7 Process of Modelling, Embedding and Enactment

Developing a pontoon application starts by identifying and implementing the object model and atomic services of the problem domain, which correspond to well-known software engineering practices as for example suggested by domain-driven design [23]. Web Pontoon provides programmers with a meta-level protocol for "exposing" them as content via concept and contract nodes.

Next, authorized non-professional programmers (could be virtually any web user) may dynamically add new activities, typically by activating the classic "add" tool proposed by content management systems, and selecting "activity" as the node type. Then, they successively get access to specialised tools to add new action and operation nodes. Adding operation nodes requires selecting a contract node and eventually a list of arguments. Once fully defined (well-formed), activities may be invoked via an *enactment link* embedded into any page node (Subsection 5.2.2). When the execution of an activity ends, the corresponding activity node may be automatically extended with instances of lifecycle nodes for inspection by end-users (Subsection 5.4).

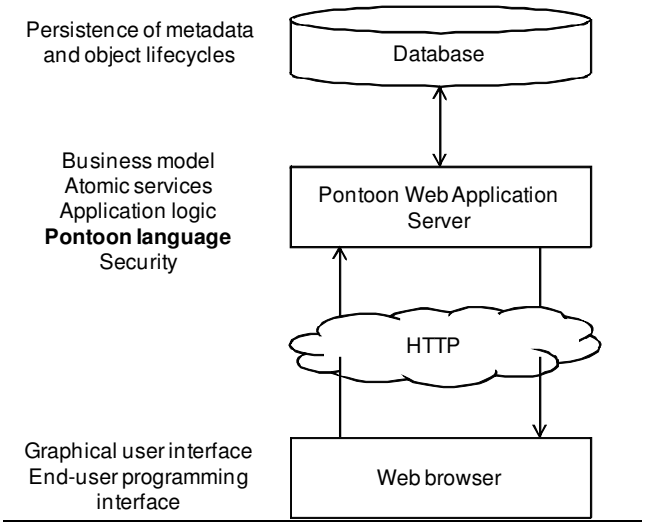

Figure 1. Architecture of pontoon applications.

3.8 Domain-specificity

End-user web programmability and user-generated services mean supporting endless ways to compose atomic services by citizens having their specific requirements and innovative approach.

Web Pontoon adheres to the research outcomes that emphasize domain-specificity as key to a better support to non-professional programmers [37]. Based on my industrial experience with end-user programmers in dimensional metrology (3D control and gauge calibration), I specifically propose to adopt a methodology for developing pontoon applications that seeks transposing systematically the domain concepts and rules into the code to ease implementation, validation and maintenance. Domain-driven design [23] is particularly adapted here since it focuses on producing domain artifacts by an iterative and incremental process.

Somehow, developing pontoon applications corresponds to automating the *ubiquitous language* of programmers and experts in domain-driven design [23]. Programmers may incrementally indentify, with the help of domain experts, and implement alongside the project the "ingredients" of that ubiquitous language, i.e. business concepts and application logic, as class and atomic services, and delegate (to the extent appropriate to the project setting) the composition of their data and control flows to business analysts themselves. A combination with other development methodologies, e.g. user-centric design for interfaces, should be considered when motivated by the user requirements.

3.9 Programming Tools

When developing pontoon applications, object-oriented programmers may keep using the whole range of their traditional software development, test and deployment tools. They however need additional supporting tools as described in the next Section.

The tooling required for non-professional, end-user programmers is basically as simple as any other content management editor, except for the editor of operation nodes, where additional entries should be provided to select the associated contract and effective arguments. Although the interface remains mostly consistent, some training may prove necessary on these new aspects to ensure an effective and productive use.

4. Supporting the Implementing of Pontoon Applications by Reuse and Extension

The implementation of industrial-scale pontoon apps, dedicated to well-identified groups of individuals such as senior citizens and their care providers, may be achieved by reusing and extending existing Smalltalk-based solutions: Seaside as a framework for developing dynamic web applications, Pier CMS and Magritte as a foundation for web content management systems, and Dart as a framework for representing explicitly and processing data and control flows in flow-independent object-oriented applications.

4.1 Seaside

Seaside is a sophisticated object-oriented framework for dynamic web application. It supports a clear separation of components from their connectors (the specification of their data and control flow) in web application development [19]. As such, it introduces flow-independency in the design and implementation of dynamic web applications, which addresses one of the major design principles of pontoon applications (Section 3.3).

4.2 Pier CMS and Magritte

Another core component that may be reused for implementing pontoon applications is Pier content management framework [21]. Pier CMS extends Seaside with several mechanisms while preserving the flow-independency of its applications. Specifically, it provides (1) a neat design and implementation for representing explicitly the web application's structure (Subsection 3.1), (2) an extensible node representation necessary for implementing the node types specific to pontoon applications (Subsection 3.2), (3) an extensible mechanism for generating different types of anchors, necessary for implementing the enactment links (Subsection 5.2.2), and (4) tools for online content management.

In addition, Pier CMS is known as a high level application framework for Seaside, since it addresses several common development concerns such as dynamic configurable integration of different application components, as well as security, authentication, change, and persistence management. It reuses Magritte [24] for many recurrent development concerns, including dynamic generation of graphical interfaces and field validation based on high-level Magritte annotations attached to its data structures.

All these properties are necessary for developing high quality pontoon applications, while making their development and maintenance feasible, productive and low cost.

4.3 Dart

Seaside and Pier CMS contributions to implement pontoon applications are necessary, and essential, but not sufficient. Additional support is required for integrating online end-user programmability, execution and closed-loop lifecycle management [25], which are provided by an extension to a meta-model called Dart.

4.3.1 Background

Dart, for Dynamic ARtifact-driven class specialization, is a meta-model that describes the semantics of a specific notation for service composition and enactment. I have specifically designed it for adaptive and flow-independent object-models [26, 27].

Dart has been implemented in different Smalltalk systems, as a framework for developing domain-specific end-user modelling languages. Dart abstracts from my experience as a software engineer and tool designer in developing industrial software products, specifically Prelude Inspection still commercialised worldwide by an IBM company (MDTVision), and is reused and matured during two research projects, namely Mobidyc [28] and Ambiance [29].

In this project, I reuse Dart as (1) a representation backbone for runtime-defined activities and their lifecycles, and (2) an execution engine for online interpretation of activity definitions.

4.3.2 Key Behaviour Modelling and Interpretation Concepts

The key elements of Dart are concept, contract, activity, action, operation, and lifecycle. The notion of *concept* in Dart corresponds to an explicit representation of a domain object (basically its class). Specifically, every concept results from the execution of one functional operation, while operations may indirectly take concepts as argument (via the operations that produce them). Dart has further a functional semantics in that instantiated concepts are immutable, which is a precondition for lifecycle management.

The notion of *contract* in Dart corresponds to an explicit representation of atomic services (comparable to Web Service). A contract may be instantiated by many operations, while each operation instantiates at most one contract. The contract determines specifically the required arguments, and the code to be executed upon the call to the underlying atomic service.

The notion of *activity* in Dart corresponds to an explicit representation of an unordered collection of actions that define its operational semantics. One of those actions is considered as the execution entry point, and is called the main action. Each activity

aggregates at least one (the main action) or more actions, while each action belongs to only one activity. Activity executions are managed as lifecycles (last paragraph of this Subsection).

The notion of *action* in Dart corresponds to an explicit representation of the flow of data and control amongst an unordered collection of operations. Each action aggregates zero or more operations, while an operation belongs to only one action.

The notion of *operation* in Dart corresponds to an explicit representation of a *call* to an atomic service (like a library function or a Web Service). Each operation instantiates zero or at most one contract, while a contract may be instantiated by zero or more operations. Dart operations are designed with extension and reuse as primary considerations. As such, programmers may add relatively easily operations with domain-specific semantics.

Some operations, specifically conditionals and iterations, may take other actions as argument and also return them as their execution results. In that, Dart operations behave as *higher-order functions*, and Dart actions as lexical closures (typically in Smalltalk). These capabilities are key to the expressiveness of Dart-based domain-specific languages like pontoon languages, specifically different forms of iterations and conditionals.

Finally, the notion of *activity lifecycle*, which is not part of the original design of Dart as reported in [26, 27], corresponds to an explicit representation of the executions of activities in pontoon applications. Each lifecycle is associated to at most one activity, and maintains with it a Type Object [41] relationship. Each activity aggregates zero (never executed) or more lifecycles. Lifecycles are further structured as *action lifecycles* (one per action belonging to the respective activity), which are in turn structured as operation lifecycles. Functional *operation lifecycles* store the execution results. Any operation lifecycle may store additional contextual information, which is key to many smart applications like online decision support, optimization and prediction as we typically discussed in [30]. Dart comes additionally with a hierarchy of *enactment policy* abstractions that support a variety execution conditions. For space reasons, please refer to previous publications for more detailed information [27 and 29].

4.4 Coupling Dart with Seaside and Pier CMS

Seaside, Pier CMS, and Dart may now be coupled together in four steps as follows:

Node Structure and Graphical Interface Integration As I described in Section 3, every key element of Dart should have its counterpart node type in the underlying content management system, here Pier CMS. This allows Dart elements to expose themselves as content for consultation but also runtime editing by end-users. The graphical interfaces for editing Dart nodes may be consistently implemented by defining Magritte annotations [24].

```
StoreTask ≫ go
1  | cart shipping billing creditCard |
2  cart := StoreCart new.
3  [   self fillCart: cart
4      self confirmContentsOfCart: cart ] whileFalse.
5  shipping := self getShippingAddress.
6  billing := (self useAsBillingAddress: shipping)
7      ifFalse: [ self getBillingAddress ]
8      ifTrue: [ shipping ].
9  creditCard := self getPaymentInfo.
10 self ship: cart to: shipping billTo: billing payWith: creditCard.
11 self displayConfirmation.
```

Figure 2. Sushi shopping checkout process in Seaside (Smalltalk).

Kernel Integration The root representation concept in Pier CMS is called *kernel*, responsible for aggregating attributes like the tree-structured representation of nodes and persistency policy. This design should be extended to aggregate the concepts, contracts, activities, and activity traces of pontoon applications.

Control Flow Integration The HTTP request handling process of Pier CMS, itself an extension to the Seaside process, needs to be extended so as (1) the Dart enactment engine takes the control flow over Pier, each time the current *command* [36] corresponds to a user defined activity, and (2) that engine returns the control back to Pier CMS upon the end of an enactment. Pier commands provide a "standard" programming interface for implementing node structure modification routines (adding, copying, moving … nodes). Each command is further wrapped by a *context*, which also aggregates the structure node it applies to. Context is the key concept in the request handling mechanism of Pier.

Dart activities may be integrated into Pier's HTTP request handling process by (1) specialising Pier contexts to additionally handle as commands Dart activities, and (2) paralleling the programming interface of Pier CMS commands for Dart activities. Specifically, by a polymorphic implementation of the `execute` method of Pier, Dart activities start their enactment. Then, upon the end of their execution, implemented by `doAnswer` method, they return control back to Pier by creating an adequate next context pointing to a Pier command (instead of Dart activity). My current implementation returns a context that refers (1) as structure to the activity trace node auto-generated at the end of the activity enactment, and (2) as command to a Pier standard view command. As such, the control is returned back to Pier, which takes care of rendering the activity trace page.

Seaside Components Integration Seaside is a component-based web application framework, while Dart is service-oriented. The two models may be integrated by (1) ensuring that Seaside components *answer* upon the end of their execution, and (2) wrapping them by means of Dar contracts.

4.5 Killer Applications for Smalltalk and Seaside?

As illustrated in the next Section, pontoon activities may implement relatively complex flows of Seaside components. To ensure their execution, the enactment engine of Web Pontoon exploits the Seaside's continuation-based *call/answer* mechanism [19], which, to the best of my knowledge, is currently an unparalleled feature.

The uniqueness of this feature together with the large adoption potential of Web Pontoon for web-based programming of situational services would offer pontoons a "killer app" opportunity.

5. Illustration

To illustrate pontoon as a method for developing end-user programmable web applications, I have decided to adopt the classic sushi store checkout process as implemented by Seaside to illustrate its technique and art in defining data and control flows in dynamic web applications. This process is complex enough to illustrate the expressiveness of pontoon languages.

Figure 2 depicts the Seaside implementation of that process, described in the literature by Ducasse *et al.* [19] as follows: "*When the user wants to check out, he first has to confirm the contents of the cart (lines 3 and 4), and if he agrees, he is asked for the shipping address (line 5). Subsequently, a dialog asks him if he wants to have a separate billing address (line 6). If he answers with yes, an additional address dialog is displayed (line 7). After having entered the payment information (line 9) the order is finished (line 10) and a confirmation page is displayed (line 11). Between each of these steps there is a validation logic that may decide to redisplay the previous dialog with an error message*".

```
Address >> contractSpecFillAddress
       ^ SeasideComponentContract
              name: 'Fill Address'
              argsSpec: (ArgumentAnnotation
                            name: 'Another Address'
                            kind: Address) beOptional
              resultSpec: (ResultAnnotation name: 'My Address' kind: Address)
              category: 'Store Specific Contracts'
              implementor: StoreAddressEditor
              invocationSelector: #editAddressWith:with:
```

Figure 3. Wrapping as a Dart contract an atomic service called Fill Address, implemented in Seaside as `StoreAddressEditor` class.

The next two Subsections illustrate respectively the role of professional and non-professional programmers in Web Pontoon method. I explain how object-oriented programmers may implement a pontoon application for programming online sushi shopping activities. Then, I explain how a non-technical end-user programmer may use it to implement online the process in Figure 2, and to further make it available for execution by ordinary end-users to buy sushi online. The corresponding self-documenting pontoon application is available online at http://www.afacms.com. To get access to password controlled features, specifically online programming, please contact the author.

5.1 Professional Programmers Develop a Language for Online Programming of Sushi Shopping Activities

The first step in implementing a pontoon application consists in identifying the core business concepts, their relationships, and core business rules, as typically recommended by domain-driven design [23].

5.1.1 Identifying and Implementing Concepts and Contracts

Thanks to the above mentioned Seaside implementation, I can safely assume as granted the business concepts as well as atomic services required for the illustration needs of our specific class of sushi shopping pontoon application. The domain object model is composed of elements such as Store Inventory, Store Item, Store Cart, Store Address, and Credit Card, with obvious relationships. The atomic services comprise Fill Cart, Confirm Cart, Fill Address, Get Payment Info, Ship Bill to and Pay with, Inform, While False, and If True Otherwise, with obvious operational semantics. In real life applications, the choice of concepts and contracts depends to the problem domain and end-users requirements.

Given these concepts (nouns) and contracts (verbs), as well as an implementation of Web Pontoon as described in Section 4, object-oriented pontoon programmers may proceed as follows to implement the code specific to our example pontoon application.

The first step consists in wrapping atomic services by Dart contracts. Figure 3 provides as example the method for transforming an address editor (the Seaside class `WAStoreAddressEditor`) into a Dart contract. The contract in Figure 3 typically specifies that calling the code implemented by `StoreAddressEditor` requires one optional argument of type `Address` and returns a result of the same type. The method to be invoked by the execution engine is `editAddressWith:with:` that in addition to the argument specified above, takes also the Seaside active component as argument, more precisely the current instance of `PRContentsWidget`. The latter is required by the Seaside call/answer mechanism.

Dart provides hooks for defining different types of contracts. For example, the contract in Figure 3 uses the `SeasideComponentContract` abstraction to wrap the Seaside component `WAStoreAddressEditor`, The former has not existed in the previous implementations of Dart. It has been added as an extension to Dart for this specific reuse case. It belongs to programmers to know the different types of contract provided by Dart and to select or extend the right one when defining a contract.

5.1.2 Exposing Concepts and Contracts as Nodes

The next step consists in exposing as node the application concepts (implemented as classes) and contracts. This is the role of a polymorphic method named `asNodeOn:`, which takes as argument the parent node, and returns as result an appropriate child node. All model classes that are supposed to expose themselves as node respond to this message as well as the class method `nodeClass`. For example, the `Contract class` metaclass responds to `nodeClass` by returning `ContractNode`, and the `Activity class` metaclass returns `ActivityNode`.

Figure 4 illustrates the result for the contract named Fill Address, as rendered by our deployed pilot pontoon application. It shows that this contract belongs to a contract category named *Store Specific Contracts*, and that it takes one argument, and returns a result. Links are systematically provided to access to relevant nodes, here respectively the contract category, type of the argument (`Address`), and type of the result (`Address`). Additionally, contracts that are implemented as a Seaside component, e.g. Figure 4, provide end-user programmers a Seaside-based live help interface. Thanks to the Pier CMS's facility for adding live components, this could also be defined and changed dynamically.

At this stage, the pontoon language is basically implemented, and the application bootstrapped and ready to be deployed as any standard Seaside application.

5.2 Non-professional Programmers Add Activities

Once the pontoon application implemented in the previous Subsection is deployed, trained and authorized end-users may login from any web navigator and define new shopping activities. They may also wrap those activities as new application functionalities, and make them available for execution by ordinary end-users.

Figure 4. A contract node exposing the Fill Address service.

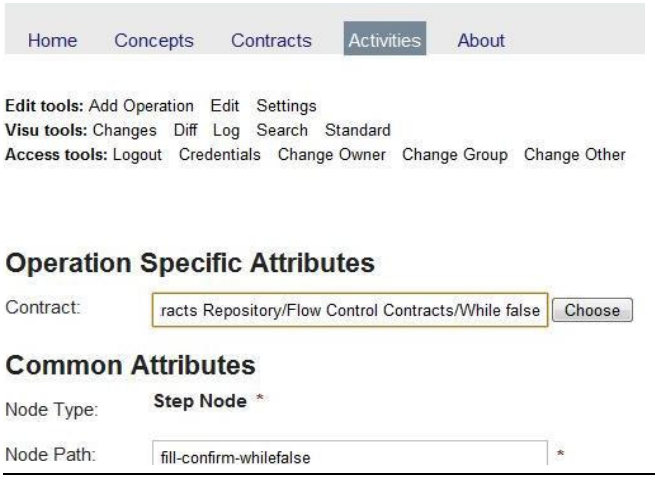

Figure 5. *While False* selected as contract for the operation (*step*) being added to the main action of the sushi shopping activity.

5.2.1 Process of Adding Activities, Actions and Operations

When logged in, the edit commands toolbar provides end-user programmers contextual links for adding activities, actions and operation (Figure 5). Both activities and actions may be basically defined by providing a name and a path string (as any other standard Pier CMS node). For example *Sushi Shopping* (classic) as an activity, and *Fill and confirm cart* as an action (Figure 7). Numbers between parentheses in this Figure, e.g. "(3)" in line N. 4, indicate the line number of each argument. The *Add Activity* link appears in the toolbar when the enclosing node is an *application node*, which is a named repository of activities. The *Add Action* link appears when the enclosing node is an activity node. The *Add Operation* link appears when the enclosing node is an action node.

```
Application: Sushi Store >> Activity: Sushi Shopping

A sushi shopping activity without transactions.

Activity  < Classic > {
  Action < Main > {
    1 - While false  ([Fill and confirm cart]*)
    2 - Fill Address  (2) returns: My shipping address
    3 - Literal String  () returns: Use shipping as billing address?
    4 - Yes or Not  (3*) returns: ?Reuse shipping
    5 - If true, otherwise  (4*, 2*, [Get billing address]*)
    6 - Get Payment Info  () returns: My payment info
    7 - Ship, Bill, and Pay with  (2*, 1@ Get billing address , 6*) returns: My transaction
    8 - Literal String  () returns: Your fish is on its way.
    9 - Inform  (8*)
    }

  Action < Fill and confirm cart > {
    1 - Fill Cart  (1) returns: My filled Cart
    2 - Confirm Cart  (1*) returns: ?Cart confirmed
    }

  Action < Get billing address > {
    1 - Fill Address  (1) returns: My billing address
    }
  }
```

Figure 7. Snapshot of a pontoon activity definition, implementing a logic equivalent to the classic sushi shopping checkout process by Seaside (Fig. 2), but defined at runtime.

Figure 6. Dialog for selecting a contract in the list of available contracts for the operation being defined, e.g. *While False* in Fig. 5.

In the latter case, when the link is pressed for example to add a *while false* operation (line 4 in Figure 2), the graphical interface illustrated in Figure 5 appears and the user is specifically invited to select a contract by pressing the *Choose* button, which pops up the contract selection dialog in Figure 6. Upon the validation of this step, if arguments are required, another dialog is displayed that invites the end-user programmer to select them. The list of options offered for each argument may comprise only operations, only actions, or a mixture of them. This list is filtered by matching the type of the argument with that of the available operations and actions in the scope of the activity under definition.

The final result is depicted by the snapshot in Figure 7, which is a pontoon activity that implements exactly the same logic than the one implemented by Seaside (Figure 2). Basically, there is a *main* action that implements the main logic of that process, and two *subordinate* actions, each for one of the block closures in the corresponding Smalltalk implementation. The correspondence between the Seaside / Smalltalk statements in Figure 2, and Dart operations in the main action in Figure 7 is as follows: Statements N. 3 and 4 go with operation N. 1 (wraps an action). Statement N. 5 goes with operation N. 2. Statement N. 6 goes with operations N. 3 and 4. Statements N. 7 and 8 go with operation N. 5. Statement N. 9 goes with operation N. 6. Statement N. 10 goes with operation N. 7. Statement N. 11 goes with operations 8 and 9.

It should be noted that operations N. 4 and 9 have each an auxiliary operation with Literal String as contract (respectively operations N. 3 and 8). Instances of this contract return at runtime the string value they hold. Here, they serve to specify the message string to be displayed by the dialogs opened each time those operations will be executed.

```
Activity: Sushi Shopping >> Trace: Classic (20 May 2010, 12:33 pm)

Activity Trace  < Classic > {
  Action Trace < Main > {
    1 - While false  ([Fill and confirm cart]*)
    2 - Fill Address  (My shipping address) returned: an Address
    3 - Literal String  () returned: a String
    4 - Yes or Not  (Use shipping as billing address?*) returned: a Boolean
    5 - If true, otherwise  (?Reuse shipping*, My shipping address*, [Get billing address]*)
    6 - Get Payment Info  () returned: a Master Card
    7 - Ship, Bill, and Pay with  (My shipping address*, My billing address, My payment info*) returned: a
Transaction
    8 - Literal String  () returned: a String
    9 - Inform  (Your fish is on its way.*)
    }

  Action Trace < Fill and confirm cart > {
    1 - Fill Cart  (My filled Cart) returned: a Store Cart
    2 - Confirm Cart  (My filled Cart*) returned: a Boolean
    }


  }
```

Figure 8. One enactment trace for the sushi shopping activity programmed online by an authorized and trained end-user.

Further, the statement N. 2 in Figure 2 has no operation equivalent *per se*. The cart object is returned when the code wrapped by Fill Cart contract is executed. The Fill Cart operation, in Fill and confirm cart action (Figure 7), takes *itself* as argument. This ensures the safe execution of this operation inside the *while false* iteration. For the first iteration, there is no cart. But, the consecutive iterations take as argument the last cart.

The activity in Figure 7 and its lifecycles like in Figure 8 are available at http://www.afacms.com/repos/applications/sushistore.

5.2.2 Exposing Activities by Enactment Links

Now, to wrap this activity as a new pontoon application functionality, and make it available to other end-users for activation, the end-user programmer may simply edit the pages where s/he wants an *enactment link* to appear, and a mark-up as follows:

+value:enact|target=/repos/applications/sushistore/classic+

The *enact* keyword is an extension to the Pier CMS *value links*, and serves to specify a request for creating and enactment link on the activity passed as argument to the *target* parameter (identified by its URL, here */repos/applications/sushistore/classic*).

Enactment links may be compared to bookmarklets, i.e. bookmarks that execute a JavaScript expression [32]. The difference between bookmarklets and enactment links is that in the former case the code is written in JavaScript and in the latter case in a domain-specific pontoon language.

5.3 Ordinary End-users Execute Activities

Now, ordinary end-users may access to the web site and press that link to launch the enactment of the underlying activity by the embedded interpreter, which ensures the flow of data and control among the involved atomic services following the specification of that activity by the authorized end-user (Subsection 5.2). In specific conditions, enactments may be launched by program.

5.4 Pontoon Applications Manage Lifecycles

Pontoon applications trace systematically their execution results while situating the relevant static and dynamic knowledge in the context of the underlying activities. For example, our sushi shopping pontoon (Section 5) knows at any moment how many times each of the available shopping activities (like the one in Figure 7) have been executed, when, by whom, in which network setting, etc., and what are the resulting objects (cart, address, order ...). It can also associate any object to its corresponding activity.

At the end of an enactment, pontoon applications extend by default the corresponding activity node by an activity lifecycle node, which encloses one or more task lifecycles, each enclosing in turn operation lifecycles. This design allows attaching URLs to individual enactment results, which may then be used for consultation and specification of individualised reports.

Figure 8 illustrates an activity lifecycle page that is autogenerated by our sushi shopping pontoon application at the end of one of the enactments of the standard sushi shopping activity (Figure 7). This trace comprises only those actions that have effectively been executed. Specifically, the action named Get billing address is absent since during that enactment, the user has preferred reusing her/his shipping address as billing address too. Links in Figure 8 allows inspecting the execution results in details. For example, when pressing the link aliased *a Store cart*, which results from the execution of the operation *Fill Cart* in the subordinate action *Fill and confirm cart*, an inspector (a Seaside component) pops up and shows the cart and its items.

5.5 Implementing atomic services

For space reasons, I'm not able to detail the implementation of Dart runtime engine, which is basically a visitor [36] of the explicit representation of the data and control flows. Other operational semantics may be implemented to address specific requirements. The description of a multi-agent execution engine for Dart may be found in [29]. Just to give an example, the logic for the *while false* atomic service is implemented by Dart as follows:

```
ActionLifecycle >> whileFalseWith: aContext
                [self evaluateWith: aContext]
                        whileFalse
```

In this (simplified) implementation, the Smalltalk `whileFalse` construct is used to iterate over the evaluations of the receiver action lifecycle (class `ActionLifecycle`). The latter corresponds to the argument passed to *while false* operations. Dart provides hooks for both implementing such atomic services, and adjusting the semantics of the underlying interpreter.

Complementary illustrations may be found at http://www.aas-platform.com/about/illustration.

6. Related Work

Web Pontoon is a method for developing object-oriented web applications that allow client-side adaptability, specifically by end-user programming. Apart from Seaside, Pier CMS and Dart (Subsections 4.1, 4.2, and 4.3), as well as Web Velocity [44] which is also Seaside-based but dedicated to professional programmers, it is related to concepts and implementations that combine the wiki metaphor and client side programming, specifically Lively Wiki and Application Wikis.

Lively Wiki [7] is designed as a combination of the wiki metaphor and a general purpose JavaScript-like programming language for client side application development. Several extensions are, however, currently under development, .e.g. Lively Fabrik [33], to make development more appealing to end-users.

Application Wikis are described by Dirk Riehle as "a natural outgrowth of more traditional wiki engines, enhanced with lightweight programming features that allow users to create lightweight applications as part of a regular Wiki, and aid in making data structures and processes explicit" [8]. Current application wikis provide end-users with a markup and scripting language that enables them to create their own wiki applications by editing wiki pages. They provide database-like manipulation of fields stored on pages, and offer a query language to embed reports in wiki pages. However, according to Thoeny, "most application wiki programming today is done by a wiki champion rather than an end-user" [8]. This observation suggests that further research into end-user programming with wikis is needed.

Within this context, and as a first step on a way to a more comprehensive integration of the SAP business application platform with wikis, the SAP Wiki Business Object project follows the goal of allowing end-users to write business queries that operate on structured data from within a wiki, while the structured data resulting from the query executions are also displayed inside a wiki page. The architectural components of their MediaWiki-based prototype comprise a description of the business objects, definition of queries that read and write business objects, and execution and presentation of queries [34].

The definition of business objects in this system is comparable to that of concept nodes in pontoon applications. However, their query definition language seems purely textual, which is more practical in some situations, and not integrated as content.

More generally, Application Wikis seem closer to pontoon applications than Lively Wiki, due to their closer integration with the wiki metaphor and their domain-specificity. But, closed-loop lifecycle management does not seem a priority for the former.

From theoretical perspective, I leverage results from activity theory [42], specifically their hierarchical representation of the structure of human activities (activity, action, and operation), their dynamics and dialectic relationship with the context of their performance. This has specifically led to the integration of a context and closed-loop lifecycle management API into Dart, which traces the execution results by situating the static and dynamic knowledge of the system about itself in the context of users' activities.

7. Summary of Contributions

I motivated (Section 2) and described (Section 3) Web Pontoon as a method for developing object-oriented, industrial-scale applications for network-based provisioning of individualised and adaptable services managed as content. Individualisation and dynamic adaptability, core quality attributes in many applications (Section 1), are achieved by a combination of content management, client side web programming, closed-loop lifecycle management and domain-driven design. Pontoon applications may be deployed with only a blank home page, and extended online with new content, presentation and functionality (e.g. http://afacms.com).

When appropriate tooling and software development methodology are deployed (Section 4), developing and maintaining pontoon applications becomes even somehow easier than developing traditional web applications. Programmers are guided through a set of well-defined steps to identify and code business concepts and atomic services, and wrap them for online programming and adaptation by end-users (Section 5). By extending existing tools from Pier CMS, I have even implemented a framework that automates to a large extent the preparation and deployment of "blank" pontoon applications on secured servers under Linux / Apache.

The core design principles of Web Pontoon include representing explicitly the application's structure, exposing as content its data structures including those involved in its self-representations (structure and behaviour), and enabling their dynamic instantiation by end-users, while supporting their immediate execution and lifecycle management. Runtime adaptations in the resulting reflective web applications are currently curried out manually, but may be automated provided on-line reasoning mechanisms. Like reflective programming languages, Pontoon applications regard and dynamically manipulate their own "code" as queryable regular objects, organized around meta-level objects that represent their concepts, contracts, structure, control flow, interpretation process, and lifecycles. This meta-level layer provides the necessary basis for their openness and runtime programmability, and allows pontoon applications maintaining a *causally connected* relationship with those meta-objects in that any modification is immediately reflected in their behaviour. These reflective facilities extend further those already provided by the host system [43].

Pontoon languages allow dynamic definition of relatively complex logics by non-professional programmers, including iterations and conditionals. Recursion is not supported. Web Pontoon brings to end-users the combined benefits of a constrained programming environment for creating productive results [34], and the power of a full-fledged server side programming platform used to create and maintain those end-user programming tools. End-user generated pontoon services may be exported as an XML file, and reused by other end-users sharing the same domain model and atomic services. Finally, due to sharing some design principles with wikis, as described by Ward Cunningham [38], pontoon applications expose a comparable behavior.

8. Outlook

Pontoon applications address adaptability in network-based applications in a usable, affordable and widely applicable way:

Acceptability and applicability to diverse and changing situations This is achieved by postponing most design decisions until runtime. This concerns not only structure, content, presentation, and components, but also the control flow of network-based applications. New functionality may be added at runtime by composing atomic services using any navigator and following an intuitive approach. For example, in applications to support older persons, this will be accessible to care givers and dedicated relatives. Further, our architecture provides for the creation and easy integration of new atomic services. We foresee a community of programmers for collaborative development and exchange of atomic services in different application areas (pontoonity.com).

Usability It is admitted that usability goes beyond interface design. It has to do with task support - notably, avoiding misfits between the work process induced by the software and the work routines and local contingencies in the workplace [44, p.14]. We achieve usability by allowing runtime adaptation of task flows and integration of adequate methodologies like user-centric design.

Affordability and privacy We build on Open Source and free technologies under MIT License. There are also free, yet robust, environments for development and privately hosting our applications, namely Squeak [45] and Pharo [46]. Users shall not be requested to grant any rights on their personal data.

We are currently defining a new project to build on this technical achievement to produce innovative network-based software and services, specifically for older persons, and to perform real-life case studies. Examples include *individualized* information (e.g. planning of visits, medication timing and dosage), communication, assistance (e.g. memory aids, activity encouragement and coaching), remote monitoring, and network-based enhanced feel of security. Current socio-economic transformations indicate many other application areas for flexibility and runtime adaptability by end-user programmers, like architectural renovation and requirements elicitation in legacy applications. Smart features such as online reasoning and predictive services may be added, provided pontoon applications are coupled with wireless sensors (WSN) as we investigated during Ambiance [29, 30].

Extensions include model-to-code generation to map activities to other runtime engines like Lively Kernel, similarly our work on dynamic macroprogramming of ActorNet-based WSNs [29, 39]. Also, integration with GLASS by Gemstone [31] for transparent persistence of terabytes of lifecycle data when necessary.

Finally, programmable web applications constitute an emerging interdisciplinary area of research, and the first Workshop on End User Development of Software Services and Applications was just held in Rome (Italy) on May 25, 2010 [40].

Acknowledgments

This project has been funded by Ambient Activity Systems, a Luxembourgish startup company supported by the Ministry of Economy and Foreign Commerce. The reported results would have not been achievable without the support of the Smalltalk, Squeak and Pharo communities in general, and more particularly Avi Bryant, Julian Fitzell, Lukas Renggli *et al.*, for having implemented and generously published as free and Open Source their outstanding work on Seaside and Pier CMS. I'm grateful to Christophe Dony, Jean-François Perrot, David Potter, Dirk Riehle, and Joe Yoder for their insightful comments on early versions of this paper. I'd also like to acknowledge the support of my family and friends, specifically Goya Razavi for his technical support.

References

[1] Ambient Assisted Living Joint Programme, Call 3, http://www.aal-europe.eu/calls/aal-call-3-2010. As of May 26, 2010.

[2] Dustdar, S., Hauswirth, M., Hierro, J.-J., Soriano, J., Urmetzer, F., Möller, K., and Rivera, I. "UGS2009 - Workshop on User-generated Services at ICSOC2009", *CEUR Workshop Proceedings* (2009).

[3] Gilles, F., Hoyer, V., Janner, T., and Stanoevska-Slabeva, K. "Lightweight Composition of Ad-Hoc Enterprise-Class Applications with Context-aware Enterprise Mashups," in [2].

[4] Nestler, T., Dannecker, L., and Pursche, A. "User-centric Composition of Service Front-ends at the Presentation Layer," in [2].

[5] Silva, E., Pires, L.-F., and van Sinderen, M. "Supporting Dynamic Service Composition at Runtime based on End-user Requirements," in [2].

[6] Taivalsaari, A., Mikkonen, T., Ingalls, D., and Palacz, K. "Web Browser as an Application Platform: The Lively Kernel Experience," TR SMLI TR-2008-175, Sun Microsystems.

[7] Krahn, R., Ingalls, D., Hirschfeld, R., Lincke, J., and Palacz, K. Lively Wiki a development environment for creating and sharing active web content. In *Proceedings of the 5th international Symposium on Wikis and Open Collaboration* (Orlando, Florida, October 25 - 27, 2009). WikiSym '09. ACM, New York, NY, 1-10.

[8] Riehle, D. "End-User Programming with Application Wikis: A Panel with Ludovic Dubost, Stewart Nickolas, and Peter Thoeny," In *Proceedings of the 2008 International Symposium on Wikis* (WikiSym `08). ACM Press, 2008. Pre-conference panel summary.

[9] SOA4All, http://www.soa4all.eu/.

[10] FAST, http://fast-fp7project.morfeo-project.org.

[11] Yoder, J.-W., Johnson, R. "The adaptive object-model architectural style," In: Bosch J, Morven Gentleman W, Hofmeister C, Kuusela J, editors. *Third IEEE/IFIP conference on software architecture*. IFIP conference proceedings 224. Dordrecht: Kluwer. p. 3–27, (2002).

[12] FaceBook, http://www.facebook.com/.

[13] iGoogle, http://www.google.com/ig.

[14] MyYahoo, http://my.yahoo.com/.

[15] Yahoo! Pipes, http://pipes.yahoo.com/pipes/.

[16] IBM Mashup Center. http://www-01.ibm.com/software/info/mashup-center/. As of May 14, 2009.

[17] Seaside. http://seaside.st.

[18] Goldberg, A. "Smalltalk-80: the interactive programming environment," Addison-Wesley Longman., Inc., Boston, MA, USA, 1984.

[19] Ducasse, S., Lienhard, A., and Renggli, L. "Seaside: A Flexible Environment for Building Dynamic Web Applications," IEEE Softw. 24, 5 (Sep. 2007), 56-63.

[20] Pier CMS. http://piercms.com.

[21] Ducasse, S., Renggli, L., and Wuyts, R. "SmallWiki: a meta-described collaborative content management system," In *Proceedings of the 2005 international Symposium on Wikis* (San Diego, California, October 16 - 18, 2005). ACM, New York, NY, 75-82.

[22] Manolescu. D. "Micro-Workflow: A Workflow Architecture Supporting Compositional Object-Oriented Software Development". PhD Thesis, Univ. of Illinois at Urbana-Champaign, Illinois (2000).

[23] Evans, E. "Domain-Driven Design: Tackling Complexity in the Heart of Software," Addison-Wesley, ISBN: 0-321-12521-5.

[24] Renggli, L., Ducasse, S., and Kuhn, A. "Magritte - A Meta-driven Approach to Empower Developers and End Users," MoDELS 2007: 106-120.

[25] Closed-Loop Lifecycle Management. http://cl2m.com/.

[26] Razavi, R. "Tools for Adaptive Object-Models – Adaptation, Refactoring and Reflection," (in French: "Outils pour les Langages d'Experts – Adaptation, Refactoring et Réflexivité") *Université Pierre et Marie Curie*, Department of Computer Science Technical Report (based on doctoral dissertation), vol. LIP6 2002/014, 285 pages, Paris, France, November 2001.

[27] Razavi, R., Perrot, J.-F., and Johnson. R. "Dart: A Meta-Level Object-Oriented Framework for Task-Specific, Artifact-Driven Behavior Modeling," 6th OOPSLA Workshop on Domain-Specific Modeling (DSM'06), Gray, J., Tolvanen, J.-P., Sprinkle, J. (eds.), Computer Science and Information System Reports, Technical Reports, TR-37, University of Jyväskylä, Finland, pp. 43-55, 2006.

[28] Ginot, V., Le Page, C., and Souissi, S. "A multi-agents architecture to enhance end-user individual-based modeling," *Ecological Modeling* 157, pp. 23-41, 2002.

[29] Razavi, R., Mechitov, K., Agha, G., and Perrot, J.-F. "Ambiance: A Mobile Agent Platform for End-User Programmable Ambient Systems," J.C. Augusto and D. Shapiro (eds.), Advances in Ambient Intelligence, (FAIA), vol. 164, pp. 81-106, IOS Press, 2007.

[30] Razavi, R., Mechitov, K., Agha, G. "Architecture Design Principles to Support Adaptive Service Orchestration in WSN Applications," ACM SIGBED Review, vol. 4, no. 3, 2007.

[31] GLASS – GemStone. http://seaside.gemstone.com/.

[32] Miller, R.-C., "End User Programming for Web Users," Workshop on End User Development, 2003.

[33] Lincke, J., Krahn, R., Ingalls, D., and Hirschfeld, R. "Lively Fabrik - A Web-based End-user Programming Environment," In *Proceedings of the. Conference on Creating, Connecting and Collaborating through Computing* (C5). IEEE, January 2009.

[34] Anslow, C. and Riehle, D. "Towards end-user programming with wikis," In *Proceedings of the 4th international Workshop on End-User Software Engineering* (Leipzig, Germany, May 12 - 12, 2008). WEUSE '08. ACM, New York, NY, 61-65.

[35] Mediawiki. http://www.mediawiki.org. As of May 14, 2010.

[36] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. "Design Patterns: Elements of Reusable Object-Oriented Software," Addison-Wesley Longman Publishing Co., Inc., Boston, USA, 1995.

[37] Nardi, B.A. "A Small Matter of Programming: Perspectives on End User Computing," *MIT Press*, 1993.

[38] Cunningham, W. "Design Principles of Wiki: How can so little do so much?,". Keynote at WikiSym 2006.

[39] Kwon, Y., Sundresh, S., Mechitov, K. and Agha, G. "ActorNet: An Actor Platform for Wireless Sensor Networks," *Fifth International Joint Conference on Autonomous Agents and Multiagent Systems* (AAMAS 2006) , Future University, Hakodate, Japan, May 2006.

[40] EUD4Services: Workshop on End User Development of Software Services and Applications, Rome, Italy, on May 25, 2010.

[41] Johnson, R. and Woolf, B. "Type object," In *Pattern Languages of Program Design 3*, R. C. Martin, D. Riehle, and F. Buschmann, Eds. Addison-Wesley Software Pattern Series. Addison-Wesley Longman Publishing Co., Boston, MA, 47-65, 1997.

[42] Center for Activity Theory and Developmental Work Research. http://www.edu.helsinki.fi/activity/pages/chatanddwr/activitysystem.

[43] Rivard, F. "Smalltalk: a Reflective Language," Proceedings of Reflection'96, pp. 21–38 (1996).

[44] Kaasgaard, K. "Software design and usability: talks with Bonnie Nardi, Jakob Nielsen, David Smith, Austin Henderson and Jed Harris, Terry Winograd, Stephanie Rosenbaum," Copenhagen Business School Press, 2000.

[45] Squeak Smalltalk. http://www.squeak.org/.

[46] Pharo. http://www.pharo-project.org/.

Selected Papers of the Int. Workshop on Smalltalk Technologies, Barcelona, Spain, September 14, 2010

BLOC: a Trait-Based Collections Library – a Preliminary Experience Report

Tristan Bourgois Jannik Laval Stéphane Ducasse Damien Pollet

RMoD Project-Team, Inria Lille – Nord Europe — Université de Lille 1 — CNRS UMR 8022

tr.bourgois@laposte.fr, {jannik.laval, stephane.ducasse, damien.pollet}@inria.fr

Abstract

A trait is a programming construct which provides code reusability. Traits are groups of methods that can be reused orthogonally from inheritance. Traits offer a solution to the problems of multiple inheritance by providing a behavior-centric modularity. Since traits offer an alternative to traditional inheritance-based code reuse, a couple of questions arise. For example, what is a good granularity for a Trait enabling reuse as well as plug ease? How much reuse can we expect on large existing inheritance-based hierarchies?

In this paper we take as case study the Smalltalk Collection hierarchy and we start rewriting it from scratch using traits from the beginning. We show how such library can be built using traits and we report such a preliminary experience. Since the Collection library is large, we focused and built the main classes of the library with Traits and report problems we encountered and how we solved them. Results of this experience are positive and show that we can build new collections based on the traits used to define the new library kernel.

1. Introduction

A trait is a programming construct which provides code reusability. Traits are groups of methods that can be reused orthogonally from inheritance. Traits offer a solution to the problems of multiple inheritance by providing a behavior-centric modularity. [12, 20].

There are different trait model variations. In the original model, *Stateless traits* [12, 20], traits only define methods, and no instance variables. *Stateful traits* [2] extend this model and let traits define state. *Freezable traits* [14] extend stateless traits with a visibility mechanism. In the context of this paper, when we use the term *trait* we mean *Stateless trait*. The reader unfamiliar with traits may read the appendix Section A for a rapid introduction to stateless traits.

Black et al. refactored the Squeak Smalltalk collection [4] hierarchy and showed a gain of 12% of code reuse [6]. Still, their solution closely followed the inheritance-based collection hierarchy. Cassou et al. rewrote the Smalltalk stream hierarchy from scratch [9]. They showed that traits support the reuse of code between a new kernel and a backward compatible one based on the same traits. Ducasse et al. reused and composed unit tests out of traits [13].

Problem: The goal of this paper is to experimentally verify the original claims of code reuse with traits, in the context of a forward engineering scenario. More specifically, our experiment looks for answers to questions that arise when using traits in practice: What is a good trait granularity which favors reuse as well as ease of reuse? Is the composition mechanism good enough to deal with common composition scenarios? What do we gain from using traits? When is it better to define a trait versus a class? Do we need state in traits?

Our approach was to redesign, from scratch, a new collection library based on traits. We identified traits for collections based on the work of Cook, who specified collection behavior [10], and by analyzing the ANSI Smalltalk standard [1]. Since elementary aspects of collections behavior are represented as traits, building new collections based on the composition of such traits is possible. We report on the creation of such new collections.

The paper contributions are:

- The identification of problems in the existing Collections library.
- The design of BLOC, a new library of collections composed from traits.
- Assessing whether traits act as reusable elements to define a library, and checking that the obtained design is clearer and more modular.
- Identifying trait related reuse.

In Section 2, we present the working hypotheses that drive this work. Then in Section 3, we highlight the existing Pharo Collection hierarchy and its modularity problems. Section 4 presents the hierarchy of traits based on the Collection behavior. Section 5 gives usage examples of the Collection Traits library to show the reusability of traits. In Section 6, we discuss the validity of traits, and finally we discuss related work in Section 7 before concluding in Section 8.

2. Working Questions

In this paper, we try to answer the following questions:

1. Trait granularity. Understanding whether a trait has a good size is a difficult topic. On one hand, we would like to reuse a coherent and a potentially large group of behaviors, but on the other hand we may want to only use part of the behavior to plug it into another scenario. Since there is no definitive answer and the answers will depend on the context and domain, we cannot draw immediate solutions. We would like, however, to empirically get an understanding of the granularity of traits that maximize reusability.
2. Trait reusability. The ideas beside traits are modularity and reusability. A non-reusable trait is useless. The question is how much code can be reused in the Collections library.
3. Trait modularity. Can we define traits as effective building blocks? The idea is to have a library of traits to easily compose new classes from different traits and obtain specific behaviors.
4. Can we identify guidelines to assess when trait composition should be preferred over inheritance? This is an important question for class modularity. Inheritance has a strong impact in a system structure, whereas traits seem to be more difficult to understand without documentation.

5. Do traits need state? In the original model, traits do not have state, but in the context of collections, we want to understand whether the initialization of state in the class is a problem. A related question is to which extent mixing-like solutions that include state are better from a user point of view [7].
6. What are the trait limits do we encounter? Traits are a new approach, we enumerate limits and problems we encountered during the study.

3. Collections in Pharo

In Smalltalk, the Collections library is a central part of the system; it is used in the whole system, from the core to the UI. We have chosen the Collections library from Pharo because it is a complete library with a lot of different behavior [5, 16].The hierarchy is composed by more than seven levels, which exhibits reuse of behavior within branches of the hierarchy, but also across different branches. The Collections library in Smalltalk defines a rich set of behavior that we need: hash with HashedCollection, unicity with Set, sequence with OrderedCollection, order, identity with Identity related classes ... One of the problems is that elementary behaviors are often defined in a branch of the hierarchical inheritance structure and this forces their duplication across branches. For example, Dictionary inherits from HashedCollection, but dictionaries have both hashed and indexed behaviors, therefore there are some duplicated methods, like at:. The case of Dictionary shows that Traits could be a good design for Collections.

3.1 The Collections Library

The collection classes form a loosely-defined group of general-purpose subclasses of Collection and Stream. The group of classes that appears in the [17] contains 17 subclasses of Collection (Figure 1), and had already been redesigned several times before the Smalltalk-80 system was released. This group of classes is often considered to be a paradigmatic example of object-oriented design. In Pharo, the abstract class Collection has 101 subclasses, but many of these (like Bitmap, FileStream and CompiledMethod) are special-purpose classes crafted for use in other parts of the system or in applications, and hence not categorized as *Collections* by the system organization. In this paper, we use the term *Collections Hierarchy* to mean Collection and its 47 subclasses that are also in the categories labelled Collections-*.

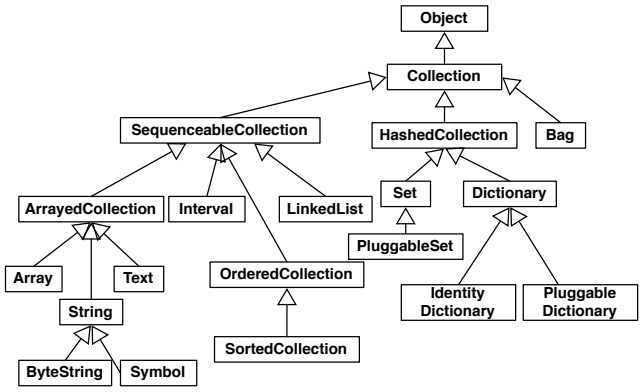

Figure 1. Current Collection hierarchy in Smalltalk

3.2 Cook Analysis

We based our decomposition of the Collections library into traits on the one of Cook [10]. Cook decomposes the Collections library in several behaviors, such as UpdatableCollection, IndexedCollection, ExtensibleCollection ... (shown in Figure 2). To decompose the Collections hierarchy, he uses the different messages and protocols that the classes define. What interested us in his work is the different behaviors that he defined.

On Figure 2 you can see the different behaviors Cook defines for collections, and the methods he selects to define each behavior. This work gives a first approach to a possible Collections library decomposition.

3.3 Single inheritance hampers reusability

The reusability in the Collection Library is limited to single inheritance. As with the example of Dictionary explained before, multiple inheritance is not available and the sole possibility without Traits is to copy and paste behaviors which are not in the same hierarchical branch.

The Collections library does not provide reusability. This library was built to be used, not extended by recomposition of elementary behaviors. If we want to create a new collection, the choice is not easy: what is the sole parent class of the new collection? Do we choose a generic class or a more specific one? The complex hierarchy of the Collections library does not help. Moreover, sometimes we need behaviors from different branches of the inheritance tree.

A simple example: OrderedSet. The new library built out of traits should support the definition of new collections easily. For example we want to be able to create an OrderedSet, a collection of unique ordered elements, which mixes the properties of a Set and OrderedCollection.

To create such a new collection, with the existing library, we have two choices either we inherit from Set and duplicate code from OrderedCollection, or we inherit from OrderedCollection and duplicate code from Set. In either case we must duplicate code because multiple inheritance does not exist in Smalltalk. This example reveals several problems:

- Of single inheritance limitations,
- necessity of code duplication, and
- lack of reuse.

Our work helps to avoid this problem. We create a library of traits which can be used and reused, either to recreate the existing collections, or to create new ones, as shown by the case studies we performed with the classes Dictionary and OrderedSet.

4. Overall design of BLOC

4.1 Traits

Traits are sets of methods designed to be reused as a group in classes. To define additional behaviors in a class, the class can compose a set of traits. A trait requires methods that are necessary to use the trait. Traits do not define state, instead, they require accessors. A complete explanation of Traits is available in Appendix A.

To define a trait, we just send the message named:uses: to the class Trait, specifying the new trait name as well as the traits it is build upon. In the following code, the Trait TOrderedAccessing is defined with the use of behavior from the Trait TSequenceableAccessing.

```
Trait named: #TOrderedAccessing
    uses: TSequenceableAccessing
```

To define a class using traits, the class should inherit from its superclass. It should list the traits is used and provide the methods that should be defined. Here the class OBOrderedCollection inherits from Collection and uses some predefined traits such as TOrderedAc-

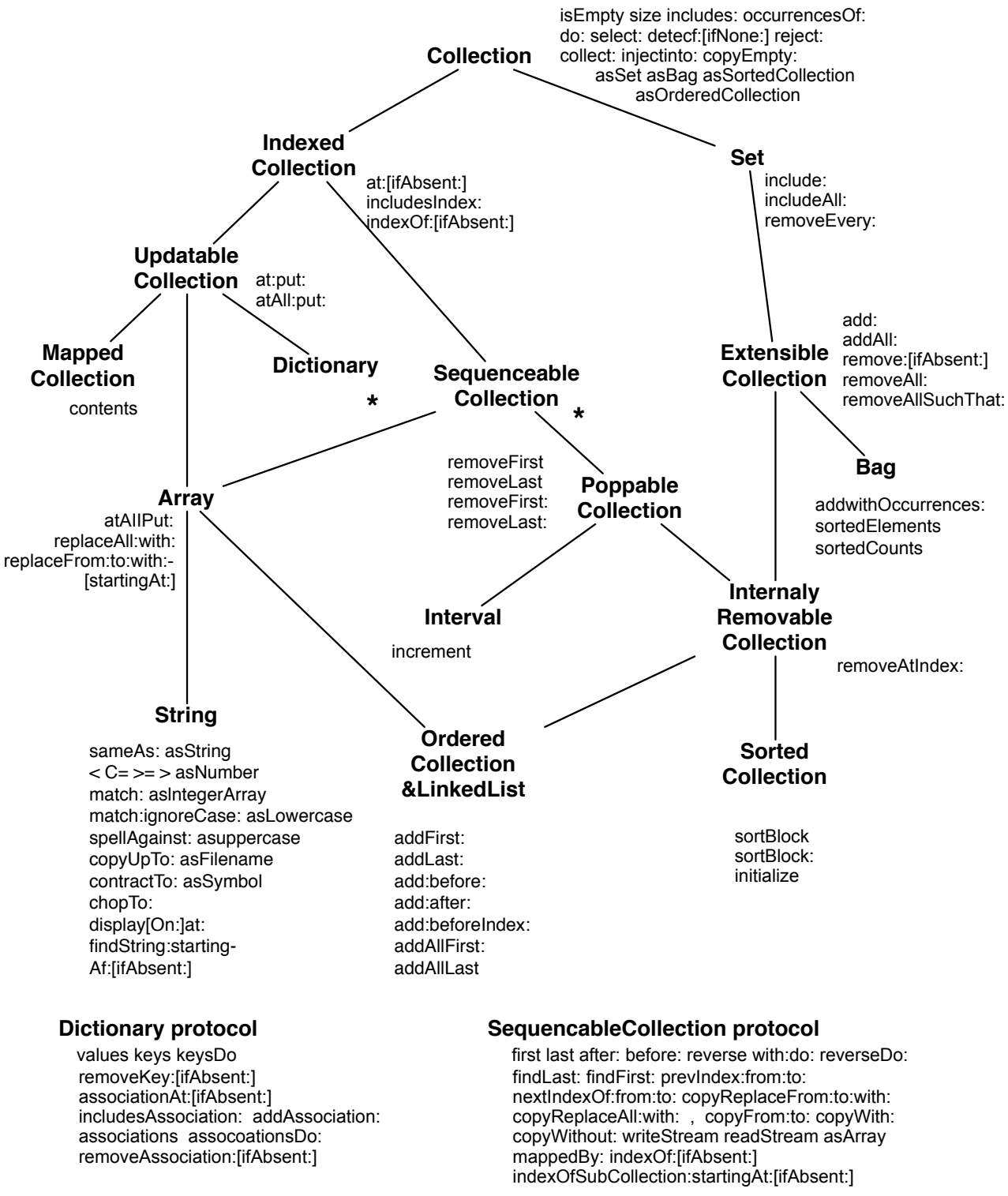

Figure 2. Hierarchy of Collection by Cook

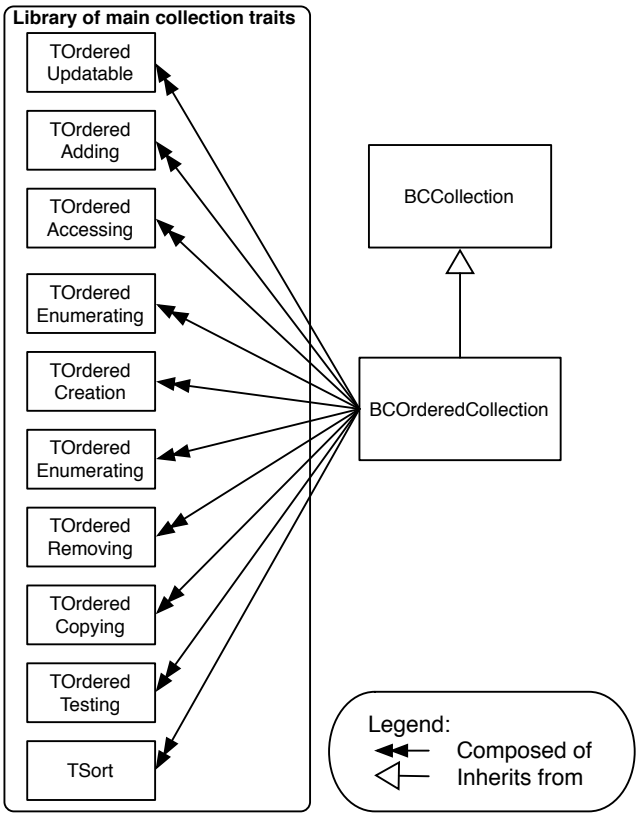

Figure 3. Overall structure

cessing, TSort or TOrderedCopying. Then specific methods should be defined.

```
Collection subclass: #OBOrderedCollection
        uses: TOrderedAdding + TOrderedAccessing + TSort
+ TOrderedIterate + TOrderedCreation + TOrderedCollection
+ TOrderedRemoving + TOrderedCopying
+ TSequenceableTesting + TOrderedUpdatable
        instanceVariableNames: 'array firstIndex lastIndex'
        classVariableNames: ''
        poolDictionaries: ''
        category: 'BLOC-OrderedCollection'
```

4.2 BLOC Elementary Traits

The design of our Collection library is completely different from the inheritance-based one. There is only one level of inheritance: each collection class is a subclass of the abstract class Collection. It brings a semantical means and a collection gets the generic behavior of Collection (like atRandom:, anyOne or ,). Then a new collection becomes specific by adding Traits from the library of Traits.

In Figure 3, the collection BCOrderedCollection is composed of the traits: TOrderedAdding, TOrderedAccessing, TOrderedEnumerating, TOrderedUpdatable, TOrderedCreation, TOrderedCollection, TOrderedRemoving, TOrderedCopying, TSequenceableTesting.

To specify the main collections: OrderedCollection, Set, SortedCollection, Dictionary, Interval and Array, we created traits representing the behaviors defined by protocols proposed in the "Pharo by Example" book [5]. We created 9 different categories of traits presented in Table 1. Each of these categories can be defined (not necessarily) for each main collection.

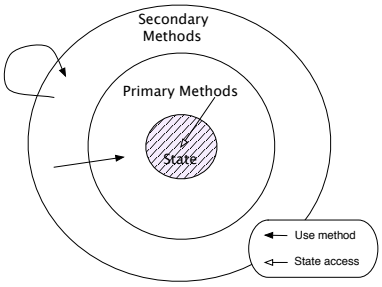

Figure 4. Encapsulation used to compose a collection

If we take the case of Dictionary, Dictionary is a subclass of HashedCollection and needs behaviors from SequenceableCollection to be indexable. If we put all the behavior of one class in one trait, we have to cancelled some methods not used in Dictionary.

4.3 Methods: primary vs. secondary

In Smalltalk, traits do not have state. Our design supports this separation between traits and object state access. Indeed, trait methods should still access object state. In fact, we isolate state access by defining methods in the class. Then, traits use these methods. This concept follows encapsulation. It allows us to make traits independent from the state and the structure of a collection.

To make it, we define two types of methods (see Figure 4): primary and secondary methods. Primary methods access directly object state. They are accessors, but also more complex methods with processing to avoid the time consumption of accessors. Secondary methods use only other methods without accessing directly state. With this differentiation, the Traits library can be used in new collections with different structure. The primary methods are *required methods* of Traits, so when we create new collection we have to define the structure and these methods.

Therefore having a total abstraction of the state enhances reuse. Note that with this approach Traits do not require accessors directly since trait methods do not access state directly.

A primary method is not necessarily a required method. Primary methods are useful for the collection, it provides primary processing. Let's look at an example: In OrderedCollection, the method insert:before: accesses the state of the collection but this method is not used directly by the trait methods. It is used by other primary methods such as add:beforeIndex: and add:afterIndex:. So, insert:before: is a primary method which is not declared as required by the trait.

4.4 Composition Map

We defined the different behaviors of main collections and implemented them as traits. Table 2 shows all main collection in Traits, for each of them we defined the traits defined in Table 1. This way we could recreate the collections but based on elementary characteristics which can be recomposed and reused to create new collection.

We made a map of traits composition. In Figure 5, there is the map of the category Accessing. It has some similarity with the current Collection hierarchy, with a principle difference: there are multiple use of Traits which represent multi-inheritance.

5. Case studies

In this Section we present how the new kernel (*i.e.,* the trait library for the core classes) let us define new collection by recomposing and extending traits.

TXXXAccessing	Contains methods to accessing to the element(s) of the collection.	at:, ...
TXXXAdding	Contains methods to add element(s) in the collection.	add:, addAll:, ...
TXXXUpdating	Contains methods to change one or several elements in the collection.	at:put:, ...
TXXXRemoving	Contains methods to remove element(s) in the collection.	remove:, remove:ifAbsent:, ...
TXXXCopying	Contains methods to copy the collection.	copy, copyWith:, ...
TXXXTesting	Contains methods which test the collection or the elements in.	includes: isEmpty, ...
TXXXCreation	Contains methods to create the collection (class methods)	with:, new:, ...
TXXXEnumerating	Contains methods to iterate on the collection.	do:, select:, ...
TXXXCollection	Contains methods which are specific of the behavior of the collection.	hash, findElementOrNil, ...

Table 1. Elementary traits for composing collection behavior

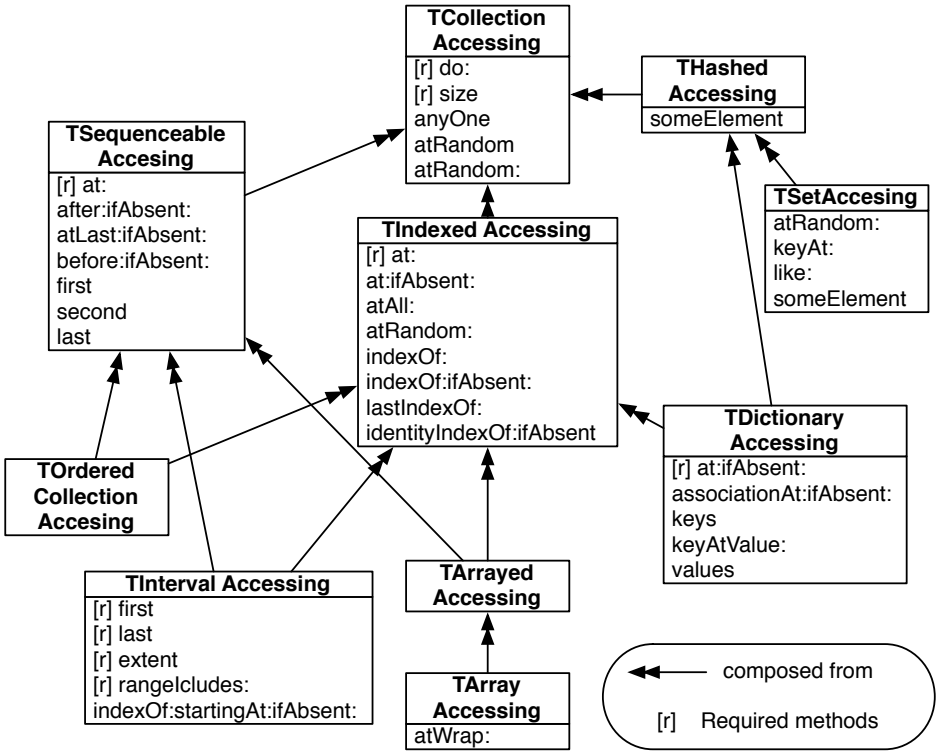

Figure 5. New trait hierarchy [r] represents required methods which are provided by primary methods or other Traits (secondary methods)

5.1 OrderedSet

We would like to define a new collection named OrderedSet that on the one hand offers a hash-based access and on the other hand an ordered access to its elements. Note that this collection is different from an UniqueOrdered collection that makes sure that its elements are ordered and not duplicated. For this goal, we create the new collection OrderedSet with the library of traits designed previously. As explained in Section 4.3, we simply create the structure to the new collection, create primary methods and use necessary traits. For this case study, we use two traits: TOrdered and TSet.

The following code shows how OrderedSet is defined. It uses all TOrdered traits and all TSet traits expect a few methods which are defined in both TSet and TOrdered. For example addAll: is defined in TOrdered, so the one in TSet is not needed.

```
Collection subclass: #OrderedSet
        uses: TSetAdding - {#addAll:} + TSetArithmetic + TSetTest-
ing - {#=. #isSequenceable} + TSetIterate - {#doWithIndex:. #se-
lect:thenCollect:} + TSetRemoving - {#removeAll} + TSetAccessing -
{#atRandom:} + TSetCopying - {#copyEmpty. #copyWith:. #copyWith-
out:} + TUnique + TSetCollection + TSetCreation + TOrderedAdding + TOrderedAc-
cessing + TSort + TOrderedIterate + TOrderedCreation + TOrdered-
Collection + TOrderedRemoving + TOrderedCopying + TOrderedEr-
ror + TSequenceableTesting + TOrderedUpdatable
    instanceVariableNames: 'array arrayO tally firstIndex lastIndex'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'BLOC'
```

OrderedSet reuses 70 methods (Figure 6). Then, we only have to reimplement or change the 38 required methods because the structure is particular: it contains two arrays to encode the two specific behaviors: hash access and order.

This example confirms the reusability of the BLOC library. Indeed, we create the new collection in less than two hours. In addition we do not have duplication code.

5.2 Dictionary

When creating the design of BLOC, we redefined certain existing collections such as Set, OrderedCollection, Interval .... This way we

TIndexed	Most indexed collections can retrieve elements with at:.
TSequenceable	Instances of all subclasses of SequenceableCollection start from a first element and proceed in a well-defined order to a last element.
THashed	Collection hashed used an hash function to store and access elements. This trait uses methods scanFor: and findElementOrNil:.
TOrdered	The *Ordered* behavior represents collections which are indexed and sequenceable.
TSet	There are methods for a set of unique elements and without nil.
TDictionary	It represents the behavior of a dictionary *i.e.,* it is an indexed collection which uses key as index. Keys permit to have attached elements. The couple key→element is stored in the collection.
TArrayed	It is a collection with a fixed size. It has the same behavior than OrderedCollection without growing behavior. This behavior is represented by a lack of the Trait: TArrayedAdding does not exist.

Table 2. Principal behavior-specific traits for collections in Pharo Smalltalk

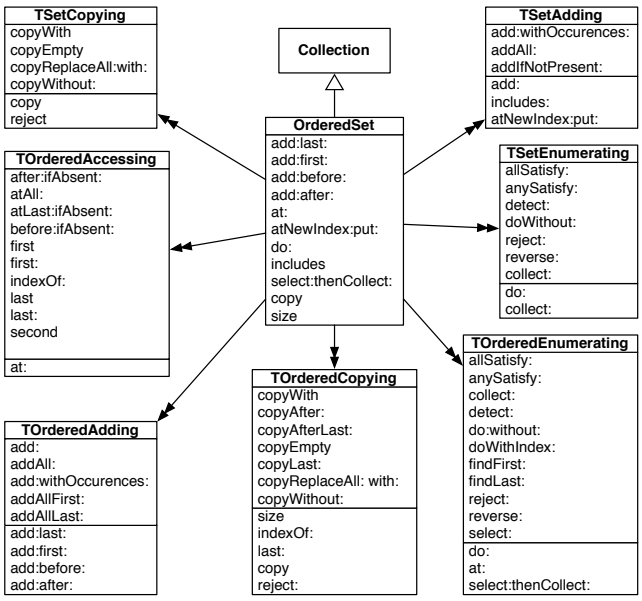

Figure 6. Map of traits used by OrderedSet

avoided code duplication. Dictionary is an example of good refactoring. In the existing library, Dictionary inherits from HashedCollection to have hash function. In addition, Dictionary is an indexed collection. This behavior is duplicated from SequenceableCollection and its subclasses.

In the original Dictionary class, there is some duplicated code, such as do: and associationsDo: which provides the same algorithm. With BLOC, Dictionary class uses 2 groups of traits TIndexed and THashed and defines some methods specific to the Dictionary.

```
Trait named: #TDictionaryAccessing
    uses: THashedAccessing + TIndexAccessing
    category: 'BLOC-Dictionary'
```

We redefined using traits the following collections: OrderedCollection, Interval, SortedCollection and Set. Now this redesign does not systematically improved existing code since some classes like Set did not present duplicated code. However, redesigning them and using traits (1) brings uniformity to the library, (2) core classes are the first clients of the traits they use, (3) it avoids duplication between such traits and their future clients. Finally an important point of the new design is that the use of traits did not hamper efficiency of the collection.

During the creation of BLOC, we discovered a difficulty: how to transform existing methods invoking super into traits. Indeed, invoking super in a trait a sign of not totally rethought functionality since it means that the trait is designed to be plugged in a hierarchy where the superclass is somehow fixed by the API it should offer to the trait. This is against the trait philosophy to be orthogonal to inheritance. Since the existing hierarchy is heavily based on inheritance, we had to face such situations. For example, OrderedCollection uses the method asSortedArray. This method uses super to call asSortedArray of SequenceableCollection but in our new hierarchy we only inherits from Collection. Therefore we have to redefine all the methods which use super to call a method that by construction may not be in the superclass. This example shows that traits are useful to avoid problems from the single inheritance. Indeed, traits permit to simulate a multiple inheritance without state.

5.3 Reusability Comparison

Table 3 presents how much the core traits are reused. For each trait it presents the number of client classes, the number of required methods and the number of methods that the trait provides. We see a good ratio provided/required for most traits, except for Interval. There are multiple reasons for this difference: Interval is more specialized than the other collections. As a consequence a large part of its API is tailored towards specific behavior and methods access directly the interval underlying structure. This explains the larger number of required methods. Note that this is a consequence of our design decision to avoid accessors and their associated cost. In presence of a JIT such point could be changed.

Table 5 presents some metrics which compare the same functionalities in the Pharo implementation and in BLOC. Note that the table presents the sum of traits for a given category: for example, TOrdered is the sum of all the traits related to the Ordered behavior. Which one indicates that BLOC has much more classes and Traits than Pharo collections and (number of methods) show that the amount of code is smaller in BLOC than in original library. BLOC has 10.9% less methods than the corresponding Pharo collection library. This means we avoided reimplementing a lot of methods by putting them in Traits. Finally, we can deduce from which that the design of BLOC is better: there are fewer cancelled methods and there are half as many methods less in BLOC than in Pharo.

Trait	client classes	required methods	provided methods	ratio prov. / req.
TCollection	3	10	92	9,20
TSequenceable	3	5	55	11
TIndexed	3	2	51	25,50
THashed	2	7	11	1,57
TOrdered	1	6	28	4,66
TSet	1	3	21	7
TDictionary	1	9	45	5
TArrayed	1	0	21	
TInterval	1	16	6	0,375

Table 3. BLOC-trait reusability.

Trait	required methods	provided methods	ratio prov. / req.
TSequenceableAccessing	1	17	17.0
TSequenceableCollection	1	3	3.0
TSequenceableConcatenation	0	1	
TSequenceableCopying	1	16	16.0
TSequenceableCreation	0	2	
TSequenceableIterate	2	9	4.5
TSequenceableRemoving	3	0	0.0
TSequenceableTesting	3	1	0.33
TSequenceableUpdatable	5	6	1.2

Table 4. Sequenceable-trait reusability.

	Pharo	BLOC	$\frac{Pharo-\text{BLOC}}{Pharo}$
# Classes and Traits	8	84	-950%
# Methods	510	454	10,9%
# Cancelled Methods	6	2	66%
# Reimplemented Methods	79	36	54%

Table 5. Some metrics comparing BLOC and pharo collection kernel.

6. Discussions

Granularity of traits. There is no definitive answer to the good granularity of traits but what we learn is that to enable reuse fine-grained traits are mandatory. Indeed, if we want to avoid duplicating code, the traits have to be small. Now pushing the idea the extreme, we could have one trait for one method. In such a case each method will be defined once. Now this is clearly not a good idea since we want also traits to represent a abstraction or a partial behavior. Adequate granularity is defined by the context. We have found a good granularity for trait in our context. Table 4 displays all traits related to SequenceableCollection. Some traits have no provided methods, because these methods are provided in TCollection. Note that the number of provided methods is variable and depends on the behavior provided.

Trait reusability. The reuse of trait depends on the behaviors. Indeed, for the collection we have a good reusability. We can now easily create different collections with BLOC. But all source code could not be reused because some methods depend on the underlying structure. In our solution, we removed a lot of reusability constraints except for methods which access state.

Trait composition vs Inheritance. One of the questions when building a system with traits is to decide when to use inheritance and when to use traits. In the Collection hierarchy (see Section 3.1), defining a class or inheriting from a class does not make sense since some of its state cannot be used or its behavior should be canceled. This is a clear motivation for using traits. Most of the time, however, the decision is not that easy to take, and the designer has to assess whether potential clients may benefit from the traits, i.e., if the defined behavior can be reused in another hierarchy.

Traits with state. In our work, we looked at the importance to have state in traits. In the context of collections we think that it is not necessary. If state is included in a trait, it also includes constraints for the implementation of future classes. In our context, to have state in traits is not necessary because of the definition of primary methods. The initialization of the state and its recomposition when used by different clients is also a problem that we did not assess but that should not be neglected.

Trait problems and limits. During our experience, we detected some limits and problems related to traits. The first problem we had was the lack of browser or tools for traits. Indeed, it is difficult to see traits, which classes use them, documentation, required methods, ... Traits are arbitrarily in the use: clause of the class definition. Therefore it becomes difficult to read what traits are used by the class.

7. Related work

Traits. We already compared our approach with the few work refactoring existing code using traits. Now we want to summarize the key differences.

Cassou et. al [9] rewrote the Stream Smalltalk hierarchy from scratch. What is interesting is that they obtained a kernel based on traits that can be assembled to reproduce the old kernel as well as express a completely new design.

Ducasse et. al reuse and compose unit tests out of traits for the collection hierarchy [13]. This work is closer to our approach since they focused on identifying elementary collection behavior. Then they used these elementary behavior to assemble tests for traits.

[6] proposed a refactoring of the existing collection but they were bound to the existing hierarchy. The work presented in this paper was focusing more on rethinking the collection as assembly of composable behaviors.

[18] and [3] proposed to use FCA to help automatically refactoring and identifying traits in Smalltalk and Java programs. The results are not as good as a manual approach because design is complex and FCA is just an indication that some methods could be optimally reused.

Automatic code reorganization of non traits code . We now present the approaches that automatically transform existing libraries using Formal Concept Analysis (FCA) or other techniques. FCA was used in different ways.

Godin [15] developed incremental FCA algorithms to infer implementation and interface hierarchies guaranteed to have no redundancy. To assess their solutions they used structural metrics. They analyzed the Smalltalk Collection hierarchy. One important limitation is that they consider each method declaration as a different method and thus cannot identify code duplication. Since the resulting hierarchies cannot be implemented in Smalltalk because of single inheritance, it would be interesting to understand whether their results could be indication for traits.

Snelting and Tip analyzed a class hierarchy by making the relationship between class members and variables explicit [21]. By analyzing the hierarchy client *usage*, they detected design anomalies such as class members that are redundant or that can be moved into a derived class. From this client perspective, Streckenbach infer improved hierarchies in Java [22]. They proposed solution that should be further be manually adapted. The tool proposes the reengineer to move methods up in the hierarchy to work around multiple inher-

itance situations generated by the generated lattice. The resulting refactoring is behavior preserving only with respect to the analyzed client programs.

Moore [19] proposes automatic refactoring of Self inheritance hierarchies. Moore factors out common expressions in methods. Resulting hierarchies do not contain any duplicated expressions or methods. Moore's factoring creates methods with meaningless names which is a problem if the code should be read. The approach is more optimizing method reuse than creating coherent composable groups of methods.

Casais uses an automatic structuring algorithm to reorganize Eiffel class hierarchies using decomposition and factorization [8]. In his approach, he increases the number of classes in the new refactored class hierarchy. Dicky *et al.* propose a new algorithm to insert classes into a hierarchy that takes into account overridden and overloaded methods [11].

The key difference from our results is that all the work on hierarchy reorganization focuses on transforming hierarchies using inheritance as the only tool. In contrast, we are interested in exploring other mechanisms, such as explicit composition mechanisms like traits composition in the context of mixin-like languages. Another important difference is that we don't rely on algorithms, to obtain the design.

8. Conclusion

In this paper we assessed the traits in a reuse context. We refactored the collection library to create a library of traits which can be composed into the behavior of main collections. This work represents a preliminary experience. A lot of questions has been raised, with no answer in this work. The need of stateful traits or the granularity is defined depending on the case. However, this study confirmed some goals of traits. The results of modularity and reusability offered by traits are good on the collection library. As future work, we need to better investigation of how to use traits, how to better define granularity. It is also important to define a browser to navigate between Traits, classes, behavior, documentation.

References

[1] ANSI, New York. *American National Standard for Information Systems – Programming Languages – Smalltalk, ANSI/INCITS 319-1998*, 1998. http://wiki.squeak.org/squeak/uploads/172/standard_v1_9-indexed.pdf.

[2] A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. Stateful traits. In *Advances in Smalltalk — Proceedings of 14th International Smalltalk Conference (ISC 2006)*, volume 4406 of *LNCS*, pages 66–90. Springer, Aug. 2007.

[3] L. Bettini, V. Bono, and M. Naddeo. A trait based re-engineering technique for java hierarchies. In *PPPJ 2008*, ACM International Conference Proceedings. ACM Press, 2008.

[4] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. *Squeak by Example*. Square Bracket Associates, 2007. http://SqueakByExample.org/.

[5] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. *Pharo by Example*. Square Bracket Associates, 2009.

[6] A. P. Black, N. Schärli, and S. Ducasse. Applying traits to the Smalltalk collection hierarchy. In *Proceedings of 17th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'03)*, volume 38, pages 47–64, Oct. 2003.

[7] G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices*, volume 25, pages 303–311, Oct. 1990.

[8] E. Casais. Automatic reorganization of object-oriented hierarchies: A case study. *Object-Oriented Systems*, 1(2):95–115, Dec. 1994.

[9] D. Cassou, S. Ducasse, and R. Wuyts. Traits at work: the design of a new trait-based stream library. *Journal of Computer Languages, Systems and Structures*, 35(1):2–20, 2009.

[10] W. R. Cook. Interfaces and Specifications for the Smalltalk-80 Collection Classes. In *Proceedings of OOPSLA '92 (7th Conference on Object-Oriented Programming Systems, Languages and Applications)*, volume 27, pages 1–15. ACM Press, Oct. 1992.

[11] H. Dicky, C. Dony, M. Huchard, and T. Libourel. On Automatic Class Insertion with Overloading. In *Proceedings of OOPSLA '96 (11th ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications)*, pages 251–267. ACM Press, 1996.

[12] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. P. Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):331–388, Mar. 2006.

[13] S. Ducasse, D. Pollet, A. Bergel, and D. Cassou. Reusing and composing tests with traits. In *Proceedings of the 47th International Conference Objects, Models, Components, Patterns (TOOLS-Europe'09)*, pages 252–271, jun 2009.

[14] S. Ducasse, R. Wuyts, A. Bergel, and O. Nierstrasz. User-changeable visibility: Resolving unanticipated name clashes in traits. In *Proceedings of 22nd International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'07)*, pages 171–190, New York, NY, USA, Oct. 2007. ACM Press.

[15] R. Godin, H. Mili, G. W. Mineau, R. Missaoui, A. Arfi, and T.-T. Chau. Design of Class Hierarchies based on Concept (Galois) Lattices. *Theory and Application of Object Systems*, 4(2):117–134, 1998.

[16] A. Goldberg. *Smalltalk 80: the Interactive Programming Environment*. Addison Wesley, Reading, Mass., 1984.

[17] A. Goldberg and D. Robson. *Smalltalk 80: the Language and its Implementation*. Addison Wesley, Reading, Mass., May 1983.

[18] A. Lienhard, S. Ducasse, and G. Arévalo. Identifying traits with formal concept analysis. In *Proceedings of 20th Conference on Automated Software Engineering (ASE'05)*, pages 66–75. IEEE Computer Society, Nov. 2005.

[19] I. Moore. Automatic Inheritance Hierarchy Restructuring and Method Refactoring. In *Proceedings of OOPSLA '96 (11th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications)*, pages 235–250. ACM Press, 1996.

[20] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable units of behavior. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'03)*, volume 2743 of *LNCS*, pages 248–274. Springer Verlag, July 2003.

[21] G. Snelting and F. Tip. Reengineering Class Hierarchies using Concept Analysis. In *ACM Trans. Programming Languages and Systems*, 1998.

[22] M. Streckenbach and G. Snelting. Refactoring class hierarchies with KABA. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 315–330, New York, NY, USA, 2004. ACM Press.

A. Appendix: Traits in a Nutshell

To ease the understanding of this paper we added this section which presents traits in a nutshell. This part is taken from [14] and is not part of the current article. It is just added here for sake of completeness and understanding the ideas presented in the paper.

Reusable groups of methods. Traits are units of behaviour. They are sets of methods that serve as the behavioural building block of classes and primitive units of code reuse [12]. In addition to offering behaviour, traits also *require methods*, *i.e.,* methods that are needed so that trait behaviour is fulfilled. Traits do not define state, instead they require accessor methods.

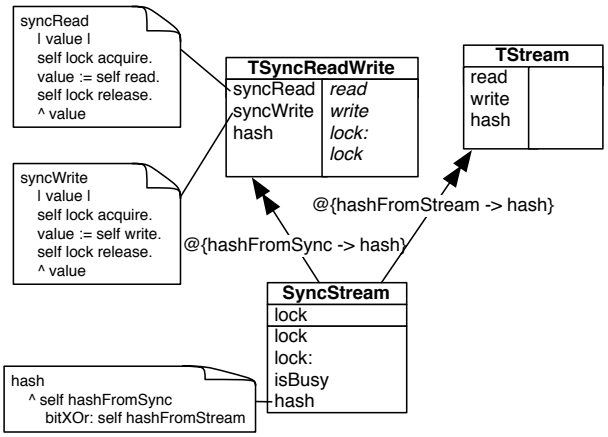

Figure 7. The class SyncStream is composed of the two traits TSyncReadWrite and TStream.

Figure 7 shows a class SyncStream that uses two traits, TSyncReadWrite and TStream. The trait TSyncReadWrite provides the methods syncRead, syncWrite and hash. It requires the methods read and write, and the two accessor methods lock and lock:. We use an extension to UML to represent traits (the right column lists required methods while the left one lists the provided methods).

Explicit composition. A class contains a super-class reference, uses a set of traits, defines state (variables) and behaviour (methods) that glue the traits together; a class implements the required trait methods and resolves any method conflicts.

Trait composition respects the following three rules:

- Methods defined in the composer take precedence over trait methods. This allows the methods defined in a composer to override methods with the same name provided by the used traits; we call these methods *glue methods*.
- Flattening property. In any class composer the traits can be in principle in-lined to give an equivalent class definition that does not use traits.
- Composition order is irrelevant. All the traits have the same precedence, and hence conflicting trait methods must be explicitly disambiguated.

Conflict resolution. While composing traits, method conflicts may arise. A *conflict* arises if we combine two or more traits that provide identically named methods that do not originate from the same trait. There are two strategies to resolve a conflict: by implementing a (glue) method at the level of the class that *overrides* the conflicting methods, or by *excluding* a method from all but one trait. Traits allow method *aliasing*; this makes it possible to introduce an additional name for a method provided by a trait. The new name is used to obtain access to a method that would otherwise be unreachable because it has been overridden [12].

In Figure 7, the class SyncStream is composed from TSyncReadWrite and TStream. The trait composition associated to SyncStream is:

TSyncReadWrite alias *hashFromSync* → *hash*
+ TStream alias *hashFromStream* → *hash*

The class SyncStream is composed of (i) the trait TSyncReadWrite for which the method hash is aliased to hashFromSync and (ii) the trait TStream for which the method hash is aliased to hashFromStream.

Method composition operators. The semantics of trait composition is based on four operators: sum (+), override (▷), exclusion (−) and aliasing (alias →) [12].

The *sum* trait TSyncReadWrite + TStream contains all of the non-conflicting methods of TSyncReadWrite and TStream. If there is a method conflict, that is, if TSyncReadWrite and TStream both define a method with the same name, then in TSyncReadWrite + TStream that name is bound to a known method conflict. The + operator is associative and commutative.

The *override* operator (▷) constructs a new composition trait by extending an existing trait composition with some explicit local definitions. For instance, SyncStream overrides the method hash obtained from its trait composition.

A trait can *exclude* methods from an existing trait using the exclusion operator −. Thus, for instance, TStream − {read, write} has a single method hash. Exclusion is used to avoid conflicts, or if one needs to reuse a trait that is "too big" for one's application.

The *method aliasing* operator alias → creates a new trait by providing an additional name for an existing method. For example, if TStream is a trait that defines read, write and hash, then TStream alias *hashFromStream* → *hash* is a trait that defines read, write, hash and hashFromStream. The additional method hashFromStream has the same body as the method hash. Aliases are used to make conflicting methods available under another name, perhaps to meet the requirements of some other trait, or to avoid overriding. Note that since the body of the aliased method is not changed in any way, an alias to a recursive method is not recursive.

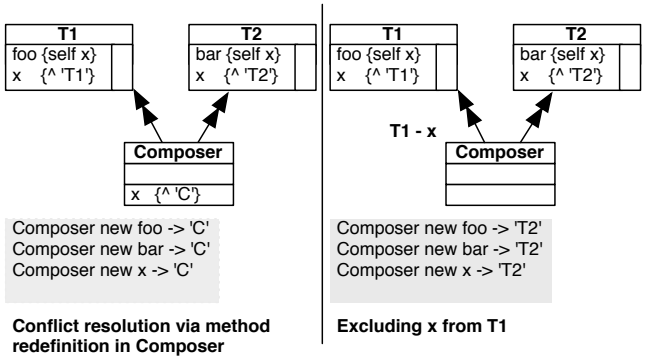

Figure 8. Trait conflict resolution strategies: either via method redefinition or via method exclusion.

Selected Papers of the Int. Workshop on Smalltalk Technologies, Barcelona, Spain, September 14, 2010

SPY: A flexible and Agile Code Profiling Framework

Alexandre Bergel Felipe Bañados
Romain Robbes
DCC, University of Chile
Santiago,Chile
www.bergel.eu
www.dcc.uchile.cl/∼fbanados
www.dcc.uchile.cl/∼rrobbes

David Röthlisberger
University of Bern
Switzerland
www.droethlisberger.ch

Abstract

Code profiling is an essential activity to increase software quality. It is commonly employed in a wide variety of tasks, such as supporting program comprehension, determining execution bottlenecks, and assessing code coverage by unit tests.

SPY is an innovative framework to easily build profilers and visualize profiling information. The profiling information is obtained by inserting dedicated code before or after method execution. The gathered profiling information is structured in line with the application structure in terms of packages, classes, and methods. SPY has been instantiated on four occasions so far. We created profilers dedicated to test coverage, time execution, type feedback, and profiling evolution across version. We also integrated SPY in the Pharo IDE.

SPY has been implemented in the Pharo Smalltalk programming language and is available under the MIT license.

1. Introduction

Profiling an application commonly refers to obtaining dynamic information from a controlled program execution. Common usages of profiling techniques include test coverage [14], time execution monitoring [5], type feedback [1, 2, 11], or program comprehension [12, 17]. The analysis of gathered runtime information provides important hints on how to improve the program execution. Runtime information is usually presented as numerical measurements, such as number of method invocations or number of objects created in a method, making them easily comparable from one program execution to another.

Even though computing resources are abundant, execution optimization and analysis through code profiling remains an important software development activity. Program profilers are crucial tools to identify execution bottlenecks and method call graphs. Any respectable and professional programming environment includes a code profiler. Pharo Smalltalk and Eclipse, for instance, both ship a profiler [6, 9].

A number of code profilers are necessary to address the different facets of software quality [15]: method execution time and call graph, test coverage, tracking nil values, just to name a few. Providing a common platform for runtime analysis has not yet been part of a joint community effort. Each code profiler tool traditionally comes with its own engineering effort to both acquire runtime information and properly present this information to the user.

Most Smalltalk systems offer a flexible and advanced programming environment. Over the years different Smalltalk communities have been able to propose tools such as the system browser, the inspector or the debugger. These tools are the result of a community effort to produce better software engineering techniques and methodologies. However, code profilers have little evolved over the years, becoming more an outdated Smalltalk heritage than a spike for innovation. A survey of several Smalltalk implementations—Squeak [13], Pharo [6], VisualWorks [19], and GemStone—reveals that none shines for its execution profiling capabilities: indented textual output holds a royal position (see Section 2).

In the Java world, JProfiler[1] is an effective runtime execution profiler tool that, besides measuring method execution time, also offers numerous features including snapshot comparisons, saving a profiling trace in an XML file and estimating method call graphs. Cobertura[2] is a tool dedicated to measure test coverage. Similarly to JProfiler, test coverage information may be stored in an XML file which contains method call graph analysis and coverage. However, JProfiler and Cobertura do not share any library besides the standard Java libraries. There are multiple reasons why JProfiler and Cobertura are separated from each other even though both have to gather similar runtime information. One of them is certainly a lack of a common profiling framework.

This paper presents SPY, a framework for easily prototyping various types of code profilers in Smalltalk. The dynamic information returned by a profiler is structured along the static structure of the program, expressed in terms of packages[3], classes and methods. One principle of SPY is structural correspondence: the structure of meta-level facilities correspond to the structure of the language manipulated[4]. Once gathered, the dynamic information can easily be graphically rendered using the Mondrian visualization engine [16][5].

SPY has been used to implement a number of code profilers. The SPY distribution offers a type feedback mechanism, an execution profiler [4], an execution evolution profiler, and a test coverage profiler. Creating a new profiler comes at a very light cost as SPY relieves the programmer from performing low-level monitoring.

To ease the description of the framework, SPY is presented in a tutorial like fashion: We document how we instantiated the framework in order to build a code coverage tool. The main contributions of this paper are summarized as follows:

- The presentation of a flexible and general code profiling framework.

[1] `http://www.ej-technologies.com/products/jprofiler/screenshots.html`

[2] `http://cobertura.sourceforge.net`

[3] In Pharo, the language used for the experiment, a package is simply a group of classes.

[4] According to the terminology provided by Bracha and Ungar [7], ensuring structural correspondence makes SPY a mirror-based system.

[5] `http://www.moosetechnology.org/tools/mondrian`

- The construction of an expressive test coverage tool as an example of the framework's usage.
- The demonstration of the framework flexibility, via the description of three additional framework instantiation, and of its integration with Mondrian and Smalltalk code browsers.

The paper is structured as follows: first, a brief survey of Smalltalk profilers is provided (Section 2). The description of SPY (Section 3) begins with an enumeration of the different composing elements (Section 3.1) followed by an example (Section 3.2 – Section 3.6). The practical applicability of SPY is then demonstrated by means of three different situations (Section 4) before concluding (Section 5).

2. Current Profiler Implementations

This section surveys the profiling capabilities of the Smalltalk dialects and implementations commonly available.

Squeak. Profiling in Squeak[6] is achieved through the MessageTally class (MessageTally>> spyOn: aBlock). As most profilers, MessageTally employs a sampling technique, which means that a high-priority process regularly inspects the call stack of the process in which aBlock is evaluated. The time interval commonly employed is one millisecond.

MessageTally shows various profiling information. The method call graph triggered by the evaluation of the provided block is shown as a hierarchy which indicates how much time was spent, and where. Consider the expression MessageTally spyOn: [MOViewRendererTest buildSuite run]. It simply profiles the execution of the tests contained in the class MOViewRendererTest. The call graph is textually displayed as:

```
75.1% {10257ms} TestSuite>> run:
  75.1% {10257ms} MOViewRendererTest(TestCase)>> run:
    75.1% {10257ms} TestResult>> runCase:
      75.1% {10257ms} MOViewRendererTest(TestCase)>> runCase
      ...
```

This information is complemented by a list of leaf methods and memory statistics.

Pharo. Pharo is a fork of Squeak and its profiling capabilities are very close to those of Squeak. TimeProfiler is a graphical facade for MessageTally. It uses an expandable tree widget to comfortably show profiling information (Figure 1).

Gemstone. The class ProfMonitor allows developers to sample the methods that are executed in a given block of code and to estimate the percentage of total execution time represented by each method[7]. It provides essentially the same ability as MessageTally in Squeak. One minor variation is offered: methods can be filtered from a report according to the number of times they were executed (ProfMonitor>> monitorBlock:downTo:interval:).

VisualWorks. The largest number of profiling tools are available in VisualWorks[8]. First, a *profiler window* offers a list of code templates to easily profile a Smalltalk block: profiling result may be directly displayed or stored in a file. Statistics may also be included.

VisualWorks uses sampling profiling. Repeating the code to be profiled, with timesRepeat: for example, increases the accuracy of the sampling. An additional mechanism to control accuracy is to graphically adjust the sampling size.

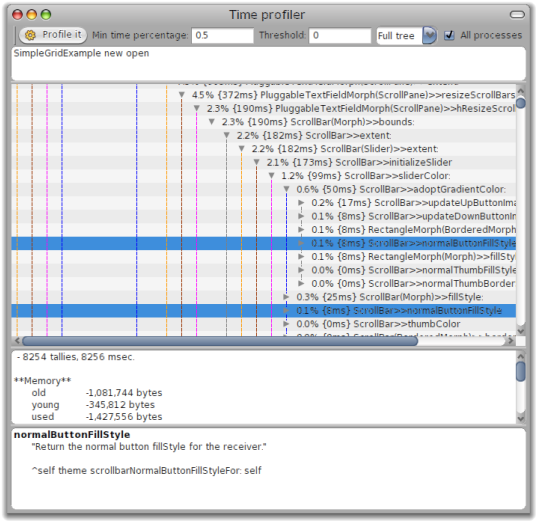

Figure 1. TimeProfiler in Pharo

The profiling information obtained in VisualWorks is very similar to MessageTally's. It is textually rendered, indentations indicate invocations in a call graph, and execution times are provided in percentage and milliseconds. Methods may be filtered out based on their computation time. Similarly to TimeProfiler, branches of the call tree may be contracted and expanded.

Conclusion. The Smalltalk code profilers available are very similar. They provide a textual list of methods annotated with their corresponding execution time share. None of these profilers is easily extensible to obtain a different profiling such as test coverage. The SPY framework described in the following addresses particularly this issue.

3. The SPY Framework

3.1 SPY in a nutshell

The essential classes of SPY are depicted in Figure 2 and explained in the following:

- The Profiler class contains the features necessary for obtaining runtime information by profiling the execution of a block of Smalltalk code. Profiler offers a number of public class methods to interface with the profiling. The profile: aBlock inPackagesNamed: packageNames method accepts as first parameter a block and as second parameter a collection of package names. The effect of calling this method is to (i) instrument the specified packages; (ii) to execute the provided block; (iii) to uninstrument the targeted packages; and (iv) to return the collected data in the form of an instance of the Profiler class which contains instances of the classes described below, essentially mirroring the structure of the program.

  Profiling results are globally accessible by other development tools. The method registryName has to be be overridden to return a name. Other IDE tools can then easily access to the profiling.
- PackageSpy contains the profiling data for a package. Each instance has a name and contains a set of class spies.

[6] http://wiki.squeak.org/squeak/4210

[7] Page 301 in http://www.gemstone.com/docs/GemStoneS/GemStone64Bit/2.4.3/GS64-ProgGuide-2.4.pdf

[8] Page 87 in http://www.cincomsmalltalk.com/documentation/current/ToolGuide.pdf

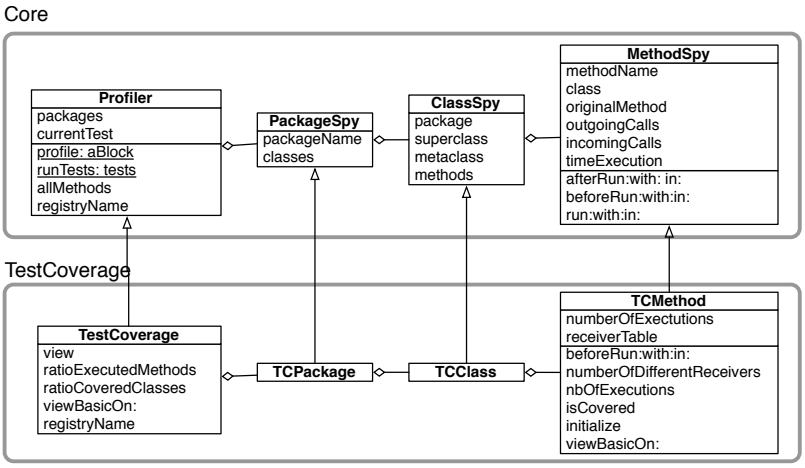

Figure 2. Structure of SPY

- ClassSpy describes a Smalltalk class. It has a name, a superclass spy, a metaclass spy and a set of method spies.
- MethodSpy describes a method. It has a selector name and belongs to a class spy. MethodSpy is central to SPY. It contains the hooks used to collect runtime information. Three methods are provided for that purpose: beforeRun:with:in: and afterRun:with:in: are executed before and after the corresponding Smalltalk method. These empty methods may be overridden in subclasses of MethodSpy to collect relevant dynamic information, as we will see in the following subsections. The run:with:in method simply calls beforeRun:with:in:, followed by the execution of the represented Smalltalk method, and ultimately calls afterRun:with:in:. The parameters passed to these methods are: the method name (as a symbol), the list of arguments, and the object that receives the intercepted message.

The SPY framework is instantiated by creating subclasses of PackageSpy, ClassSpy, MethodSpy and Profiler, all specialized to gather the precise runtime information that is needed for a particular system and task.

3.2 Instantiating SPY

Test coverage. We motivate and demonstrate the usage of the SPY framework by building a test coverage code analyzer. Identifying the coverage of the unit tests of an application may be considered as a code profiling activity. A simple profiling reveals the number of covered methods and classes. This is what traditional test coverage tools produce as output (e.g., Cobertura).

We go one step further with our test coverage tool running example. In addition to raw metrics such as percentage of covered methods and classes, we retrieve and correlate a variety of dynamic and static metrics:

- number of method executions – *how many times a particular method has been executed.*
- number of different object receivers – *on how many different objects a particular method has been executed.*
- number of lines of code – *how complex the method is.* We use the method code source length as a simple proxy for complexity.

The intuition behind our test coverage tool is to indicate what are the "complex" parts of a system that are "lightly" tested, and what are the "apparently simple" components that are "extensively" tested. There is clearly no magic metric that will precisely identify such a complex or simple software component. However, correlating a complexity metric (*i.e.,* number of lines of code in our case) with how much a component has been tested (*i.e.,* number of executions and number of different receivers) provides a good indication about the quality of the test coverage.

Instantiating SPY. The very first step to build our test coverage tool is to subclass the relevant classes. TestCoverage, TCPackage, TCClass, and TCMethod, respectively, subclass Profiler, PackageSpy, ClassSpy and MethodSpy.

```
Profiler subclass: #TestCoverage

PackageSpy subclass: #TCPackage

ClassSpy subclass: #TCClass

MethodSpy subclass: #TCMethod
  instanceVariableNames: 'numberOfExecutions receiverTable'
```

TCMethod defines two variables, numberOfExecutions and receiverTable. The former variable is initialized as 0 and is incremented for each method invocation. The latter keeps track of the number of receiver objects on which the method has been executed. Recording the hash value of each receiver object can be easily implemented to provide a good approximation of the number of receivers in most cases.

```
TCMethod >> initialize
  super initialize.
  numberOfExecutions := 0.
  receiverTable := BoundedSet maxSize: 100
```

The class BoundedSet is a subclass of Set in which the number of different values is no greater than a limit. In our case, no more than 100 different elements may be inserted in a bounded set. This value is actually arbitrary and depends very much on how the related metric will be used. In our environment, for the types of programs we write, given the resources we can expend, we have not been able to devise a way to efficiently and easily keep track of all receiver objects of a method call. Using an ordered collection in which we insert the object receiver at each invocation is not practically exploitable. There is a number of reasons for

this. As soon as a method is called many times, *e.g.,* one million times, then one million elements have been added to the collection. Allowing the ordered collection to grow up to one million elements significantly slows down the overall program execution. In addition to this, identifying the number of different elements in a list with one million elements is also slow. The same schema applies for all the recursively called methods.

The method beforeRun:with:in: is executed before the original method. We simply increment the execution counter, and record the receiver.

```
TCMethod>> beforeRun: selector with: args in: receiver
  numberOfExecutions := numberOfExecutions + 1.
  receiverTable at: receiver hash put: true.
```

A number of utility methods are then necessary:

```
TCMethod>> isCovered
  ^ numberOfExecutions > 0

TCMethod>> numberOfExecutions
  ^ numberOfExecutions

TCMethod>> numberOfDifferentReceivers
  ^ (receiverTable select: #notNil) size
```

The ratio of executed methods and covered classes are defined on TestCoverage:

```
TestCoverage>> ratioExecutedMethods
  ^ ((self allMethods select: #isCovered) size /
        self allMethods size) asFloat

TestCoverage>> ratioCoveredClasses
  ^  ((self allClasses
        select: [ :cls |  cls methods anySatisfy: #isCovered ]) size /
        self allClasses size) asFloat
```

The method allClasses is defined on Profiler, the superclass of TestCoverage.

3.3 Running Spy

Our TestCoverage tool can be run using the profile:inPackagesNamed: class method. In this example, we run it on the test cases of the Mondrian visualization framework.

```
coverage :=
  TestCoverage
        profile: [ MOViewRendererTest buildSuite run ]
        inPackage: 'Mondrian'
```

Executing the code above returns an instance of TestCoverage.

3.4 Visualizing Runtime Information

The Mondrian visualization engine framework [16] easily produces visualizations. Mondrian is a visualization engine that offers a rich domain specific language to define graph-based rendering. Each element of a graph (*i.e.,* node and edge) has a shape that defines its visual aspect. Nodes may be ordered using a layout. Consider the method:

```
TestCoverage>> viewBasicOn: view
  view nodes: self allClasses forEach: [ :each |
        view shape rectangle
          height: #numberOfLinesOfCode;
          width: [ :m |  (m numberOfDifferentReceivers + 1) log * 10 ];
          linearFillColor:
                [ :m |  ((m numberOfExecutions + 1) log * 10) asInteger ]
          within: self allMethods;
          borderColor:
                [:m |  m isCovered
                      ifTrue: [ Color black ] ifFalse: [ Color red ] ].
        view interaction action: #inspect.
        view nodes: (each methods
                        sortedAs: #numberOfLinesOfCode).
        view gridLayout gapSize: 2.
  ].
  view edgesFrom: #superclass.
  view treeLayout
```

The visualization is rendered by evaluating:

```
coverage viewBasic
```

An excerpt of the visualization obtained is depicted in Figure 3. The displayed class hierarchy represents Mondrian shapes. The root is MOShape. The visualization has the following characteristics:

- Outer boxes are classes.
- Edges between classes represent class inheritance relationships. A superclass appears above and a subclass below a particular class node. A tree layout is used to order classes which is adequate since Smalltalk uses single inheritance.
- Inner boxes are methods. Methods are sorted according to their source code length.
- White boxes with a red border are methods that have not been executed when running the coverage.
- The height of a method is the number of lines of code.
- The width of a method is the number of different receivers. We use a logarithmic scale to accommodate the variability of this metric.
- The color of a method is the number of method executions. We use a logarithmic scale also for this metric.

From what is depicted in Figure 3, a number of patterns can be visually identified:

- Some classes contain red methods only. This means that the class is absent from all the execution scenarios specified in the tests.
- Red methods that are tall and thin are long, untested methods. They are excellent targets for new test additions.
- Gray methods (few executions) and narrow methods (few receivers) are probably good candidates for further testing.
- Dark and large methods are extensively tested.
- Horizontally flat methods are very extensively tested more since they contain just a few lines of code and are still executed many times.

As it is the case for most software visualizations, the goal of our test coverage visualization is not to precisely locate software deficiency. Rather, it aims at assisting the programmer to identify candidates for software improvement. In this case, the visualization pinpoints red methods, and thin, gray methods, as likely candidates to consider in order to improve the coverage of the code by tests.

3.5 Call graph and execution time

Profiler defines an instance method getTimeAndCallGraph which simply returns false. By overriding this method in a subclass to make it return true, the execution time (in milliseconds and percentage) and the call graph for each method is computed during the block execution.

```
TestCoverage>> getTimeAndCallGraph
  "Each instance of TCMethod contains information about
```

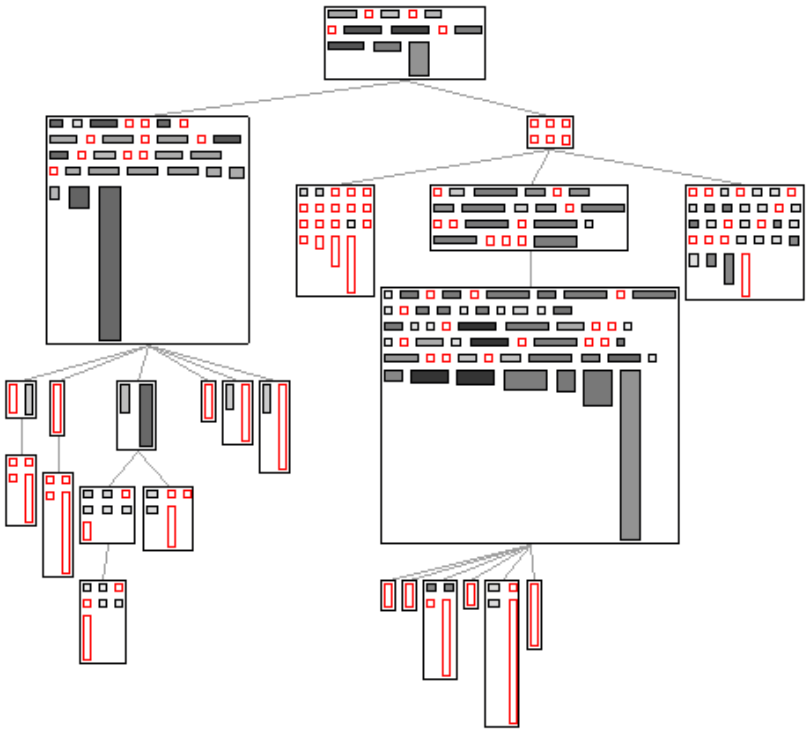

Figure 3. Test coverage visualization

```
    execution time and outgoing and incoming calls"
  ^ true
```

The call graph and execution time is estimated by regularly sampling the method call stack. For that very purpose, SPY contains a class called Sampling, which is a simplified version of MessageTally[9]. Each method spy will now store the execution time it took, as well as a list of outgoing calls and incoming calls.

By determining the method call graph from these incoming and outgoing calls, all packages involved during the block evaluation are easily identified. The profiling can now be realized using the profile: method. There is no need to provide a package name to extract the call graph of the execution.

```
coverage :=
  TestCoverage
        profile: [ MOViewRendererTest buildSuite run ]
```

Now that the method call graph is computed, we can add an entry point to a new visualization. The script defined in TestCoverage>> viewBasicOn: may be refined with a new menu item for methods:

```
        ...
        view interaction action: #inspect;
              item: 'view call graph' action: #viewBasic.
        view nodes: (each methods
                        sortedAs: #numberOfLinesOfCode).
        ...
```

For a user-selected method, the following script renders the method call graph, using the outgoingCalls method of MethodSpy:

```
TCMethod>> viewBasicOn: view
  | methods |
  methods := self withAllOutgoingCalls asSet.
  view shape rectangle
        height: #numberOfLinesOfCode;
        width: [:m | (m numberOfDifferentReceivers + 1) log * 10 ];
        linearFillColor: [ :m |  ((m numberOfExecutions + 1) log * 10)
                                    asInteger ]
        within: self package allMethods;
        borderColor: [ :m |  m isCovered
                                    ifTrue: [ Color black ]
                                    ifFalse: [ Color red ] ].

  view nodes: methods.
  view shape arrowedLine width: 2.
  view edges: methods from: #yourself toAll: #outgoingCalls.
  view treeLayout
```

The visualization we provide may be enriched with information about the method execution time. Overriding the printOn: method will change the text that is displayed by Mondrian when hovering the mouse over a node.

```
TCMethod>> printOn: stream
  super printOn: stream.
  stream nextPutAll: self executionTime printString, ' ms'
```

By right-clicking on a method node, a menu item render the call graph for the method (Figure 4). Methods are ordered from top to down. The arrowed edges represent the control flow between methods.

3.6 Summary

This section presented a simple application of SPY. It described the essential steps to create a code profiler: (i) recovering the required

[9] Sampling is not represented in Figure 2 since a user is not expected to use it directly.

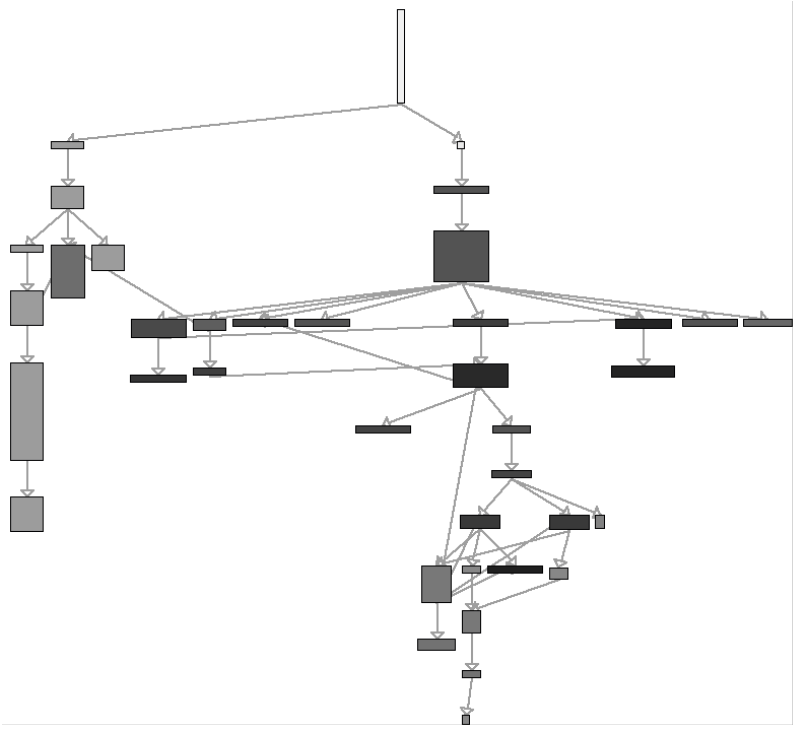

Figure 4. Call graph of the method MOViewRenderer>> testTranslation

profiling information by instantiating the framework; (ii) visualizing this information with Mondrian; (iii) gathering further execution and call graph information; and (iv) visualizing this additional information.

Effective profiling visualizations may be produced using Mondrian. The fact that the profiling information follows the code structure leads to comprehensive and familiar visualizations that are easy to implement as the profiling information's representation matches the one often used by Mondrian visualizations.

4. Applications

In this section, we present some of the profiling tools we built on top of SPY.

4.1 Extracting types from unit tests

As a first application of SPY, we proposed a mechanism for extracting type information from the execution of unit tests[10] [4]. For a given program written in Smalltalk, we can deduce the type information from executing the associated unit tests., as has been proposed by other researchers as well [18]. The idea is summarized as follows: (i) we instrument an application to record the runtime types of the arguments and return values of methods; (ii) we run the unit tests associated with the application; and (iii) we deduce the type information from what has been collected. The idea is to record the type of each message argument and return value to later deduce the most specialized types for each argument and return type. We refer to the most specialized type as the most direct supertype that is common for a set of classes. Method signatures of the base program are then determined by the values provided to and returned by method calls while the tests are being executed.

As a concrete use case, we exploit the extracted type information to find software faults. Type information combined with test coverage helps developers identifying methods that were not invoked with all possible type parameters. By covering these missing cases, we identified and fixed four anomalies in Mondrian.

4.2 Time profiling blueprints

As a second application, we proposed a time execution profiler[11]. Time profiling blueprints are graphical representations meant to help programmers (i) assess the program execution time distribution and (ii) identify and fix bottlenecks in a given program execution. The essence of profiling blueprints is to enable a better comparison of elements constituting the program structure and behavior. To render information, these blueprints use a graph metaphor, composed of nodes and edges.

The size of a node gives hints about its importance in the execution. When nodes represent methods, a large node means that the program execution spends "a lot of time" in this method. The expression "a lot of time" is then quantified by visually comparing the height and/or the width of the node against other nodes.

Color is used to either transmit a boolean property (*e.g.,* a gray node represents a method that always returns the same value) or a metric (*e.g.,* a color gradient is mapped to the number of times a method has been invoked).

We propose two blueprints that help identify opportunities for code optimization: the structural profiling blueprint visualizes the distribution of the CPU effort along the program structure and

[10] `http://www.moosetechnology.org/tools/Spy/Keri`

[11] `http://www.moosetechnology.org/tools/Spy/Kai`

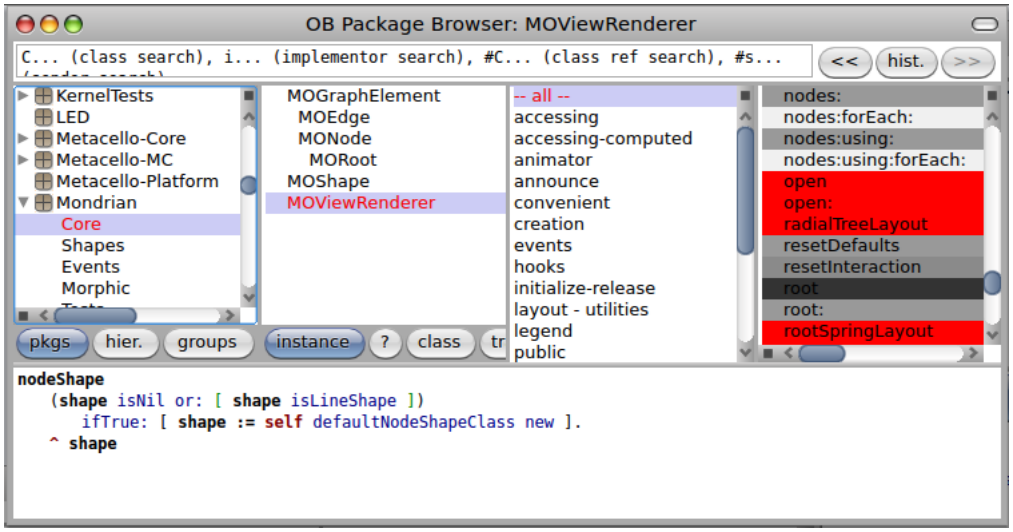

Figure 5. Integration of profiling information into the Pharo IDE

the behavioral profiling blueprint along the method call graph. These blueprints provide hints to programmers to refactor their program along the following two principles: (i) make often-used methods faster and (ii) call slow methods less often. The metrics we adopted in this paper help developers finding methods that are either unlikely to perform a side effect or always return the same result, good candidates for simple caching-based optimizations.

4.3 Profiling differentiation

The use of profiling information might be taken a step further by profiling different versions of an application. Spotting differences between them provides insights on the causes of slowdowns, and what should be improved next. Comparing, e.g., time profiling throughout a package's history allows one to confirm an optimization trial as an improvement and to find the potential bottlenecks that remain. The package Hip helps us in this task. Hip allows one to build a collection of history profiles, following a schema similar to the Hismo model [10]. Each method, class, and package profile can access the profiles of its previous and next version. Queries about metrics may be then formulated (e.g., *has a metric increased?*) as well as "differential measurements"[12] (e.g., *how much has a metric increased?*).

Hip provides facilities to automatically profile a block throughout a set of package versions available from a Monticello[13] repository by loading each version, profiling it, and adding the gathered profiling information to a Hip version collection structure.

Hip opens the door to a wide range of options to visualize the evolution of a program's runtime behavior. As an example, we propose a semaphore-like view that helps to identify bottlenecks. For a particular profiled object and version, Hip assigns one of five colors. In the case of a metric such as the execution time—where lower is better—source artifacts with a lower metric value compared to the previous version are colored green; those with a greater value red; unchanged artifacts are colored in white; removed ones black; and new ones yellow. The emphasis is on red and green artifacts for obvious reasons, and also on yellow artifacts, as from that version onward the developers should put focus on newly created artifacts, as they were not available before.

4.4 IDE integration

The primary tool developers use to develop and maintain software systems is the integrated development environment (IDE). For this reason we integrate profiling information gathered by SPY into Pharo's IDE which is implemented using the OmniBrowser framework [3]. As soon as a system's test suite has been executed with SPY, the IDE can access the test coverage information using the following statement:

```
Profiler profilerAt: #testCoverage
```

The Pharo IDE exploits the profiling information resulting from the execution of tests to highlight in the source code perspectives methods and classes that have been covered by the system's test suite. The same color scheme as introduced in Section 3.4 is used to highlight the source artifacts. A non-executed method is colored red to raise the awareness for untested code while methods colored dark (*e.g.,* in a gradient from gray to black) have been executed often and are hence tested extensively. Gray methods, that is, methods that have not been executed often by the test suite, are good candidates to look at in detail in order to reveal whether they could benefit from more extensive testing. Visualizing profiling information directly in the IDE hence helps developers to easily locate methods that should be better covered with tests to improve a system's test coverage. Figure 5 illustrates how profiling information is visualized in the Pharo IDE.

5. Conclusion

SPY is a profiling framework for the Pharo Smalltalk environment designed to easily build application profilers. Profiling output is structured along the static structure of the analyzed program composed of packages, classes and methods. The core of SPY is composed of four classes, Profiler, PackageSpy, ClassSpy and MethodSpy. These classes represent the profiler itself and profiling information for packages, classes and methods.

Once the data about a program's execution is gathered by SPY, one can explore the data by visualizing it using a dedicated visualization framework such as Mondrian.

[12] This term is commonly employed in electronic and voltage measurement. We consider it to be descriptive in our context.

[13] Monticello is the version control mechanism commonly employed in Pharo.

However, SPY is not cost free. Mondrian tests are 3 times slower when the coverage is computed. Future effort of SPY will be dedicated to reducing information gathering overhead based on bytecode transformation [8] and DTrace[14]. When method time execution matter, the user has always the option to rely on a second profiling "pass" triggered with the getTimeAndCallGraph option. The piece of code to profile is then executed a second time, using a sampling approach, less costly, but also less precise.

We have shown by a simple example how one can instantiate SPY for a given problem, such as building a code coverage tool. Furthermore, we have demonstrated the flexibility of SPY by presenting three additional applications we built on top of it, namely a type extraction profiler, a time profiling visualization tool, and an evolutionary time profiling visualization tool. Finally, we demonstrated that the information gathered via SPY is useful beyond visualization, as we integrated our code coverage profiler with the regular IDE, allowing a more direct interaction between the source code and its dynamic aspects.

Acknowledgment We gracefully thank Dave Ungar for his comment and feedback of our paper.

References

[1] O. Agesen and U. Holzle. Type feedback vs. concrete type inference: A comparison of optimization techniques for object-oriented languages. Technical report, Department of Computer Science, University of California, Santa Barbara, Santa Barbara, CA, USA, 1995.

[2] M. Arnold. *Online Profiling and Feedback-Directed Optimization of Java*. Ph.D. thesis, Rutgers University, Oct. 2002.

[3] A. Bergel, S. Ducasse, C. Putney, and R. Wuyts. Creating sophisticated development tools with OmniBrowser. *Journal of Computer Languages, Systems and Structures*, 34(2-3):109–129, 2008.

[4] A. Bergel, R. Robbes, and W. Binder. Visualizing dynamic metrics with profiling blueprints. In *Proceedings of the 48th International Conference on Objects, Models, Components, Patterns (TOOLS EUROPE'10)*. LNCS Springer Verlag, July 2010. to appear.

[5] W. Binder. Portable and accurate sampling profiling for java. *Softw. Pract. Exper.*, 36(6):615–650, 2006.

[6] A. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. *Pharo by Example*. Square Bracket Associates, 2009.

[7] G. Bracha and D. Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04), ACM SIGPLAN Notices*, pages 331–344, New York, NY, USA, 2004. ACM Press.

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

[9] Eclipse. Eclipse platform: Technical overview, 2003. http://www.-eclipse.org/whitepapers/eclipse-overview.pdf.

[10] T. Gîrba, M. Lanza, and S. Ducasse. Characterizing the evolution of class hierarchies. In *Proceedings of 9th European Conference on Software Maintenance and Reengineering (CSMR'05)*, pages 2–11, Los Alamitos CA, 2005. IEEE Computer Society.

[11] M. Haupt, R. Hirschfeld, and M. Denker. Type feedback for bytecode interpreters. In *Proceedings of the Second Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS'2007), ECOOP Workshop*, pages 17–22. TU Berlin, July 2007.

[12] D. Holten, B. Cornelissen, and J. J. van Wijk. Trace visualization using hierarchical edge bundles and massive sequence views. In *Proceedings of Visualizing Software for Understanding and Analysis, 2007 (VISSOFT'07)*, pages 47 – 54. IEEE Computer Society, 2007.

[13] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: The story of Squeak, a practical Smalltalk written in itself. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'97)*, pages 318–326. ACM Press, Nov. 1997.

[14] M. Marré and A. Bertolino. Reducing and estimating the cost of test coverage criteria. In *ICSE '96: Proceedings of the 18th international conference on Software engineering*, pages 486–494, Washington, DC, USA, 1996. IEEE Computer Society.

[15] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, second edition, 1997.

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

[17] D. Röthlisberger, M. Denker, and É. Tanter. Unanticipated partial behavioral reflection: Adapting applications at runtime. *Journal of Computer Languages, Systems and Structures*, 34(2-3):46–65, July 2008.

[18] D. Röthlisberger, O. Greevy, and O. Nierstrasz. Exploiting runtime information in the IDE. In *Proceedings of the 16th International Conference on Program Comprehension (ICPC 2008)*, pages 63–72, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[19] VisualWorks. Cincom Smalltalk. http://www.cincomsmalltalk.com/, archived at http://www.webcitation.org/5p1rRxls5, 2010.

[14] `http://www.adrian-lienhard.ch/blog?dialog=smalltak-meets-dtrace`

Selected Papers of the Int. Workshop on Smalltalk Technologies, Barcelona, Spain, September 14, 2010

Smalltalk debug lives in the Matrix

Loic Lagadec Damien Picard

Université Européenne de Bretagne, France.
Université de Brest ; CNRS, UMR 3192 Lab-STICC, ISSTB,
20 avenue Le Gorgeu
29285 Brest, France.
{damien.picard, loic.lagadec}@univ-brest.fr

Abstract

Agile programming aware computer scientists know how much productivity they owe to their development environments, and more precisely to advanced debuggers. Indeed, debuggers are mandatory to support an optimistic do-fix-rerun approach.

This development scheme does not make sense in hardware design where agile has a different meaning; it refers to reconfigurable architectures. Despite such architectures support tailoring and refactoring application circuits and promote short development cycles, the overall programing scheme still conforms to waterfall models and component based integration.

This paper presents a path to offer probe-based development to hardware designers, and introduces our Red Pill environment that merges several abstraction levels ranging from C like parallel coding to hardware realization embedding debug facility. Red Pill is developed using VisualWorks and reproduces some of Cincom Smalltalk browser well known features that traditionally lack when validating circuits.

General Terms Agile programming, Debugging, Reconfigurable computing

Keywords System-on-Chip, Modeling, Testing Methodology, Software Engineering, Morpheus.

1. Introduction

1.1 The red pill issue

Smalltalkers are software experts, and as so, are often poorly aware of the underlying hardware they use when running their applications. Nevertheless, by ignoring Virtual Machine structure and physical platform characteristics, software experts deprive themselves from potential massive speedups.

In addition, despite promoting reuse and expertise captured through design patterns and guidelines, they clearly fail in sharing their expertise with electrical engineers. As an example, setting up a cycle accurate bit accurate (CABA) simulator - that is very valuable and hard to get working in an efficient way - is a simple matter of combining several design patterns : template method, observer, state, visitor, composite. Similarly, providing a unified object view of several computer-aided design tool suite would be very a significant improve in term of usability and reusability.

However, several recent works have reported the use of (simplified) CORBA architecture to favor interoperability at low level [14, 11] or a growing interest for dynamic languages as specification platforms. Besides, a noticeable shift in the panel sessions topics of hardware designers conferences happened these last two years [3], what must be analyzed a promising move. Also, some educational conferences encourage cross-expertise curricula [15].

Obviously, one perfect example of such a missing win-win approach lies in adapting the debug facilities provided within advanced environment such as Smalltalk to hardware debugging. This requires to merge multi-level executable specifications (code, abstract syntax trees, graphs of primitive operators, netlists, etc.).

This translation from code to circuit in an automated manner is referred as HLS (High Level Synthesis). Not only gaining these tools to HLS would help the hardware designers to speed up their development process but this would offer a direct path to implement portions of high-level code as hardware circuit operating at two orders of magnitude faster then original software.

Every software designer is offered the choice to remain in his everyday life - and to focus on purely software issues - or to learn what the Matrix is. The real question is : will we take the red pill?

1.2 Reconfigurable architecture : entering the matrix

Reconfigurable architectures can be seen as hardware frameworks embedding resources (computation, communication, memories) that are further combined to form a circuit. To illustrate the nature of the architectures and the way they operate, one can say they act as skeletons providing a connecting scheme between elements, with computing block responsible for the inversion of control. The topology generally exhibits regularity so that reconfigurable architectures are sometimes referred as cells matrix.

The circuit implements an application in space, compared to a software solution that scales up in time. To figure out this intrinsic difference in kind, twice a bigger application takes roughly twice area on a reconfigurable architecture and twice execution time on a processor. This explains why some applications offer tremendous speed-up while other ones take no benefit from a hardware implementation. The more parallel and regular an application appears, the better it fits to a hardware solution.

Compared to classical hardware support, the reconfigurable architecture's agility comes from the ability to reconfigure the architecture - potentially in the field, partially, and on the fly - what means to re-allocate resources to form a new circuit. This favors fast prototyping and early circuit implementation - even prior to full specification availability. This carries the same benefits as software late binding (early availability, reuse, tailoring).

Despite being admitted that reconfigurable architectures increase the designer productivity by providing flexible hardware support, productivity remains strongly dependent on development environment and ease of validation.

1.3 There is only one real truth: causality

Software validation mainly covers two activities: testing and debugging. Testing refers to the error detection, as an example using characterization tests, while debugging is the task of tracking the causes of a failure. This carries the obvious need for strong

observability and controllability but also abstract analysis and fast changes. Observability ensures the designer knows what is going on. Controllability is the ability to control the execution flow. Abstract analysis means offering a programmer oriented view of the execution by preserving the programming model (e.g. source code vs byte-code, variables vs registers, etc.). Abstract analysis speeds up understanding by focusing on key aspects. Changes are required to fix some deviations.

Debugging means identifying and understanding the deviations, to allow fixing their consequences by invalidating their cause. Hence, debugging is a very iterative process looping over hypothesis-experiment-conclusion cycles.

As software agile programming promotes a just-fit approach, debugging has come to be a key piece of the designer toolbox. Because any evolution may cause a regression, and because only the debugger can provide a significant insight, even component based/ platform based development makes massive use of the debugging environment.

Software debugging happens through multiple back and forward navigation steps into the stack of contexts/current state. A common practice consists in scrolling down back in the history stack, assigning a value to a variable, then going back to the future looking forward to observing the impact of changes over the execution.

On the opposite, when designing a hardware product, the main design scheme conforms to waterfall, with early decisions that shall be revised as little as possible. Out of dynamic languages world, debugging is definitively not a mainstream way of development, but a stage that designers suffer. This comes from extremely long cycles time compared to software, with specific issues in testing, as operating testing at speed often requires to test *in-situ*. Time has now come for electronic design automation (EDA) tool suite to offer advanced debugging functionality to preserve the reconfigurable architectures time-to-market benefit. This happened 20 to 30 years ago for software engineering, when software designers shifted from assembly code hand writing to compilers and comprehensive debuggers use.

This paper presents the Red Pill tool, our contribution to this group effort. Red Pill brings ideas from typical Smalltalk-like IDEs to the world of reconfigurable hardware, especially assertion based debugging. It also takes advantage of Smalltalk polymorphism to support domain variability (reconfigurable target), and benefits from a wide legacy work [7]. Red Pills intents both to concur to open the hardware up to the ”lambda” dynamic language developer and both to offer a real IDE for hardware development.

The rest of the paper is structured as follows: section 2 is a comparative study of debugging techniques, section 3 focuses on the Red Pill while section 5 summaries some interesting results.

2. Debugging techniques: a comparative review

Nowadays, most common methods for hardware design validation are based on software or hardware simulation with RTL (Register Transfer Level) as the highest abstraction level, despite RTL is pretty close to assembly code.

Software simulation is a widely used debugging method since it is cost affordable and provides complete controllability and observability. But it suffers from performance drawbacks when simulating large and complex designs.

To overcome the speed problem validation can be done directly in hardware (built-in self test). Of course, despite being efficient for error detection, embedding testing does not fully support debugging activity. One challenge is, once detected a deviation, to restore observability on demand, to let the designer control the execution flow (step-by-step, continue, etc.), to allow running multiple scenario, and finally to support design evolution.

2.1 What good is a phone call if you’re unable to speak?

Be it software or hardware oriented, efficient debugging relies on probes to provide a way to check the state (observe) of the system at a specific point.

When validating software, a probe does not change the source code design, but will affect the timing of the program execution. Similarly, using an electronic probe does not change the design of an electronic circuit but, when used, it may change the circuit’s characteristics slightly. There are two basic types of probes: watchpoint, which logs status information without disturbing the execution, and breakpoint, which interrupts processing.

During software execution, a breakpoint immediately opens the system debugger when triggered. It shows the last several functions executed and the top function in the stack is the function containing the breakpoint. The debugger tool allows extensive exploration of the history of execution flow, code or variables’s value changes on the fly, and program execution control. After a breakpoint has triggered a function can be continued, executed step by step with or without diving into functions call.

To preserve its speed advantage, a hardware that is being debugged cannot offer full observability, nor execution stack like traveling. Observability means more than simply getting access to current state of internal signals. This would require some logging mechanism that are not scalable and would slow down the execution. As a consequence, the hardware designer has almost no information regarding the past states of the circuit. This feature really lacks as understanding the circuit’s current state cannot spring up out of previous states blindness.

One fundamental difference between software and hardware debug lies in that software debug approaches an intellectual game whereas hardware debug is still a pain. This comes from the difficulty to gain both observability and controllability. As a consequence, software debug is much more integrated as a development technique.

2.2 Hardware observability : to hell and back

When debugging circuits, designers can use embedded logic analyzers [1] or connect some IOs to a mixed-signal oscilloscope (MSO). The logic analyzes offer an insight to understand the behavior of the reconfigurable circuit (e.g FPGA standing for Field Programmable Logic Arrays), in the context of the surrounding system. This goes through connecting some internal signals to physical pins, only a small number of which are commonly available. Agilent Technologies however provides a software solution that overcomes some of these limitations by offering dynamic probes [17].

Another solution lies in bitstream instrumentation [5][18]. Xilinx ATC2 cores [19] can be added either during the design stage or within a post-synthesized netlist (similar to byte-code transformation [4]), to offer access to any internal signal and communication with external MSO. From a Smalltalk point of view, this is similar to request the designer to load a parcel in order to support `Transcript show:` operations.

ChipScope [21] is another solution to reflect activity after signals capture. Also some FPGAs offer some read back capability [20, 8, 12], and internal signals can be retraced. In a sense, Chipscope is more or less a Smalltalk inspector, but that would be available, as an example, only on Squeak, not on VisualWorks. It brings nice features, at the expense of linking the legacy to one platform.

These several solutions brought pieces of the observability designers missed for years in reflecting the FPGA internal state. Although, these functionalities remain available at a very low-level, compared to functional specification, and debugging requires more than just observability. Besides, some timing windows are critical as there happen the critical operations (inter process synchronization, looping, conditions, etc) while digging into some other should

be avoided to prevent over consumption of logging resources. Controllability enables to discretize time and to isolate hot-time windows, hence all the designer needs is to focus on these windows. The observability policy can then be tuned up depending on the window tag so that only hot-spots are considered with care.

2.3 Multiple runs under control: back to the future

Offering to hardware designers a way back to previous states while debugging however remains possible thanks to combining controllability and multiple runs, as long as the circuit execution stays deterministic (what means the debugging environment must capture and re-issue proper external stimuli on-time). However, unlike software debugging add-on, debug circuitry or tools scripting must be issued along with the design itself, hence a special attention must be carried out to plan for debug prior to physical realization (i.e. identify any test and measurement the designer would need during the verification phase).

Our approach combines the best of both simulation-based and *in-situ* solutions, by focusing on controllability and observability, while preserving the speed of hardware execution. Our original contribution lies in offering in-time traveling capabilities, while keeping the link active to/from behavioral specification of the circuit. Our solution is integrated into a high-level synthesis framework (from high-level code to circuit translator), and validation/debug of applications happens at different abstraction levels depending on the designer's needs [13]. The high-level specification acts as characterization for the circuits at selected milestones of the execution flow.

In the hardware design world, execution is fast, but debugging lacks some facilities that exist in software design (contexts stack analysis, replay, continuation, potentially hot code replacement). Therefore, what matters is combining software debugging flexibility with hardware execution performances, and no other solution that *in-situ* (using a hardware circuit) and at-speed (preventing the software IDE to slow down the execution through implementing a loose coupling update mechanism) execution raises up. Our approach ensures hardware performances while carrying controllability, and then supports focusing on some hot windows, rather than paying the cost of a full and useless observability.

3. Red Pill

Our high-level synthesis framework — that can be considered as a retargetable compiler — addresses fine-grained reconfigurable devices (IPs) connected to local memories. This conforms to recent works such as [2] that focus on system-on-chip (SoC) integration of reconfigurable IPs.

Application appears as a set of concurrent processes, some of which are granted for storage access. C has been elected as our entry-point format as it offers the promise of a wide spread and well understood language, although being enriched with specific constructs such as CSP [6] inter-process communications and looping structures for spatial unrolling. Based on a set of working files (C + XML address generators + processes topology), a flatten C code is generated. The synthesizer operates on this code, and the designer adds/removes some breakpoints to.

All the variables are looked up and logged, during either assignment, or channel based inter-processes communication. Breakpoints over variables are conditional, while breakpoints over operators are simple breakpoints. We consider three kinds of probes : watchpoints provide full visibility over a variable, breakpoints freeze the execution and conditional breakpoints are breakpoints that react to assumptions over some variable's value.

Our front end (figure 1) invokes the Biniou synthesizer [7] and either simulates or implements the resulting RTL netlist.

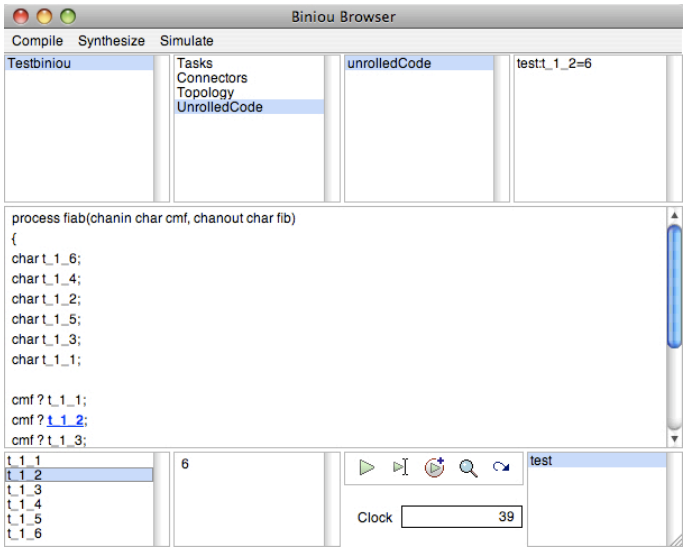

Figure 1. The front-end we use gather together per application (top left) C source code, XML address pattern, generated C-code (top middle) and debugging environment (top right/bottom). Variables are probed (bottom left), breakpoints are set (top right/code highlighting), and the execution is controlled (bottom right) through run/step/restart commands plus internal view access on demand.

Software execution serves as a characterization test during validation. A gcc-compatible C program is generated out of the source specification. This program is based on POSIX threads and emulates memories as files in which to read/write.

This feature saves the designers's time by favoring simple validation schemes. Early verification cuts off most of the functional mis-coding. Editing files in a common environment is very convenient to get checkable-oriented multiple runs.

Besides, despite execution time grows up along with complexity, it remains an affordable non recurring cost.

4. Providing debug capabilities to reconfigurable hardware

As reconfigurable architectures allow design modification, they are good candidates to embed support for software-like debug capabilities. This support is automatically inserted/removed in the design by our framework.

In Red Pill, observability happens through adding watchpoints. Preserving the genericity of the approach forbids picking up internal signals at run-time using dedicated tools such as Chipscope. Instead, extending the top module interface with probeable signals at compile-time is far less intrusive. The gain is portability as observability is restricted to IOs at the expense of oversizing the interface. This would be equivalent to adding to a class some new accessors over private states compared to using an inspector that can access any instance variable, private variables included.

Controllability comes from using embedded controllers. The circuit is based on the coupling of hierarchical controllers and datapath portions [9]. Hence, freezing a controller propagates progressively a lock within the full circuit — be this portion connected and synchronized with other sub-circuits. Locks are also connected to a specific controller called "debug controller" that can reset to low these signals on demand. The direct benefit is to preserve the speed the simulation deserves while stopping the circuit on demand (in-

ternal conditions), what ensures a full compatibility with the software environment speed (read-back, etc.).

Watchpoints, breakpoints and conditional breakpoints combine together to ease hardware debugging. However, the cost of adding these extensions differs and should be considered with care.

Conditional breakpoints have to be very flexible, being either valid or not, supporting hot replacement of conditions. The ones to be validated must be selectable on demand so that only a subset of them is active at a time. This offers the promise of multiple runs, with different active breakpoints so that the designer can speculate on error's cause and roll back to the original deviation.

The change over signals value must be reflected as variables updating within the high-level specification, as this is much easier to understand that bit-level signals.

4.1 Hardwired Watchpoints

Hardware counterparts to software variables are signals. Watchpoints are automatically inserted in the design on demand, by wiring to the top interface the probed signals. Then traces can be analyzed, making visible any internal signals by extending the interface of the modules recursively, and performing signals binding.

As debugging often stays a matter of "what if..." and "it's when...", watchpoints can also be conditional. In this case an operator is used to perform a boolean operation on the probed signal. It takes as input the probed signal, the value to compare and control bits for selecting the comparison operator. The condition's result is wired to the top hierarchy giving its status. Hence inserting a conditional watchpoint, consists of adding an operator and two simple wachpoints: one on the value to probe, and the other one on the boolean output of the operator.

4.2 Software-Like Execution Control

Software debug tools enable to stop the execution at a given point, to process step-by-step or to restart the execution of a function. In hardware, the execution control is performed by the local controller (top left, figure 2). In order to provide software-like control capabilities the local controller must support start/stop/restart actions. This happens by injecting some debug modules (which can be removed once application is validated).The execution stops every time a trigger raises up. At this point, watchpoints are read back and the breakpoints can be updated (condition, status)[1]. The variables's current values are then updated in the front-end (bottom left, figure 1) tool; the clock cycle and the active breakpoints are provided as an assistance to the designer.

This allows to dynamically change the argument values, to activate/invalidate probes and to change the probe's relation. What remains static is only the pool of probed variables/signals. As a consequence, this allows stopping under a given condition, then tracing internal signals on which to tune the probes' conditions. Not only the execution can be resumed but also it's cost free to start again a new run with these tailored conditions in order to reach an earlier stop point. This delivers the standard software debug facilities, such as digging in the functions call stack, the hardware designers traditionally lack.

4.3 Hardwired Breakpoints

Figure 2 (right) depicts the injection of controllers for two breakpoint schemes, respectively basic breakpoints (top), and conditional breakpoints (bottom). These breakpoints feed the debug controller with some probe-status. The debug controller itself can be considered as a simple finite state machine.

[1] Using conditional watchpoints instead of basic watchpoints drastically reduces the amount of logged information hence improves readability.

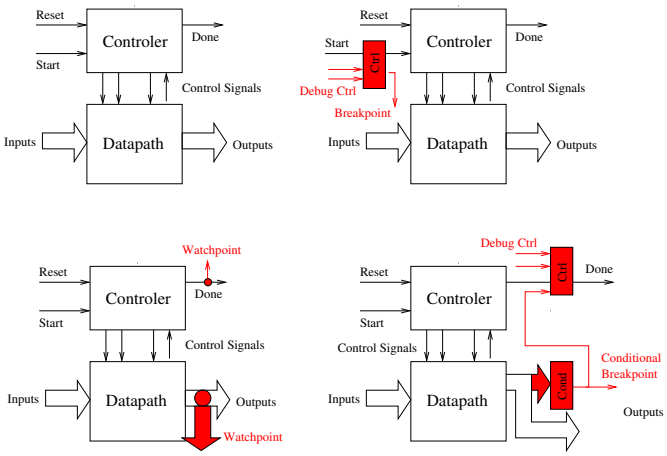

Figure 2. The circuit operates classically as a hierarchy of datapaths owning their private controller (top left). Observability happens through adding some watchpoints (bottom left) while execution control is provided by the additional local controllers freezing the execution either when activated (top right) or depending on some conditions (bottom right) .

5. You mean I can dodge bullets?

The debug controller owns a debug status register, that covers the probes's activation signals. A pooling policy — besides being inefficient as any active-waiting — is enough to launch a global state capture process as soon as the status reflects a probe activation. Extracting the global state can be achieved with no timing pressure as the circuit stays frozen until a step or a continue command is issued to the controller. The current state of the circuit appears then as an object.

5.1 The bullet time

The figure 3 illustrates a common view of the internal states and signals of a circuit under test. It demonstrates the controllability over a circuit based on probes's activation. The clock signal is highlighted and the circuit activity is reflected by low-to-high and high-to-low transitions of signals. The first circle points out the raising edge of a conditional breakpoint. This leads to an inactive period where no transition happens. As soon as the restart signal is issued, the execution resumes. This period enables to extract the internal state of the circuit for *in-situ* test. This internal state can be further post-processed to favor abstract analysis, as an example through an object representation.

5.2 Going further : Entering the object world

The object paradigm is very convenient to handle structures owning states and behavior. A circuit keeps its state in registers while its behavior is hardwired. Offering an object view of circuits eases debugging. A simple way to gain this is to redefine the $doesNotUnderstand$: method so that instance variable accessors read/write from/to registers. The figure 4 illustrates through a simple example characterization tests for component based circuits, taking advantage of virtual wires. A three inputs adder is under test. Combining two two-inputs adders provides the characterization test (code 1).

Obviously, once the application programming interface defined, these circuits are isomorphic to Smalltalk blocks. Going further than simply focusing on debugging issues, a direct benefit is the promise of "blind" mixing of software objects and hardware cir-

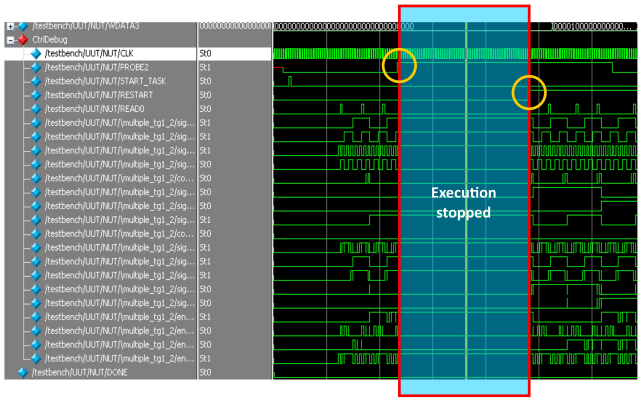

Figure 3. The design's debug view "bullet time" in mainstream simulator (Modelsim) [10]. The probe freezes the circuit until a new signal is issued to resume the execution.

cuits with hardware-software task migration support in an OS scope.

```
a := 9
b := 2.
c:= 1.
t1 := ThreeInputsAdder new.
t2 := TwoInputsAdder new.
t3 := TwoInputsAdder new.
self assert:
        (t1 value: a value: b value: c )
        =
        (t3 value: ( t2 value: a value: b)
            value: c)
```

Listing 1. SUnit code operating on circuits

Figure 5 illustrates Smalltalk blocks isomorphic circuits. In this case, IOs are organized as columns, only some cells of which are relevant (green rectangle for input cells, and red rectangle for output cells). Values are unsigned, and the second argument has an extra bit.

These two cases illustrate how object oriented modelling can improve abstract analysis. In Red Pill, this mechanism complements rebuilding of variables based on signal values to offer abstract analysis, and observability. Combining it with a fast changes ability (through HLS) and controllability (through adding dedicated controllers) brings up debugging facility.

6. Conclusion

This paper presents Red Pill, a debugging methodology for application mapped on reconfigurable platforms. The goal is to apply software-engineering methodology to hardware design for reducing design cycles and so time-to-market.

This requires to combine observability and controllability, while sustaining performances by in-situ and at-speed execution. This also requires a multi-level approach that supports merging RTL level information with C-like specification and software execution. Object modelling favors abstract analysis by hiding the hardware realm to the designer.

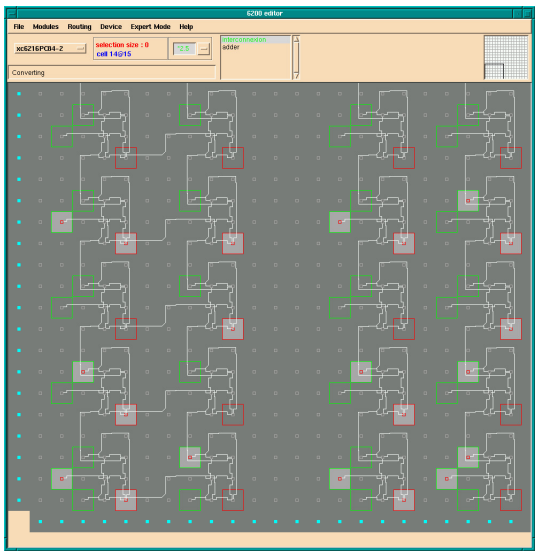

Figure 4. Two adders with hardware connections versus two adders (right) with software read/write connections.

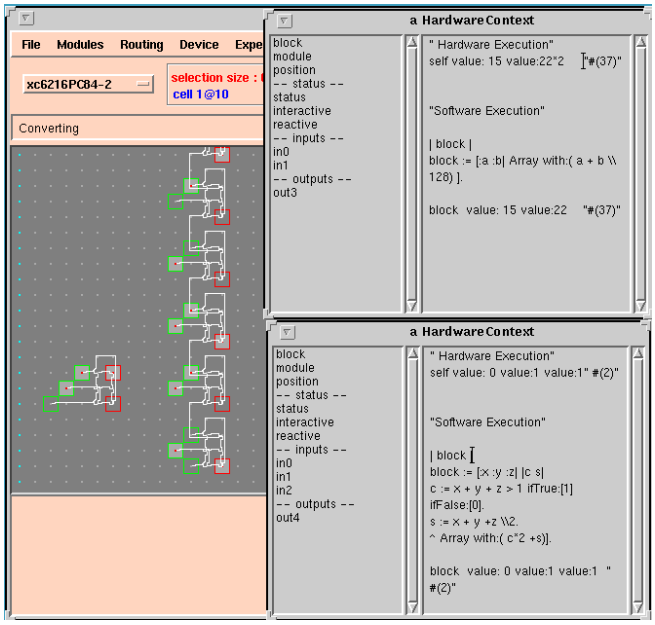

Figure 5. Circuit versus block : sharing the same API.

The successful design of significant test cases has confirmed that this methodology valuable when validating applications running on reconfigurable systems on chips. Our contribution does not pretend to be a replacement methodology but, instead, to be fully compatible and highly synergetic with legacy tools.

Future work will focus on two complementaty directions. First, refactoring Red Pill in order to integrate our front end deeply into the existing smalltalk host environment and reuse the infrastructure offered by it instead of using a DSL compiler, as decribed by [16]. Then, pushing ahead software integration with legacy tools such as Chipscope.

References

[1] Datasheet, signaltap embedded logic analyzer megafunction, 2001.

[2] Morpheus fp6 integrated project. http://www.morpheus-ist.org/, 2006.

[3] Reconfigurable communication-centric systems on chip conference panel, 2008-2010.

[4] M. Denker, S. Ducasse, and ric Tanter. Runtime bytecode transformation for smalltalk, 2005.

[5] P. Graham, B. Nelson, and B. Hutchings. Instrumenting bitstreams for debugging fpga circuits. In *FCCM'01*, pages 41–50, 2001.

[6] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[7] L. Lagadec and D. Picard. Teaching reconfigurable processor: the biniou approach. In *KIT Scientific report 7551, isbn: 978-3-86644-515-4*, 2010.

[8] P. Lucent Technologies Allentown. Orca series 4 field-programmable gate arrays, 2000.

[9] G. D. Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[10] ModelSim. Modelsim. http://www.model.com/.

[11] ORBexpress. *ORBexpress FPGA Provides CORBA Communications Implemented Directly in Hardware*.

[12] K. Paulsson, U. Viereck, M. Hubner, and J. Becker. Exploitation of the external jtag interface for internally controlled configuration readback and self-reconfiguration of spartan 3 fpgas. *ISVLSI*, 2008.

[13] D. Picard and L. Lagadec. Multi-level simulation of heterogeneous reconfigurable platforms. In *ReCoSoC'08*, Barcelona, Spain, 2008.

[14] PrismTech. Openfusion corba overview whitepaper, Jan. 2008.

[15] rcedu. Reconfigurable computing education. http://helios.informatik.uni-kl.de/RCeducation/.

[16] L. Renggli and T. Gîrba. Why smalltalk wins the host languages shootout. In *IWST '09: Proceedings of the International Workshop on Smalltalk Technologies*, pages 107–113, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-899-5. doi: http://doi.acm.org/10.1145/1735935.1735954.

[17] A. technologies. *Agilent technologies infiivision MSO N5434A FPGA dynamic probe for Altera*, 2010.

[18] A. Tiwari and K. A. Tomko. Scan-chain based watch-points for efficient run-time debugging and verification of fpga designs. In *ASPDAC*, 2003.

[19] Xilinx. *Revolutionary FPGA Real-Time Logic Debug Technology from Xilinx Slashes Verification Times by up to* 50%, march 2004.

[20] Xilinx. Virtex fpga series configuration and readback. *Xilinx Application Note*, 138, 2005.

[21] Xilinx. *UG029, ChipScope Pro 10.1 Software and Cores User Guide*, 2008.

Aktuelle Technische Berichte des Hasso-Plattner-Instituts

Band	ISBN	Titel	Autoren / Redaktion
39	978-3-86956-092-2	**Dritter Deutscher IPv6 Gipfel 2010**	Hrsg. von Christoph Meinel und Harald Sack
38	978-3-86956-081-6	**Extracting Structured Information from Wikipedia Articles to Populate Infoboxes**	Dustin Lange, Christoph Böhm, Felix Naumann
37	978-3-86956-078-6	**Toward Bridging the Gap Between Formal Semantics and Implementation of Triple Graph Grammars**	Holger Giese, Stephan Hildebrandt, Leen Lambers
36	978-3-86956-065-6	**Pattern Matching for an Object-oriented and Dynamically Typed Programming Language**	Felix Geller, Robert Hirschfeld, Gilad Bracha
35	978-3-86956-054-0	**Business Process Model Abstraction : Theory and Practice**	Sergey Smirnov, Hajo A. Reijers, Thijs Nugteren, Mathias Weske
34	978-3-86956-048-9	**Efficient and exact computation of inclusion dependencies for data integration**	Jana Bauckmann, Ulf Leser, Felix Naumann
33	978-3-86956-043-4	**Proceedings of the 9th Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS '10)**	Hrsg. von Bram Adams, Michael Haupt, Daniel Lohmann
32	978-3-86956-037-3	**STG Decomposition: Internal Communication for SI Implementability**	Dominic Wist, Mark Schaefer, Walter Vogler, Ralf Wollowski
31	978-3-86956-036-6	**Proceedings of the 4th Ph.D. Retreat of the HPI Research School on Service-oriented Systems Engineering**	Hrsg. von den Professoren des HPI
30	978-3-86956-009-0	**Action Patterns in Business Process Models**	Sergey Smirnov, Matthias Weidlich, Jan Mendling, Mathias Weske
29	978-3-940793-91-1	**Correct Dynamic Service-Oriented Architectures: Modeling and Compositional Verification with Dynamic Collaborations**	Basil Becker, Holger Giese, Stefan Neumann
28	978-3-940793-84-3	**Efficient Model Synchronization of Large-Scale Models**	Holger Giese, Stephan Hildebrandt
27	978-3-940793-81-2	**Proceedings of the 3rd Ph.D. Retreat of the HPI Research School on Service-oriented Systems Engineering**	Hrsg. von den Professoren des HPI
26	978-3-940793-65-2	**The Triconnected Abstraction of Process Models**	Artem Polyvyanyy, Sergey Smirnov, Mathias Weske
25	978-3-940793-46-1	**Space and Time Scalability of Duplicate Detection in Graph Data**	Melanie Herschel, Felix Naumann
24	978-3-940793-45-4	**Erster Deutscher IPv6 Gipfel**	Christoph Meinel, Harald Sack, Justus Bross
23	978-3-940793-42-3	**Proceedings of the 2nd. Ph.D. retreat of the HPI Research School on Service-oriented Systems Engineering**	Hrsg. von den Professoren des HPI

ISBN 978-3-86956-106-6
ISSN 1613-5652