ESUG 2004 Research Track

Stéphane Ducasse

ESUG 2004 Research Track

Editor: Stéphane Ducasse

published as Technical Report
of the Institut für
Informatik und Angewandte Mathematik
University of Bern, Switzerland

IAM-04-008

April 14, 2005

Abstract

Classification: 68-02 Research Exposition, 68N30 Mathematical aspects of software engineering (specification, verification, metrics, requirements, etc.) [New MSC2000 code] 68U35 Information systems (hypertext navigation, interfaces, decision support, etc.) [New MSC2000 code] D.2 Software Engineering, D.2.2 Tools and Techniques, D.2.7 Distribution and Maintenance. D.3.1 [Programming Languages]: D.3.2 Language Classifications, D.3.3 Language Constructs and Features (E.2), 68N15 Programming languages, 68N19 Other programming techniques (object-oriented, sequential, concurrent, automatic, etc.) [New MSC2000 code]

Table of Contents

Smalltalk is a pure object-oriented, dynamically typed class-based object-oriented programming language. The first official version dates from 1976, but the language used today is the ANSI standard Smalltalk-80. Since it was around for quite a long time and because it was purely object-oriented from its inception, lots of pioneering effort for object-oriented languages was done in Smalltalk. These advances spread to mainstream object-oriented languages and systems: virtual machines using byte codes, garbage collection, reflection and integrated development environments have been part of the language since its early stages. A big open-source library of classes (e.g., collections and graphical user interface support) was also there. More recently we have seen how refactoring, unit testing frameworks and extreme programming were developed in Smalltalk before gaining widespread acceptance in other languages and systems. The impact of Smalltalk becomes obvious when looking at modern current-day development environments featuring "advanced" concepts such as fix & continue debugging and realizing that the same functionality has been around in Smalltalk environments since the early eighties.

But Smalltalk is not just an old language that one can marvel at to see glories long past. Recent years have seen a come-back of novel research using Smalltalk, as evidenced from publications in major object-oriented conferences such as ECOOP or OOPSLA. Smalltalk's flexibility is once more put to good use to research and validate novel ideas in diverse areas. The papers published here can be seen as a glimpse in the future, not only for Smalltalk but for object-oriented languages and systems in general. They were a selection of papers presented at the Research Track of the yearly Smalltalk event organized by ESUG - the European Smalltalk Users Group[1]. Like last year, the goal of the research track is to give wide academic recognition to high-quality research done in or with Smalltalk. The topics we were interested in are (in a non-exhaustive list): language features, development process, meta and reflective programming, code analysis, applications, virtual machines, integrated development environments, and software maintenance evolution.

Each paper received five reviews by members of the following international program committee:

- Prof. Dr. Andrew Black (Oregon Health and Science University, USA)
- Dr. Gilad Bracha (SUN, USA)
- Dr. Noury Bouraqadi (École des Mines de Douai, France)
- Prof. Dr. Serge Demeyer (University of Antwerpen, Belgium)
- Prof. Dr. Theo D'Hondt (Vrije Universiteit Brussels, Belgium)
- Prof. Dr. Christophe Dony (University of Montpellier, France)

[1] http://www.esug.org

- Prof. Dr. Stéphane Ducasse (University of Berne, Switzerland)
- Dr. Robert Hirschfeld (DoCoMo Euro-Labs, Germany)
- Prof. Dr. Ralph Johnson (University of Urbana Champain, USA)
- Prof. Dr. Michele Marchesi (University of Cagliari, Sardinia, Italy)
- Prof. Dr. Kim Mens (Université catholique de Louvain la Neuve, Belgium)
- Dr. Serge Stinckwich (Université de Caen, France)
- Prof. Dr. Dave Thomas (Bedarra, USA-Canada)
- Prof. Dr. Roel Wuyts (Université Libre de Bruxelle, Belgium)

Papers accepted for the ESUG 2004 Conference Research Track give a glimpse of high quality work conducted using Smalltalk. The spectrum of these research projects ranges from new programming concepts and reflection to applications in various domains such as Embedded Software and Robotics. The best six papers we selected for a special issue in the Journal Computer Languages, Systems and Structures. The current proceedings contains the ten papers accepted for the ESUG Research Track held at Kothen the 6th September 2004.

Session: Program Analyzis

- "Delving Source-Code with Formal Concept Analysis", Kim Mens and Tom Tourwé
- "Power Laws in Smalltalk", M. Marchesi, S. Pinna, N. Serra, and S. Tuveri

Session: AI and Robotics

- "A Dynamic Graph Implementation in Smalltalk for Self-Reconfigurable Robots Simulation", S. Saidani and M. Piel
- "An Aspect-Based Multi-Agent System", R. Robbes, N. Bouraqadi, and S. Stinckwich

Session: Language Development

- "Design, Implementation and Evaluation of the Resilient Smalltalk Embeded Platform", J.R. Andersen, L. Bak, S. Garup, K. V. Lund, T. Eskildsen, K. M. Hansen, and M. Torgersen
- "Prototalk: an Environment for Teaching, Understanding and Prototyping Object-Oriented Languages", A. Bergel, Ch. Dony, and S. Ducasse

Session: Reflection and Meta-Level Programming

- "Language Support for Adaptive Object-Models using Metaclasses", R. Razavi, N. Bouraqadi, J. W. Yoder, J. F. Perrot, and R. Johnson
- "Uniform and Safe Metaclass Composition", S. Ducasse, N. Schrli, and R. Wuyts

Session: Modules, Packaging, and Deployment

- "Classboxes: Controlling Visibility of Class Extensions", A. Bergel, S. Ducasse, O. Nierstrasz and R. Wuyts
- "Parcels: A Fast and Feature-Rich Binary Deployment Technology", E. Miranda, D. Leibs, and R. Wuyts

We're sure that you'll enjoy reading those papers and hope to see you at 2005 ESUG Research Track.

October 2004
Dr. Noury Bouraqadi, Prof. Stéphane Ducasse and Prof. Roel Wuyts

Delving Source-code with Formal Concept Analysis

Kim Mens *

Département d'Ingénierie Informatique (INGI)
Université catholique de Louvain (UCL)
Place Sainte Barbe 2, B-1348 Louvain-la-Neuve, Belgium

Tom Tourwé

Centrum voor Wiskunde en Informatica (CWI)
P.O. Box 94079, NL-1090 GB Amsterdam, The Netherlands

Abstract

Getting an initial understanding of the structure of a software system, whether it is for software maintenance, evolution or reengineering purposes, is a nontrivial task. We propose a lightweight approach to delve a system's source code automatically and efficiently for relevant concepts of interest: what concerns are addressed in the code, what patterns, coding idioms and conventions have been adopted, and where and how are they implemented. We use formal concept analysis to do the actual source-code mining, and then filter, classify and combine the results to present them in a format that is more convenient to a software engineer. We applied a prototype tool that implements this approach to several small to medium-sized Smalltalk applications. For each of these, the tool uncovered several design pattern instances, coding and naming conventions, refactoring opportunities and important domain concepts. Although the tool and approach can still be improved in many ways, the tool does already provides useful results when trying to get an initial understanding of a system. The obtained results also illustrate the relevance and feasibility of using formal concept analysis as an efficient technique for source code mining.

Key words: Source-code mining, formal concept analysis, software classification.

* Corresponding author.
Email addresses: Kim.Mens@info.ucl.ac.be (Kim Mens), Tom.Tourwe@cwi.nl (Tom Tourwé).
URLs: http://www.info.ucl.ac.be/people/cvmens.html (Kim Mens), http://www.cwi.nl/~tourwe (Tom Tourwé).

1 Introduction

When maintaining or evolving a software system it is important to gain some knowledge of its overall design first. Demeyer et al. [1, Chapter 4] explain how obtaining such an *initial understanding* is crucial for the success of a reengineering project, and discuss some techniques and lightweight source-code analysis tools to get such an understanding. The tool proposed in this paper can be seen as another such tool in the software engineer's toolbox.

The tool implements a bottom-up approach that can help in getting an initial idea of the coding conventions, idioms and patterns used in the source code of a software system. In a sense, it is related to the "Study the Exceptional Entities" reengineering pattern documented in [1, Chapter 4], but in a complementary way, as it focusses on finding commonalities in the structure of the source code, rather than potential design problems.

Our tool builds on the technique of *formal concept analysis* [2], which has many known applications in data analysis and knowledge processing, and some in software engineering. The essence of our contribution lies not in the idea of applying formal concept analysis (FCA) to source code, but in our particular choice of elements and properties for the FCA algorithm, and how we filtered and classified the discovered concepts in order to mine a system's source code in a lightweight way that is independent of the actual system being analyzed.

Although the proposed approach can still be improved in many ways, and in spite of its apparent simplicity, our case studies show that it allows us to delve Smalltalk source code for many interesting symptoms of good design, like design patterns, programming idioms and naming and coding conventions. It also allows us to discover symptoms of bad design which may provide opportunities for refactoring, as well as some features of which the implementation is spread throughout the source code. Most of the discovered information provides a good starting point for understanding the source code in more detail.

The remainder of this paper is structured as follows. Section 2 briefly introduces the mathematical technique of formal concept analysis. In Section 3 we explain our approach and how we use formal concept analysis to delve the source code for relevant concepts of interest. Section 4 briefly presents the tool and Section 5 gives an overview of the experiments we conducted on five different small to medium-sized case studies and presents the design symptoms we discovered. Sections 6 and 7 discuss related and future work. We conclude the paper in Section 8.

2 Formal Concept Analysis

Formal concept analysis (FCA) [2] is a branch of lattice theory that can be used to identify meaningful groupings of *elements* that have common *properties.*[1] The FCA algorithm takes as input a relation, or Boolean table, T between a (potentially large, but finite) set of elements and a set of properties of those elements. An example of such a table is given in Table 1, in which different programming languages and properties are related. A mark in a table cell means that the programming language in the corresponding row has the property of the corresponding column.

Table 1
Programming languages and their supported programming paradigms.

Progr. language	OO	Functional	Logic	Static typing	Dynamic typing
Java	√	-	-	√	-
Smalltalk	√	-	-	-	√
C++	√	-	-	√	-
Scheme	-	√	-	-	√
Prolog	-	-	√	-	√

Taking such a table T as input, the FCA algorithm determines *maximal* groups of elements and properties, called *concepts*, such that:

- each element of the group shares the properties,
- every property of the group holds for all of its elements,
- no other element outside the group has those same properties, and
- no other property outside the group holds for all elements in the group.

Intuitively, a *concept* corresponds to a maximal 'rectangle' in the table T, up to a permutation of the table's rows and columns.

All concepts are ordered into a *concept lattice*, an example of which is depicted in Figure 1. The lattice's bottom concept contains those elements that have all properties. Since there is no such programming language in our example, that concept contains no elements. Similarly, the top concept contains those properties that hold for all elements. Again, there is no such property. Other concepts represent related groups of programming languages, such as the concept *({ Java, C++ }, { static typing, object oriented })*, which groups all statically-typed object-oriented languages.

[1] As in Arèvalo et al. [3], in this paper we prefer to use the terms *element* and *property* instead of *object* and *attribute* used in traditional FCA literature, because these latter terms already have a very specific meaning in OO software development.

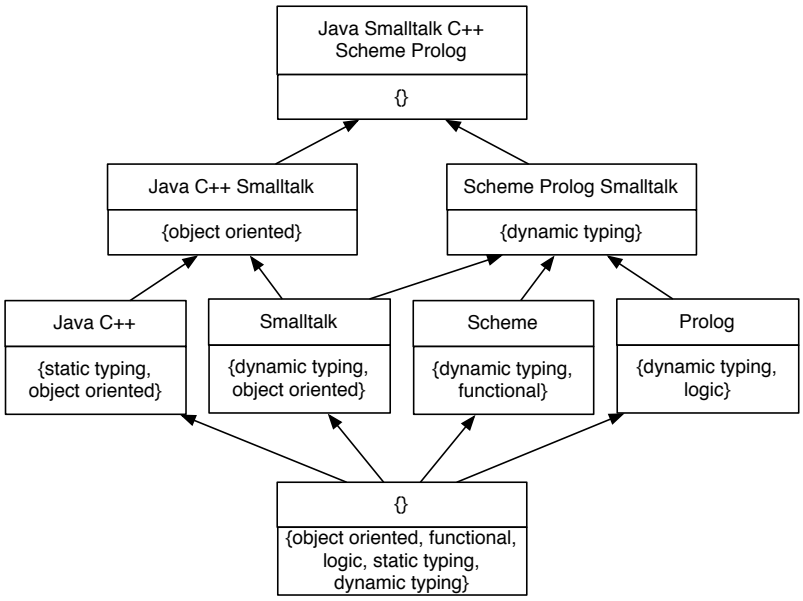


Fig. 1. Concept lattice corresponding to Table 1.

For more details on formal concept analysis we refer to [2]. The next section explains the details of our approach to use FCA for source code mining.

3 Delving Source Code

When applying FCA for delving Smalltalk source code, we first have to choose the elements and properties to compute the concept lattice (§3.1). When computing the lattice (§3.2), lots of concepts are produced, many of which are irrelevant or redundant. Therefore, we filter the discovered concepts (§3.3) and classify (§3.4), combine and annotate them (§3.5) in a way that is more relevant to a software engineer. It is important to stress that the FCA algorithm, filters and analyzers need to be tuned only once, for the specific kind of symptoms that are of interest. Afterwards, this tuning can simply be reused "as is" in order to mine other applications for similar design symptoms.

The novelty of our contribution is not in the idea of applying FCA to source code. What is more important is our particular choice of elements, properties, filters and analyzers, and how these allow us to discover interesting design symptoms in the source code, in a way that is independent of the considered application, and even largely independent of the considered programming language. Proof is that the tool has recently been generalized to support delving Java source code as well.

Since our goal in this paper is to mine Smalltalk source code, as FCA *elements* we choose source-code entities like classes, methods and method parameters. The reason why we did not (yet) consider additional entities like bundles, protocols and categories, is two fold. Firstly, in order to avoid cluttering the results, we choose to be pragmatic and include initially only the most relevant source code entities. Secondly, we want our approach to be language independent, and most other programming languages do not feature protocols, categories, etc.

As *properties* we take simple substrings of the names of the chosen source-code entities. Evidently, an application's source code contains a wealth of other information, such as call-graph or parse-tree information. Computing such information and reasoning with it, requires vastly more resources, however, which would make our approach much less lightweight. For this very reason, we chose to resort to a simple "name matching" approach at first. Preliminary experiments that reason about similarities in parse trees are ongoing at the moment, but should still be considered future work.

Because we take as properties simple (sub)strings occurring in the names of the considered source-code entities, the discovered concepts will group entities with similar names. Our motivation for choosing these properties, is that in Smalltalk in particular and in many other programming languages, programmers often rely on naming conventions to reveal their intentions and to implement certain programming idioms and design patterns. Keeping the properties simple has the additional benefit that they can be generated and manipulated efficiently.

Nevertheless, in order to limit the number of properties, we do not consider all possible substrings. Instead, we split class, method and parameter names in substrings according to the capitals and other separators occurring in them. In addition, we discard substrings with little conceptual meaning or that are used too often, such as '`with`', '`from`', '`the`', '`object`', as well as substrings that are too small (i.e., less than 3 characters). We also ignore colons, plurals and the difference in case when comparing substrings. For example, the properties associated with a class `QuotedCodeConstant` are the substrings '`quoted`', '`code`' and '`constant`'. The properties corresponding to a method named `#unifyWithDelayedVariable:inEnv:myIndex:hisIndex:inSource:` are '`unify`', '`delayed`', '`variable`', '`index`' and '`source`'.

3.2 Compute the concept lattice

Applying an FCA algorithm to the elements and properties generated in the previous step, results in a large *concept lattice* of several hundreds to thousands of concepts, depending on the size of the application Table 2, Section 5, shows some quantitative results of applying our approach on five different Smalltalk applications.

For now, let it suffice to give an illustrative example of a simple kind of concepts that is discovered by the FCA algorithm: *accessing methods*. Indeed, since both accessor and mutator methods are named after an instance variable, they share the same substring. E.g., the following concept we discovered in one of our case studies groups the `#callStack` accessor and `#callStack:` mutator methods of the `callStack` instance variable defined in the `Environment` class:

```
Environment >> callStack
    ^ callStack
Environment >> callStack: aStack
     callStack := aStack
```

They are grouped based on the properties 'call' and 'stack' that are shared by these methods, but by no other methods, classes or parameters in the application.

3.3 Filter the concepts

As we will see in Section 5 (see column *#raw* of Table 2), for the considered cases the number of concepts discovered by FCA, before applying any filtering, is of the same order of magnitude as the number of considered elements. This would imply that a software engineer needs to look at a significant number of concepts in order to try and understand the source code. However, the concepts do contain a large amount of redundancy and noise that we can easily filter.

A first filter ignores all concepts that contain two or less elements, since these concepts are generally too small to provide relevant information. Note that this filter discards most *accessing method* concepts, since these typically contain only two elements: an accessor and a mutator method. However, since accessing methods are rather fine-grained, since there are a lot of them, and since they can be inspected with standard browsers easily or they can be retrieved with more dedicated tools, we don't mind that they get discarded.

A second filter ignores all concepts that share only one property (substring). Although this filter may discard some interesting concepts, it does throw away

many more irrelevant concepts. We think that, during initial understanding of an application, getting a quick and focussed idea of certain commonalities in the source-code is more important than getting a precise list of *all* possible commonalities. A nice improvement of this filter would be to discard those concepts of which the properties 'cover' only a small fraction, based on some threshold, of the concept elements' names. As such, the filter becomes relative to the size of the elements' names.

Whereas these two generic filters are independent of the kinds of elements being analyzed, our third filter is more targetted. It discards concepts that contain only classes (with a similar name) in the same hierarchy. These concepts typically do not provide very useful information — since classes belonging to the same hierarchy often have similar names — except if we want to discover exactly which naming convention these classes are relying upon.

3.4 Classify the filtered concepts

Being mere sets of elements (classes and methods) and properties (substrings of the elements' names), the concepts that remain after filtering are rather unstructured. Therefore, we reorganize the concepts automatically in a way that is more easy for a software engineer to analyze and interpret.

More precisely, we flatten the concept lattice and *classify* the concepts in a way that makes more sense to the software engineer. Our classification distinguishes 3 main groups of concepts:

(1) *Single class concepts* group concepts of which all elements are methods (or parameters of those methods) belonging to one and the same class;
(2) *Hierarchy concepts* have a larger scope as they group classes, methods and parameters of those methods, that belong to a single class hierarchy;
(3) For *crosscutting concepts* we explicitly require that at least two different class hierarchies are involved. We do this by verifying that the *most specific* common superclass of the considered classes is `Object` *and* that none of the methods in the concept are defined by the `Object` class itself (which would be a degenerate case of a *hierarchy concept*).

Such a classification helps a software engineer in several ways. Knowing that a certain concept belongs to a given classification helps him to better understand that concept. For example, knowing that a concept containing several methods with exactly the same name belongs to the *hierarchy concepts* classification allows him to qualify those methods as *polymorphic methods*. Or, observing that a *crosscutting concept* contains polymorphic methods that exhibit a lot of duplication, may point out the need for an aspect-oriented solution.

By organizing the concepts in a classification like the above, the structure of the lattice is lost. Concepts that were nearby in the lattice (e.g., that were in a subconcept relationship) will not necessarily belong to the same classification, and vice versa. As a consequence, since there is a lot of overlap between concepts that are nearby in the lattice, when reorganizing the concepts this may lead to redundancy among concepts that get classified into *different* classifications. Whenever possible, however, when nearby concepts in the concept lattice are put in the *same* classification, we automatically reconstruct part of the original structure of the lattice, in order to reduce redundancy. More specifically, we recombine highly overlapping concepts into a single nested one.

In addition, we automatically regroup and annotate the concepts belonging to each classification, in order to present them in a way that is more convenient to the software engineer: different concepts related to the same class(es) are combined, methods are annotated with the classes they belong to, and concepts are annotated with their properties.

4 DelfSTof, our Conceptual Code Mining tool

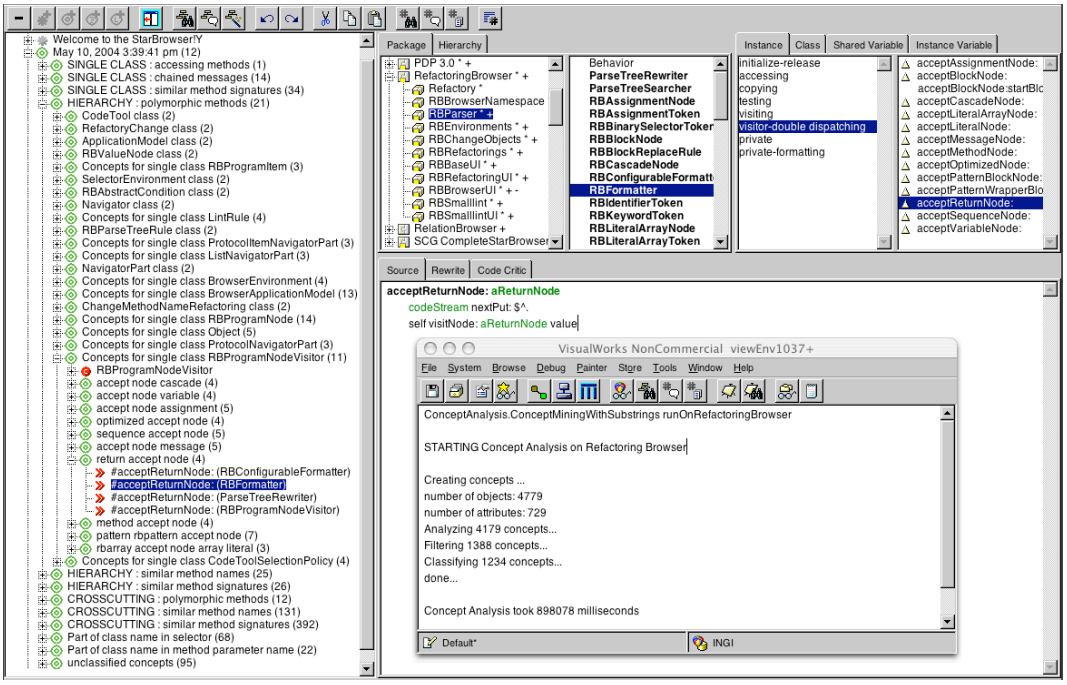


Fig. 2. DelfSTof : Discovered concepts for the Refactoring Browser.

We developed a prototype tool, *DelfSTof* [2], that implements the approach outlined above, and presents the discovered concepts in a way that is easy to use and manipulate. We capitalize the letters "ST" because the tool is implemented completely in Smalltalk and originally only analyzed Smalltalk source code.

The tools consists of an efficient FCA algorithm, a set of filters, and a set of 'analyzers' that are in charge of the classification, combination and annotation of concepts. The resulting classification of concepts is directly visualized with the *StarBrowser* [4]. A screenshot of the tool, which is essentially a StarBrowser plugin, is presented in Figure 2.

5 Discovered Design Symptoms

We applied our tool to five different cases, as summarized by Table 2. Every row in this table corresponds to a different application of which we delved the source code for design symptoms. *SOUL* is an interpreter for a Prolog-like language and *DelfSTof* is our own conceptual code mining tool. We chose these two applications because we know their implementation well, which allowed us to better assess the relevance of the discovered results. Both the *StarBrowser* and the *Refactoring Browser* are advanced Smalltalk browsers and *CodeCrawler* is a language-independent reverse engineering tool which combines metrics and software visualization. These cases were chosen because they are relatively stable and well-developed, have been around for several years, and are of perfect size for initial experimentation.

Table 2
Quantitative results of FCA applied to some Smalltalk applications.

Case	#elements	#properties	#raw	#filtered	time (sec)
DelfSTof	756 (135)	237	617	126	5
StarBrowser	731 (52)	352	740	115	7
SOUL	1469 (111)	434	1188	281	22
CodeCrawler	1370 (93)	477	1419	327	24
Refactoring Browser	4779 (271)	729	4179	1234	414

[2] The name coins the Dutch word 'delfstof', which designates the result of a delving process. In English, the first meaning of the verb "to delve" is "to make careful investigation for facts and knowledge". Coincidentally, the pronunciation of the word "delfstof" sounds like the English "delve stuff" which is indeed what the tool does.

The column *#elements* gives an indication of the size of each considered case as it equals the total number of classes plus methods in that case. (The number between brackets is the number of classes.) Note that we did not consider method parameters as separate elements since they are part of the method signature.

The column *#properties* shows the number of substrings that have been generated from the considered elements. We observe that the number of properties is always a factor 2 to 4, in the case of the Refactoring Browser even almost a factor 7, *less* than the number of elements. This is a good sign as it implies that a significant amount of the properties are actually shared by the elements.

The third column *#raw* shows the raw number of concepts discovered by FCA. Column *#filtered* shows how many concepts remain after having applied the simple filters that were explained in §3.3. We observe that, after filtering, there remain about 4 to 7 times less concepts than the number of considered elements. We still think that this is a bit too much, especially for larger cases, but we will come back to this discussion in Section 7.

For all considered cases, the time of computation — which includes all steps explained in Section 3 and not only the computation of the concept lattice — was acceptable (ranging from a few seconds to a few minutes), although theoretically it increases in a non-linear way with the number of considered objects.

The remainder of this section discusses some of the "design symptoms" we discovered when manually analyzing the results of our FCA experiments. As a matter of fact, we could refine the classification of §3.4 to classify explicitly and automatically most of these symptoms as well (for example, by using our logic meta-programming environment SOUL). However, a certain trade-off needs to be made, since this would require more automated analysis and thus slow down the tool. Also, we want to keep the tool sufficiently general so that it still can be applied to other languages (maybe even non OO languages). Nevertheless, we already extended our tool so that it classifies automatically the programming idioms listed in Subsection §5.1 : *polymorphic methods*, *chained messages* and *delegating methods*.

5.1 Programming idioms

Polymorphic methods are a symptom of good design that is readily recognized by our tool, since polymorphic methods have exactly the same name. Consequently, they are grouped together in a concept. In addition, if there are several polymorphic methods for a same class hierarchy, these will all be grouped together automatically in a single combined concept.

For example, in the source code of the *Refactoring Browser* we discovered a concept containing 4 methods named `#acceptReturnNode`, implemented on different subclasses of `RBProgramNodeVisitor`. This is a typical example of polymorphism of which many examples can be found in any OO application. In fact, for the class `RBProgramNodeVisitor` alone many more polymorphic method concepts were discovered, as can be seen from Figure 2.

Polymorphic methods across class hierarchies are equally interesting to detect as they may trigger interesting refactorings or tell us something about possible multiple inheritance problems the original developers encountered. We will come back to this in Section 5.5

Chained messages are concepts that group a method together with some of its auxiliary methods in the same class. These chains are recognized by FCA since the auxiliary methods often have a name that is similar to that of the originating method, though sometimes slightly longer and taking an extra parameter. E.g., in the *CodeCrawler* application the class `CCMetricsChooserDialog` implements a method `#applyChosenMetrics`, which calls an auxiliary method `#detectChosenMetrics`, which in turns calls `#assignChosenMetricsTo:`. These 3 methods share the substrings 'chosen' and 'metrics'.

Delegating methods delegate responsibility by calling a method with the same name. Our tool discovered some interesting sets of delegating methods with a similar name that all belonged to the same class. The presence of many such delegating methods in a single class may indicate that the class is a *Decorator* [5].

5.2 Code duplication

By closely inspecting the discovered concepts, we also detected several cases of copy and paste reuse: several concepts contain methods that not only have a similar name, but a similar implementation as well. This may seem logical, since methods that implement similar behavior can be expected to have similar names. However, from an implementation point of view, this duplicated code could just as well be factored out and reused.

For example, in *CodeCrawler*'s `CEVModelHistory` class we discovered a concept with 2 methods `#predecessorModelNameOfModel:` and `#predecessorModelNameOfModelNamed:` which had nearly the same implementation. Since one of them is no longer being used, we assume that the original developer(s) created one of the methods by copying it from the other one, then replaced all calls to the old one to the new one, but in the end forgot to remove the old method.

In general, particular reasons for duplication may become clear by looking at the classification of the concepts:

- a developer who was not aware that a method implementing the desired behavior was already defined, may accidentally implement a method with a similar signature and behavior. Such methods are often grouped in concepts classified as *hierarchy concepts*, because the method that already implements the desired behavior is probably not implemented by the class the developer is looking at, but by one of its sub- or superclasses.
- a method was needed whose behavior differed slightly from the behavior already implemented by an existing method, and this behavior was copied and adapted slightly, without extracting the common code into a separate method (or merely deleting the original version if it is no longer needed, such as in the example of the `CEVModelHistory` class above). Concepts containing such duplicated methods tend to be classified as *single class concepts*, because such duplication typically occurs inside a single class.
- the duplicated behavior could not be factored out into a single piece of code, and thus could not be reused. This is mostly due to the fact that the duplicated code occurs in classes defined in different class hierarchies. As such, these concepts are often classified as *crosscutting concepts*.

5.3 Design patterns

As many design patterns [5] use certain naming conventions, it is no surprise that they are detected by our tool. For example, the *Visitor* pattern uses the convention that each *visit* method defined by a *visitor* class encodes the name of the class being visited. Since they clearly share some substrings, our tool will group a class and its corresponding visit methods inside a single concept.

The *Refactoring Browser* uses a *Visitor* design pattern in order to perform a variety of operations on source code entities, such as renaming and moving them. These entities are represented as subclasses of the `RBProgramNode` hierarchy, while the visitor hierarchy is defined by the `RBProgramNodeVisitor` class hierarchy. Our tool recovered this design pattern instance in two concepts:

(1) A combined concept in classification *hierarchy concepts* which groups all *polymorphic methods* in the `RBProgramNodeVisitor` hierarchy, that implement the behavior to be executed when a particular term is visited. The concept consists of a several sub-concepts, each of which contains all methods defined by subclasses of class `RBProgramNodeVisitor`, dealing with one particular term. For example, one such concept is defined by the properties 'accept', 'return' and 'node', and contains various implementations of the `acceptReturnNode:` method, defined in the `RBProgramNodeVisitor` hierarchy and

responsible for implementing behavior associated to a `RBReturnNode` object. More specifically, the concept consists of four `acceptReturnNode:` methods, implemented in the classes `RBFormatter`, `RBConfigurableFormatter`, `ParseTreeRewriter` and `RBProgramNodeVisitor`.

(2) The second concept is also a *hierarchy concept* and contains all `acceptVisitor:` methods defined by subclasses of `RBProgramNode`. These methods are responsible for calling the appropriate method, corresponding to the node being visited, in the supplied visitor object. They are grouped based on the 'visitor', 'accept', 'program' and'node' substring properties. This is an example of a concept that takes into account both the method's name and the name of its formal parameter, since the `acceptVisitor:` methods always define a formal parameter named `aProgramNodeVisitor`.

Note that the *polymorphic methods* depicted in Figure 2 are also a part of a visitor pattern, used in the *Refactoring Browser* implementation.

Another example our tool detected is the *Abstract Factory* design pattern that is used in the *Soul* application. The factory is responsible for creating new instances of many different classes in the system, among others, subclasses defined in the `AbstractTerm` and `HornClause` hierarchies. Its presence was recognized by one classification that groups different concepts, and that looks similar to the first classification for the visitor design pattern. The classification groups all concepts that contain methods that instantiate new objects. Each such concept groups an abstract method of the `Factory` class and its concrete counterpart defined in the `StandardFactory` class. For example, a concept based on the properties 'make' and 'cut' is identified, that contains the two implementations of the `makeCut` method in the `Factory` hierarchy. In addition, the choice of the word 'make' strengthens our belief that it indeed is a factory.

Several other design patterns, such as the *Builder*, *Observer* and *Decorator* design patterns, were detected in other applications as well.

5.4 Relevant domain concepts

Frequently occurring properties give a good idea of what the important concepts in the application or problem domain are. This information is very useful to understand the domain, and to provide a common vocabulary which can be used to talk with maintainers. For example when applying our FCA tool to the source code of *DelfSTof* itself, we found several concepts with properties like 'concept', 'attribute', 'analyze(r)', 'filter' or 'classification'. Likewise, we found concepts with properties 'lint rule' and 'browser' in the *Refactoring Browser*, as well as concepts whose properties name the different refactorings, such as 'add method', 'rename variable' and 'change name space'. In *SOUL*,

we found concepts with properties 'resolution' and 'repository'. Clearly, these strings convey information that is important in the domain of the respective applications.

5.5 Opportunities for refactoring

In addition to revealing interesting symptoms of good design and important domain concepts, our approach can identify opportunities for refactorings [6] that improve the source code quality.

An obvious opportunity for refactoring is to get rid of some of the code duplication that was detected (§5.2). The way a concept containing duplicated methods is classified, can provide useful hints about which refactorings to apply. If the concept is classified as a *single class concept*, the duplication occurs in a single class, and an *Extract method* refactoring is appropriate. If the concept occurs in the *hierarchy concepts* classification, a combination of *Extract method* and *Pull up method* refactorings is probably more appropriate. Of course, if the duplication is caused by having copied a method that is no longer used, it suffices to simply remove that method.

Concepts that were recognized as *polymorphic methods* can also be inspected for refactoring opportunities, to ensure that polymorphism is well-implemented:

- We could check whether all classes in a class hierarchy understand the polymorphic method. If not, we may need to add an additional one.
- A polymorphic method might also be implemented by several subclasses of a particular superclass, but not by that superclass itself. In that case, a *Pull up method* or *Add class* refactoring may be appropriate to define the methods in the superclass, or to insert an intermediate superclass.

  A particular example we discovered in *SOUL* is the `#updateRepositories` method, which is only defined separately in subclasses `RepositoryBrowser`, `SoulQueryBrowser` and `SoulClauseBrowser` but not in their common superclass `ApplicationModel`. Introducing an intermediate superclass here might be appropriate.

  In the *CodeCrawler* case, we also found several examples where the *same* polymorphic method appeared in several subclasses, but not in their common superclass. However, in most of these examples the code in one subhierarchy did not seem to be used, which made us believe that the code was moved from one part in the hierarchy to another, but that the developer(s) forgot to remove the original code.
- *Crosscutting polymorphic methods* are also suspicious. For example, we discovered that in *SOUL*, the `#usesPredicate:multiplicity:` method is implemented in both the `AbstractTerm` and `HornClause` hierarchies. Smalltalk, which has dynamic typing, still allows classes of these hierarchies to be used polymor-

phically. A static type system, as in Java, would prohibit such polymorphic use. In that case, an interface probably has to be introduced in order to avoid explicit type checks and type casts. If the crosscutting polymorphic methods also happen to exhibit a lot of code duplication, the solution might be to introduce aspects into the application. In that respect, our tool might also be considered as an aspect mining tool, capable of detecting aspect opportunities. Future research into this area appears very interesting.

Concepts that represent design pattern instances can also be scanned for particular design flaws. For example, the *Visitor* design pattern requires that each visitor class defines an appropriate *visit* method and, vice versa, that each element class defines an *accept* method that calls the appropriate *visit* method. The concepts that identify instances of the *Visitor* design pattern can be used to inspect the implementation in a quick and straightforward way, and verify whether these constraints are adhered to.

6 Related Work

The use of FCA in software engineering is not new. Snelting et al. [7] use FCA to reengineer C++ class hierarchies, while Arévalo et al. [3] analyze object-oriented framework reuse using FCA. Closer to our work are the techniques by Tonella et al. [8] and Dekel et al. [9]. The former use FCA to detect instances of design patterns in source code. Since they specifically tune the FCA algorithm for detecting such instances, they are not able to detect other kinds of design symptoms, as our approach does. The latter use FCA to reveal the structure of single classes only. They partition the methods of a class, according to the fields these methods use, and then use the concept lattice to visualize and understand the structure of that class. Tilley et al. [10] provide an overview of the use of FCA for several other software engineering purposes.

A large number of tools to verify the quality of the source code of an application exists. The spectrum ranges from very simple tools that detect basic coding errors [11], over specialized clone detection tools [12,13,14,15], to tools that detect high-level bad smells [16,17,18] and propose appropriate refactorings.

Other tools exist that are capable of detecting high-level structures in source code, such as coding conventions and design patterns [19,20,21,22]. The main difference between these tools and ours, is that our approach requires no a priori knowledge. Most of these existing tools, however, rely on the fact that design pattern implementations follow *particular* naming conventions and guidelines. Our approach is not targeted to detecting a specific kind of conventions, but is able to detect a variety of symptoms that reveal bad design (bad smells,

duplication), good design (design patterns, programming idioms), and opportunities for refactoring. This is particularly useful during initial understanding, whereas in a later phase, when the code is better understood, a more directed tool is preferable.

Research in the domain of aspect mining is also related. Tonella et al. [23] and Breu et al. [24] find aspect instances in existing applications by means of dynamic execution traces. The former work even uses formal concept analysis to that extent. Our approach seems to complement these approaches, since we combine static analysis techniques with formal concept analysis. Marin et al. [25] use the fan-in metric in order to mine for aspects, whereas Shepherd et al. [26] and Bruntink et al. [27] use clone detection techniques.

7 Future work

An important topic of future work is to further *improve the filtering* of the concepts discovered by FCA, so as to reduce the remaining redundancy in the discovered concepts. The problem is that this redundancy occurs between concepts that are classified in different categories. As was briefly mentioned in §3.3, this redundancy is a consequence of having flattened the concept lattice. By doing so we lost some important dependencies between concepts. But exactly this information may be useful to get rid of the redundancy. Therefore, we propose to keep the lattice as an internal representation, so that advanced filters can take advantage of it to resolve the remaining redundancy.

Since the time of computation of our tool theoretically increases in a non-linear way with the number of considered objects, this may pose problems regarding the scalability of the approach. However, we do not think that it is a good idea to apply the approach to *very* large amounts of source code, since the tool assumes that certain naming convention are adhered to in a consistent way. This is unlikely for very large cases. In such a context it would be better to apply the approach multiple times on several smaller pieces of which we know they are more or less independent and have been developed by a same development team. As a side-effect, this will also resolve the problem with the time of computation.

8 Conclusion

In this paper we proposed a fairly efficient tool, capable of delving an application's source code for meaningful and interesting symptoms of good and bad

design. The tool combines formal concept analysis with filtering and classification techniques, in order to provide simple and effective views on structural commonalities in the source code. In order to validate the approach we applied the tool on a number of small to medium-sized Smalltalk applications. In spite of relying on nothing more than similarity of names of source-code entities, we discovered symptoms of programming idioms, code duplication, design patterns and domain concepts, as well as interesting opportunities for refactoring. Although the approach can still be improved in many ways, in particular to further reduce the redundancy, we do believe it can be very useful when an application has to be understood, little a priori knowledge about the source code is available and time is scarce.

9 Acknowledgements

We thank T. Mens, R. Wuyts, R. Riquelme, S. González and G. Arévalo as well as the anonymous reviewers of the ECOOP'2004 *workshop on object-oriented reengineering* and the AOSD'2004 workshop on *foundations of aspect languages* for their feedback on earlier versions of this paper. We thank F. Spiessens for his efficient implementation of the FCA algorithm in Smalltalk. Finally, we thank the ESUG'2004 reviewers and our shepherd Serge Demeyer in particular, for their useful comments.

References

[1] S. Demeyer, S. Ducasse, O. Nierstrasz, Object-Oriented Reengineering Patterns, Morgan Kaufmann and DPunkt, 2002.

[2] B. Ganter, R. Wille, Formal Concept Analysis: Mathematical Foundations, Springer-Verlag, 1999.

[3] G. Arévalo, T. Mens, Analysing object-oriented application frameworks using concept analysis, in: Proceedings of the Inheritance Workshop at the European Conference on Object-Oriented Programming (ECOOP), 2002.

[4] R. Wuyts, S. Ducasse, Unanticipated Integration of Development Tools using the Classification Model, Journal of Computer Languages, Systems and Structures 30 (2003) pp. 63–77.

[5] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, Massachusetts, 1994.

[6] M. Fowler, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999.

[7] G. Snelting, F. Tip, Reengineering Class Hierarchies Using Concept Analysis, in: Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering, 1998.

[8] P. Tonella, G. Antoniol, Inference of object oriented design patterns, Journal of Software Maintenance - Research and Practice 13 (5) (2001) 309 – 330.

[9] U. Dekel, Y. Gil, Revealing Class Structure with Concept Lattices, in: Proceedings of the Working Conference on Reverse Engineering (WCRE), 2003.

[10] T. Tilley, R. Cole, P. Becker, P. Eklund, A Survey of Formal Concept Analysis Support for Software Engineering Activities, in: Proceedings of the International Conference on Formal Concept Analysis, 2003.

[11] S. Paul, A. Prakash, A Framework for Source Code Search using Program Patterns, IEEE Transactions on Software Engineering 20 (6).

[12] B. S. Baker, On Finding Duplication and Near-Duplication in Large Software Systems, in: Proceedings of the IEEE Working Conference on Reverse Engineering (WCRE), 1995, pp. 86–95, received IEEE Outstanding Paper Award.

[13] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, L. Bier, Clone detection using abstract syntax trees, in: Proceedings of the International Conference on Software Maintenance (ICSM), IEEE Computer Society Press, 1998, pp. 368–377.

[14] S. Ducasse, M. Rieger, S. Demeyer, A Language Independent Approach for Detecting Duplicated Code, in: H. Yang, L. White (Eds.), Proceedings of the International Conference on Software Maintenance (ICSM), IEEE Computer Society Press, 1999, pp. 109–118.

[15] R. Komondoor, S. Horwitz, Using Slicing to Identify Duplication in Source Code, in: Proceedings of the International Symposium on Static Analysis, Springer-Verlag, 2001.

[16] Y. Kataoka, M. D. Ernst, W. G. Griswold, D. Notkin, Automated Support for Program Refactoring using Invariants, in: Proceedings of the International Conference on Software Maintenance (ICSM), 2001, pp. 736–743.

[17] T. Tourwé, T. Mens, Identifying Refactoring Opportunities Using Logic Meta Programming, in: Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR), IEEE Computer Society, Benvento, Italy, 2003, pp. 91 – 100.

[18] E. van Emden, L. Moonen, Java Quality Assurance by Detecting Code Smells, in: Proceedings of the Working Conference on Reverse Engineering (WCRE), IEEE Computer Society Press, 2002.

[19] H. Albin-Amiot, P. Cointe, Y.-G. Guéhéneuc, N. Jussien, Instantiating and Detecting Design Patterns: Putting Bits and Pieces Together, in: Proceedings of the International Conference on Automated Software Engineering (ASE), 2001.

[20] K. Brown, Design reverse-engineering and automated design pattern detection in smalltalk, Master's thesis, University of Illinois at Urbana Champaign (1996).

[21] K. Mens, I. Michiels, R. Wuyts, Supporting Software Development through Declaratively Codified Programming Patterns, Journal on Expert Systems with Applications .

[22] R. Wuyts, Declarative Reasoning about the Structure of Object-Oriented Systems, in: Proceedings of TOOLS USA'98, IEEE Computer Society Press, 1998, pp. 112–124.

[23] P. Tonella, M. Ceccato, Aspect Mining through the Formal Concept Analysis of Execution Traces, in: Proceedings of the Working Conference on Reverse-Engineering (WCRE), 2004.

[24] S. Breu, J. Krinke, Aspect Mining Using Dynamic Analysis, in: GI-Softwaretechnik-Trends, Mitteilungen der Gesellschaft für Informatik, Vol. 23, Bad Honnef, Germany, 2003, pp. 21–22.

[25] M. Marin, A. van Deursen, L. Moonen, Identifying Aspects using Fan-In Analysis, in: Proceedings of the Working Conference on Reverse-Engineering (WCRE), 2004.

[26] D. Sheperd, E. Gibson, L. Pollock, Automated mining of desirable aspects, Tech. Rep. 4, Department of Computer and Information Sciences, University of Delaware, Newark, DE 19716 (2004).

[27] M. Bruntink, A. v. Deursen, R. v. Engelen, T. Tourwé, An Evaluation of Clone Detection Techniques for Identifying Crosscutting Concerns, in: Proceedings of the International Conference on Software Maintenance (ICSM), IEEE Computer Society, 2004.

Power Laws in Smalltalk

Michele Marchesi, Sandro Pinna, Nicola Serra, Stefano Tuveri

Dipartimento di Ingegneria Elettrica ed Elettronica, Università di Cagliari,
Piazza d'Armi, 09123 Cagliari, Italy
{*michele,pinnasandro,nicola.serra,stefano.tuveri*}*@diee.unica.it*

Abstract

Many real systems have been described as complex networks, where nodes represent specific parts of the system and connections represent relationships among them. Examples of such networks come from very different contexts. Traditional theories suggest to represent complex systems as random graphs, according to the models proposed by Erdős and Rényi. However, there is an increasing evidence that many real world systems behave in different ways displaying *scale-free* distributions of nodes degree. Random graphs are not suitable to describe such behaviors, thus new models have been proposed in recent years. Recently, it has emerged the interest in applying complex network theories to represent large software systems. Software applications are made of modules and relationships among them. Thus, a representation based on graph theory is natural. This work presents our study [1] in this context. We analyzed four large Smalltalk systems. For each one we built the graph representing all system classes and the relationships among them. We show that these graphs display scale free characteristics, in accordance to recent studies on other large software systems.

Key words: software graphs, smalltalk, power-laws, scale-free networks, software architecture

1 Introduction

In recent years, many studies have been carried out on the application of complex networks theory to physical, biological and human phenomena. In fact, many systems can be described as complex networks, whose nodes represent the elements of the system, and edges represent the interactions between them. Examples of such systems are living systems, whose nodes are macromolecules with different functions, connected through chemical links; the stock

[1] This study is part of MAPS research project (Agile Methodologies for Software Production, funded by FIRB research fund of MIUR.)

market, where many traders are connected through information and opinion exchanges; the Hollywood movie system, where stars take part together in a movie; the Web, where pages link other pages. Such networks grow and evolve increasing their complexity in an apparently disordered way.

Using the classical random graph theory developed by Erdős and Rényi [10][11] to model such systems looks sensible, but most of these systems in fact behave according to laws significantly different from those predicted by Erdős and Rényi theory.

More precisely, there is increasing evidence that several real networks behave as *small worlds*, simultaneously showing a short average minimum length path and a high clustering behavior [16]. Moreover, many real networks show interesting laws in the distribution of the number of links connected to a node. The tails of such distributions follow a *power law*, that is a significant deviation from the Gaussian behavior that would be expected if links were randomly added to the network [3]. The meaning of the term *small worlds* and the difference between *power law* and *gaussian law* will be clarified within the next section.

Surprisingly, such properties are common to very different real systems. This suggests that the modeled behaviors are independent of the particular nature of the represented real system. By the contrary, it seems like these are general and universal behaviors which raise as a consequence of the sole complexity of the system.

Large software applications are considered to be among the most complex artifacts ever produced by man [6], and consequently are good candidates to be modeled as complex networks, and to be studied with complexity theory. This is particularly true for object-oriented systems, where objects and classes are natural candidates to be represented as nodes, and the various possible relationships between them – such as inheritance, instantiation, composition, dependence – are represented as arcs connecting nodes. Thus, it is natural to try to describe such systems by complex systems theory models. Very recently, some studies have been performed on software systems showing that run-time objects [13] and static class structures of object oriented systems are in fact governed by scale free power law distributions [14][15]. Other studies have been performed by Myers [12] and Wheeldon and Counsell [17], leading to similar results. Most of these studies of complex software systems are based on C++ and Java code.

In this paper we present a study on the Smalltalk system, a very complex software application that is considered the most pure object-oriented system. Namely, we study the Squeak [9] and VisualWorks [8] Smalltalk implementations. The dynamic typing nature substantially differentiates Smalltalk by

other languages such as C++ and Java. This makes the study of Smalltalk systems particularly interesting. Accordingly to our previsions and similarly to other software systems developed using different languages, we will show that Smalltalk systems display link distributions whose tails follow a *power law*. We think that this approach could be an important starting point to better understand the nature and evolution of large software systems. Therefore, with this paper we want to suggest a new point of view for reviewing well known properties of software systems and for leading to new results that are not reachable with the classic software engineering measurement approaches.

2 Small Worlds and Scale Free Networks

Traditionally, complex networks have been described as completely random or completely deterministic. Only recently it has been shown that many real technological, biological and social networks lie somewhere between these extremes. Erdős and Rényi pioneered the study of the behavior of random graphs. According to their model, a "random graph" is made starting from a set of n nodes and connecting each pair independently with a given probability p. Random graphs exhibit many interesting properties, depending on how large is the number of nodes n, and on the connection probability p. The probability distribution of the nodes degree, k in a random graph with n vertexes and z connections is the well-known Poisson distribution:

$$p_k = \binom{n}{k} \cdot p^k \cdot (1-p)^{n-k} \cong \frac{z^k \cdot e^{-z}}{k!} \tag{1}$$

where the second approximation becomes exact as the number of nodes n becomes very large. Watts and Strogatz [16] observed that a very important property of random graphs is the small value for the average minimum length path, coupled with the low clustering coefficient. In a graph, the minimum length path between two nodes is the shortest path of consecutive edges connecting the two nodes. The average minimum length path is the average of such value taken among all possible node pairs. The clustering coefficient of a graph is a measure of how clustered, or locally structured, a graph is. Specifically, if node i is linked to K_i neighbor nodes, then we define the clustering coefficient for node i, C_i, as the fraction of the total possible $\frac{K_i \cdot (K_i - 1)}{2}$ arcs that are realized between i's neighbors. Empirical evaluation of many real networks shows that they are characterized by a small value of the minimum length path, and by a large clustering coefficient. This means that random graphs are not suitable to model these real-world networks. Watts and Strogatz defined as *small worlds*, the networks characterized by a small average minimum length path, and by a large clustering coefficient, and proposed an interesting method

to build small words networks starting by a lattice network. The other main obstacle to apply Erdős and Rényi theory to real networks is the empirical observation that the predicted Poisson distribution of the nodes degree given in equation 1 does not fit the observed nodes degree distribution of a great number of real technological, social and biological networks. Barabasi and Albert [3] showed how this incongruence between the random graphs theory and real networks is due to two main factors not accounted for by the Erdős and Rényi model:

- Real networks expand continuously by the addition of new vertexes
- Connection between nodes are preferential rather than independent

The above conditions are demonstrated to be sufficient conditions for displaying a power law tail in the distribution of the node degrees:

$$p_k \propto k^{-\gamma} \tag{2}$$

The theoretical model provided by Barabasi asserts the value for the exponent to be $\gamma = 3$. Empirical studies on real networks such as the Web, the North American power grid, the Hollywood actors network and others, display values for the exponent very close to the theoretical γ.

It is very surprising how many real systems modeled as networks reflect this power law distribution in accordance with theoretical evidence. This suggests that many real systems share common properties and behaviors in their structure, which are independent of the particular nature of the system itself. Those properties seem to be related exclusively with the complexity of the system modeled. Software systems play an important role in this context, as they are obviously characterized by a high level of complexity.

3 Software Applications as Complex Systems

Software applications and architectures have become ever larger and more complex over the past years. So, it is sensible to apply complex systems and graph theories to model and to study large software applications. The study of software systems as complex networks has a great scientific interest. It is acknowledged that software networks play a special role in the context of random networks theory, as it seems reasonable that the small world and scale-free signature origin from different factors compared to other real networks. In fact, software is built up out of many interacting modules and subsystems at many detail levels (methods, classes, hierarchies, libraries, etc.), and good software is developed following collective and collaborative designs, design patterns and

optimization principles, rather than an explicit preferential attachment. This suggests an alternative scenario for generating scale-free networks, which is more related to optimization rather than to Barabasi hypothesis[14]. This enforces the idea that the power law shape of degree distribution tails describes general behaviors of complex networks.

Moreover, the analysis of software systems structure and evolution as complex networks could also be of practical interest, as it might provide an alternative perspective able to help a better understanding of the mechanisms ruling software production or the relationships among network structure and software quality. For instance, an entire new bunch of metrics computed on the program graph could be introduced, and correlated with external software metrics.

One of the first studies of this kind[13] has shown that the graphs formed by run time objects, and by the references between them in object-oriented applications are characterized by a power law tail in the distribution of node degrees. Valverde and Sole[14][15] found similar properties studying the graph formed by the classes and their relationships in large object-oriented projects. They found that software systems are highly heterogeneous *small worlds* networks with scale-free distributions of connections degree. Wheeldon and Counsell[17] performed similar studies on Java projects. Myers[12] found analogue results on large C and C++ open source systems, considering the collaborative diagrams of the modules within procedural projects and of the classes within the OO projects. He also computed the correlation between complexity metrics and topological measures, reveling that nodes with large output degree tend to evolve more rapidly than nodes with large input degree.

Our study is a contribution to the open issues quoted above. This paper introduces a procedure for building the class graphs of a group of Smalltalk programs, and presents the related results. Smalltalk language provides a powerful environment to easily build and analyze graphs representing its own classes. Moreover, the dynamically typed structure of the language adds more interest as it represents a fundamental difference with respect to other related studied languages. On the other hand, for the same reason, it is hard to statically resolve all class relationships.

4 Building the Smalltalk Graph

A software system is made of modules and relationships between them. More precisely, a software application developed according to the object-oriented paradigm is made of classes and well defined relationships between classes. Representing such a software system as a graph is natural. In this graph, each node represents a class within the system, while the arcs between nodes repre-

sent the relationships among classes. An arc links a couple of nodes, one representing the starting class, and the other the ending class. The graph is thus an oriented graph. An arc has also a weight, which is a measure of the degree of the relationship between the two classes. While there may be many kinds of relationships between classes, for the sake of simplicity, in this study we have considered just two kinds of relationships: inheritance and dependence, each one giving a different contribution to the arc weight. The inheritance is represented by an arc with unitary weight, from the subclass to the superclass.

As regards dependence, we say that class A depends on class B when one of the following conditions holds:

- class A has an instance variable whose type is B (composition);
- a method of class A has a variable (parameter or local variable) whose type is B;
- an object of class B is obtained by a method call, or created, inside a method of class A.

The dependence analysis must consider that Smalltalk is not a typed language. It is a dynamic, implicitly typed language where objects, not variables, carry type information. This frees the programmer from declaring variable types, but embeds less information into the source code. None of the three conditions reported above can be easily tested by an analysis of Smalltalk code. We observe, however, that having access in the code of a method to a variable of a given class is significant only if one or more messages are sent to this variable. So, we define that class A depends on class B if a method of class B is called from within a method of class A. If the considered method has only one implementor, class B, the dependence link is clearly unambiguous. If the method has more than one implementor class, say n, it is not possible to ascertain with a static analysis which one is the right one.

In this situation, we decided to introduce a dependency towards all implementor classes, weighting such a dependency with weight $1/n$. If a method of class B is called by more than one method of class A, or more than once within the same method of class B, we introduce weighted arcs for every call. Inheritance is not considered in the dependency analysis. For instance, let us suppose that class C is subclass of class A, that class D is subclass of class B, and that a method of B – which is not overridden in class D – is called inside a method of A, which in turn is not overridden in class C. At run time, it may be possible that an object belonging to class C calls the method, and/or that such a method is in fact executed on an object belonging to class D. Nevertheless, in this case only a dependence between class A and class B is recorded. Note that this issue holds also for typed languages, such as Java or C++, when dynamic binding is used.

In conclusion, our graph is a collection of nodes:

$$G \equiv \{N_i\}_{i=1..N_c} \tag{3}$$

where N_c is the total number of classes within the represented software system. Thus, the graph contains one node for each system class C_i. A node N_i has a collection of links, each representing a relationship that the class represented by N_i has with other classes in the system. Links, or arcs, are pairs of nodes representing respectively the starting and the ending class:

$$L_{ij} = (N_i, N_j) \tag{4}$$

A link L_{ij}, carries also a weight that we indicate as $W(L_{ij})$. Now let's introduce the symbols we use for representing methods, messages and implementors:

- $M(C_i) = \{methods\, of\, the\, class\, C_i\}$
- $S(m) = \{messages\, sent\, inside\, m\}, m \in M(C_i)$
- $P(s) = \{implementors\, of\, message\, \mathbf{s}\}, s \in S(m)$
- $dim(P(s)) = number\, of\, implementors\, of\, message\ \mathbf{s}$

We define the weight of a link as the sum of related dependence, W_{dep}, and inheritance, W_{inh}, contributions:

$$W(L_{ij}) = W_{dep}(L_{ij}) + W_{inh}(L_{ij}) \tag{5}$$

$$W_{dep}(L_{ij}) = \sum_{m \in M(C_i)} \sum_{s \in S(m)} \frac{1}{dim(P(s))} \cdot I_s(C_j) \tag{6}$$

$$I_s(C_j) = \{^{1\ if\ C_j \in P(s)\ (C_j\, is\, an\, implementor\, of\, s)}_{0\ otherwise} \tag{7}$$

$$W_{inh}(L_{ij}) = \{^{1\, if\, C_j\, is\, the\, direct\, superclass\, of\, C_i}_{0\, otherwise} \tag{8}$$

Now we can define both the input degree, and output degree of node N_i:

$$outputDegree(N_i) = \sum_{j=1}^{N_c} W(L_{ij}) \tag{9}$$

$$inputDegree(N_j) = \sum_{i=1}^{N_c} W(L_{ij}) \tag{10}$$

The graph built in this way reflects the relationships between the classes, using in the least biased possible way the information that can be inferred by static analysis of a Smalltalk system. The input degree of a class is directly linked to the usage of this class in the system.

We performed also another measurement on Smalltalk system, considering the distribution of classes implementing a method. For each method selector, M, we easily found in the system its implementors, $P(M)$, and their number, $dim(P(m))$.

5 Analyzing Smalltalk Classes

Once defined the graph model used to capture the structure of the software system under study, we need to build an analyzer able to generate such a graph, starting from the software system. With strong typed languages, this would be accomplished by parsing the source code, recognizing class and variable definitions, and generating the graph, as in [17]. Another approach is to use a CASE tool able to reverse-engineer the code, building the related UML class diagram [5]. The code analyzer can then take advantage of the navigation API of the CASE tool, to explore class relationships and to build the graph. Following this approach, it is possible also to directly analyze UML design models. This is the approach followed in [15].

With Smalltalk, we can take full advantage of the introspection of the language. The whole system is accessible from within itself, and no source code parser or reverse engineering is needed. Moreover, Smalltalk is endowed of powerful query methods able to return useful information, as for instance the set of all the classes of the system, the set of classes implementing a given selector, the set of messages sent within a given method, and so on. We remember that in Smalltalk everything is an object, including the classes, the methods, the message calls themselves. We analyzed two dialects of Smalltalk – Squeak (version 3.5), an open-source implementation [9], and VisualWorks (version 7.2), a commercial implementation which is freely available for non-commercial uses [8]. The system classes and methods to perform queries about the system differs between Squeak and VisualWorks, but they are substantially equivalent. We defined a simple data structure able to hold the needed information on the graph describing the system. Figure 1 shows the UML class diagram depicting this structure. A **Graph** is a collection of nodes; a **Node** has a related class and a collection of links. A **Link** has a startNode, an endNode and a weight.

To give an insight on how the dependence analysis was performed, we shortly describe its implementation in Squeak. The main classes involved in the computation are **SystemNavigation**, **Class** and **CompiledMethod**. The method

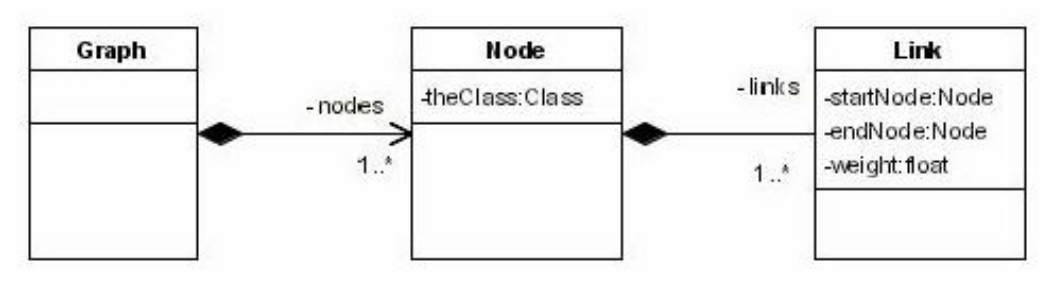


Fig. 1. *The graph structure*

allClasses of **SystemNavigation** returns the collection of all classes in the system. For a given class, the method **selectors** returns all selectors of that class. A selector is a Symbol (a kind of String) representing a method signature. The whole method, instance of class **CompiledMethod**, can be obtained by sending the message **compiledMethodAt: aSelector** to the class. The **CompiledMethod** class implements the method **messages** which returns the collection of all messages sent inside the CompiledMethod. A message is in turn a Symbol representing the selector of the corresponding invoked method. The **SystemNavigation** class implements the method **allClasses-Implementing: aMessage** which returns a collection of all classes implementing the given message. In this way, it is possible to navigate the system, building the graph described in the previous section.

A similar approach has been adopted for Visual Works even if the involved classes are not the same in the two systems. For example in VW we use the class **Refactory.Browser.BrowserEnvironment** instead of **SystemNavigation**, and so on. The graph construction has taken about four hours in Squeak (1797 classes), and three hours in VisualWorks (2227 classes) on a PC running Windows, powered by a 2.6 GHz processor.

6 Results

Our study has been structured across the following points:

- Building of the class relations graph
- Computing the survival distributions of the input degree and output degree
- Plotting the survival distributions on a Log-Log plot
- Discussing the results

Our main purpose is to verify that the distributions tails are better fitted by a power law rather than by the Poisson distribution typical of the random graphs. Moreover, we want to interpret the results in a less general context and from the more practical point of view of the software developer.

Namely, we analyzed four Smalltalk systems:

- Squeak
- Visual Works with three different parcels installed:
  - Base image
  - Base image plus Jun parcels
  - Base image plus VisualWave parcels

Jun is a large 3D graphical application. VisualWave is the VisualWorks extension to support the development of Web applications. Note that the VisualWorks parcels are analyzed together with the base image, so the results referring to these cases are clearly correlated to the results referring to the base image.

6.1 Distributions of input degree and output degree

Figure 2 and figure 3 display the distribution tails of the input degree and output degree respectively for VisualWorks and for Squeak systems. More precisely, we computed the survival distributions and plotted them in a log-log plot. The survival distribution depicts the probability that a value exists that is greater than the current one. The log-log plot performs a transformation of the power law function into a straight line function, with slope equal to the power exponent of the original curve. This allows to visually better verify the presence of a power law and to estimate the γ coefficient.

As expected, the plots show that distribution tails are actually well fitted by a power law. Table 1 shows the fitted γ exponent of the power law distributions obtained for each system graph, compared with the number of classes of the analyzed systems. The power exponent γ of the distributions is quite close to the theoretical value ($\gamma = 3$) obtained by Barabasi and it is coherent to the empirical values ($2 < \gamma < 4$) obtained for many real networks. An higher exponent denotes a tail decreasing quicker. A small exponent denotes a "fatter" tail, that is a bigger deviation from the behavior of a Gauss or Poisson distribution.

Despite this general result, which is in fact shared by various kind of different real networks, our main interest is to interpret its meaning in the specific context of software systems. The number of outgoing links of a node is a measure of how many messages are sent from within the methods of the corresponding class. A high value of the number of outgoing links denotes that the class performs many message calls from within its methods. This can be considered also as a measure of the coupling between classes. Thus, a power law distribution with fat tails is a clue that the system is characterized by a certain number of classes with an high level of coupling, while the bulk of the classes tends

to be much less coupled. This interpretation is supported by other previous studies. For example, Chidamber and Kemerer [7] and Basili [4] found similar distributions for coupling metrics.

Note also that, since the output degree of a node is not divided by the number of methods of the related class, it is clearly proportional to the number and size of methods of the class. Consequently, the power law behavior seems to be related also with the distribution of class sizes, and with the fact that, when a new method is added to the system, it is more likely that it belongs to a class that already has many methods. However, both high coupling and big classes are not considered good object oriented programming style. A good object oriented system should be composed of many cohesive classes, at different detail levels, each one focusing on a single task, and endowed with a few, short methods. Our empirical analysis of the Smalltalk system shows that this is not the case, because the distribution of the number of outgoing links decays with a power law. This observation is confirmed by other studies, showing the same behavior in Java applications [15].

A broad analysis of many OO systems is outside the scope of this paper. However, these behaviors clearly point out an interesting research direction, suggesting to perform more detailed analysis on this indicator, for instance analyzing separately system classes and classes written by programmers, and analyzing the values of the outgoing links power law degree in systems with different quality rating, or before and after an extensive refactoring.

The other main indicator we examined is the input degree, which gives a measure of how a certain class is referenced by other classes in the system. The shape of the distribution is due to the fact that in Smalltalk system, there are "service" classes that are used by almost other classes. Classes such as **Object**, the **Collection** and the **Magnitude** hierarchies, come immediately to mind. There are also classes which are fairly used, both in the system and in specific parcels, while most other classes are rarely used. The usage rate is certainly not random. This behavior suggests the hypothesis that when defining a new class, it is likely to be subclass of a class already having more subclasses, and that when writing a new method the probability to send a message implemented in another class is roughly proportional to the number of sending of that message in the system. In fact, this probability is more than proportional to this number, as the coefficient values smaller than 3 suggest.

6.2 Distributions of method implementors

We also computed the distribution of the method implementors on the VisualWorks system, respectively for the base image and for the base image with

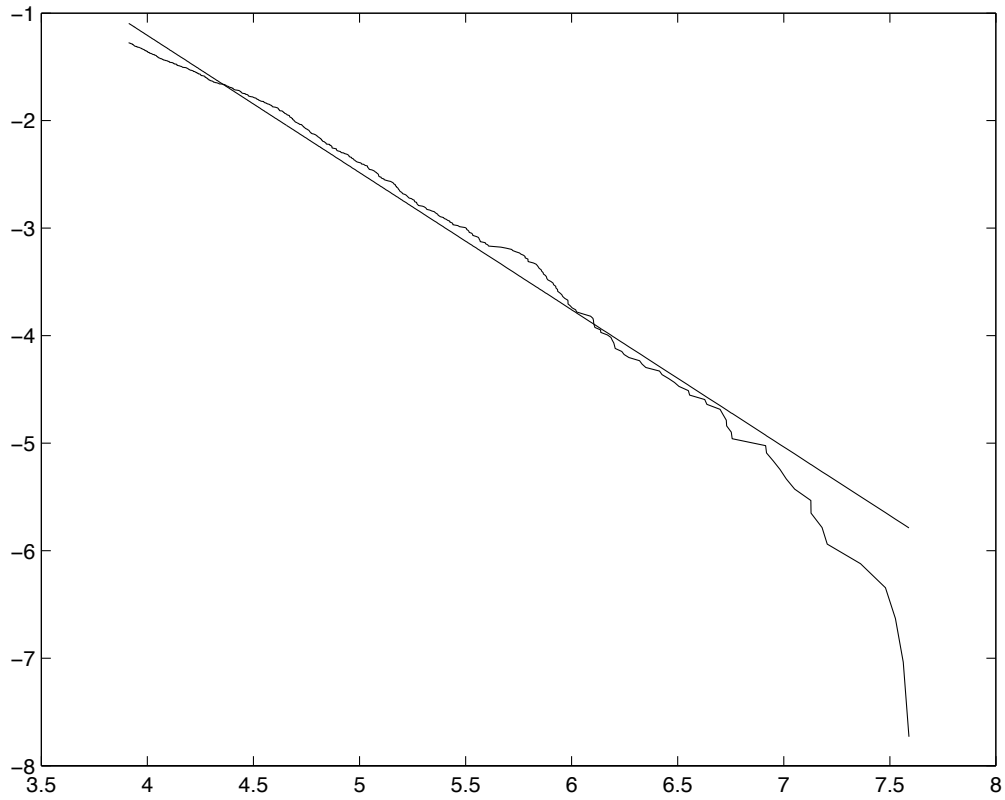


Fig. 2. *Log-Log plot of input degree survival distribution computed on VisualWorks system. Similar plots are obtained for other analyzed systems: Squeak, VisualWorks with Jun and VisualWorks with VisualWave.*

	inDegree	**outDegree**	**number of classes**
Squeak	-2.07	-2.3	1797
Visual Works	-2.28	-2.73	2227
Jun	-2.39	-2.55	3022
Visual Wave	-2.26	-2.53	2648

Table 1
Power Law exponents for input degree and output degree distributions.

Jun and VisualWave parcels loaded. Figures 4 and 5 show the log-log plots of the survival distributions for the number of implementors of each method for VisualWorks base image and Jun, together with best-fit Poisson distribution of the same data.

Table 2 shows the exponent γ of the power law distributions obtained for each implementors graph, compared with the total number of considered methods.

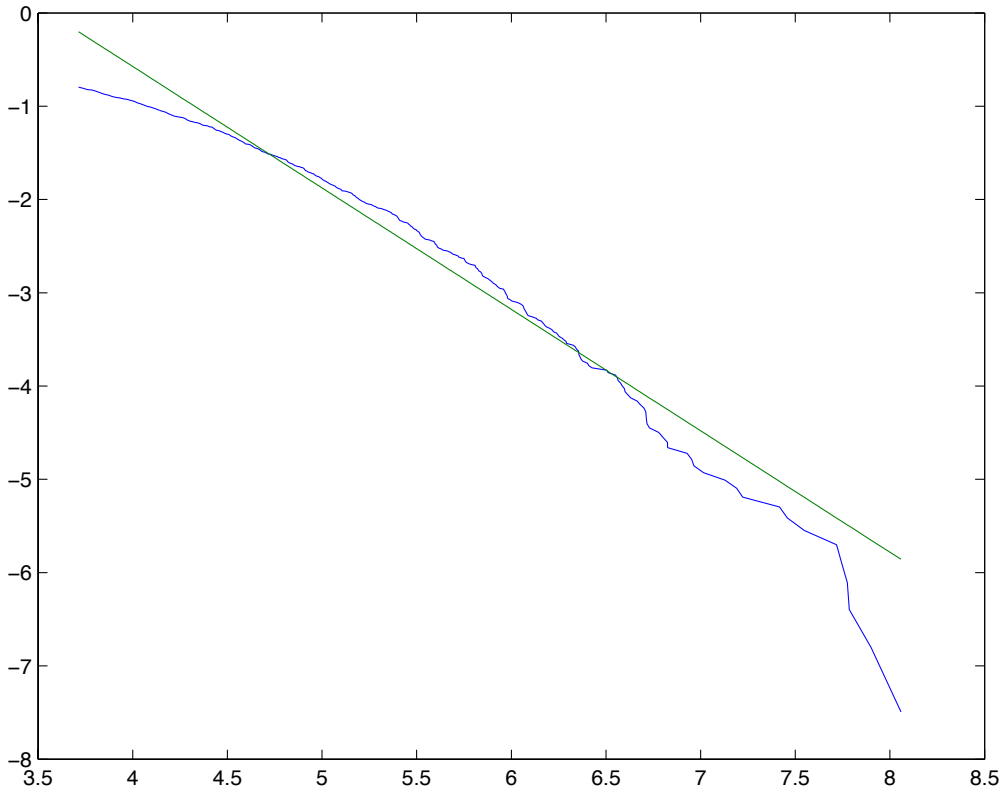


Fig. 3. *Log-Log plot of output degree survival distribution computed on Squeak system. Similar plots are obtained for other analyzed systems: VisualWorks, VisualWorks with Jun and VisualWorks with VisualWave.*

Clearly, also the tail of these distributions show a power-law behavior, with exponent lying in ranges similar to those of previous distributions.

This behavior denotes that, in the Smalltalk system, there are methods with the same name implemented in many classes, while most methods are implemented in one or few classes. This means that, when defining a new method, the probability that its name is the same of a method already present in the system is roughly proportional to the number of times that method is implemented in different classes. We know that it is good programming practice to give the same name, in different classes, to methods performing the same kind of service. This seems to be confirmed by the behavior of these distributions.

7 Conclusions and further works

In this paper we have studied distribution laws related to object-oriented class relationships of large system and application libraries of a "pure" object-

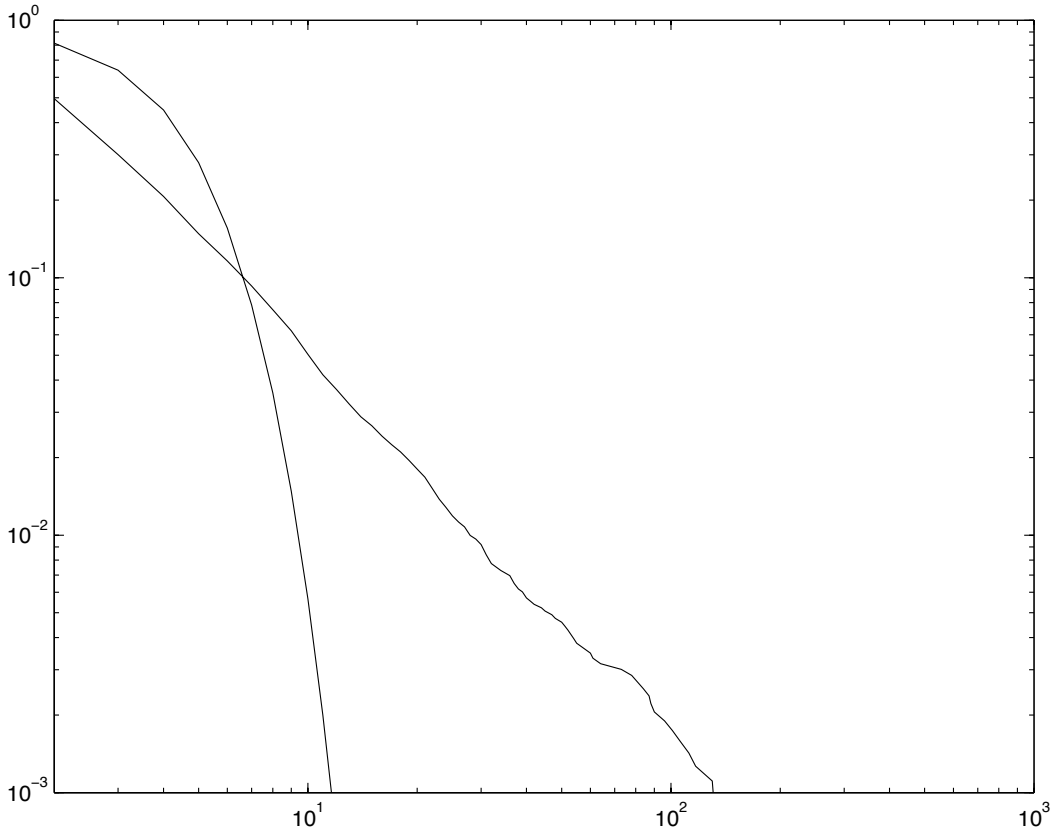


Fig. 4. *VisualWorks, log-log plot of survival distributions of the number of implementors of each method. The best-fit Poisson distribution of the same data is also shown.*

oriented system as Smalltalk. Not unlike very recent findings by others on different object-oriented designs and systems, we found that key statistical distributions of the class-relationship graph exhibit scale-free and heavy-tailed degree distributions qualitatively similar to those observed in recently studied biological and technological networks. Moreover, these distributions show strong regularities in their characteristic exponents.

Following Wheeldon and Counsell [17], we believe that these regularities are common across all non-trivial object-oriented programs. This is a strong impli-

	$-\gamma$	**Number of Methods**
Visual Works	-2.56	23883
Jun	-2.27	33489
Visual Wave	-2.54	27780

Table 2
Power Law exponent for the distributions of the number of implementors of each method

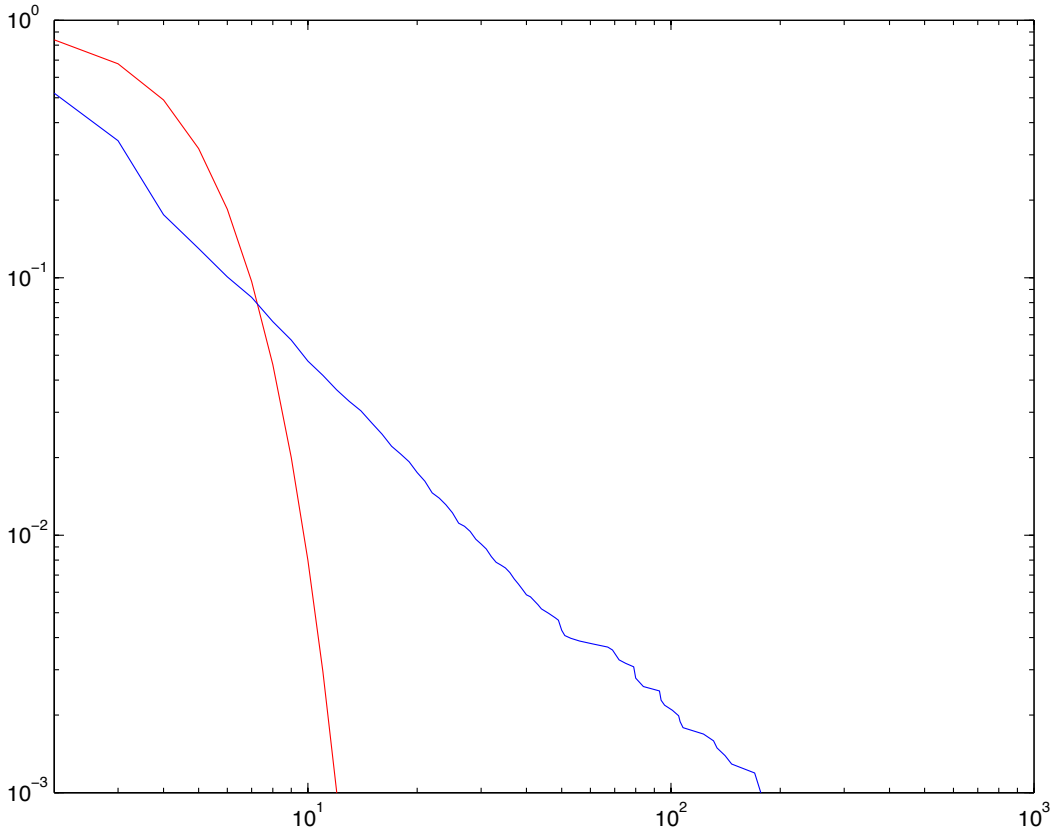


Fig. 5. *VisualWorks with Jun parcel loaded, log-log plot of survival distributions of the number of implementors of each method. The best-fit Poisson distribution of the same data is also shown.*

cation indeed, and could show the path of future research along the following directions:

- What are the statistical mechanisms involved in software development, and how it is possible to model them, accounting for the hierarchical nature of software design, and for the relationships among network structure, object interactions, and system evolution?
- During system development, when these scale-free patterns do emerge? Is it possible to develop a growth theory of software graphs, which in turn could be used to predict the dimensions of future systems and to estimate the complexity of developing and maintaining those systems?
- Is it possible to correlate statistical graph properties with software quality? What about the impact of refactoring on the software graph?
- Are there differences in statistical graph properties between open-source software, which is developed by many programmers in a somewhat destructured way, and commercial software developed following a strict process by full-time developers?
- Are there differences in statistical graph properties between software de-

veloped in the agile way, with short iterations, test-driven development, continuous integration and extensive refactoring, and software developed with waterfall-like approaches?

Answering to these questions using an innovative approach such as the augmented random graph theory could break new grounds in software engineering, and could be very valuable.

As a further conclusion, we agree with Valverde [15] that software systems present novel perspectives also to the study of complex networks. Software must be both functional and evolvable, and unlike biological systems it is to a large extent designed in advance. Traditional software does not emphasize redundancy to support fault tolerance, but it presents other degrees of freedom that play a central role in supporting evolvability, such as modularity, layering, cohesion, genericity, polymorphism, and collaboration. Are some of these degrees of freedom relevant also to the organization and evolution of biological networks? The study of software systems from the new perspective of statistical graph properties may be also useful in suggesting novel insights into collective biological function.

References

[1] Réka Albert, Hawoong Jeong, Albert-László Barabasi: Attack and Error Tolerance of Complex Networks. Nature Vol. 406, pp 378-382 (2000)

[2] Albert-László Barabasi: Linked: the new science of networks. Persus Press, New York (2002)

[3] Albert-László Barabasi, Réka Albert: Emergence of scaling in random networks. Science Vol. 286, pp. 509-512 (1999)

[4] Victor R. Basili, Walcelio L. Melo, A Validation of Object-Oriented Design Metrics as Quality Indicator, IEEE Transaction on Software Engineering, Vol.22, No. 10, (October 1996)

[5] Grady Booch, Ivar Jacobson, James Rumbaugh: The Unified Modeling Language User Guide. Addison-Wesley, Reading, MA, (1999)

[6] Frederick P. Brooks: The Mythical Man-Month. Addison-Wesley, (1995)

[7] Shyam R. Chidamber, Chris F. Kemerer, A Metric Suite for Object Oriented Design, IEEE Transaction on Software Engineering, Vol.20, No. 6, (June 1994)

[8] Cincom Corp., VisualWorks Application Developer's Guide. Cincom, (2004)

[9] Mark J. Guzdial: Squeak: Object-Oriented Design with Multimedia Applications. Prentice-Hall, (2000)

[10] Paul Erdős, Alfred Rényi: On random graphs.I, Publ. Math. Debrecen 6, pp. 290-291, (1959)

[11] Paul Erdős, Alfred Rényi: On the Evolution of Random Graphs.Publ.Math. Inst. Hungar. Acad. Sci. 5, pp. 17-61, (1960)

[12] Christopher R. Myers: Software systems as complex networks: structure, function, and evolvability of software collaboration graphs. Phys. Rev. E 68, 046116 (2003), preprint at cond-mat/0305575

[13] Alex Potanin, James Noble, Marcus Frean, Robert Biddle: Scale-free geometry in Object Oriented programs. Victoria University of Wellington, New Zeland, Technical Report CS-TR-02/30 (2002)

[14] Sergi Valverde, Ramón F. Cancho, Richard V. Sole: Scale-Free networks from optimal design, Europhisics Letters 60, pp. 512-517 (2002)

[15] Sergi Valverde, Richard V. Sole: Hierarchical small worlds in Software Architecture. Submitted to IEEE Transactions of Software Engineering (2003)

[16] Duncan J. Watts, Steven H. Strogatz: Collective dynamics of 'small-world' networks. Nature Vol. 393, pp. 440-442 (1998)

[17] Richard Wheeldon, Steve Counsell: Power law distributions in class relationships.Proc. Third IEEE Int. Workshop on Source Code Analysis & Manipulation, Amsterdam, The Netherland, (Sept. 2003)

DynaGraph : a Smalltalk Environment for Self-Reconfigurable Robots Simulation

Samir Saidani

Laboratoire GREYC, Université de Caen, France, saidani@info.unicaen.fr

Michaël Piel

Laboratoire GREYC, Université de Caen, France, mpiel@etu.info.unicaen.fr

Abstract

In the field of self-reconfigurable robots, a crucial problem is to model a modular robot transforming itself from one shape to another one. Graph Theory could be the right framework since it is widely used to model different kind of networks. However this theory in its present state is not suitable to model dynamic networks, *i.e.* networks whose topology changes over time. We propose to inject a dynamic component into graph theory, which allows us to talk about dynamic graphs in the sense of a discrete dynamical system. To address this problem, we present in this paper the theoretical framework of dynamic graphs. Then we describe the Smalltalk implementation of a dynamic graph, allowing us to perform simulations useful to understand the dynamics of such graphs.

Key words:
Self-Reconfigurable Robots, Dynamic Graphs, Dynamic Cellular Automata

1 Introduction

1.1 Self-Reconfigurable Robots

Our work is situated in the more general field of modular robotics research and specifically in the area of self-reconfigurable robots. Self-reconfigurable robots have the ability to adapt to their physical environment and their required task by changing shape. Some of the self-reconfiguring robot systems are heterogeneous : some modules are different from the others, whereas in homogeneous approach all the modules are identical.

In the literature, we distinguish two kinds of work. The first one tries to define elementary modules which can be human-assembled to rapidly build robots dealing with specific problems. For example, RMMS [1] applies this approach to build manipulators, as well as the Golem project [2] where "robots have been robotically designed and robotically fabricated" and Swarm-bots [3] for the design and implementation of self-organizing and self-assembling artifacts.

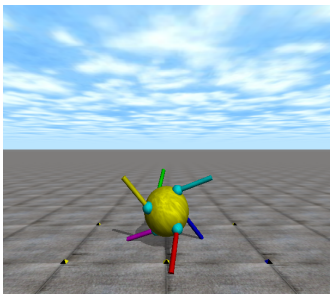
On the other hand, modular robotics look for self-reconfigurable structures. In this case, the problem is often to build identical components which dynamically reconfigure themselves to adapt their behavior to a specific task. Some works already done on this kind of self-organizing components include the Molecule Robots [4], based on a single component with two elementary movements, USC/ISI Conro [5] assembled as a serial chain with two degrees of freedom and docking/dedocking connectors, I-Cube [6] modules, quite similar but with three rotations. In the Crystalline robot [7], the elementary component uses a 2D translation movement, Telecube [8] implements a 3D version of this Crystalline component, PolyBot/Polypod [9] has a very rich structured based on simple components and finally the MEL [10] proposes a two-rotation element with a universal connecting plate allowing dynamic coupling. There is already a lot of proposals, but we think there is still an interest in building another kind of robotic atom : in our project, each module can move around autonomously and connect each other to form a new robot. In this manner, they act as ants : they can cooperate to achieve various tasks or they can form a new structure by connecting themselves to cross a hole for instance.

1.2 The MAAM Project

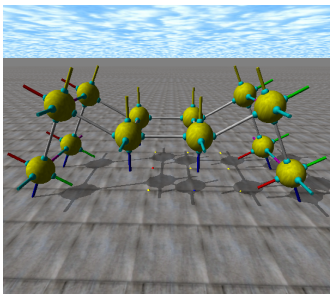
The MAAM [1] project, supported by the Robea project of the French CNRS, aims at specifying, designing and realizing a set of elementary robots able to connect each other to build reconfigurable systems. The self-reconfigurable robot is based on a basic component called atom. An atom is defined as a body with 6 orthogonal legs (see Figure 1). Each leg moves in a cone around his axis and the extremity of a leg holds a connector allowing connection between atom legs.

The goal of MAAM is to leverage this kind of atom to build dynamic reconfigurable structures called molecules. An advantage of such reconfigurable structures is that atoms move autonomously and they find and move to other atoms to connect with them by using their sensors on the end of each leg. In a 2D plan, they build an active carpet by array connection. A snake robot can be assembled to move into encumbered world. With a 3D structure, the molecules climb on objects, transform itself into a tool, surround objects and

[1] MAAM is a recursive acronym that means : Molecule = Atom | Atom + Molecule



(a) An atom



(b) A spider

Fig. 1. MAAM Project atoms and molecules

manipulate them. This project leveraged the cooperation of researchers from different fields : robotics, electronics, computer science, mechanics... Several french universities are involved in this project: Université de Paris VI (LIP6), Université de Caen (GREYC), Université de Bretagne Sud (LESTER and VALORIA), Laboratoire de Robotique de Paris.

1.3 Self-Reconfiguration Algorithms

A self-reconfigurable robot is ideally composed of cheap elementary units. The cheapness constraint makes an industrial manufacture possible and has a direct consequence on self-reconfiguration algorithms design. These algorithms must be the simplest one since the power calculus of the units is very weak. Therefore we think that reconfiguration algorithms requiring the description of the target shape are not relevant in this perspective.

So we are especially interested in distributed reconfiguration algorithm not requiring the exact description of the target shape. Thus Bojinov et al. (2000) [11], [12] proposed biologically inspired control algorithms for chain robots, using growth, seeds and scents concepts to make the target shape emerge from local rules. Another approach designed by Butler et al. (2003) [13] [14] is based on architecture-independent locomotion algorithms for lattice robots, inspired by the cellular automata model. Abrams and Ghrist (2003) [15] considered geometrical properties on a shape configuration space adapted to parallelization.

Actually we are looking for a general approach to tackle self-reconfiguration for different kind of modular robots. Modular robots are modules networks, and networks are usually modeled by graphs, ideal to stress the relation between entities. We can for example express the modules connectivity by bounding the degrees of a graph, unidirectional and bidirectional connections by directed or undirected edges... However, the reconfiguration of a modular robot implies

the evolution of the modules network topology, and thus of its underlying graph.

Modeling the evolution of a network topology is quite hard to capture in graph theoretical model, which is essentially static. The fundamental work on random graphs, by Erdös and Rényi [16] was the very first attempt to add dynamicity in a graph. Recently, interest has grown among graph theoretician in dynamic graphs, and especially in dynamic algorithms able to incrementally update a solution on a graph while the graph changes. To represent a dynamic graph, Harary (1997) [17] proposed dynamic graph models based on logic programming and the study of the sequence of static graphs. Ferreira (2002) [18] proposed a model called "evolving graph", whose definition is based upon an ordered sequence of subgraphs of a given digraph. But this given digraph induces an *a priori* knowledge on the dynamic process.

In the following sections, we will try to answer two main questions : how to model the modules network of a self-reconfigurable robot and how to design self-reconfiguration algorithms so that a self-reconfigurable robot converges to the required shape ?

2 Modeling Self-Reconfigurable Robots with Dynamic Graphs

2.1 Emergent Calculus

We are looking here for a way to control the convergence of dynamic graphs - modeling modules network - to a target topology by emergent calculus, that is to say the modules do not know the goal configuration and the final configuration emerges from the modules collective behavior.

We would like too to found the notion of dynamic graph in the area of dynamical systems. A dynamical system is characterized by a configuration space and a function defined on this space: the goal is to understand the dynamical behavior of this function according to the structure of the configuration space and to the property of the function. Dynamical system theory is a rich and well developed field and we hope to benefit of research and results of this field by defining a dynamic graph as a special dynamical system. Moreover, extending the modeling power of graph theory on dynamic network should be interesting since dynamic networks seem to appear fundamental in very different fields as it was recently stressed by Albert and Barabasi [19].

In addition, to take into account the distributed nature of a lot of observed phenomena, we would like the graph topology to evolve in a decentralized way

with local and simple rules : each node has a local knowledge on its neighborhood and a limited power for computation and communication. Cellular automata are well known for their ability to express complex dynamics from the local knowledge of the cells. The underlying lattice of a cellular automata is usually static, but Ilachinski and Halpern (1997) [20] developed a cellular automata model in which the underlying infinite d-dimensional array (*i.e.* the metric space $\mathbb{Z}^d$) evolves according to link transition rules. But the very formulation of this model, expressed through the array data structure, is not relevant to express a graph topology. Let us note also that link transition rules depend on the states of cells neighborhood and we are looking for purely "topological" rules.

To combine graph theory expressness with richness of cellular automata dynamic, we define the notion of topodynamic of a graph, with the assumption that a module only knows about its neighborhood and the neighbors of its neighborhood.

The last part is devoted to the validation of the framework proposed in the following section, by building a dynamic graph implementation in Squeak [21]: we present the overall architecture behind the DynaGraph software and show an example of simulation where the final topology emerges from the dynamic nodes collective behavior.

By this way, we hope to transform the shape controlling problem into the study of graph topodynamic, where simulation should be a valuable tool in discovering of topodynamics converging towards a given topology.

2.2 Topodynamic of a Graph

We first remind basic notions in graph theory and then state a topological definition of a graph independent of its embedment in metric space. We finally define the notion of graph topodynamic.

2.2.1 Preliminaries

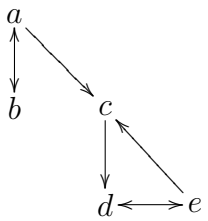
Definition 2.1 (Graph) *A graph is a pair (V, E) with V a finite set of vertices and E a set of edges, finite subset of $V \times V$.*

A graph is undirected if the relation defined on V is symmetric, otherwise the graph is directed, and edges have a direction. The *order* of a graph is its number of vertices $|V|$. Two vertices are said to be *adjacent* if they are joined by an edge.

The *neighborhood* of a vertex v is the set $\tau(v)$ of vertices such that they are adjacent to v. If there is no ambiguity with the context, we note a neighborhood $\tau(v) = \{x, y, \ldots, z\}$ by $xy \ldots z$. The *out-neighborhood* of a vertex v is the set $\tau^+(v)$ of vertices outgoing from v. The *in-neighborhood* of a vertex v is the set $\tau^-(v)$ of vertices pointing to v. We see that an undirected graph (resp. directed graph) is completely described by giving the set of its vertices and a neighborhood (resp. in-neighborhood *or* out-neighborhood) on each vertex. The *degree* (resp. *indegree*, *outdegree*) of a vertex is the number of its neighbors (resp. in-neighbors, out-neighbors). The degree of a graph, noted $d^\circ(G)$ is the maximum degree of a vertex. Note that for an undirected graph, each edge v, w could be considered as a double arrow (v, w) and (w, v), so the in-neighborhood is equal to the out-neighborhood of a vertex.

Definition 2.2 (Graph Topology) *The topology τ (resp. τ^+, τ^-) of a graph G is the family of its neighborhoods $(\tau_v)_{v\in V}$ (resp. out-neighborhoods $(\tau_v^+)_{v\in V}$, in-neighborhoods $(\tau_v^-)_{v\in V}$).*

Example 2.3 *Let G be the following graph :*



The neighborhood $\tau(a)$ of a is $\{b, c\}$ or in short bc. We have also the following relations: $\tau^+(a) = bc$, $\tau^-(a) = b$, $\tau(c) = ade$, $\tau^+(c) = d$, $\tau^-(c) = ae$, $d^\circ(G) = 3$ because $|\tau(c)| = 3$. The graph topology is the family $(\tau^+(a), \tau^+(b), \tau^+(c), \tau^+(d), \tau^+(e))$

2.2.2 Topodynamic

Let us now define the notion of a sequence of graphs.

Definition 2.4 (Sequence of Graphs) *A sequence of graphs is a family of graph $(\mathcal{G}_i)_{i\in\mathbb{N}}$ with $\mathcal{G}_i = (V_i, \tau_i)$.*

For simplicity, we consider from now only sequence of graphs with constant order, *i.e.* $\forall i \in \mathbb{N}$, $V_i = V_0$.

What is the difference between a sequence of graphs and a dynamic graph ? Usually, a dynamic system is characterized by its transition function : we can compute the state of the system from an initial state and past states.

Basically, the graphs in a sequence of graphs cannot change their topology on their own : the evolution of the topology is predetermined by giving a family τ_i of topologies. However, we can associate a transformation function to a sequence of graphs. So we call dynamic graph a sequence of graphs consisting of an initial graph and a function which transforms its topology to a new topology.

Definition 2.5 (dynamic graph - global transition function) *A dynamic graph is the pair* (G_0, Δ)*, such that* $G_0 = (V, \tau_0)$ *is an initial graph and* $\Delta : (V \mapsto 2^V) \mapsto (V \mapsto 2^V)$ *define a* topodynamic *by mapping a topology on* V *to a new topology.*

So the topodynamic is an algorithm taking as input a graph and giving as output a graph of the same order. More precisely, this algorithm takes a node and its neighborhood of the input graph and computes a new neighborhood for this node. It continues to compute over all the nodes of the input graph to render the output graph. Then this output graph becomes the input graph and we apply the topodynamic again. This whole process draws the dynamic graph.

Nevertheless, we would like to have dynamic graph vertices more active than in a sequence of graphs, namely able to change their own degrees by accepting, keeping or removing its adjacent edges, according to local transition rules inducing the graph topodynamic. Local transition rules are widely used in cellular automata area: the state of an automaton depends on its own state and the state of its neighbors. Local transition function, simultaneously applied to each cell, determine the dynamic of a cellular automaton. Although cellular automata are usually defined on regular lattices, this definition can be extended to more complicated graph: graph of automata (connected bounded degree graph), first introduced by Rosenstiehl (1966) [22]. In a graph of automata, each node has a state and the next state depends of its current state and the state of its neighbors.

Definition 2.6 (graph of automata) *A graph of automata is a triplet* (S, G, δ) *where* S *is a finite set called set of states,* $G = (V, \tau)$ *is a graph,* $\delta : S \times \{(S \cup \epsilon)^{d^\circ(G)} / \sigma\} \mapsto S$ *is the transition function where* ϵ *is a special element used when the vertex has less than the maximum degree of the graph.* σ *is the equivalence relation defined on the cartesian product* S^n *with* $x \sigma y$ *if* x *is a permutation of* y*. So* S^n / σ *is the unordered set* S^n*.*

In this definition of graph automata , the underlying graph is static. We study here the possibility to have an evolving underlying graph : this evolution may be controlled by active vertices, kind of automata able to connect and disconnect their own edges in the network. We give to the automata the control of its underlying graph by slightly modifying the graph automata definition as

following.

Definition 2.7 (dynamic graph - local transition function) *A dynamic graph is the pair* (G_0, δ) *where* $G = (V, \tau_0)$ *define an initial graph, and* $\delta : S \times \{(S \cup \epsilon)^{|V|-1} / \sigma\} \mapsto S$ *with* $S = 2^V$ *the set of states, define the local transition function, where* ϵ *is a special element used when the vertex has less than* $|V| - 1$ *(the maximal degree of the dynamic graph).*

A node chooses its next neighborhood according to its current neighborhood and the current neighborhood of its neighbors. If we replace "neighborhood" in the precedent sentence by the word "state", we retrieve the usual definition of the evolution of a cell in cellular automata.

To deal with the different neighbors of a given neighborhood, we build from the application τ an application $\vec{\tau}$ which for each vertex gives its *neighbors vector* : $\vec{\tau} : \{v\} \mapsto (\{1, \ldots, |\tau(v)|\} \mapsto V)$ such that $\bigcup_{i=1}^{|\tau(v)|} \vec{\tau}(v)(i) = \tau(v)$ where $v \in V$

If a vertex v has a degree n, its next state $\tau_{i+1}(v)$, namely its next neighborhood, is given by :

$$\tau_{i+1}(v) = \delta(\tau_i(v), \tau_i(\vec{\tau_i}(v)(1)), \ldots, \tau_i(\vec{\tau}(v)(n)), \epsilon, \ldots, \epsilon)$$

We define now the fix-point topology for a given topodynamic as a graph topology unchanged by applying this topodynamic.

Definition 2.8 (fix-point topology) *A fix-point topology* τ *for the topodynamic* Δ *is a topology such that* $\Delta(\tau) = \tau$

A question we may ask is for which topodynamic a given topology is the fix-point, *i.e.* the so-called *inverse dynamic problem*. For the moment the design of such topodynamic rules is rather a matter of intuition supported by computer experimentations, but resolving the *inverse dynamic problem* would provide us with a means of constructing a desired topology, or at least to show us the impossibility to automate this construction. Anyway the simulation of dynamic graphs should be valuable to address this problem.

3 DynaGraph : an Environment for Dynamic Graph Simulation

We present here the DynaGraph environment intended to understand and simulate the dynamic of a graph, following the theoretical framework stated in the first section. Through this environment, we hope a better understanding of the dynamic graphs behavior. For instance, we could ask what kind of dynamic graph converges to a path graph when the initial graph is a star

graph, so we have to define the relevant rules allowing such a convergence. We also need to implement this rules to gain trust on this rules before proving their correctness.

3.1 The DynaGraph Environment

Let us imagine that we want to reconfigure a spider robot, represented to the left of Figure 2, to a caterpillar robot. Each node represents a module, and directed edges the connection between modules.

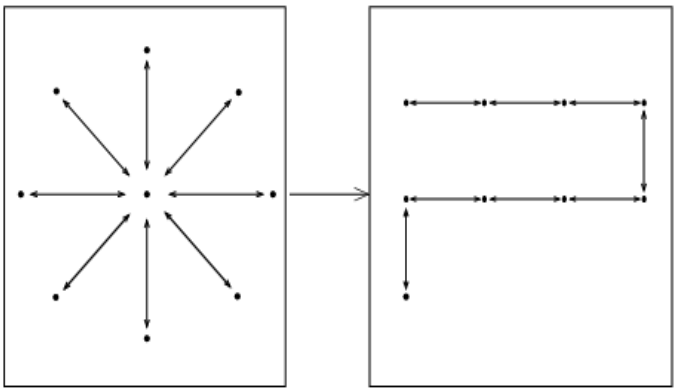


Fig. 2. A star to chain reconfiguration

The expression `DynaGraphSystemWindow open` launches the DynaGraph GUI (Fig. 3).

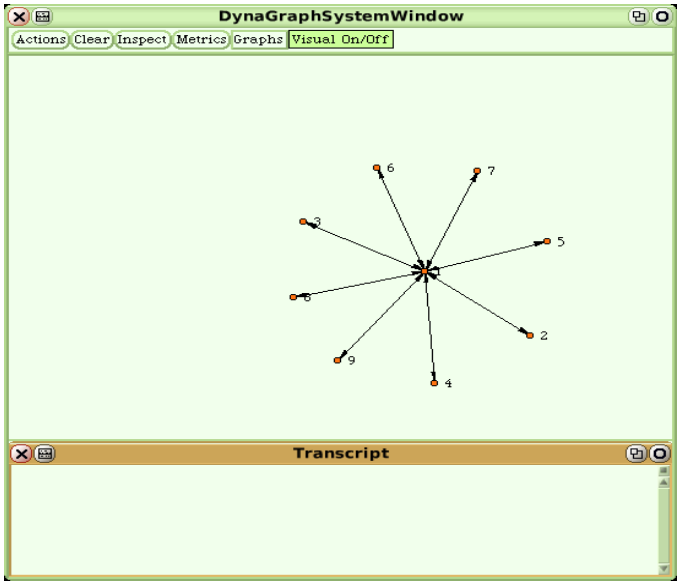


Fig. 3. The DynaGraph System Window

The DynaGraph window (Fig. 3) is currently composed of two parts : the first one is the gui interface and the second one is for debugging purpose.

Several buttons allow the user to start, stop or run the simulation step by step. The user chooses the initial graph of the dynamic graph thanks to the `Graphs` button. Several metrics *i.e.* function of the graph at the current step are available, like degree distribution, clustering coefficient and average path length, to gain some information during the evolution of the dynamic graph. We use the `PlotMorph` package [23] to display such kind of informations (see Figure 4).

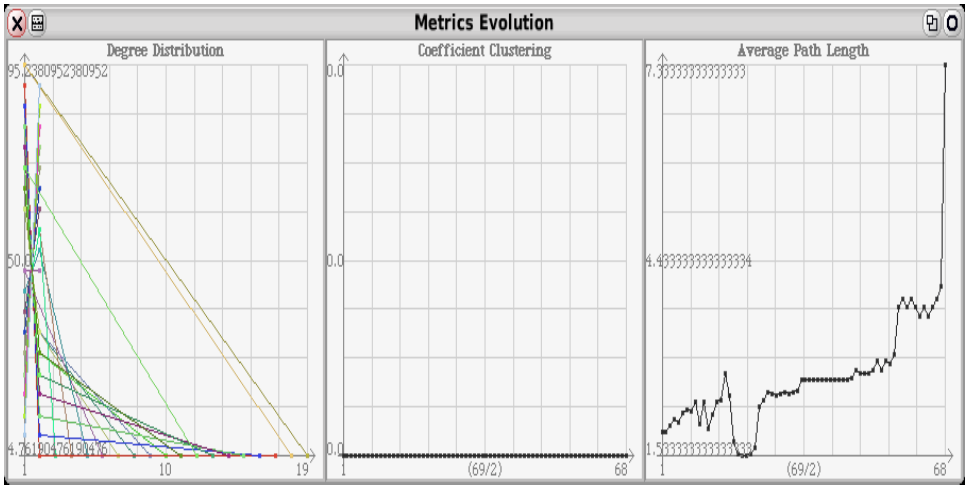


Fig. 4. Dynagraph metrics

We have enhanced this package to export the curves in the gnuplot format to exploit thouroughly the collected data. Note that we have also implemented a GraphSystemWindow to deal with graphs, *e.g.* random graphs, so the metrics are available for this graphs. For instance, Figure 5 shows the degree distribution of a single random graph of 10000 nodes and a connection probability of 0.0015, generated through the `exampleRandomGraphWithOrder:probability:` method. We verify that the deviation is small between the generated random graph and the theoretical result stating that for large N, the probability for a node to have a degree k follows roughly a Poisson distribution $P(k) \simeq e^{-pN} \frac{(pN)^k}{k!}$.

The simulation shows in Figure 6 that after few steps the initial star topology is transformed into an undirected acyclic graph to finally reach a fix-point. The local topodynamic rules, when applied to the neighborhood of each node, does not change their neighborhood anymore. We show thanks this simulation that it is possible to make a topology emerge from a local topodynamic rule.

3.2 Overview of the DynaGraph Architecture

The dynamic graph classes are build on top of graph classes, designed in Smalltalk by Mario Wolczko and ported for Squeak by Gerardo Richarte and

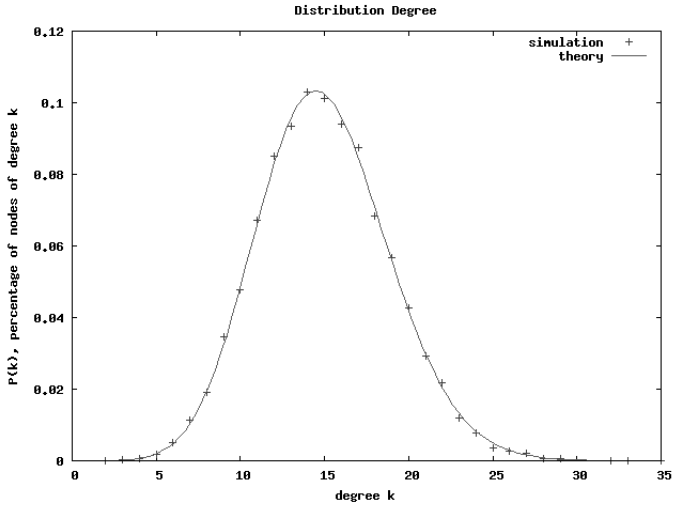


Fig. 5. The degree distribution resulting from the simulation of a random graph ($N = 10,000$ nodes, connection probability $p = 0.0015$).

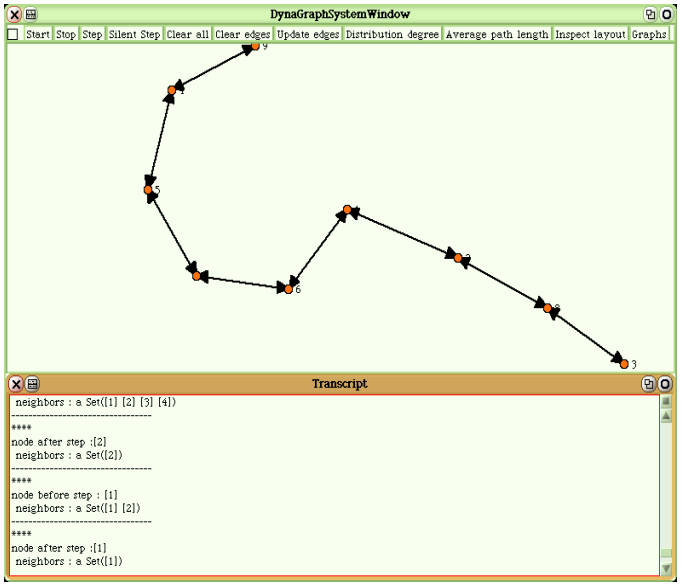


Fig. 6. The chain, fixpoint topology

Luciano Notarfrancesco. A GraphMorph class is available, allowing us to layout a graph through two main methods : a springs-gravity model where each node behave like negative electrical particles and an animated radial layout. We use the springs-gravity model to display a dynamic graph.

Several principles are used behind a local topodynamic design. These principles are not really specific to one particular transformation, but they are rather a guide for designing a topodynamic :

- Reconfiguration : the reconfiguration is done thanks to the knowledge of a neighborhood node and of the neighborhood of the node neighbors.

It can disconnect from its current nodes to reconnect itself to a neighbor of its neighbors.

- Local knowledge : A node knows only its own indegree and outdegree (computed from its knowledge of its in and out-neighbors) and the in and outdegrees of its direct neighbors (computed from its knowledge of in and out-neighbors of its neighbors).
- Outgoing connection control : A node only controls its outgoing connections, it cannot decide to disconnect itself from an ingoing connection but can connect to or disconnect from its outgoing connections.
- Decision process : To take a decision, for instance a reconnection to another node or a disconnection, a node may exploit the dissymmetry of its neighborhood. Indeed if a node wants to reconnect itself to a neighbor of one of its neighbors, it has to check its neighbors degree and take a decision according to the neighbors degree. If all neighbors have the same degree, the reconnection to the neighborhood neighbors will be random : from the node point of view, there is no way to decide which neighbor of its neighbors to choose since all its neighbors have the same degree. Otherwise it will exploit the dissymmetry related to the difference of degree. For that reason, we have to avoid the ring topology because of the possibility to lose graph connectivity.
- Connectivity : A node must never be isolated during the reconfiguration process.
- Uniformity : All nodes have the same set of rules.
- Synchronicity : The topodynamic rules, *i.e.* a combination of disconnection and connection rules, are applied *simultaneously* over all the nodes. The computation must hold synchronous nodes reconfiguration although it could be interesting to study how asynchronicity influences the reconfiguration process. But we choose to study first a synchronous dynamic since it is the simplest one : we have not to choose which nodes the rules must apply first, all nodes are equivalent according to the application of rules.

These principles constraint the implementation of a dynamic graph : for instance, synchronicity means that we have to take care about the way each node changes its neighborhood. Indeed if a node changes its neighborhood, one of the neighbor of this node must not see immediately this change... The global architecture of a dynamic graph, or dynagraph, is pictured in Figure 7.

Moreover, to control the evolution of a dynamic graph, we have added the `MetaGraph` class, which is composed of the list of graphs computed during the evolution of a dynamic graph. This class could be seen as an equivalent of the space-time diagram used in cellular automata simulation and allows us to keep trace of the different states of a dynamic graph. The more important and critical method of this class is the method `oneStep` which is implemented as following :

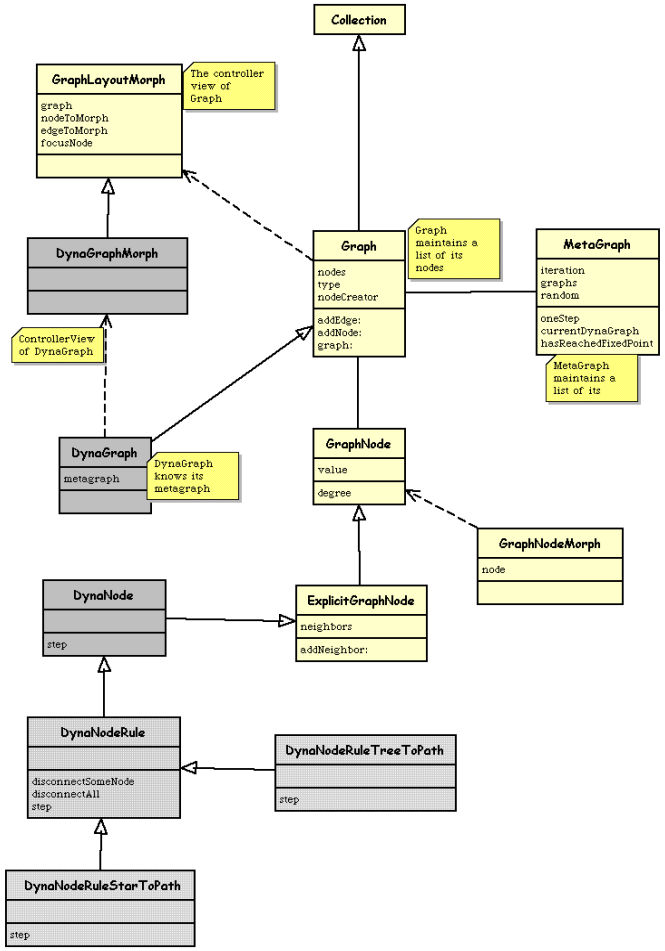
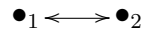


Fig. 7. The DynaGraph Architecture

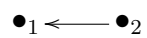
```
| nextDynaGraph |

nextDynaGraph := self currentDynaGraph veryDeepCopy.
"we enter to the next iteration, all nodes will refer
to the precedent iteration to compute their neighborhood"
iteration := iteration + 1.
(nextDynaGraph nodes asArray shuffledBy: random)
        do: [:node | node step].
self addGraph: nextDynaGraph.
^ nextDynaGraph
```

Let us take an example to understand how to get synchronous processes. Suppose that there is only one rule applied to all nodes : a node disconnects from the other one if and only if its indegree is 1, that is to say if there is a node connected to itself.



The node labelled 1 is in this case, so it will disconnect from the node 2 :



If all this process was sequential, at the next step, the node 2 will keep its connection since its indegree is now O, but we would like to focus our study on parallel updates. So this behavior is not the desired one, the right result for parallel update is the following:

So we will copy the current graph to a new graph, then we will modify the nodes neighbors of the new graph according to the neighborhood, and neighbors neighborhood of the previous graph. In this manner, we simulate the fact that each node changes their neighborhood simultaneously.

3.3 Overview of the Implementation

The `DynaNode` class corresponds to the notion of dynamic node as explained in the Topodynamic section and implements the abstract method `step`. A dynamic node is essentially a set of rules defining the dynamic aspect of a node, so it makes sense to derive the `DynaNodeRule` class from the `DynaNode` class. Then we can use this facility to give a name to a specific `DynaNode` like `DynaNodeRuleStarToPath` and so on. The `DynaNodeRule` class implements the main methods allowing the evolution of the neighborhood of a given graph. Here is a sample of main methods involved in the graph reconfiguration:

- The method `disconnectNodeOfOutDegree: anInteger` tries to disconnect the current node from a neighbor of outdegree *anInteger* and return true. If it fails, return false.
- The method `connectToNgbOfMyNgb` connect the current node to any neighbor of its neighbor.
- The method `connectToNgbOfNgb:` connect the current node to any neighbor of its neighbor v.
- The method `connectToNodeOfInDegree: anInteger` connect the current node to a node of indegree *anInteger*.
- The method `connectToOneOfMyStrictInNgb` connect the current node to one of its strict in-neighbors, namely in-neighbors which are not out-neighbors.

- The method `connectToOutNgbOfNgb:` connect the current node to somekindof out-neighbor of its neighbor v.
- The method `connectToNode:` connect the current node to the node w.
- The method `disconnectToAnyOutNode` disconnect the current node from one out-neighbor.

From the `DynaNodeRule` class, we derive a class which implements the method `step` expressing the local topodynamic as explained in the first section, *i.e.* a topodynamic depending of the neighborhood and of the neighbors neighborhood. For instance, the method `step` of the `DynaNodeRuleStarToPath` class implements four local reconfiguration rules [24] :

```
DynaNodeRuleStarToPath>>step

        self extremeNodeClosure.
        self extremeNodeReconfiguration.
        self starLosingLeaves.
        self middleNodeReconfiguration.
        ^ self.
```

The self-reconfiguration algorithm is based on three main rules applied one time for each step :

- RECONFIGURATION: Threads want to find an extremity. (`extremeNodeReconfiguration` and `middleNodeReconfiguration`)
- NODE CLOSURE: Closing all links when a path is formed. (`extremeNodeClosure`)
- STAR LOSING LEAVES: Disconnect from some of many out-neighbors. (`starLosingLeaves`)

It would be very long to explain and develop the code of all this rules, which are currently available in the SqueakSource site under the name DynaGraph. Here is a piece of code giving an idea of how the rules are generally coded :

```
DynaNodeRuleStarToPath>>starLosingLeaves

        self currentOutDegree >= 3
                ifTrue: [self disconnectAnyOutNode]

DynaNodeRuleStarToPath>>middleNodeReconfiguration

 "principle : a middle-initial vertex walks around the graph
  till it finds an extremity"
 self isMiddleInitial
```

```
ifTrue:
[self currentUnclosedOutNeighbors loneElement isAnExtremity not
 & self currentUnclosedOutNeighbors loneElement isMiddle not
 ifTrue: [
  self reconnectToOutNeighborhoodOfNode:
   self unclosedOutNeighbors loneElement]]
```

Let us notice an important point when we design rules : the difference between the methods inNeighbors, outNeighbors... and the methods prefixed with the term "current", like currentInNeighbors, currentOutNeighbors... The first ones are the accessors of the nextDynaGraph (see the method oneStep above) whereas the second ones refer to the DynaGraph of the current iteration. This distinction is necessary to keep the dynamic synchronous, as it was explained above. So the conditional part of a rule must refer to the DynaGraph of the current iteration whereas the action part must refer to the DynaGraph under construction, that is to say the nextDynaGraph. So we have always to care about which kind of DynaGraph is concerned : the one of current iteration or the one in progress.

3.4 Building its own dynamic graph

To create a dynamic graph with a specific rule, we have implemented the class method `dynamicDirectedWith:` which takes as argument a DynaNodeRule class:

```
DynaGraph dynamicDirectedWith: DynaNodeRuleStarToPath.
```

The usual way to add a new dynamic graph in the DynaGraph environment is to add a class method to the class `DynaGraph` :

```
exampleMyDynaGraph
        | d |
        d := self dynamicDirectedWith: DynaNodeMyOwnRule.
        d addEdge: 1 -> 2.
        d addEdge: 2 -> 4.
        d addEdge: 4 -> 2.
        d addEdge: 2 -> 3.
        d addEdge: 3 -> 2.
        ^ d
```

Of course, we have to first create the `DynaNodeMyOwnRule` class by deriving it from the `DynaNodeRule` class and using the methods defined in this class, or creating our own methods. The when we start the GUI, the `exampleMyDynaGraph`

appears when we press the `Graphs` button in a pop-up menu :

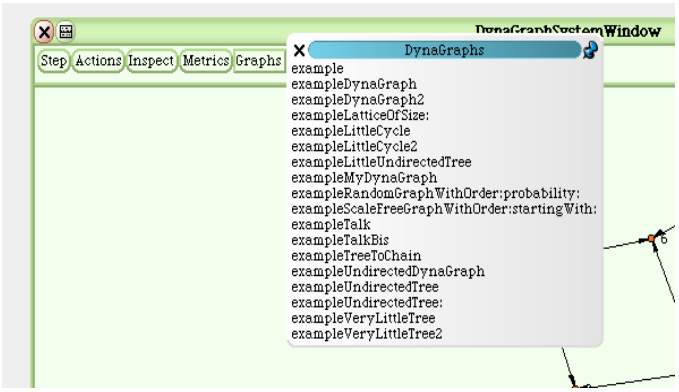


Fig. 8. Running a dynamic graph

If we want to keep trace of the dynamic graph evolution, we can use the `MetaGraph` class as following: `MetaGraph new setInitialDynaGraphTo: (DynaGraph dynamicDirectedWith: DynaNodeRuleStarToPath)`

4 Conclusion

This work presents a framework based on graph topodynamic and cellular automata intended to address the problem of controlling modules network topodynamic.

After introducing the field of self-reconfigurable robots, we have defined the notion of dynamic graph by proposing to make a distinction between sequence of graphs and dynamic graphs: a dynamic graph is a sequence of graphs where a local or global topodynamic determines the topological evolution of a graph.

Then we have described the smalltalk implementation of this framework : since a topodynamic can be defined from the local knowledge of each vertex, we can use an oriented-object language, and Squeak helps us to prototype and validate quickly and easily our framework.

The most interesting feature offered by Squeak is the debugging facilities which are crucial for developing self-reconfigurable algorithms. For example, we have implemented the inspection of a node during the simulation by shift-clicking on a node in the DynaGraphWindow. This is a great advantage compared since it is possible to modify topodynamic rules during the simulation of a dynamic graph. This possibility allows us to find out several reconfiguration algorithms in few weeks, including the time spent to develop the gui. For a research project, it is really valuable since we would like to test quickly some concepts without spending a lot of time to implement those ideas, and Squeak

plays the perfect role of "notebook for concepts implementation". But we feel at the present state of this work one limitation : it is hard to simulate dynamic graphs with a large number of nodes. The current limitation is around 30 nodes which is sufficient to test our ideas but not sufficient to gain some statistical information about the dynamic of a graph. Maybe have we now to implement some parts in C which fortunately is one of a feature of Squeak, although we would prefer to stay in the Squeak environment.

This simulator should help us in rules discovery involved in the emergence of different network topologies : from a connected graph to a chain graph, from a lattice to a chain, from a chain to a lattice, and so on. This software [25] is available as a package in the Squeakmap site, a server providing applications designed for Squeak, and is available too in the SqueakSource site.

Let us note that the MAAM project is in fact a long-term project composed of several subprojects like sQode, a plugin to interface Squeak with ODE (Open Dynamic Engine), SqueakSimulAtom, a 3D robots simulator, LCSTalk, a Learning Classifier System framework... We expect to have the first molecule with 10 elementary components in 3 years.

Acknowledgment

This work was supported by an MENRT Research Studentship, and is a part of the MAAM Project. We thank Serge Stinckwich for its valuable comments and to allow us to participate in the MAAM project, with the support of François Bourdon. We wish to thank also Stéphane Ducasse and the anonymous reviewers whose comments permit us to really improve this paper.

References

[1] C. J. J. Paredis, P. K. Khosla, Fault tolerant task execution through global trajectory planning, Reliability Engineering and System Safety (special issue on Safety of Robotic Systems) 53 (3) (1996) 225–235.

[2] H. Lipson, J. B. Pollack, Automatic design and manufacture of robotic lifeforms, Nature 406 (2000) 974–978.

[3] Swarmbots.
URL `http://www.swarmbots.org`

[4] K. Kotay, D. Rus, M. Vona, C. McGray, The self-reconfiguring molecule: Design and control algorithms, in: Workshop on Algorithmic Foundations of Robotics, 1999.

[5] W. Shen, P. Will, Docking in self-reconfigurable robots, in: Proceedings IEEE/RSJ, IROS conference, Maui, Hawaii, USA, 2001, pp. 1049–1054.

[6] C. Unsal, P. Khosla, A multi-layered planner for self-reconfiguration of a uniform group of i-cube modules, in: IROS 2001, 2001.

[7] R.Fitch, D. Rus, M.Vona, A basis for self-repair using crystalline modules, in: Proceedings of Intelligent Autonomous Systems, 2000.

[8] S. B. H. John W. Suh, M. Yim, Telecubes: Mechanical design of a module for self-reconfigurable robotics, in: Proceedings, IEEE Int. Conf. on Robotics and Automation (ICRA'02), Washington, DC, USA, 2002, pp. 4095–4101.

[9] M. Yim, D. Duff, K. Roufas, Polybot: a modular reconfigurable robot, in: Proceedings, IEEE Int. Conf. on Robotics & Automation (ICRA'00), Vol. 2, San Francisco, California, USA, 2000, pp. 1734 –1741.

[10] A. Kaminura, al, Self reconfigurable modular robot, in: Proceedings IEEE/RSJ, IROS conference, Maui, Hawaii, USA, 2001, pp. 606–612.

[11] H. Bojinov, A. Casal, T. Hogg, Multiagent control of self-reconfigurable robots Comment: 15 pages, 10 color figures, including low-resolution photos of prototype hardware.
URL `http://arXiv.org/abs/cs/0006030`

[12] H. Bojinov, A. Casal, T. Hogg, Emergent structures in modular self-reconfigurable robots, in: Proceedings, IEEE Int. Conf. on Robotics & Automation (ICRA'00), Vol. 2, San Francisco, California, USA, 2000, pp. 1734 –1741.

[13] Z. Butler, D. Rus, Distributed planning and control for modular robots with unit-compressible modules, International Journal of Robotics Research 22 (9) (2003) 699–716.

[14] Z. Butler, K. Kotay, D. Rus, K. Tomita, Generic decentralized control for a class of self-reconfigurable robots, in: Proceedings, IEEE Int. Conf. on Robotics and Automation (ICRA'02), Washington, DC, USA, 2002, pp. 809–815.

[15] A. Abrams, R. Ghrist, State complexes for metamorphic robots, International Journal of Robotics Research In press.

[16] P. Erdős, A. Rényi, On the evolution of random graphs, Publ. Math. Inst. Hung. Acad. Sci. 5 (1960) 17–61, a seminal paper on random graphs. Reprinted in *Paul Erdős: The Art of Counting. Selected Writings*, J.H. Spencer, Ed., Vol. 5 of the series *Mathematicians of Our Time*, MIT Press, 1973, pp. 574–617.

[17] F. Harary, G. Gupta, Dynamic graph models, Math. Comput. Modelling 25 (7) (1997) 79–87.

[18] A. Ferreira, On models and algorithms for dynamic communication networks: The case for evolving graphs, in: 4^e rencontres francophones sur les Aspects Algorithmiques des Télécommunications (ALGOTEL'2002), Mèze, France, 2002.

[19] R. Albert, A.-L. Barabàsi, Statistical mechanics of complex networks, Rev. Mod. Phys. 74 (2002) 47–97.

[20] Ilachinski, Halpern, Structurally dynamic cellular automata, COMPSYSTS: Complex Systems 1.

[21] Squeak, an open-source smalltalk.
URL `http://www.squeak.org`

[22] P. Rosenstiehl, Existence d'automates finis capables de s'accorder bien qu'arbitrairement connectés et nombreux, International Computer Science Bulletin 5 (1966) 245–261.

[23] Plotmorph, morphs to draw xy plots.
URL `http://minnow.cc.gatech.edu/squeak/DiegoGomezDeck`

[24] S. Saidani, Self-reconfigurable robots topodynamic, in: Proceedings, IEEE Int. Conf. on Robotics & Automation (ICRA'04), New Orleans, Louisiana, USA, 2004, pp. 2883–2887.

[25] Dynagraph, a dynamic graph simulator.
URL `http://www.squeaksource.com/DynaGraph`

An Aspect-based Multi-Agent System

Romain Robbes

Université de Caen - GREYC - CNRS - Romain.Robbes@info.unicaen.fr

Noury Bouraqadi

École des Mines de Douai - Département GIP - bouraqadi@ensm-douai.fr

Serge Stinckwich

Université de Caen - GREYC - CNRS - Serge.Stinckwich@info.unicaen.fr

Abstract

We present how we used Aspect-Oriented Programming (AOP) to ease the development of Multi-Agent Systems (MAS). Our study focuses on the Aalaadin MAS model. We make use of AOP at both the conceptual level of Aalaadin and the implementation level. On the conceptual level, we introduced in Aalaadin AOP concepts. It results in unifying the *group* concept of Aalaadin with the *aspect* concept. This unification relies on reflection to allow the definition of groups with *intrusive* processing. On the implementation level, we used AOP to ease the implementation of a MAS infrastructure. It's worth noting that the aspects we identified are not specific to the Aalaadin model. Indeed, aspects such as *agent message building*, *messaging strategy* or *agent lookup* are applicable in a variety of MAS.

Key words: Aspect Oriented Programming, Multi-Agent Systems, Reflection, Organizational Model, Group, Role.

1 Introduction

Although Object-Oriented Programming (OOP) has been a major progress in software engineering, it has some limitations that led in 1997 to the introduction of Aspect-Oriented Programming (AOP) [1]. OOP suffers particularly from code *cross-cutting* and *tangling*. Indeed, modern software does not only include functional code, but also code describing non-functional properties, such as persistency, remote communications or database management. Each of these properties cross-cuts classes describing entities of the application domain. Moreover, code tangling arises since implementation of such transverse properties are melted in *the base code* (*i.e.,* application "core" code).

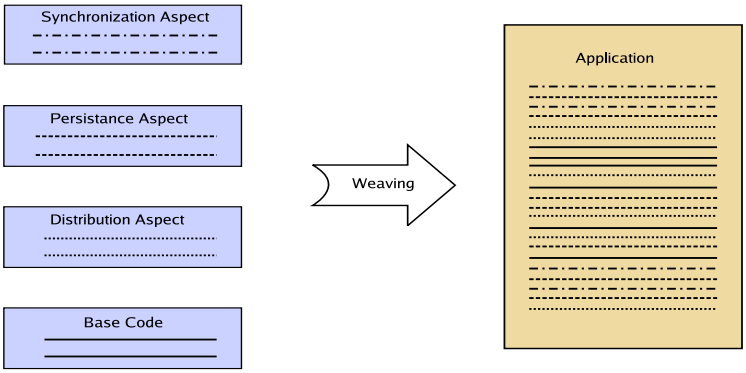


Figure 1. Building an application using AOP

AOP deals with the above mentionned limitations by introducing a new dimension to structure applications [2][3]. It strengthens modularity by allowing developers to isolate the definitions of transverse properties. As shown in Figure 1, each property is defined totally and exclusively in a single module, called *aspect*. Development is therefore simplified, and becomes less error-prone, as developers can focus on one concern at a time. Once the various aspects have been defined, they are assembled with base code to build the application. This integration process called *weaving* is essentially automatic. It consists of linking the aspects with the base code in particular points, called *join points*. These points are either in the base code structure (definitions, ...), or control flow (message sends, field read/write, ...).

In the field of Multi-Agent Systems (MAS), separation of concerns as promoted by AOP is not clearly supported. This is the case of the numerous existing platforms, such as MadKit, Brainstorm/J, JAF, Mobydic, ZEUS, ... [4][5][6][7]. No modularity is provided when it comes to defining non-functional properties even if they are bound to the MAS infrastructure (distribution, security, real-time behaviour) or to the agent's intrinsic nature (autonomy, bounded rationality, organizational aspects). In this paper, we address this issue by exploring two areas where MAS can make use of AOP:

(1) Introducing AOP concepts in MAS models and designs. Indeed, AOP concepts can be used at the conceptual level of a MAS. This facet is the one we will mainly talk about in this article. To our knowledge, it hasn't been studied yet.
(2) Using AOP to implement a MAS infrastructure. A MAS being a complex application, with numerous interrelating, and even sometimes conflicting functionalities, one can easily imagine the advantages of using AOP at this level. So far, some works have been done on this facet [5], [8]. They have identified several generic aspects of MASs (*e.g.,* exception handling, redundancy, persistence) and also specific aspects of agents (*e.g.,* autonomy, collaboration, mobility, learning). But in these works, the agent concept is not first class, as it is build upon a domain object woven with agent aspects.

The paper is organized as follow. Section 2 sums up what has to be known about agents, particularly the Aalaadin [9] model, as our study is based on it. This will bring us in section 3 to discuss the common points between AOP and Aalaadin. Then, we exploit the result of this comparison, to suggest some extensions to the Aalaadin model meant to unify groups and aspects. Section 4 describes and hints at how to implement some infrastructure aspects needed for a MAS. Finally, future works in section 5, and conclusion in section 6 closes this article. A comprehensive example including code from our prototype is shown in an appendix.

2 Background: Agents and the Alaadin Model

2.1 Agents

As H. Van Dyke Parunak said [10], agents are entities able to say "go and no", *i.e.,* they can decide and act autonomously. An object deals with *how* to perform an action, whereas an *active object* deals with *how and when* to perform it. An agent, on the other hand reflects on *how, when and why* he should perform this action. Hence an agent has a notion of finality which lacks from objects, thus Castelfranchi argues [11] that the agent concept is denied if there is no intention notion.

An agent is an entity which commonly has the following properties:

- Autonomy: an agent decides on its own, taking into account its available ressources: computing time, energy, space ... There is a conflict between this property and the agent conception, since conception explicitely supposes that a function is given to the agent, whereas autonomy offers the agent the

possibility to evolve freely.
- Adaptativity: This property can be seen as a consequence of the previous property, as autonomy makes it necessary to have an open conception that agents can modify.
- Situationness: the agents are continuously interacting with their environment. They sense events happening in it with *sensors*, and can act upon the environment using *actuators*. This environment reported by the sensors can be uncertain and noisy, while the effects of the actuators cannot be guaranteed.
- Agency: This property stipulates that agents organize themselves in groups, thus forming societies, and communicate using messages.

There have been traditionally two epistemological approaches to design agents able to interact in a physically complex environment:

- The symbolic approach [12] adopts a top-down conception and is mainly interested in the inferencing capabilities of an agent. Logic and symbolic programming are then used to implement this behaviour.
- The behaviour-based approach [13] adopts a bottom-up one where simple, reactive behaviours are used. This approach uses non-symbolic technologies.

Even if they are still difficult to conceive, both approaches tend to be merged uniformly, in a *hybrid* approach.

A multi-agent system (MAS) is composed of a set of agents living in a logical or physical environment, interacting with each other and with the environment. These agents can be competiting, or more often cooperating, to solve a given task in the environment. MAS are commonly used in:

- distributed problem solving,
- simulation of ecosystems, chemical or physical phenomenon,
- control of complex systems,
- man-machine interaction, using interface agents.

2.2 Description of the Aalaadin Model

Aalaadin's authors describe it as a meta-model based on organizational concepts such as roles, groups and organizational structures [9]. Its primary goal is to reduce the intrinsic complexity of MAS systems (agent heterogeneity, interaction between different MAS, security ...). Aalaadin is a meta-model in the sense that it does not force the designer to use a specific agent or organizational model. It rather provides a uniform framework allowing the modeler to define his inter-agents coordination structures, such as hierarchic or market-like organizations.

Aalaadin's main concepts are:

- *Agents* are autonomous and communicating entities, taking *roles* in *groups*. No behavioural or implementation constraints are imposed to them.
- A *group* is a dynamic set of agents. Each agent is a member of one or several groups. New groups can be created by any agent, and an agent must request its admission in a group.
- A *role* is an abstract representation of the functionalities or services of an agent inside a group. Each agent can take the responsibility of several roles, each role being locally associated to a group. Similarly as the admission in a group, being in charge of a role in a group must be requested by the candidate, and may not be allowed. This role notion allows agents to follow several heterogenous dialogues simultaneously. A role is characterized by:
  its *cardinality*, which can be unique or multiple in a group, *i.e.,* only a single agent can wear the role or multiple agents can wear it,
  its *competences*, which are the prerequisites that the requesting agent must fulfill to bear this role in the group, and
  its *capacities*, which are extra competences acquired by the agent when he plays a given role.
- A *group structure* is the abstract description of a group, where the set of roles composing it are described, like the possible interactions between those roles. A group structure is then used to instantiate concrete *groups*.
- An *organizational structure* is built with a set of group structures

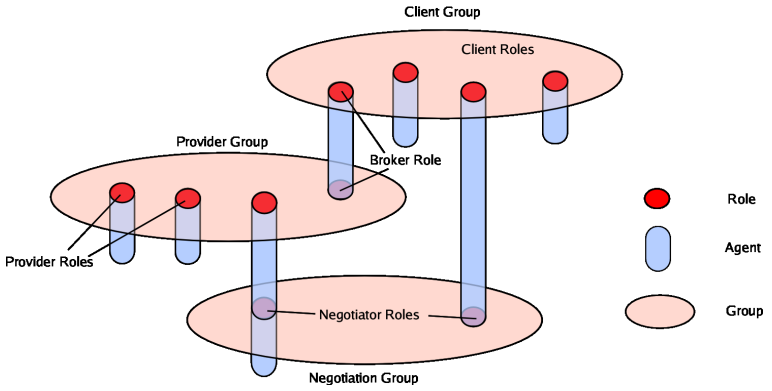


Figure 2. Sample application using the Aalaadin model: the market structure

Figure 2 shows a comprehensive example of the previous concepts. The provided sample application is a market organization structure that includes 3 groups. The provider group includes a provider role, worn by multiple agents, as well as the client role of the client group. Both the provider group and the client group have a broker role that should be worn by a single agent. The broker role consists of allowing clients needing some products to meet the right providers. Clients and providers that get matched this way join a negotiation group to discuss the price. Hence, they wear the negotiator role.

3 Aop at the Conceptual Level of the Aalaadin Model

3.1 Groups vs Aspects

The comparison between Aalaadin and AOP consists mainly in comparing groups and aspects. As we explain below, these two notions are indeed very close, especially as they both describe transverse properties. Because our goal is to import interesting AOP features in Aalaadin, we make the group/aspect comparison based on properties of the latter one. We thus list aspect characteristics and show whether Aalaadin's groups provide them or not.

3.1.1 Transversality

Groups and aspects share the property of being *transverse* to the application they are defined in. They both indeed describe facets of applications.

In object-oriented applications, aspects modify the control flow of computations performed by instances of *several* classes. For example, concurrent access synchronization is a facet of an e-commerce application. This facet, described in an aspect, is tranverse to several objects (clients, orders, products ...) since computations performed by these objects all need synchronization.

Groups are tranverse to agents in the same manner aspects are transverse to objects. A group indeed describes roles which must be worn by several agents in a MAS. Each role features some capacities (such as access rights) given to the agent wearing the role. Thus the group "modifies" the different agents who join it by augmenting their capacities. For example, a "negotiation" group is defined by the "buyer" and "seller" roles, which must be worn by two different agents. This group is therefore transverse to these two agents.

We can also note that groups have a richer semantic than aspects, as they allow one to define the cardinality of the roles which they are constituted of. In addition, they are more dynamic than most AOP implementations as agents can join or leave groups at anytime.

3.1.2 Weaving and Join Points

The process of weaving an aspect into a given application consists of linking the processings defined in the aspect with those of the application. The corresponding operation in Aalaadin is performed when an agent wears a role to join a group in a MAS. However, these two processes differ on a few points:

- First, computations defined in an aspect are intrusive to the rest of the application: an aspect can modify the control flow of an application and the processings performed by its objects. For example, an authentification aspect can forbid the execution of a message if the given password is wrong. On the other hand computations performed inside a group are much more local. They dont influence directly processings performed in the other groups.
- An immediate consequence of the previous point is the absence of an AOP join point equivalent in Aalaadin. Groups are indeed not directly affecting the rest of the application contrary to aspects. The orthogonality of groups prevents this type of intrusions.
- Another difference is that weaving an aspect can provoke conflicts whereas joining a group does not. Conflicts happen when two aspects need to be woven on the same join point. A log aspect can for example produce a trace when an object receives a message, while an authentification aspect might prevent messages to be executed. To solve this conflict, the integrator must decide whether to always produce a trace, or to restrain it to the messages whose anthentication was successful. Such situations cannot happen in Aalaadin, as groups are orthogonal to each other, so the integration step can be totally automatized.

3.1.3 Modularity and Reusability

Aspects in AOP strengthen modularity of applications, as each aspect is a module describing totally and exclusively a unique facet of an application. It is hence possible to develop separately (in time, or by several persons) each aspect, since there is no coupling between them. This modularity also allows new reusability perspectives: generic application-independent aspects can be spread and reused in different contexts.

Reusability and modularity seems not to have interested Aalaadin's designers. If groups and roles are well identified conceptually, this is not the case for the implementation. This is shown in Aalaadin's reference implementation, MadKit, for which no support whatsoever is provided to modularly define groups and roles. Thus, most of the time, role implementation is hardwired in the agent code, preventing all reuse opportunities.

3.1.4 Functional and Non-Functional Properties

Most of the works in the AOP community are done on non-functional properties of applications. These works suppose that a monolithic base code exists. Aspects are then added to define infrastructure properties.

The situation is different in applications based on the Aalaadin model. Groups are mostly used to define functional facets of the application, as each group

supports the interaction between agents trying to solve a domain problem. The application is then defined with groups, the "base code" being the code in the agents unrelated to roles (message handling, ...). In MadKit, there is also a notion of "system groups" based on "hooks" allowing the definition of non-functional groups. Hooks allow agents to override the way the platform evaluates some primitive operations such as message sendings. Agents can register themselves to hooks. These non-functional properties are not totally satisfactory, as hooks are limited (only one agent can control a hook, composition is thus not taken into account) and seem an *a*d hoc solution (they seem to be implementation-specific and are not mentioned in the Aalaadin model itself). Another limitation is due to the fact that dependencies exist from functional to non-functional properties. Roles in functional groups that require some non-functional property *explicitely* refer to a role to wear in a non-functional group.

3.2 Importing Aspects Properties Into Groups

Our goal is to use AOP in order to benefit the design and implementation of MAS systems, mainly by improving modularity. In the following, we propose an extension to the concept of Aalaadin groups. Groups are already well-suited to the separation of functional concerns. We rely on this feature to support separation of non-functional concerns as supported by AOP.

Even if modularity was one of Aalaadin's main concerns, it is only fulfilled at the conceptual level. The first extension we propose allows developer to implement each group in an individual module, by adding to the transversality found in group the modularity which characterizes aspects. Development is thus simplified as it becomes possible to add or remove individual groups. It also becomes possible to split developement in time or to share it among several persons. And finally, groups can be reused in different MAS applications, much like generic aspects.

Another interesting characteristic of AOP is that aspects are intrusive with respect to the base code. Infrastructure properties can then be defined using these AOP properties. Multi-agent systems need to define such an infrastructure with properties varying among time or applications, so AOP seems to be the best suited technology for this task. Our second extension is the introduction of AOP concepts in Aalaadin. We were however reluctant to have the two concepts of groups and aspect coexisting in the model, given how close they are, so we unifyied them. Furthermore, implementing aspects by means of groups and roles follows the line defined by the reflexive nature of Aalaadin, which "agentifies" services by implementing them with groups of agents collaborating to perform the desired function [4].

3.2.1 Modularizing Groups Using Role Reification

The main goal of this extension is to allow groups to be defined in individual modules rather than dispersed in the body of several agents. This enforces modularity and maximize reusability. Since a group is a set of roles, isolating the definition of a group requires separating role definitions from the agent code.

One could think that since join points are missing from the definition of groups, we could use Smalltalk's packaging mechanism and class extensions to provide group modularity. This is not possible for several reasons :

- This mechanism is not present in most languages. One of the features of AspectJ is that it can add methods to existing classes, which is close to class extensions. Hence using class extensions could be seen as using a subset of AOP.
- Using only class extensions would populate the agent class with all the role code, which is not the desired effect since we want different agent to have different roles. Hence we want to have an entry point in the agent where we can dispatch methods correctly. Having all the roles as extensions to the agent class will also have a much higher probability of conflicts.
- We provide a way to have join points later on, relying on the role reification that is dealt with in the following paragraph.

This separation of roles from the agents bearing them is achieved by the *reification* of roles. They hence become full-fledged objects still existing and manipulable during execution. Wearing a role is then essentially, for the agent implementation, obtaining a reference to an object representing this role. An agent then becomes an "empty shell", able to wear roles necessary for it to perform its task. The agent is additionally responsible of low-level responsabilities, such as message dispatching, life cycle, managing and executing roles ....

Once roles are reified, they can be separated from the agent's implementation and be logically put with the group implementation they belong to. The group then becomes a module containing all the code necessary to its execution and can be shipped separately.

On the implementation side, each role is represented by a class. The *agent messages* [1] the role can respond to, are implemented by methods of this role. One responsibility of the agent (as a role manager) is to automate the message delivery by executing the right method on the right role, extracting arguments from the message in the meantime.

[1] These messages are not like object messages, as they are much more complex: they for example contain information about the sender, and group and role information for sender and receiver.

One must note that reified roles have an impact on modelling in addition to implementation. This refinement of the Aalaadin model allows the acquisition and the dynamic transfer of roles from an agent to another, and permits the implementation of meta-roles, which is explained in section 3.2.2.

One benefit of the use of a class per role (and one method per possible request), is that roles can reuse or inherit behaviour from each other by only using mechanisms provided by object-oriented programming. This greatly enhances role reusability.

A similar approach to our reification of roles can be found in Philippe Mathieu's (et al.) MAGIQUE MAS platform [14]. MAGIQUE introduces the notion of reified competences, which are rather similar, though finer grained, to roles. As our model, this allows for initially behaviourless agents to dynamically acquire competences and to specialize them. MAGIQUE does not however have a group notion, and thus does not deal with modularity.

3.2.2 Defining Intrusive Groups Using Meta-Roles

We introduce *meta-roles* in order to allow the definition of intrusive groups. Intrusive groups are groups able to modify the execution of other groups. Hence meta-roles in intrusive groups allows us to define an equivalent of AOP join points when such a behavior is needed [2]. The meta-role concept is directly spawned from the meta-object concept. We first remind below the definition of the latter, before explaining the former.

3.2.2.1 Meta-objects and aspects

Meta-objects have been introduced in the fields of reflection and meta-programing in object-oriented languages [15][16]. A meta-object is an object which can act like a "virtual machine" for other objects called "base objects", thus defining the way they behave. For example, a meta-object managing concurrent access can modify the semantics of message sendings so as to control accesses to a base object with semaphores. Such modifications can be defined with a MOP, short for *Meta-Object Protocol*.

One of the uses of MOPs is the exploitation of meta-objects to support AOP [17]. Each aspect is defined with a set of meta-objects which are linked to the base objects where the aspect must intervene. A trace aspect for example is represented by meta-objects augmenting the semantics of message reception with a trace of the message to be recorded in a file. Weaving such an aspect consists of linking the meta-objects to the base objects which need tracing. Thus the meta-object associated to such an object adds a line to the log file each time a message is received.

[2] See section 3.2.2.3, page 75 for uses of this behavior

3.2.2.2 Meta-roles and aspects A meta-role is a role which controls the activity of other roles, and so can reason and act upon their execution. Meta-roles are quite close to meta-objects, as the roles are reified. A meta-role can therefore define the semantics of other roles, and so can act in an intrusive way on base roles which it is linked to via a *meta link*. This action consists of intercepting the control flow in the agent when the base role:

- receives an *agent message*,
- sends an *agent message*,
- requires that the agent enters in a group,
- requires that the agent exits from a group.

This intrusion of the meta-role is materialized for each of the intervention points by a message describing the semantics of this particular point. The set of signatures of these messages is the equivalent of a MOP.

This intrusive capacity allows us to use meta-roles to define aspects. An aspect is more precisely represented by a group which contains meta-roles. We name such groups "aspectual-groups". We thus reach our objective of representing aspects using groups. Facets belonging to the functional code of a MAS are described with groups made of base roles, and aspects representing infrastructure facets are defined with aspectual-groups containing meta-roles.

3.2.2.3 Example We use here the classic example of the market organization (Figure 3), adding the following aspects:

- A logging aspect: the broker traces the messages it sends and receives, to keep proofs in case of contestations for example.
- A cryptography aspect: the client/provider negotiation groups use secure communications.

We implement these aspects using the following aspectual-groups:

- The "Log" aspectual-group contains *1* "Logger" role, and n "Logged" roles. Logged roles send a message to the "Logger" role for each message their base role sends or receives. This aspectual-group is instantiated once, and each of the roles of the "Broker" agent are linked with a meta-role "Logged".
- Each "Secure" aspectual-group has two "Crypted" meta-roles. When two base roles establish a communication, their meta-roles exchange public keys in order to cypher their subsequent messages. This aspectual-group is linked to the "Negotiation" group structure. So, an instance of the aspectual-group will be woven with every instance of the Negotiation group. When an agent joins a Negotiation group, it will automatically join the Secure aspectual-group.

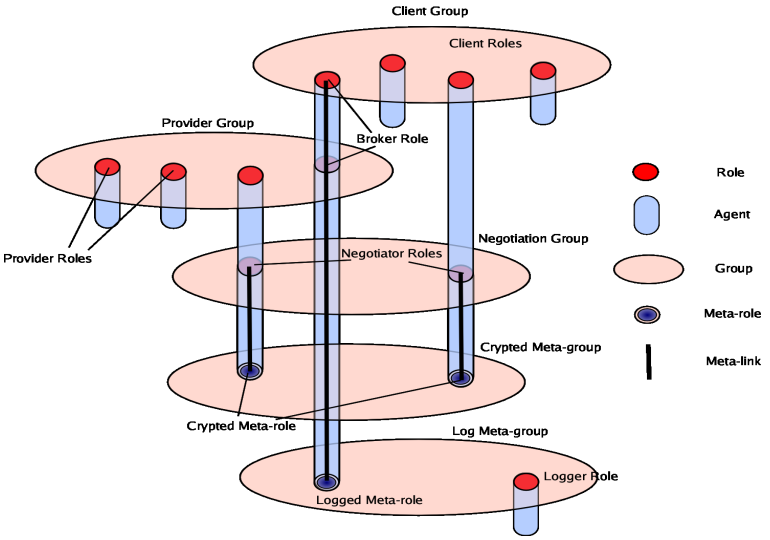


Figure 3. Meta-roles example

3.2.2.4 Weaving The weaving process consists in linking the "functional groups" to the aspectual-groups, by correctly setting meta-links between roles and meta-roles. The controls performed by a meta-role MR on a role R is materialized by messages from R to MR at each join point. To preserve the autonomous property of agents, an agent willing to wear R must prealably wear MR.

In order to keep the modularity and reusability properties defined previously, the meta link between a role and a meta-role is soft. There is no direct reference between roles and meta-roles. In the implementation phase, a role ignores the meta-role it will be linked with. Conversely, a meta-role knows the role it is linked with only at weave time. It is therefore possible to link different roles and meta-roles in different contexts at separate times. This low coupling needs a complementary mechanism allowing us to specify which meta-role will be woven to which role. This is done by group parametrization. The meta-roles to attach to a role are specified as prerequisites to wear the role, so that an agent willing to bear this role must also bear the associated meta-roles.

Please note that to simplify the discourse, we have until now essentially dealt with a *1-1* relationship between roles and meta-roles, but this relation can be possibly as wide as *n-n*. A meta-role in an aspectual-group can indeed be linked with several base roles, and conversely several meta-roles can be linked to a single base role. In the latter case, conflicts such as masking [3] may arise, as it is the case in classic AOP. A possible solution is to use the "chain of responsibility" design pattern, as described for meta-objects in [17].

[3] An example has been given earlier, in section 3.1.2, page 71.

4 Aspects for MAS Infrastructure

We list some infrastructure properties we have detected and which are suitable to be implement as aspects in a MAS platform. When possible, these aspects should be implemented using meta-roles in aspectual-groups rather than full-fledged aspects to follow Aalaadin's reflexive trend.

4.1 Messaging

The messaging strategy, albeit higher level, can also be isolated in a single aspect. This aspect comprises two parts:

(1) automatization of the message building process,
(2) parametrization of the message sending strategy.

4.1.1 Message Building

An object message sent to an agent is automatically promoted to an *agent message*. This avoids the burden of constructing *agent messages* manually, as most of the work can be automated.

An *agent message* have a much more complex structure than a regular object message. The one used here is a dictionary of keys and values allowing the receiver to reflect upon various parameters of the message, not only the request asked, but maybe who the sender is, when the message was sent or received, ...

The aspect can use the sender role and its context to fill the `senderRole`, `receiverRole`, `group`, `date`, ..., fields of the message. As for some other aspects mentioned, this aspect's join points are message sends on agent objects. On the receiving side, the agents can add extra parameters to the message before dispatching it to the role, such as the date when the message has been received.

To dispatch an *agent message* to a role, the agent performs two actions. On the one hand, it makes the receiver role refer to the *agent message*. And, on the other hand, it triggers the action to perform by sending an object message to the receiver role. So, when performing the action, the role can access to the *agent message* extra parameters such as the date if required.

4.1.2 Messaging Strategy

There are three messaging strategies so far:

- eager: This is the message sending strategy commonly used for objects. The message is sent to the other agent, and the sending role waits until the message is answered, the answer being received as the return value of the sent message, like in conventional object-oriented programming.
- asynchronous: This is the strategy often used by agents. The message is sent, and control returns immediately to the sender. The message has no answer, if one is needed, it will be given later by the receiver when it sends a message to the sender. The message will be processed normally by the agent, and the receiver may answer as many times as it wants to.
- lazy (using future objects): The message send returns immediately, as in the asynchronous case, but returns an answer like an eager message send. The answer is a "Future" (or Promise) object, of any type. When it it used, two cases can occur: the answer is either already computed, and evaluation proceeds immediately, or evaluation has not finished yet, and the calling role has to wait for it to end. In the current implementation, future objects are actually future roles, who wait for the response in place of the calling role.

The messaging policies are set group-wise to allow maximal flexibility. Messaging policies are high-level aspects, which are set by the application depending on its needs and its implementation, as asynchronous and synchronous messaging are fundamentally different. Lazy and eager policies are easier to swap on the other hand.

4.2 Agent Lookup

The join points of this aspect are the accesses to an instance variable of a role, as each role represents its acquaintances with some of its instance variables. If one of them represents a role, the lookup process is started. An instance variable represents a role if it contains a role, or if its name corresponds to a role available in the group the agent is in with the current role.

The procedure the aspect uses to find a suitable agent is as follows:

(1) The aspect first verifies if the variable really references an active role, and if so, if the role and the agent it represents are still in the system, and are still able to process requests.
(2) If both conditions hold, it proceeds and sends the message to the agent, which dispatches the message to the correct role as usual.
(3) If not, the searching process is started: the aspect searches an agent wear-

ing the correct role in the group the sender is in, sets its target to be the agent it just found, and finally sends the message to it.

A few exceptional handling strategies comes to mind, if no agent is found. They could be implemented using variants of this aspect:

- Wait until an agent holding the role appears in the group, checking from time to time if this event happens. The role sending the message is meanwhile blocked.
- Ask agents with the valid capacities if they can take over the role (including the message sender itself if it is eligible), then send the message to the selected one. One could imagine a variant of this where the search would consider all agents in the system if none is found in the group, asking agents to take the role and join the group.
- Fail, notifying the agent about the absence of such an agent, letting the sending role make a decision based on this information.

The current implementation uses the first scheme only. It proceeds by asking the agents responsible of the group (it wears a "group manager" role) if an agent fulfilling the given role is in the group, and waits on this event.

This aspect allows us to search for available agents more reliably and more naturally, as a reference to an agent instance variable will trigger the searching and verifying process as needed. This frees us from the burden of checking every time that the receiver is still available, hence the increased reliability as well as the increased focus on the functional code. It is of course possible to have various strategies in the different groups of the application or platform, depending on the needs of those groups.

4.3 Distributed Messaging

Distributed messaging must be dealt with transparently if we want to use an application indifferently on a single or on several machines. The use of a remote messaging framework such as SOAP, RST (Remote SmallTalk[4]), XML/RPC, CORBA ..., would provide us with a good basis to build agent distributed messaging on top.

This aspect can deviate control flow when the agent sends a message to another role. The aspect has ways to find where the agent really is and is able to send messages to him, using one of the previously mentioned remote messaging framework if he is outside. Moreover, this aspect could check the location where the agent is before sending the message, thus keeping track of mobile

[4] A framework that supports distribution of Squeak objects

agents, or use different remote messaging frameworks if they want to talk to an agent on a different platform, using a different communication medium.

The aspect must continue its processing on the receiving side, to properly decode the message and make it be received. One could implement this aspect using meta-roles previously mentioned, as this "remote" property could then be set on a by-group basis. Since this aspect was not considered a priority at the time of writing, some problems such as partial failures are not considered, though replication could be used.

4.4 Visualization and Debugging

As multi-agent systems are applications that are rather hard to develop, bugs can often happen. Hence the need to have good debugging tools is crucial. We believe a debuging aspect can be implemented to help the developer in bug tracking. We have implemented a first version of it in the form of a graphical visualisation tool of inter-agent communications. The structure of the application, in terms of groups, agents and roles is materialized onscreen, as are the messages sent between agents. Each agent is shown as an entity linked to the role it wears, and these roles are displayed in the group they are in. The messages exchanged are shown using arrows between the roles.

The screen capture provided in Figure 4 shows the visualization of a market organization with 4 agents (the red squares): two clients, a provider and a broker. Each agent wears several roles in various groups. Arrows link each agents and the roles it wears. Each role is contained in a group, represented by a rectangle.

Some work still remains in order to have a full-fledged debug aspect. Indeed, this aspect should also allow one to show and modify the state of the agents and roles, and to act upon the control flow of the application, as described in the next section.

4.5 Flow Control

This aspect should deal with the evaluation strategy of the agents: should they be all evaluating in parallel, or should they be synchronized? If they are synchronized, should they "step" in a repeatable order? This aspect should also govern whether time is equally shared between agents, or if each agent should have as much time as it needs and a limited number of actions.

This should be a pluggable policy at the group level, as the desirable strategies

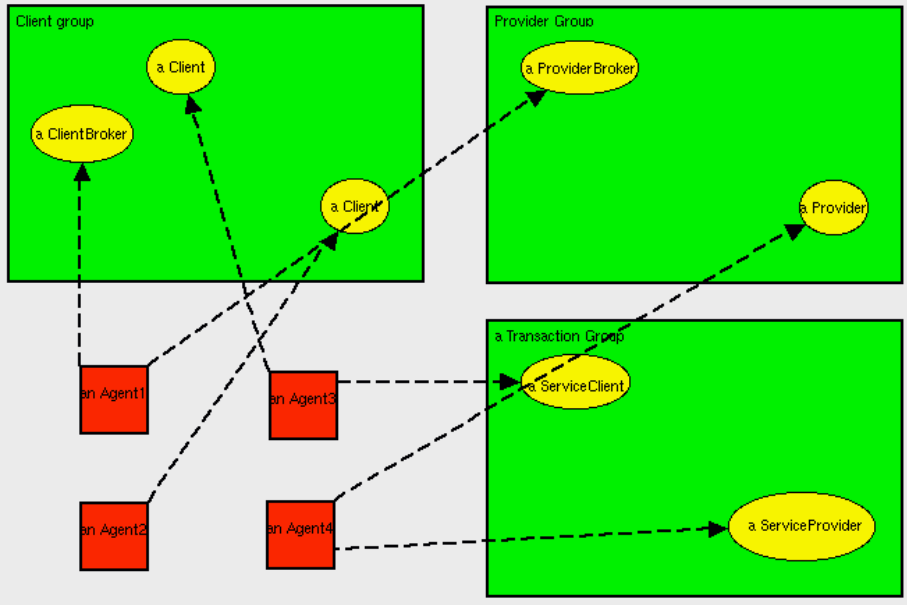


Figure 4. Broker example in Squeak, using the debug aspect

are greatly varying depending on the application designed. Different roles for the same agent could have different time-sharing strategies.

Another part of this aspect is more closely linked to the debug aspect described earlier: this aspect could deal with breakpoint and stepping support, so that one can define when the simulation should halt, and how its evaluation should continue after a halt. It can step at a pace indicated by the user, or restart normally, ... This debugging policy could be set at the group level as usual, but care must be given concerning synchronization-related issues.

5 Future Work

5.1 Improving Identified Aspects

Aspects we identified so far still require some improvements:

- While the messaging aspect is able to do automatic filling of *agent messages*, sometimes manual filling is needed. This could be done with a special syntax such as capitalizing the extra parameters. Suppose for example that we need to set the priority of a message to `urgent`. We could use the following syntax: **broker** provide: #apples atPrice: 200 **P**riority: #urgent (note the capitalized "P" to set a priority field, which is close to the one used in Seld (ref)) .
- Some aspects are designed but not yet implemented. This is mainly the case

of the Remote messaging aspect. Other aspects do not have all the "bells and whistles" mentionned, such as the agent lookup aspect.
- The visualisation aspect could be enhanced to be a debugging aspect if we add the notions of breakpoints, to halt all or parts of the platform on special conditions, and if we allowed groups, agents and roles to be easily inspected.

5.2 Other Aspects Envisioned

5.2.1 Bounded rationality

"Bounded rationality" as defined by Herbert Simon [18] is a compromise between the quality of a solution and the cost invested in computing and interaction. In order to embed such principles in a computer, several approaches have been proposed in the litterature: cost-calculus [19], anytime algorithms [20], approximate processing, ...

As the real-time aspects have been already studied in the litterature, the anytime properties of an algorithm could be considered an "anytime aspect". So a working direction would be to see if these algorithms could be extracted out of the base code, and implemented in a generic aspect.

5.2.2 Decision Making

The decision making process governs how the agent reasons and chooses tasks to do. There are indeed several ways for an agent to determine actions it should take next, the most common trends being:

- The reactive behaviour is the simplest. The agents responds a direct stimuli (here a message send) in a predictable way.
- A more complex behaviour can be built upon the latter one by combining several layers of reactive behaviour in a subsomption architecture. In this architecture, several simple reactive behaviours are layered and prioritized, so that basic behaviours are trigerred as needed (*e.g.,* avoiding obstacles), and other, more complex behaviours are trigerred when the situation is less urgent (*i.e.,* path planning).
- The deliberative behaviour: the agent evaluates a set of rules to determine the best action to take given the environment it observes.
- Finally, Markov Decision Processes can be used. In those, a policy is first computed for all possible states the agents can perceive, which associates each environmental state with the optimal action to take.

All these processes should ideally be replaceable, and would all fit in variants of a generic decision-making aspect. This aspect would require reifying the

tasks that roles have to do. So, the agent can reason upon, prioritize them, in addition to simply performing them .... This reification effort have not yet been done, but could have a lot of benefits. With this, a role could just be a list of tasks to perform, and could be worn by several agents having various decision methods to perform a role. This would even more separate the content of the role from the evaluation strategy of it.

5.3 Introducing AOP concepts into other MAS Models

We focused in this paper on extending the Aalaadin model with AOP concepts. We have chosen Aalaadin, since it has a good basis for separating concerns with its concepts of groups and roles. Obviously, the way to introduce AOP concepts and the impact of this introduction depends on the extended MAS model. Thus, it should be interesting to conduct other studies with different MAS models.

6 Conclusion

Building Multi-Agent Systems is a hard task because of the multiplicity of simultaneous concerns that developers should take care of. We therefore use AOP to separate the multiple, transverse facets of a multi-agent system and its software platform. In this paper, we focuse on the Aalaadin MAS model.

In order to take benefit of aspect properties in Aalaadin, we unified the concept of group with the one of aspect. We thus made Aalaadin groups inherit the modularity and reutilisability of aspects. This gives groups the same qualities aspects have, and even more, since groups can also be used to decompose the functional code of an application, something AOP can not yet easily do. The unification is done by reifying roles and by introducing the concept of *meta-roles*. Role reification allows the separation of their definitions from the agents ones. Hence, the definitions of all roles constituting a given group can be packaged together within a single module. This enables us to achieve modularity and furthermore allows reuse of generic groups.

The concept of meta-roles permits the creation of *aspectual groups*. Since meta-roles can reason and act upon roles, they allow defining groups that alter the execution of roles belonging to other groups.

We experimented with our model by developping a MAS infrastructure on top of Squeak and MetaclassTalk[21][17]. We defined several aspects of this infrastructure using aspectual groups, acheiving hence separation of concerns.

Using the same broker example as above, the resulting implementation is by far more modular and understandable compared to Aalaadin's reference implementation, MadKit.

References

[1] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin, Aspect-Oriented Programming, in: M. Akşit, S. Matsuoka (Eds.), Proceedings of ECOOP'97, no. 1241 in LNCS, Springer-Verlag, Jyväskylä, Finland, 1997, pp. 220–242.

[2] T. Elrad, R. E. Filman, A. Bader, Aspect-oriented programming, Communications of the ACM 44 (10) (2001) 29–32.

[3] R. Filman, S. Clarke, M. Aksit, T. Elrad (Eds.), Aspect-Oriented Software Development, Addison-Wesley, 2004, (to appear).

[4] O. Gutknecht, J. Ferber, MadKit: Organizing heterogeneity with groups in a platform for multiple multi-agent systems, Tech. Rep. 97188, LIRMM, 161, rue Ada - Montpellier - France (Dec. 1997).
URL http://citeseer.nj.nec.com/gutknecht97madkit.html

[5] A. Zunino, A. Amandi, Brainstorm/J: a Java framework for intelligent agents, in: Proc. of the 2^{nd} Argentine Symposium on Artificial Intelligence (ASAI 2000 - XXIX JAIIO), SADIO, Tandil, Buenos Aires, Argentina, 2000.
URL http://www.exa.unicen.edu.ar/~azunino/asai2000.ps.gz

[6] R. Vincent, B. Horling, V. Lesser, An Agent Infrastructure to Build and Evaluate Multi-Agent Systems: The Java Agent Framework and Multi-Agent System Simulator, Lecture Notes in Artificial Intelligence: Infrastructure for Agents, Multi-Agent Systems, and Scalable Multi-Agent Systems. 1887.
URL http://mas.cs.umass.edu/paper/200

[7] H. S. Nwana, D. T. Ndumu, L. C. Lee, J. C. Collis, ZEUS: a toolkit and approach for building distributed multi-agent systems, in: O. Etzioni, J. P. Müller, J. M. Bradshaw (Eds.), Proceedings of the Third International Conference on Autonomous Agents (Agents'99), ACM Press, Seattle, WA, USA, 1999, pp. 360–361.

[8] A. Garcia, C. Chavez, O. Silva, V. Silva, C. Lucena, Promoting advanced separation of concerns in intra-agent and inter-agent software engineering, in: Workshop on Advanced Separation of Concerns in Object-oriented Systems (ASoC) at OOPSLA'2001, 2001.
URL http://citeseer.nj.nec.com/460915.html

[9] J. Ferber, O. Gutknecht, A meta-model for the analysis and design of organizations in multi-agent systems, in: Third International Conference on Multi-Agent Systems (ICSMAS'98), 1998, pp. 128–135.

[10] H. V. D. Parunak, Go to the ant: Engineering principles from natural multi-agent systems, Annals of Operations Research (Special Issue on Artificial Intelligence and Management Science) (75) (1997) 69–101.

[11] Castelfranchi Cristiano, Guarantees for Autonomy in Cognitive Agent Architecture, in: Wooldridge Michael J, Jennings Nicholas R. (Eds.), ECAI-94 Workshop on Agent Theories, Architectures, and Languages, no. 890 in Lecture Notes in Artificial Intelligence, Springer Verlag, 1994, pp. 56–70.

[12] A. Newell, Physical symbol systems, Cognitive Science (4) (1980) 135–183.

[13] B. R. A., The whole iguana, in: SDF Benchmark Symposium, Robotics Science, MIT Press, 1989, pp. 432–456.

[14] J. Routier, P. Mathieu, Y. Secq, Dynamic skill learning: A support to agent evolution, in: Proceedings of the AISB'01 Symposium on Adaptive Agents and Multi-Agent Systems, 2001, pp. 25–32.

[15] P. Maes, Concepts and experiments in computational reflection, in: Proceedings of OOPSLA'87, ACM, Orlando, Florida, 1987, pp. 147–155.

[16] G. Kiczales, J. des Rivières, D. G. Bobrow, The Art of the Metaobject Protocol, MIT Press, 1991.

[17] N. Bouraqadi, T. Ledoux, Aspect-Oriented Software Development, Addison-Wesley, 2004, Ch. 11 – Supporting AOP using Reflection, (to appear).

[18] H. Simon, A behavioral model of rational choice .

[19] E. Eberbach, $-Calculus Bounded Rationality = Process Algebra + Anytime Algorithms, 2001.

[20] S. Zilberstein, Operational rationality through compilation of anytime algorithms, Ph.D. thesis (1993).

[21] N. Bouraqadi, Safe metaclass composition using mixin-based inheritance, Journal of Computer Languages and Structures 30 (1-2) (2004) 49–61, special issue: Smalltalk Language.

Design, Implementation, and Evaluation of the Resilient Smalltalk Embedded Platform

Jakob R. Andersen, Lars Bak, Steffen Grarup, Kasper V. Lund

OOVM A/S
Ellevej 2
DK-8310 Tranbjerg J

Toke Eskildsen, Klaus Marius Hansen, Mads Torgersen

Computer Science Department, University of Aarhus
IT-Parken, Aabogade 34
DK-8200 Aarhus N

Abstract

Most microprocessors today are used in embedded systems and the percentage of microprocessors used for embedded systems is increasing. At the same time development of embedded systems is very resource-consuming among other due to the lack of support for incremental development and for support for dynamic servicing and upgrading of deployed systems. This paper introduces the design and implementation of the *Resilient System* for embedded systems development which has as a design goal to support exactly this. Programs are written in a dialect of Smalltalk and executed on a compact, efficient virtual machine running on embedded systems. Programmers may connect to running virtual machines and service, monitor, or change the running systems. Furthermore, we present an evaluation of the Resilient platform in relation to the design goals through a case study of two development projects which successfully used the platform.

Key words: Virtual Machines, Smalltalk, Evaluation, Case Study

Email addresses: jakob@oovm.com (Jakob R. Andersen), lars@oovm.com (Lars Bak), steffen@oovm.com (Steffen Grarup), kasper@oovm.com (Kasper V. Lund), darkwing@daimi.au.dk (Toke Eskildsen), klaus.m.hansen@daimi.au.dk (Klaus Marius Hansen), madst@daimi.au.dk (Mads Torgersen).

1 Introduction

More than 90% of microprocessors produced today are used in embedded devices, ranging from dishwashers, automobiles, to mobile phones. The embedded industry each year spends more than 20 billion dollars developing and maintaining software in these products [1]. Development of software for embedded devices has traditionally been very cumbersome: source code is compiled and linked on a development host, whereupon the resulting binary image is transferred as a whole onto the actual device, typically into flashable memory. If the source code is changed, the entire process must be restarted. Making a change effective can thus easily take several minutes, severely limiting development productivity. This is problematic in an industry where software development and testing already comprises more than 50% of R&D budgets [1].

Another problem facing the embedded industry is the lack of serviceability. Software content in embedded devices doubles every two years, making exhaustive testing virtually impossible [2]. Deployed products will inevitably contain software errors, leading to expensive recalls. As an example, the telecommunications industry currently spends as much as 8 billion dollars each year fixing faulty handsets [3]. Debugging and profiling is sometimes supported during development, typically through instrumentation of code, making a debuggable system larger and slower than a non-instrumented version. For this reason such support is removed in deployed devices, making later error diagnostics exceedingly difficult.

Similarly, software content in embedded devices often requires updates in the form of upgrades. In a static software model, this is commonly accomplished by downloading and re-flashing the entire binary image. In a more dynamic software development model where component code can originate from several locations, such updates become extremely hard to administrate.

Finally, source code is typically highly platform-specific, and written in unsafe, low-level programming languages such as C or assembler. As a result, reusability is limited and software development requires a large degree of expertise regarding the particular target platform and its low-level details.

The Java™ 2 Micro Edition (J2ME) has been proposed as a solution to some of these issues [4]. It comes with a debugging interface, but this is again commonly removed due to space concerns. Furthermore, the dynamic code loading model in J2ME is severely limited compared to its Java 2 Standard Edition (J2SE) counterpart. Finally, the runtime environment for J2ME requires more memory than what is available on most low-end embedded systems.

Future pervasive computing scenarios in which a large number of embedded devices are communicating will require a much more flexible software model, allowing for dynamic and changing functionality over time [5]. In the face of this,

the *Resilient System* has been designed and implemented. The design goal for the Resilient System is to have an always running serviceable platform for embedded devices with the following characteristics:

Incrementality and serviceability: The platform should support incremental programming enabling rapid development and full product serviceability.

Accessibility and reuse: Furthermore, it should be accessible to non-expert programmers, by being based on a safe, high-level programming language with easy support for reuse of platform-independent code.

Low resource consumption: Finally, the platform should be compact enough to be useful on low-end embedded devices, including system-on-chip (SOC) solutions, yet efficient enough for most real-time applications.

In this paper we present the design of the Resilient platform as driven by these goals. Furthermore we evaluate the extent to which this was successful, based on experiences from two development projects using the Resilient platform, both conducted in a cooperation between academia and industry.

1.1 Contributions

The expected contributions of this paper are:

- to introduce the innovative design and implementation of the Resilient System which provides incremental development, dynamic upgrading, and serviceability in embedded systems development and
- an evaluation of the Resilient System in terms of its design goals and of the relevance of the design goals to the development of two embedded systems.

1.2 Paper Structure

The rest of this paper is structured as follows: Section 2 presents the design and implementation of the Resilient System in terms of its associated programming language (Section 2.2), embedded platform (Section 2.4), and programming environment (Section 2.5). Next, Section 3 presents our evaluation of the Resilient System through the conduction of two projects; the “Digital Speakers” project (Section 3.1) and the “LIWAS” project (Section 3.2). Section 3.3 discusses the experiences of using the Resilient Systems and finally Section 4 concludes.

2 Design and Implementation of the Resilient System

2.1 Overview

The complete Resilient System comprises a development environment running on a PC and a running Resilient Virtual Machine deployed on an embedded system or run locally. Figure 1 shows a deployment view of the Resilient System using a Unified Modeling Language (UML; [6]) deployment diagram. The Resilient Program

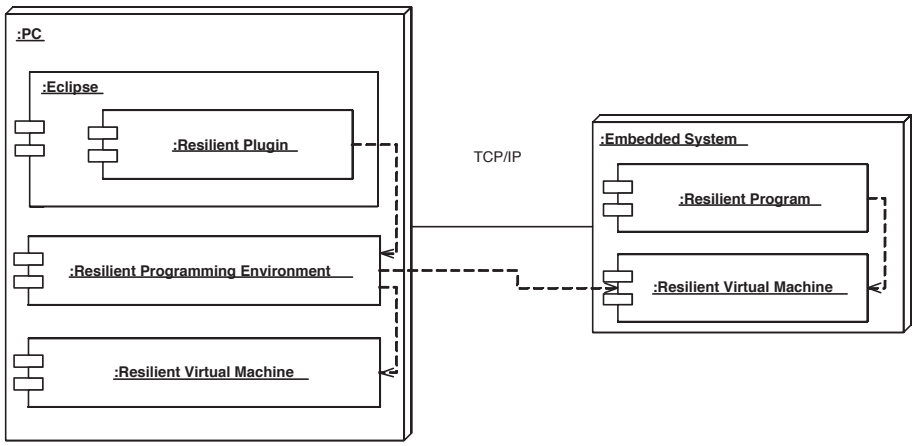


Fig. 1. Deployment Overview of the Resilient System

Environment, accessed using Eclipse with Resilient plugins, communicates with the running Resilient Virtual Machine using any available network connection.

The virtual machine connects to the underlying operating system (if available) in order to establish network connections. If the virtual machine is running without

an underlying operating system, a TCP/IP networking stack running on the virtual machine is used.

In order to create snapshots for booting and deployment, the programming environment uses a local virtual machine. Due to the platform independence of the Resilient code, the local virtual machine is independent of the target platform for the snapshots.

Having created an initial snapshot, the developer transfers it to the target hardware and starts the virtual machine. Incremental development is performed by connecting to the virtual machine from Eclipse, which controls debugging as well as updating of classes and evaluation of code on the target virtual machine.

Changes to the running system are not persistent, so the creation and transfer of a new snapshot is necessary if the changes are to survive rebooting of the virtual machine.

The following sections describe the design of the Resilient System in detail.

2.2 The Resilient Programming Language

The Resilient programming language is a dialect of Smalltalk [7] designed for simplicity, compactness, and performance. Smalltalk was chosen for several reasons: Smalltalk is a simple, dynamically typed, object-oriented programming language; everything is an object and behavior is described as message sends between objects; Smalltalk has proven ideal for supporting incremental program modification in that a programmer can freely modify a program without the need for recompiling and restarting the application; and most Smalltalk systems use a snapshot model allowing the same program execution to survive for years.

The Resilient programming language differs from Smalltalk in the following ways:

- Full syntax for classes is provided.
- Last-In-First-Out (LIFO) blocks are enforced.
- An atomic test-and-store statement for synchronization is introduced.
- A namespace hierarchy for modularization of libraries is introduced.

Traditionally, Smalltalk systems have forced programmers to use the integrated programming environment for all program manipulations. Methods are the unit of manipulation and for that reason a full syntax for classes does not exist. We introduce a full syntax for classes to allow programmers to use standard tools for program manipulation and source control management. The class syntax has been inspired by the syntax for Self [8]. The example below shows the source code for `Mutex`, a class that implements a simplified lock structure.

```
Mutex = Object (
  | owner |
  "acquire the lock prior to evaluating the
   block and then release the lock"
  do: [action] = (
    ["repeat the atomic test and store until it succeeds"
      owner ? nil := Scheduler current
    ] whileFalse: [ Scheduler yield ].
    action value.
    owner := nil
  )
)
```

Most object-oriented systems provide high-level synchronization mechanisms as part of the programming language [9] or as prebuilt data structures [10]. Instead, we have introduced a very low-level and simple synchronization mechanism; an atomic test-and-store statement. Advantages of this approach are minimal virtual machine support and a very flexible building block for implementing a wide range of high-level synchronization mechanisms. The above `Mutex` example uses the atomic test and store statement to busy wait for exclusive access. In the example, the current thread is computed, and then we atomically compare and do a conditional store (`owner ? nil := Scheduler current`). The instance variable owner is compared to nil, and if they match the current thread is assigned to it. The boolean result of the statement indicates whether the atomic test-and-store succeeded.

In order to support independently developed program parts, a solution for preventing name collision has been introduced. Resilient provides a simple form of namespaces. Any class can act as a namespace. For instance, the class describing elements in a list resides inside the List class. Classes in two different namespaces will therefore not be subject to name collisions.

2.3 Typed LIFO Blocks

Creating blocks in Smalltalk has always been a potential source of performance problems. Blocks might survive the lifespan of the creating activation forcing the underlying implementation to heap allocate not only the blocks themselves but often also the method invocations in their scope. This is expensive and also stresses the memory management system. In Resilient, we have restricted blocks to be last-in-first-out (LIFO), which means a block cannot survive the creating activation. This allows Resilient to stack allocate blocks, thereby eliminating most costs associated with block creation.

To guarantee this behavior, we have introduced a type declaration for blocks: square

brackets around a formal parameter specifies that it is a block. An example is the parameter `action` in the above `Mutex` class. This separation between objects and blocks makes it straightforward for the byte-code compiler to statically enforce LIFO behavior, by preventing blocks from being stored in objects or used as return values.

The graph in figure 2 on the following page shows the execution time of a simple, recursive, block-intensive micro-benchmark on a number of Smalltalk virtual machines. The benchmark constructs a linked list and uses blocks and recursion to compute its length:

```
Element = Object (
  | next |

  length = (
    | n |
    n := 0.
    self do: [ :e | n := n + 1. e ifLast: [ ^n ]. ].
  )

  do: [block] = (
    block value: self.
    next do: block.
  )

  ifLast: [block] = (
    next isNil ifTrue: [ block value ].
  )
)
```

It follows from the implementation that the micro-benchmark allocates at least one block-context per level of recursion, and that the non-local return in the `[ ^n ]` block must unwind at least as many contexts as the length of the linked list.

The graph shows that the execution time is linearly dependent on the recursion depth for all virtual machines. It also shows that enforced LIFO blocks makes our virtual machine almost 78% faster than the virtual machines for Squeak and Smalltalk/X, when it comes to interpreting block-intensive code. Better yet, our interpreter outperforms the just-in-time compiled version of the Smalltalk/X system by more than 16%.

Of course this approach has some language implications. On the negative side, the purity of the standard Smalltalk “everything is an object” credo is somewhat hampered with two static types rather than one. In practice the generality of standard Smalltalk blocks is rarely used, but there are a few (we know of two) com-

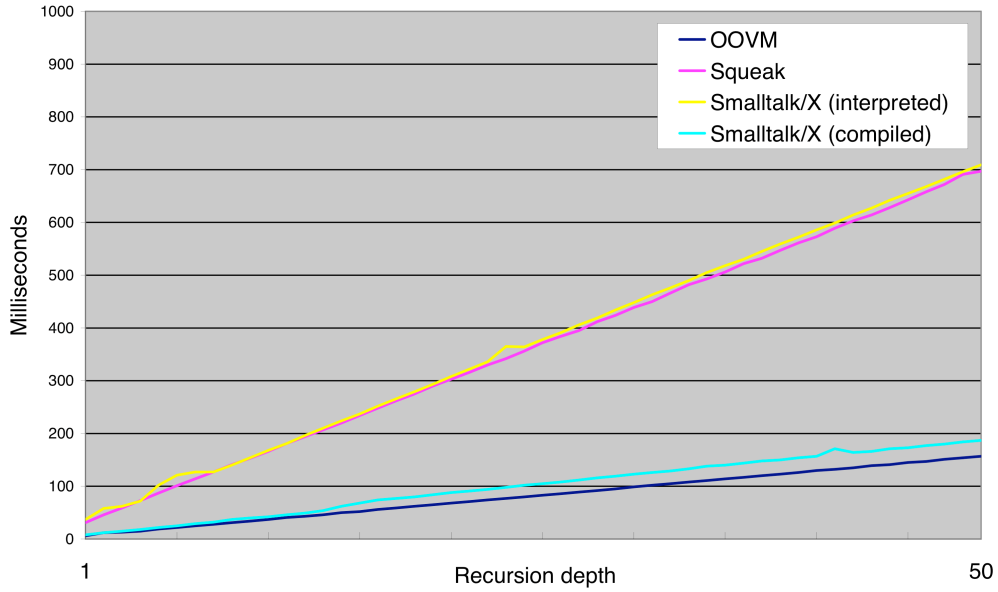


Fig. 2. Execution time for block-intensive micro-benchmark

mon situations where it would be natural to store blocks inside objects in order to dynamically parameterize these objects with behaviour. One example is sorted collections, which should be parameterizable with the comparison operator to use for the sorting. The other example is Graphical User Interface (GUI) widgets like buttons, which should be able to store a callback function for when the button is pressed. In neither case can Resilient LIFO blocks be used, and one must instead apply the “function object” technique from, e.g., Java Comparators, where a full object is supplied, implementing the desired behaviour (comparison or callback) as a method.

From a language point of view it should be noted that general blocks themselves are not without problems, especially in light of the non-local return mechanism. Non-local return makes the block return to its creating context rather than its calling context, but that is meaningful only when the creating context is still on the stack, and otherwise gives rise to a runtime error (after possibly having peeled apart the whole stack in search of the missing activation). This undesirable situation is naturally prevented in Resilient, so the added static typing of blocks does in fact have an error-preventing as well as an efficiency-related effect.

LIFO behaviour also means that blocks can never be transferred between concur-

rent threads, avoiding the similar issue of what to do in the case of non-local returns to a different stack. All in all we think that this language restriction has important semantic advantages along with the efficiency gain, and that the loss of expressiveness is a minor problem in practice.

2.4 The Embedded Platform

The embedded platform is based on a small object-oriented virtual machine. All software components are compiled to safe, compact bytecodes and executed on top of the virtual machine. The compactness makes it possible to fit the virtual machine, core libraries, device drivers, TCP/IP networking stack, and user applications in less than 128KB of memory.

The embedded platform can be configured to run directly on hardware without the need for an operating system. This accommodates for the most resource constrained devices, onto which it is impossible or impractical to fit a full operating system. However, it is also possible to run the embedded platform on top of existing embedded operating systems, such as Embedded Linux [1] or Symbian OS [2]. This option is useful in projects that depend on existing applications or device drivers.

Software components running on top of the virtual machine are mostly platform independent. The virtual machine bytecodes themselves are independent of the hardware on which they run, so that it is possible to build applications that run on any device equipped with the embedded platform. However, the virtual machine allows device drivers to be implemented on top of it, and therefore some software components may depend on external input/output devices available only on some hardware platforms. The example below briefly illustrates how a sample device driver for the Intel StrongARM general-purpose I/O (GPIO) module might be implemented:

```
GPIO = Object (
  | io |

  initialize = (
    io := Memory at: 16r90040000 size: 16r20.
  )

  setOutput: pins = (
    io longAt: 16r08 put: pins.
  )
```

[1] http://www.embedded-linux.org
[2] http://www.symbian.com

```
  clearOutput: pins = (
    io longAt: 16r0C put: pins.
  )
 )
```

The example also illustrates the most common way of providing device access, viz., by mapping device address spaces as part of system memory space. Access to memory-mapped I/O is provided through external memory proxy objects (here through `Memory` objects). The virtual machine makes sure that are driver cannot allocate a proxy that refers to the object heap, thereby possibly corrupting the heap. Interrupt requests from devices are reified as *signal objects* by the Resilient system software; device drivers consequently handles interrupt requests at the level of these objects.

Software components running on top of the virtual machine are fully serviceable through a reflective interface in the embedded platform. The reflective interface allows the programming environment running on the developer's personal computer to inspect the state of the running system and possibly change it. Section 2.5 gives more details on the connection between the programming environment and the embedded platform.

Most of the reflective interface is implemented in the Resilient Programming Language itself as a component running on top of the virtual machine. The virtual machine provides hooks for manipulating breakpoints and threads and for inspecting and changing classes, methods, and objects. The communication protocol for reflection is handled by a service implemented entirely in the Resilient Programming Language. The reflective interface is always available, even on devices deployed at customer sites. The result is that developers can debug, profile, and update running code on any device that runs the embedded platform.

The embedded platform guarantees predictable response times. It supports scheduling of interrupt handlers and time critical tasks. The non-disruptive, real-time garbage collector handles all resource management in the background. Furthermore, the virtual machine prevents user code from performing malicious operations, and as such it provides a secure, device-independent platform for real-time software execution and delivery.

2.5 The Programming Environment

The Resilient programming environment is written as a plug-in to the Eclipse extensible IDE [3] . The Eclipse framework provides well-known methods and abstractions for managing projects and browsing and editing source code. The Resilient plugin

[3] `http://www.eclipse.org`

provides the Resilient-specific components, such as source code compiler, debugging, and the reflective connection to a running embedded platform. Because the IDE is based on the industry-standard Eclipse framework, developers are able to transfer existing knowledge and practices of software development into the realm of embedded development.

An important part of the Resilient development model is the ability to connect the IDE to a running embedded platform. When connected, the IDE is able to inspect and make changes to the object model on the running platform. The connection is made using the network stacks present on the embedded platform, such as standard TCP/IP. The reflective interface on the embedded platform allows the IDE to inspect, pause and terminate running threads, inspect and modify objects in the object heap, and update code on the running platform.

The reflective interface is key to the Resilient way of developing software. Since developers can evaluate chunks of code on the target device, and upload changes while the system is running, it is possible to immediately see the effects of changes in source code. Debugging is also easier, since it is closely coupled with the source code development.

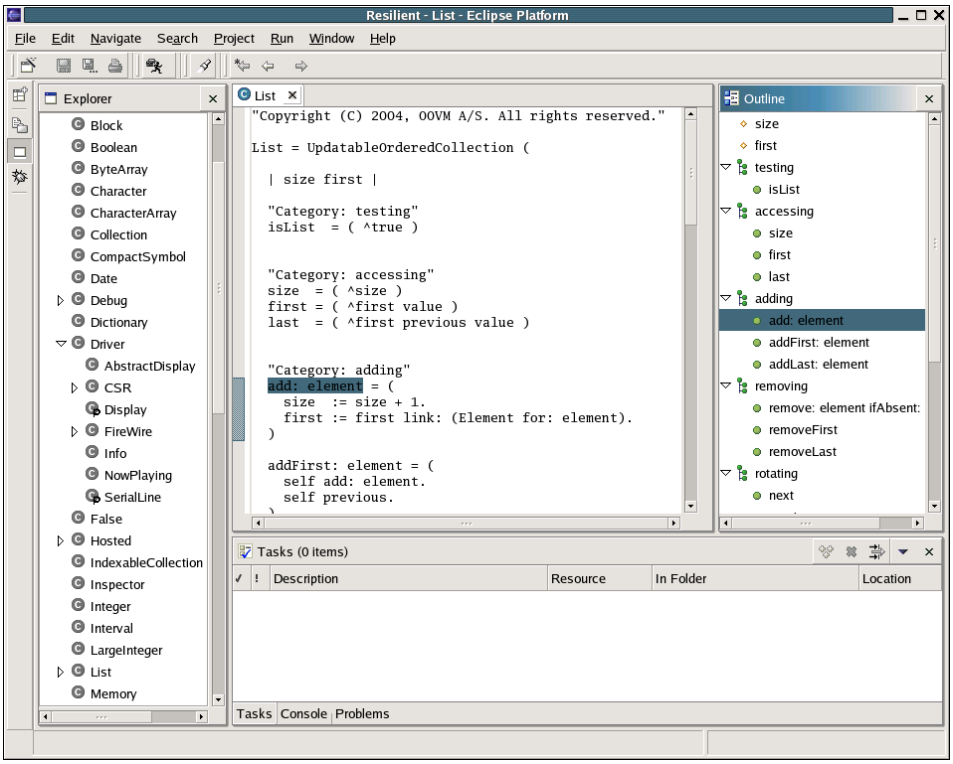


Fig. 3. Resilient IDE with a sample project open

Classes in Resilient are organized in namespaces, which may be nested, providing a means for grouping of related classes. The Resilient IDE, shown in Figure 3, offers

advanced features for navigating and searching the source code.

3 Evaluation

This section relates the experience of two research projects – both including academic as well as industrial participation – in which the Resilient platform was used. One project was aimed specifically at evaluating the platform, whereas the other uses Resilient as a vehicle for industrial prototype development. We first introduce the projects and then discuss experiences of using the Resilient System.

3.1 The "Digital Speakers" Project

The purpose of this project was to evaluate the Resilient platform in an industrial setting, targeting as a test case the next generation digital speakers of Bang and Olufsen (B&O)[4], which are to be connected using the IEEE1392 Firewire standard.

The virtual machine was ported to a development board with a number of Firewire-related hardware components. A prototype "speaker" was set up by connecting the board to a Firewire source (in this case a laptop) and a digital audio destination (in this case a stereo). In a production speaker, the Firewire source would be a B&O stereo and the digital audio destination would be a digital amplifier and a number of speaker units, boxed up with the board as an active speaker.

A programmer with no previous experience in embedded software was employed to develop a full test application ranging from low-level platform-specific Firewire driver code to a high-level platform-independent webserver component.

3.2 The LIWAS Project

The LIWAS project is a three year research and development project aiming at producing a framework, *Ex Hoc*, for hybrid communication in sensor networks [11]. The initial use scenario is a network of communicating sensors for measuring road condition, i.e., whether the road is dry, icy, snowy, or rainy. This scenario has applications in traffic safety as well as for resource consumption in connection to road maintenance.

[4] `http://www.bang-olufsen.com`

The sensors will be deployed in cars and on stationary road signs along roads. This will allow for ad hoc networking among cars [12] in combination with centralized communication through either Internet-connected road signs or mobile gateways in the form of mobile phones in cars. The communication will, e.g., allow cars to be warned by passing cars if the road conditions are problematic further ahead.

The current status is that the first communicating mobile sensor system with a very simple dissemination protocol is built and tested. Figure 4 shows a deployment view of Ex Hoc. The major part of the software on the *Mobile Unit*, viz. the *Com-*

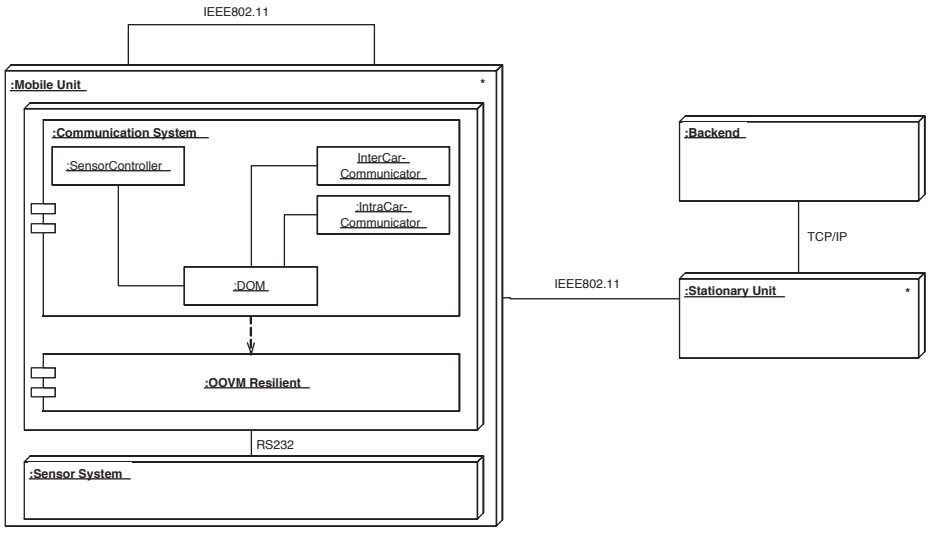


Fig. 4. Deployment View of Ex Hoc

munication System is written using the Resilient System. It currently runs on a CerfCube [5] board with an XScale processor.

The *Sensor System* is mainly low-level microcontroller code written in C. The *Sensor System* interfaces to the various sensors needed for reliably classifying road conditions. The *Sensor System* delivers measurements 36 times per second (which is equivalent to each meter at 130 kph), each of which needs to be classified by the *Domain Object Model (DOM)*. The ad hoc communication puts real-time constraints on the system as well since among others classifications need to be commu-

[5] `http://www.intrinsyc.com`

nicated among cars passing each other at high speed. The *Stationary Unit* contains software analogous to that of the *Mobile Unit*.

The implementation of the communication system of the sensors is done using the Resilient System by two programmers with little experience in embedded software. The development is done in collaboration with the private company LIWAS A/S [6] which is also responsible for the necessary hardware construction.

3.3 Experiences

This section presents and discusses experiences from the "Digital Speakers" and LIWAS projects in relation to their use of the Resilient System as a basis for implementation.

3.3.1 Batch and Incremental Development

The "speaker" application was the first major application to be developed on the Resilient platform, and the project took place while the platform was still being developed and thus suffering from a number of bugs and flaws. One of these turned out to be a major show stopper: a problem with the serial line meant that for some time the platform could not survive receiving incremental program modifications. Thus we were temporarily forced into the batch-oriented style more conventionally used for device programming, and inadvertently ended up being able to compare it with the incremental development model of the Resilient system.

The incremental mode of work proved a huge benefit to productivity. The ability to query and modify the state of the running device (by sending code to it – at the time the debugger was not yet functional), and to modify the running code for programming or test purposes gave rise to a very tight development loop. Compared to desktop development environments the immediacy and integration of the code-test-run cycle on the Resilient platform was found to be more in line with dynamic systems such as conventional Smalltalk than the sopmewhat slower turnaround of static systems such as Java and C++. The experiences in the LIWAS project concur with this.

3.3.2 Dynamic Update in Development

An example of the power of serviceability comes from the LIWAS project: part of the prototyping work has been to equip a trailer with among others light reflection, road temperature, dewpoint, and air temperature sensors. This was done to collect

[6] `http://www.liwas.com`

as much data as possible on as varied road conditions as possible in order to be able to create a suitable classification algorithm. While using the trailer with measurement collection and classification controlled by software written in the Resilient System, it was possible to connect to the running Resilient System with a PC running the Resilient programming environment. This was very useful among others in deploying the system, since it was possible to monitor the system, see what was wrong if anything, and possibly fix the problem online.

One problem encountered during development of the LIWAS application was the inability to update the running code for a thread handling serial communication. By refactoring the code to a tight loop containing a method invocation, we sidestepped this limitation. This experience lead to the standard practice of factoring all looping threads.

A probable future use of the incrementality and serviceability in the LIWAS project will be the dynamic update of the classification part of deployed systems. Currently, we are testing two different strategies of classification: one based on a physical model and one based on a statistical classification. It may possibly turn out that the best strategy for classification will change after the LIWAS system has been deployed in a number of test installations and that the type of change is one that requires major code changes which cannot be parametrized in any single algorithm. The Resilient System contains the main functionality for this, but probably defining a component model (which enables developers to make deployable, versionable collections of classes and objects) will be necessary on top of this.

3.3.3 The Resilient Programming Language

Within the resources of the projects we did not have the opportunity to directly compare programming in Smalltalk with solving an equivalent task in a traditional embedded programming language like C or C++. We have however written both low level (e.g. the FireWire driver in the Digital Speakers project) and high level (e.g. the webserver in the Digital Speakers project) code.

At the low level, Smalltalk lacks the ability of C-like languages to directly map declared datastructures onto the very specific bit layout of data in memory which is often characteristic of low level programming. The abstraction mechanisms of Smalltalk are strong enough, however, that it is straightforward to represent the individual data components symbolically by means of accessor methods.

In the “speakers” application, the construction and decoding of FireWire packages was as simple and elegant as any C version, the bit manipulation localized to a few package-specific methods. For parts of the FireWire code, we did indeed have C source available for comparison. In conclusion, writing driver software and related low level programming presents no special challenges in the Resilient Programming Language.

At the high level, the use of object-oriented abstractions is a well documented advantage, which was also heavily leveraged in the project. A similar effect could probably have been achieved with the more traditional C++. However, apart from being far from platform independent, C++ as a compiled language inherently lacks the possibility of incremental update.

The combination of platform independence and object-oriented abstraction meant that the whole web server component, first developed on top of Linux, could be migrated to the evaluation board with only one minor source code adjustment before the code was running again. Thus, the on-the-board development philosophy is balanced by the possibilty of writing extensive parts of an application off-board if necessary. Moreover, this suggests the possibility for reuse of large amounts of application code across different platforms, something which is rarely seen in the embedded world.

In the LIWAS project, a “proof-of-concept” prototype was first developed on a Resilient System running without operating system. Using Smalltalk, it was possible to implement a special-purpose serial driver which hooked into the running Ex Hoc system with miminal overhead. The system was later ported to Embedded Linux for further prototyping since a large number of drivers had to be used including Bluetooth, USB, and IEEE802.11 drivers. Except for the driver part, there was little code that had to be removed or replaced. Currently, we are primarily developing directly on the CerfCube, but also developing on a version of the Resilient System running on Microsoft Windows XP without any code changes being necessary between the setups. The end goal is that the system (i.e., drivers) will be fully implemented on a version of the Resilient System running without an operating system.

3.3.4 Interpretation and Performance

With a purely interpreted architecture, certain performance characteristics are to be expected. Bytecode representations of code are generally compact, whereas compiled code is several times more memory consuming. This is the reason for having Just-In-Time (JIT) compilers in many systems in order to compile only the most used code. That code will be doubly represented however, and more importantly in a small system, the JIT compiler itself takes up considerable space. But compiled code undoubtedly does run a lot faster.

The running “speakers” application, including the VM itself, webserver, firewire driver, TCP stack, etc., fitted comfortably within the 128K RAM available on the development board. There were no real-time constraints in the functionality, and therefore little opportunity to evaluate efficiency. Engineers at B&O estimated that the Resilient VM might have a hard time keeping up with e.g. real-time filtering of audio streams, at least on affordable hardware. For this purpose, interfacing to compiled or assembler code would be crucial for the performance-intensive parts

of the code.

The LIWAS application on the other hand is inherently a real-time application in which the system has to handle a set of measurements 36 times per second and make a classification based on this for each set of measurements. Our current implementation clearly supports this also with the statistical classification. And even though the CerfCube board has a large amount of RAM available, only little is used, and the application would be able to be ported to a device with 256K RAM.

4 Conclusion

4.1 Incrementality and serviceability

Incremental development with a smooth and tight integration of coding and running has always been a hallmark of Smalltalk and related dynamic programming languages. A major contribution of the Resilient development model is to make this manageable on an embedded system, by separating out the reflective parts of the running program onto a separate IDE on a different platform.

Experience from both development projects shows that the Smalltalk programming feel does indeed carry over to the embedded world, where it is an even greater relative advantage, because the traditional batch-style alternative here is so much more costly than in a desktop environment.

Where the update of running code in a desktop Smalltalk environment is a possibility, in Resilient it is almost inevitable. This brings to focus some of the standard problems in this area, most notably the fact that code has to be structured in anticipation of later changes.

4.2 Accessibility and reuse

Traditional embedded programming tends to require a great deal of platform specific expertise. Not only do programs have to be written in platform-dependent dialects of assembler, C or C++, but the tools used for testing and debugging are highly platform-specific and often even require special hardware. With the Resilient system, the execution semantics as well as the development tools are independent of the execution platform, as long as a virtual machine has been ported to it. Of course, drivers etc. must be implemented specially for each device, but the object-oriented abstraction mechanisms of Smalltalk help encapsulating and isolating these parts. Thus, code reuse across platforms becomes a real possibility.

The projects have confirmed this situation. Programmers with no embedded experience or device-specific knowledge have found it easy to develop applications on the system, and cross-platform reuse has also been employed successfully. Compared to standard Smalltalk, the LIFO restrictions on blocks have proven a nuisance at some points, but generally the language restrictions of the Resilient Programming Language have not been too inhibitive.

4.3 Compactness and efficiency

Being something as unusual as a fully interpreted bytecode-based embedded platform, the Resilient system emphasises size constraints over speed considerations. The bytecode format, memory layout, and virtual machine itself are all optimized for compactness. That said, of course a lot of devotion has gone into running code as fast as possible. The block limitations are an example of that.

From especially the "speakers" project we can conclude that a good deal of software fits onto a rather small device, although very low-end devices will be out of reach for the Resilient model. As for speed, we have no measurements, but can only conclude that in both projects it was fast enough for our purposes.

Acknowledgements

The academic part of this work has been supported by the software part of the ISIS Katrinebjerg competency centre [7] . We thank the industrial participants in the projects which evaluated the Resilient System.

References

[1] Wind River Systems, Inc., Investor Relations Presentation (September 2003).

[2] Wind River Systems, Inc., Changing Software Development in the Electronics Industry, Prudential Securities Technology Conference (October 2002).

[3] K. Belson, Beware of the Worm in Your Handset, The New York Times Technology Section (November 28, 2003).

[4] R. Riggs, A. Taivalsaari, M. VandenBrink, Programming Wireless Devices with the Java™ 2 Platform Micro Edition, The Java Series, Addison-Wesley, Reading, Massachusetts, USA, 2001.

[7] `http://www.isis.alexandra.dk`

[5] ISTAG, Scenarios for Ambient Intelligence in 2010, Tech. rep., European Commision - Community Research, ISTAG: European Commission's Information Society Technologies Advisory Group (February 2003).

[6] OMG, Unified Modeling Language specification 1.5, Tech. Rep. formal/2003-03-01, Object Management Group (2003).

[7] A. Goldberg, D. Robson, Smalltalk-80: The Language and its Implementation, Addison-Wesley, Reading, Massachusetts, USA, 1984.

[8] D. Ungar, R. B. Smith, Self: The power of simplicity, in: Proc. of the OOPSLA-87: Conference on Object-Oriented Programming Systems, Languages and Applications, Orlando, FL, 1987, pp. 227–242.

[9] J. Gosling, B. Joy, G. Steele, The Java Language Specification, Addison Wesley, 1997.

[10] B. B. Kristensen, O. L. Madsen, B. Møller-Pedersen, K. Nygaard, Abstraction mechanisms in the beta programming language, in: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, ACM Press, 1983, pp. 285–298.

[11] K. Hansen, T. Eskildsen, L. Kristensen, K.-D. Nielsen, R. Thorup, J. Fridthjof, U. Merrild, J. Eskildsen, The Ex Hoc Infrastructure - Enhancing Traffic Safety through LIfe WArning Systems, in: In Proceedings of Trafikdage 2004 (11th Danish Conference on Traffic Research), Aalborg, Denmark, 2004.

[12] E. Royer, C.-K. Toh, A review of current routing protocols for ad hoc mobile wireless networks, IEEE Personal Communication 6 (2) (1999) 46–55.

Prototalk : an Environment for Teaching, Understanding, Designing and Prototyping Object-Oriented Languages.

Alexandre Bergel [a] Christophe Dony [b] Stéphane Ducasse [a]

[a] *Software Composition Group*
Institut für Informatik und angewandte Mathematik
Universität Bern, Switzerland

[b] *LIRMM, Montpellier, France*

Abstract

With prototype-based languages, concretization and abstraction are unified into a single concept a *prototype*. Prototype-based languages are based on a simple set of principles: object-centered representation, dynamic reshape of objects, cloning and possibly message delegation. However, they all differ in the interpretation and combination of these principles. Therefore there is a need to compare and understand.

In this paper we present Prototalk, a research and teaching vehicle to understand, implement and compare prototype-based languages. Prototalk is a framework that offers a predefined set of language data structures and mechanisms that can be composed and extended to generate various prototype-based language interpreters. It presents a classification of languages based on different mechanisms in an operational manner.

1 Introduction

Born at the end of the eighties [1][2][3][4], and influenced by *frame* and *actors* languages, prototype-based languages propose an object-oriented model based on the object as the sole entity of structure and computation. The prototype paradigm and the semantics of its main aspects have been declined into several variations. In the first age of prototype-based languages, research languages such as Self [3][5], Agora [6] [7], Kevo [8], Obliq [9], Garnet [10], Moostrap [11],

Email addresses: bergel@iam.unibe.ch (Alexandre Bergel), dony@lirmm.fr (Christophe Dony), ducasse@iam.unibe.ch (Stéphane Ducasse).

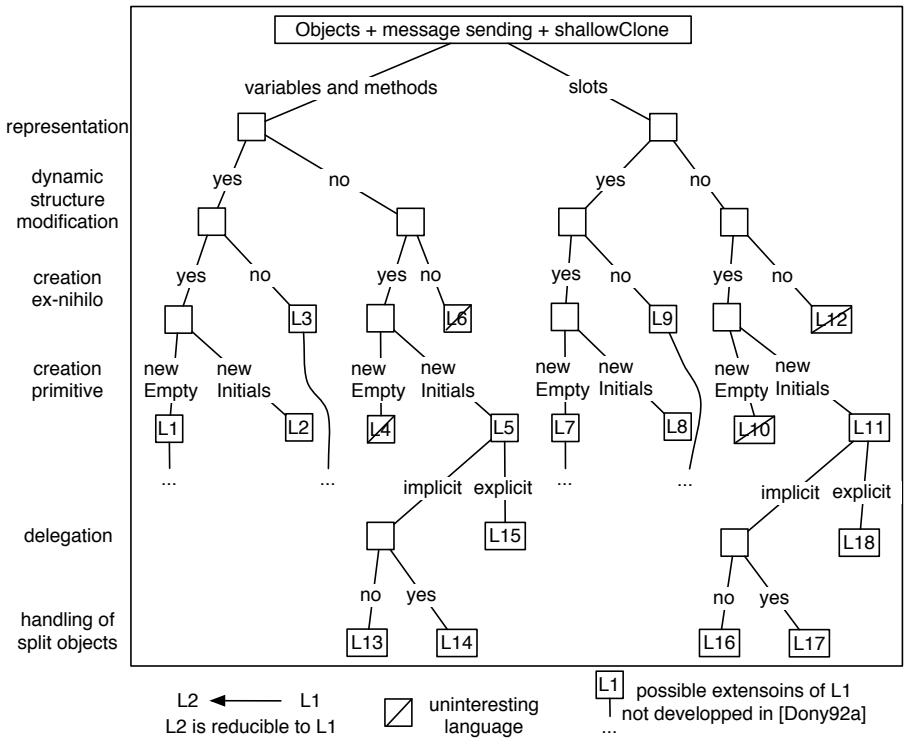


Fig. 1. Classifying prototype-based languages [16]

and ACT-1 [2] [12] have been developed and others such as NewtonScript [13] [14] and GlyphicScript have been used in industrial setting.

While the concept of prototype which results from a desire for simplification and unification seems simple at first glance, prototype-based languages come with different models and mechanisms in key language aspects: knowledge representation, object creation and modification, information sharing, message delegation, method lookup. The Treaty of Orlando was the first attempt at defining more clearly the notion of prototypes and as such acknowledged that understanding the differences and grouping was needed [15]. Later on, various studies have proposed taxonomies and analyses to explain and compare the mechanisms and bring to the fore problems [16] [8] [6] [17] [18] [19]. Figure 1 presents a first taxonomy based on five basic criteria [16]. Most of the languages mentioned in the taxonomy are implemented in Prototalk. Along this paper references to languages named like L1 or L5 refer to the one classified in Figure 1.

Today new prototype-based languages are continuously appearing. Lua [20] is used as a light scripting language. JavaScript with its prototypical instance is still the web-client programming language [21] [22]. More recently IO, a new and extremely compact language with NewtonScript's double inheritance [23], Pic% [24] a new prototype language with mixins, Slate a mix between

Smalltalk and Self [3] and Prothon [25] a mix between Python and Self have been released. New areas such as mobile and distributed applications [26] are discovering the potential benefits of prototype-based languages or of some of the mechanisms or aspects they advocate. Therefore the diversity of prototype-based languages has been and is again quite real. Comparing these languages, being able to design alternative ones, by plugging together adequate aspects, being able to check how far some aspects are compatible is a challenging and useful task since language facets can interact.

The paper presents Prototalk, a framework for implementing interpreters for various prototype-based languages by specializing and composing existing components. The goal of Prototalk is twofold: Firstly, it supports the understanding, classification or teaching of existing languages in a syntactically *uniform* environment. Second it helps designing and testing new languages.

Prototalk follows the approach *"by construction"* promoted by Actalk [27] and ObjVLisp [28] where concurrent languages or reflective class-based ones are decomposed and built from scratch. Like ObjVLisp and Actalk, Prototalk is also an open pedagogical laboratory used to teach object-oriented development. Enabling students compose or design their own language in a concrete way and get an executable language is a pedagogical advantage.

This article is structured as follows. Section 2 presents the goal and the design decision, from the research and pedagogic point of views, that motivated the initial need of a platform to model prototypes-based languages[16]. Section 3 describes the architecture of the framework and shows how the Self and NewtonScript language can be simulated. Section 4 presents the bases for programs interpretation. Section 5 details the step to integrate a new interpreter in the platform. An interpreter for NewtonScript is presented. This language introduced the idea of having two inheritance hierarchies based on two kinds of parent links. Section 6 presents an evaluation of the use of Prototalk.

2 Goals and Design Decisions

While designing the Prototalk platform we had the following goals in mind:

(1) *Uniformity:* The first objective has been to establish an operational semantics for the basic primitives of the basic prototype-based languages and to to unify the languages description to achieve easy comparisons.
(2) *Minimality:* In the tradition of ObjVLisp, Classtalk and Actalk, we wanted through a minimal kernel to express the most general semantics of prototype-based languages,
(3) *Extensibility:* We wanted Prototalk to be a framework so that new lan-

guages simulation can be integrated efficiently and easily.

(4) *Usability:* We didn't want to restrict our system to a semantic model and a raw implementation but we wanted it to be a fully usable environment, including dedicated browsers and inspectors, to achieve actual experiments with prototype-oriented programming.

Prototalk has been initially conceived and used for research purposes but has rapidly been used for pedagogical purposes to teach object-orientation and program interpretation. Its implementation uses all important object-oriented concepts and techniques. It is a framework composed of three main class hierarchies that also uses the Interpreter design pattern. Besides, one of the most interesting points is that it is not so obvious for a student to understand how a class can describe the structure and behavior of objects of a classless world or what distinguishes primitive methods from user-defined methods.

3 The Prototalk Architecture

In Prototalk, a prototype language is represented by an *object model class* which implements all the language primitives and prototype internal representation (See Section 4.2). In addition, the object model class related to a dedicated program node builder which is responsible for emitting the abstract syntax tree of the program (see Figure 2). Programs written in Prototalk are parsed by the Smalltalk parser that generates an abstract syntax tree (AST). The nodes composing this tree are specified by the program node builder associated with the object model. The resulting abstract syntax tree is used to interpret a program following the Interpreter Design Pattern [29].

Prototalk is a framework that can be used at different levels: At the most basic level a user can experiment with languages by combining existing object models provide by Prototalk and existing interpreters. For example, a language L1 making a distinction between the variable and method representation and supporting encapsulation of the variables is created by executing the following expression:

```
(LanguageBuilder new)
   name: #L1
   model: VariableMethodProto
   builder: EncapsulationProgramNodeBuilder
```

The way to define a new language is to (1) define a new object model with its associated program node builder, (2) define new program nodes specifying the semantics of the language, (3) define new primitives and associating them to a language. These tasks are often based on refining or extending existing classes proposed by the framework as shown in Figures 5 and 4 which refers

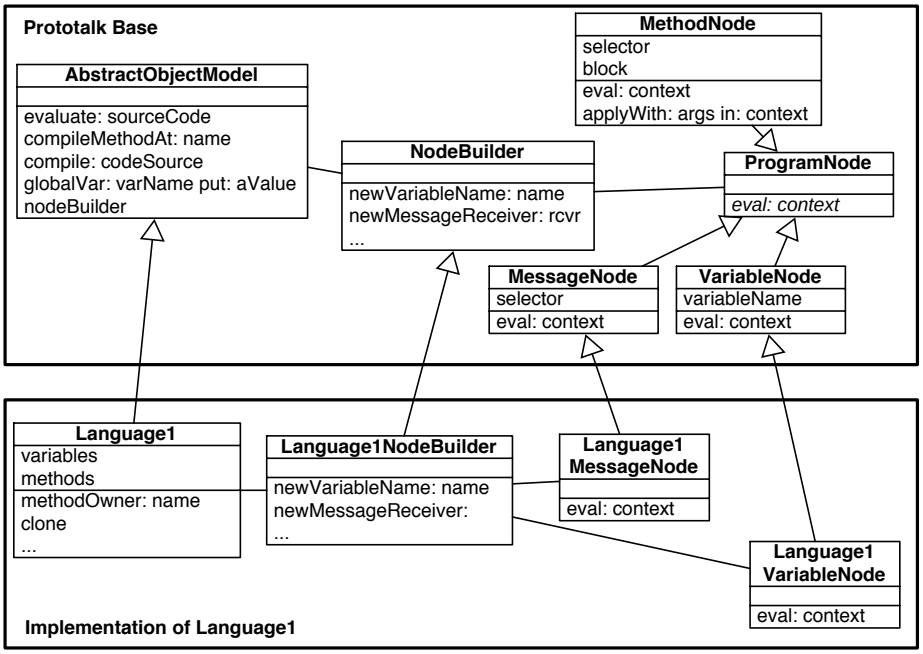


Fig. 2. An object model, its associated program node builder and the corresponding AST elements.

to the taxonomy presented in [16] and in Figure 1. We illustrate these steps in Section 5. Now we present the core elements of the Prototalk architecture.

3.1 Object Model Class

For a given prototype language, the *object model class* has the following responsibilities: (i) it executes programs written in this language by invoking an interpreter on the program nodes, (ii) implements new concepts of the prototype language such as methods or variables lookup, (iii) represents in the Smalltalk memory prototypes created at runtime within a program, and (iv) provides predefined objects such as the empty prototype PRoot or namespace for prototypes.

Executing. A program written in a prototype language is executed by invoking the method evaluate: sourceCode implemented on the class side of the object model class implementing the corresponding language. This method is the starting point of the program interpretation which is based on the Interpreter design pattern [29]. The argument passed to this message is the source code of the program to be executed. The return value of this method execution is the result of the very last statement specified in the program.

The object model class defines all the concepts for a specific prototype language. However, it delegates to the program node builder the responsibility

to assemble a program representation that will then be interpreted (see Section 3.2). A dedicated program node builder is associated with an object model class by the method nodeBuilder. Figure 3 shows a part of the program node builder hierarchy defined in Prototalk.

Prototype Language Primitives. An object model class gathers the different concepts defining the prototype language by implementing primitives related to the following aspects:

- *Object variable and method management.* An object model class has to provide the necessary primitives (i) to add or retrieve methods or variables (*e.g.*, addMethod: code, addVar: name, or addSlot: name) and (ii) to perform some basic object creation operations (*e.g.*, clone or newEmtpy). These primitives are implemented as instances methods defined on the object model class.
- *Method Lookup and Delegation.* Primitives related to message sending and delegation have to be implemented by the object class model through the method methodOwner: name. This method returns the prototype owner of the method name. It defines the lookup semantics. Traditionally it follows the chain of parents objects. This method can be specialized in order to simulate other behaviors such as multiple-parents or double inheritance (Section 5.1).

The method AbstractProto class ≫addPrimitivesOn: defines the primitives that are shared by all the prototype languages. By default any prototype can be cloned, printed and saved in textual format. The method #clone is defined when the structure of the prototype is defined. For example the class SlotProto which unifies variables and methods into slots redefines the primitive #clone. The language L7AndImplicitDelegation which introduces the concept of parent adds two primitives #newSon and #parent.

```
AbstractProto class ≫ addPrimitivesOn: aRoot
    aRoot addPrimitiveNamed: #clone.
    aRoot addPrimitiveNamed: #printOn:.
    aRoot addPrimitiveNamed: #fileIn:.
    ^aRoot

AbstractProto≫ clone
    self subclassResponsibility

L7AndImplicitDelegation class≫ addPrimitivesOn: aRoot
    super addPrimitivesOn: aRoot.
    aRoot addPrimitiveNamed: #newSon.
    aRoot addPrimitiveNamed: #parent:.
    ^aRoot
```

```
SlotProto≫ clone

    | nDic |
    nDic := Dictionary new.
    slots associationsDo: [:each | nDic add: each copy].
    ^ self shallowCopy initMethDic: nDic
```

Prototype Representation. Smalltalk instances of the object model class represent prototype objects. Therefore the object model class defines the common structure of any prototype. For instance prototypes in the SelfLike language are represented as instances of this object model class SelfLike. Proto-

types are Smalltalk objects, therefore Prototalk does not have to deal with the garbage collector.

Predefined Prototypes. Often predefined objects such as the default object to be cloned have to be provided to properly build programs. Usually such a root object offers primitives (cloning, adding variables or methods, ...) accessible to all of its children. For instance the PRoot object is created by the method createRoot defined on the class AbstractProto.

For a given language, PRoot is contained in the global environment and it provides a prototype normally intended to be the root of the object hierarchy.

3.2 Abstract Syntax Tree and Program Node Builder

The method evaluate: sourceCode implemented on the object model class calls the Visualworks Smalltalk parser to generate the abstract syntax tree (AST) of the program source code passed as argument. This AST is composed of nodes obtained from the program node builder associated with object model class. Each language L in our platform is associated with program node builder class that determines, via a set of methods, which kind of interpreter program nodes represent an L expression. This is a hook offered by the Smalltalk compilation framework to specialize the generated AST. The class ProgramNodeBuilder is implemented by the Smalltalk compilation framework. This class emits specific AST nodes when the parser requests it. It offers a common protocol that can be specialized to get different ASTs from the same or different source code. For example, the ProgramNodeBuilder defines the following methods: declareVariableName:, newCascadeReceiver:messages:, newAssignmentVariable:value:, newMessageReceiver:selector:arguments: or newSelfMessageSelector:.

Syntax. In Prototalk, programs are written using a Smalltalk syntax (cf. Section 6). Therefore syntactically, a Prototalk program is a valid Smalltalk one. However, the semantics are different, variable accesses and method invocations depend on the semantics of the language implemented. Most of the nodes composing a prototalk program AST are the Smalltalk ones, except for a few of them that need to be specialized to reflect prototype languages features such as MessageNode and SuperMessageNode for method lookup and VariableNode and AssignmentNode that represent variables.

3.3 Object Model Class Hierarchies

Prototalk heavily uses inheritance to structure and reuse the various aspects of prototype-based languages. Figure 4 gives the hierarchy of object model

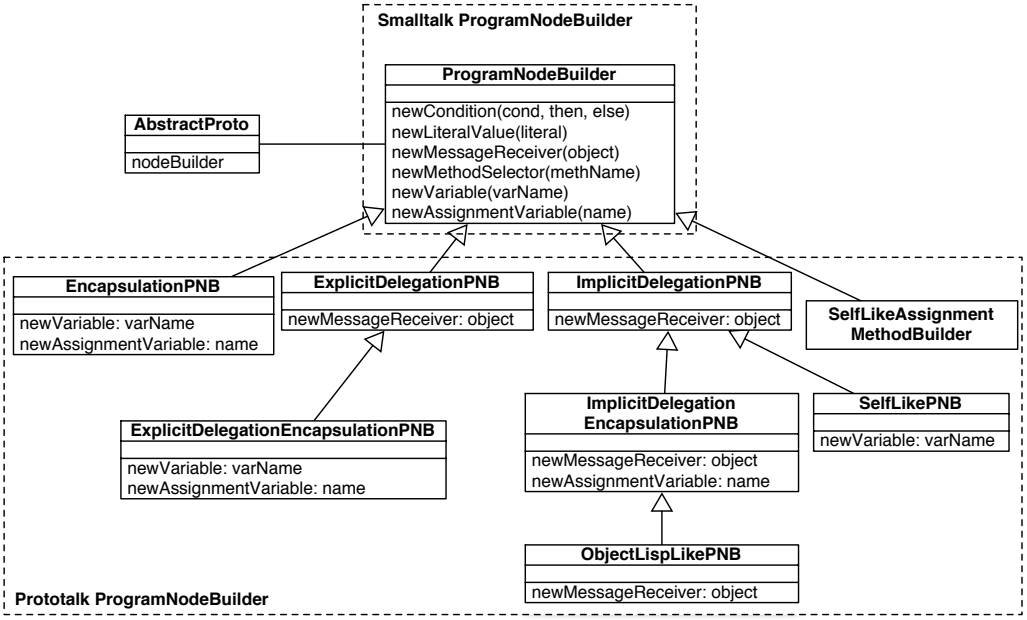


Fig. 3. Hierarchy of the program node builder.

classes representing languages in which prototype objects are defined by a set of slots. For example, the class SlotProto has an instance variable slots and some primitive methods to add new slots (addSlot: name), and to retrieve slots values (slotNamed: name).

Having a class hierarchy for the object models facilitates the extensibility. The definition of a new language is done by simply defining the differences with a previous language. For example the class L11 is a refinement of the class SlotProto that allows prototypes to be created and initialized with a set of slots (method newInitials: slots). The language L11 is refined to the language L11AndImplicitDelegation. The object model class adds an instance variable parent and implements methodOwner: name, parent and newSon. This language is specified by defining the differences with L11.

The object model class hierarchy that defines languages having variables and methods is shown in Figure 5. The class VariableMethodProto defines two instance variables: variables and methods. These variables are initialized in the constructor of this class to an empty set. It also contains all the necessary primitives to manipulate them properly (*e.g.*, addVar: name and addMethod: code to add a variable or a methods). Each prototype objects of this language has its own set of variables and methods.

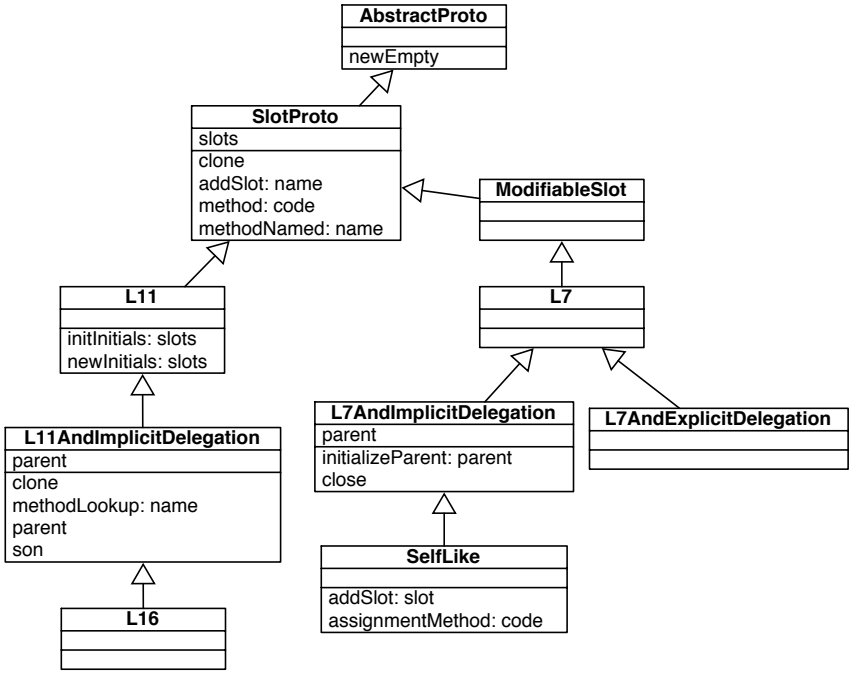


Fig. 4. Hierarchy of object model classes for languages having slots.

4 Basics for Language Interpretation

In Prototalk programs are not compiled, instead interpreted directly from the AST. This AST implements the Interpreter design pattern [29]: each node of the hierarchy is evaluated (through a method eval: context). The interpretation (*i.e.*, execution of a program) starts in its root node. This method is triggered by the object model class when the method evaluate: sourceCode is invoked. Evaluating a node requires a context containing information related to the action performed. The context refers to the receiver of the message currently sent (self), possibly the super one, the variables passed as argument within a method. Access to the self reference is necessary when new variables are defined (*i.e.*, method eval: context of class VariableNode and its subclasses), when new slots are added (methods addSlot: code in the class SelfLike), or when message are sent (method eval: context class MessageNode and its subclasses).

4.1 Variable and Message Nodes

Modeling prototype-based languages has to deal with object representation and message lookup semantics. This means that embedding a prototype language within a class-based language requires the redefinition of the semantics of messages passing and variables lookup. In Prototalk, the AST used to represent prototype program is quite similar to the one offered by Smalltalk except

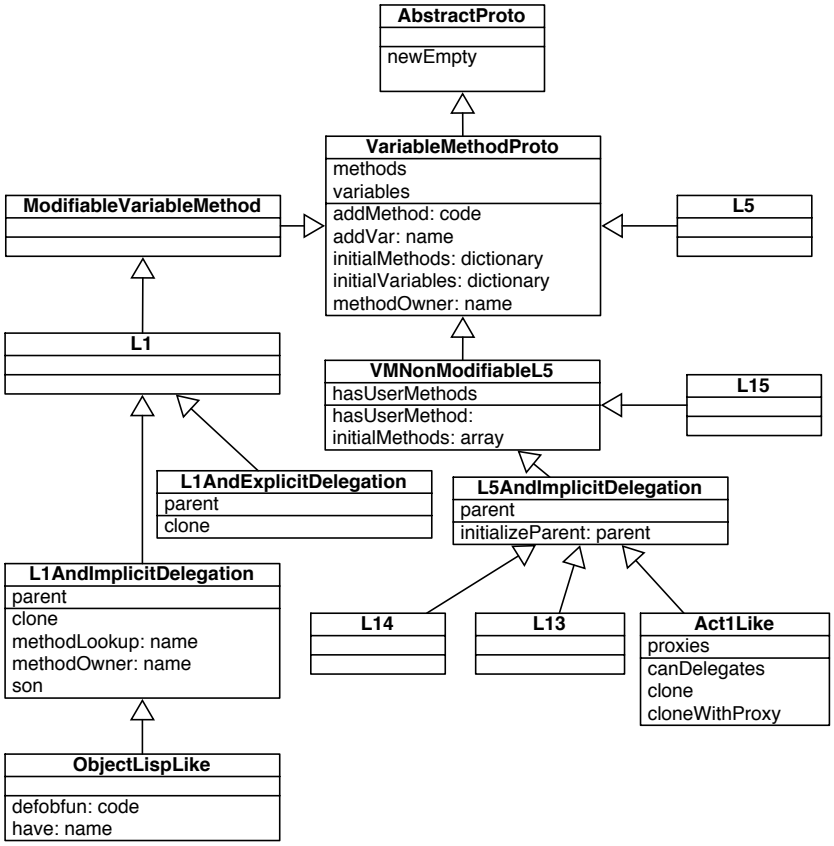


Fig. 5. Hierarchy of object model classes for languages having variables and methods.

that the nodes MessageNode, AssignmentNode and VariableNode have several subclasses to represent the different semantics for sending messages and manipulating a prototype state. Four different message send semantics are implemented in Prototalk: (i) explicit delegation to represent languages like ACT-1 where the delegation has to be done manually (no parent relationship), (ii) implicit delegation to the parents for messages that are not understood by the receiver (*e.g.*, Self), (iii) invocation of a message by invoking the function *ask(receiver, messageName, arguments)* (à la ObjectLisp) and (iv) super call used to explicitly forward a message to the object contained in the *super* slot. Figure 6 shows the specialization of the Smalltalk AST to allow Prototalk to use different language semantics.

4.2 Primitives and Basic Behavior

Primitives refer to the methods implemented on the object model class that can be invoked (by a direct or indirect method call) from within code written in the prototype language implemented by the object model class. While the object model class instance represents prototype, methods defined by the user

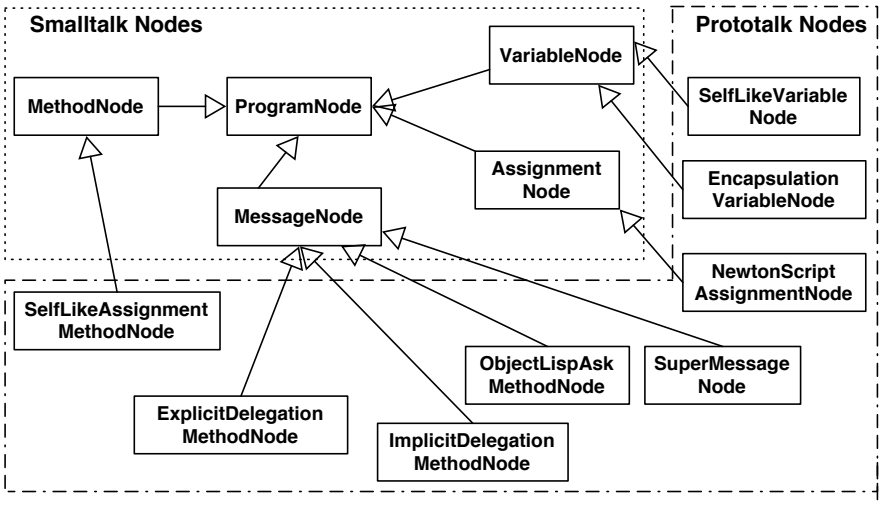


Fig. 6. Specialization of Smalltalk nodes to simulate prototype language behavior.

on these instances are not defined on the object model class. Such methods are stored either in the methods dictionary of an object defined by the class VariableMethodProto or on slot dictionary of an object defined by the class SlotProto (see Figures 5 and 4). These two method holders contain Smalltalk compiled methods in case of primitives, or AST in case of user defined methods.

Some utility methods are provided to enable the programmer to easily manipulate methods such as the method Object≫lookup: name that returns the class implementing a particular method name. In addition the method compiledMethodAt: has been specialized to take care of the fact that the owner of a method is now an object instead of a class (*i.e.*, Smalltalk).

4.3 Default Primitives

The class AbstractProto, which is the root of the object model classes, offers severals methods that can be used as primitives in the prototype-based languages. This class offers eleven primitives.

Four primitives are defined on the instance side of the class AbstractProto. (i) New primitives can be added using addPrimitive: aCompiledMethod named: sel. However it is declared as abstract but it is redefined in its two direct subclasses VariableMethodProto and SlotProto: either to store a primitive in a method or a slot. Global environment for a language can be accessed through a prototype (ii) globalVar: varName put: aValue and (iii) globalVarValue: varName are used for that. (iv) Creation from ex-nihilo (*i.e.*, empty object) is available with newEmpty.

On the class side of AbstractProto seven primitives are offered. Even if these methods are usually not directly invoked during the interpretation of a program they are essential. (i) New primitives are added to the default objects offered to the user (*e.g.*, PRoot) using addPrimitivesOn: aRoot. (ii) New method are added by the user through compile: aString (*e.g.*, this method is indirectly called by addSlot:). (iii) The definition of the PRoot object is contained in the createRoot method. (iv) Compilation and execution of a program is done with evaluate: aStream. The result of this method is the result of the very last statement defined in the program passed as a Stream. The global environment for a language is managed by (v) globalVar: varName put: aValue and (vi) globalVarValue: varName. These methods are invoked by the corresponding method on the instance side. And the reference to the node builder is got by invoking (vii) nodeBuilder.

4.4 Evaluation

Even if programs expressed within Prototalk have a particular semantics, their syntax is always the one of Smalltalk (Section 6). Prototalk extends all of the Smalltalk AST nodes with the bare minimum, that defines a prototype language where: (i) variables do not need to be declared (they are created when first referenced) and (ii) any node of the AST can be particularized using subclassing.

The following classes implement an eval: context methods that simulate the normal Smalltalk behavior (which is identical to Prototalk's *i.e.*, control flow structure): ArithmeticLoopNode, BlockNode, CascadeNode, ConditionalNode, LiteralNode, LoopNode, MessageNode, ReturnNode, SequenceNode.

Differences between Prototalk and Smalltalk only rely on a few nodes points: variable assignment, sending messages, and accessing variables.

As described below, an assignment refers either to binding a new value to a key existing in the context (*i.e.*, self, super, or arguments passed to a method or a block) or creating a new global variable (explained later in this section). Accessing a variable looks at a binding up either in the current context or in the global environment.

```
AssignmentNode≫ eval: context
    | varName val client |
    varName := variable name asSymbol.
    val := value eval: context.
    (context hasLocalVar: varName)
        ifTrue: [ ^ context at: varName put: val ]
        ifFalse:
            [ client := context at: #self.
            ^ client globalVar: varName put: val ]
```

```
VariableNode≫ eval: context
    | client varName |
    varName := name asSymbol.
    ^ context at: varName
        ifAbsent:
            [ client := context at: #self.
            client globalVarValue: varName ]
```

The class SimpleMessageNode is the node that represents a message send. It implements the default message send semantics. Its eval: context method performs the following steps: (i) evaluation of the receiver in the local context, (ii) evaluation of the arguments related to this message (the method evalList: is implemented on the class Collection), (iii) lookup of the prototype that defines the looked up method and if it is not found then raise an error, (iv) if the method retrieved is a CompiledMethod (*i.e.*, if it is a primitive or if it correspond to a Smalltalk message) then the result of the methods is its execution [1] else, (v) the retrieved method is an AST (it corresponds to a method defined by the user) and it is applied to the arguments using a new context where the receiver is bound to #self.

```
SimpleMessageNode≫ eval: context
    | method rec args newContext owner |
    rec := receiver eval: context.
    args := arguments evalList: context.
    owner := rec lookup: selector.
    owner isNil ifTrue: [AbstractProto unknownMethod raiseWith: selector].
    method := owner compiledMethodAt: selector.
    (method isKindOf: CompiledMethod)
        ifTrue: ["this is either a call to a primitive or a smalltalk message"
            ^ method valueWithReceiver: rec arguments: args ].
    "this is a user-defined method"
    newContext := PContext new.
    newContext at: #self put: rec.
    ^ method applyWith: args in: newContext
```

4.5 Application

The classes BlockNode (representing a Smalltalk block closure) and MethodNode (representing a method definition) implement a method applyWith: args in: context invoked when a block or a method has to be evaluated (cf. last example in the previous paragraph).

Applying a block on a set of arguments (i) extends the local context with the arguments, and (ii) evaluate the block's body.

```
BlockNode≫applyWith: args in: context
  1 to: arguments size do: [:i |
                context at: ((arguments at: i) variable name asSymbol)
                        put: (args at: i)].
  ^ body eval: context.
```

If no error occurs during the application of a method then the result is the application of the related block defining a method. In Smalltalk the body of a method is represented as a block closure referenced by the instance variable block.

```
MethodNode≫applyWith: args in: context
```

[1] note that the context is discarded because the method is not interpreted by prototalk

```
^ AbstractProto returnFromSignal
        handle: [:ex | ex returnWith: ex parameter]
        do: [block applyWith: args in: context]
```

4.6 Extensions in basic classes

The classes Object and Behavior offered by Smalltalk has to be extended in order to make them normal smalltalk objects able to receive message from prototypes. The goal is to make Smalltalk objects usable in any language expressed in Prototalk.

Object is extended with a method lookup: selectorName used to return the class contained in the class hierarchy of the receiver that defines a method named selectorName. Methods are retrieved with methodNamed: name.

```
Object≫ lookup: selectorName
    ^ self methodOwner: selector
```

```
Object≫ methodOwner: selectorName
    ^ self class
        smalltalkMethodOwner: selectorName
```

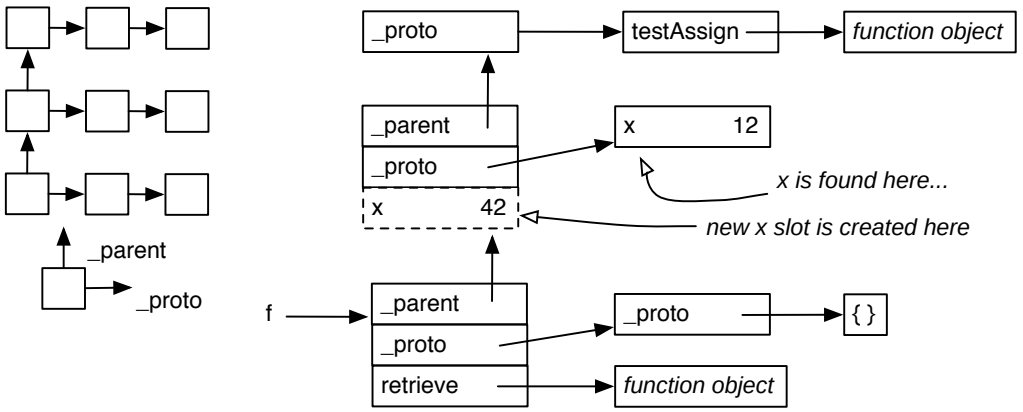
The class Behavior is extended with two methods useful to retrieve a class contained in the super chain of the receiver that defines a method (smalltalkMethodOwner: selector) or a variable (smalltalkVariableOwner: selector).

```
Behavior≫smalltalkMethodOwner: selector
    "Answer the class owner of the method or nil."
    (self includesSelector: selector) ifTrue: [^ self].
    superclass == nil ifTrue: [^nil].
    ^superclass smalltalkMethodOwner: selector
```

```
Behavior≫smalltalkVariableOwner: selector
    "Answer self if the variable is one the receiver variables
    or one of its superclasses variables and nil otherwise."
    (self allInstVarNames includes: selector)
        ifFalse: [^nil].
```

5 Implementing NewtonScript's Double Inheritance Semantics

Prototalk served as a basis to implement the key features of prototype-based languages as discussed. In this section we present how double inheritance of NewtonScript is implemented in Prototalk. We choose NewtonScript since it was one of the few prototype-based language used in production for the Apple Newton PDA [14]. The major design goals of the language were to minimize memory consumption and support graphical user interface definition by introducing an environmental acquisition mechanism [14] [30] that allows sharing between composite graphical objects and their parts.




```
f := {retrieve: func () x, _proto: {_proto: {}}};
f._parent := {_proto: {x: 12}};
f._parent._parent := {_proto: {testAssign: func () x := 42}};
f:testAssign();
Print (f:retrieve());
```

Fig. 7. NewtonScript double inheritance method lookup and variable access.

5.1 NewtonScript

In NewtonScript an object is called a *frame*. A frame element is a slot which can either be an attribute or a function reference. Slots can be inserted or deleted dynamically at run time. New frames can be created from ex-nihilo (empty frame), by cloning and by extending an existing frame.

NewtonScript has a double inheritance scheme in which a prototype link (_proto) is searched prior to a parent one forming a comb-like graph traversal as shown in Figure 7. This specific inheritance supports the building of user-interfaces as well as the minimization of memory [31]. Each frame has a prototype frame from which it is derived (refers to by a slot called _proto). Each time the NewtonScript interpreter does not find a slot variable or function locally it looks into the frame's prototype and then recursively into its prototype until the whole proto chain is searched through. When the slot is not found in the prototype chain of the receiver, it is looked in the parent (following the _parent slot) and the parent prototype chain.

The semantics of assignment is slightly different and supports space consumption minimization: the same lookup occurs to locate the frame containing the variable, but the assignment is only allowed to alter an object in the _parent chain, not the one in a _proto chain. If necessary, a new slot is created in the object on the _parent chain nearest the object where the slot was found as shown in Figure 7. It provides a form on copy-on-write in which the initial value of for a slot comes from the prototype. This way the allocation of the slot to hold a new value is delayed until a new value is actually assigned.

Here is an example of NewtonScript code and its implementation in Prototalk. Even if syntactically the two programs are different, their execution leads to the same objects and construction.

```
Point := {
        x : 0,
        y : 0,
        move: func (newX, newY)
                        begin
                                x := newX;
                                y := newY;
                         end,
        same : func (pt)
                        begin
                         (pt.x = x) and (pt.y = y)
                         end
};
```

```
Point := PRoot clone.
Point addSlot: 'x = 0'.
Point addSlot: 'y = 0'.
Point addSlot: 'moveToX: newX andY: newY
                                x := newX.
                                y := newY.'.
Point addSlot: 'same: p
                        (p x = x) & (p y = y)'.
```

5.2 In Prototalk

The first step is to create the object model class NewtonScriptLike as shown below. Because of the similarity between Self and NewtonScript (implicit delegation, reference to a parent object, slot management, ...) the object model class inherits from the class SelfLike. In NewtonScript a frame has a parent (instance variable inherited from SelfLike) and a prototype named _proto instance variable which has to be defined in the object class model.

```
Smalltalk defineClass: #NewtonScriptLike
        superclass:  #{Smalltalk.SelfLike}
        instanceVariableNames: 'proto '
```

The method methodOwner has to return the object that implement the looked up method. The NewtonScript's method lookup semantics described previously is specified by specializing the method methodOwner: name as follow: if the current frame provides an implementation of a name method then return this frame, otherwise check if one of the frame's prototypes implements it. If not then we lookup in the frame's parent. The lookup in the prototype chain is simply done by running over it: for each frame it is checked if it implements a method named name. Note that this lookup does not have to run over the parent link.

```
NewtonScriptLike≫methodOwner: name
  "return the owner of the method or nil"

 ^ (self includesSelector: name)
       ifTrue: [ self ]
       ifFalse: [ | t |
          t := self methodOwnerInProtoChain: name.
          t isNil
             ifTrue: [
                parent isNil
                  ifTrue: [ nil ]
                 ifFalse: [ parent methodOwner: name ]]
             ifFalse: [ t ]]
```

```
NewtonScriptLike≫methodOwnerInProtoChain: name

 ^ (slots includesKey: name)
       ifTrue: [ self ]
       ifFalse: [
          proto isNil
             ifTrue: [ nil ]
             ifFalse: [ proto methodOwner
                              InProtoChain: name ]]
```

The class NewtonScriptLike has to provide an accessor and a mutator for its instance variable proto (code not shown). These have to be accessible from a program intended to be interpreted. Therefore two primitives have to be added to the PRoot object which is the first prototype from which others can be cloned. By having the primitives proto and proto: defined in PRoot, they are accessible from each object.

```
NewtonScriptLike class≫addPrimitivesOn: aRoot
      super addPrimitivesOn: aRoot.
      aRoot addPrimitiveNamed: #proto.
      aRoot addPrimitiveNamed: #proto:.
      ^ aRoot
```

The object model class has to implement a method nodeBuilder that returns the *class* of its associated node builder (NewtonScriptPNB).

```
NewtonScriptLike class≫nodeBuilder
     ^ NewtonScriptPNB
```

Program Node Builder. The class NewtonScriptPNB inherits from ProgramNodeBuilder. It defines the program node builder used to yield proper nodes when requested by the Smalltalk parser. When the Smalltalk parser parses an assignment, the method newAssignmentVariable: var value: val leftArrow: bool is invoked on the program node builder. A proper node representing an assignment has to be emitted.

```
NewtonScriptPNB≫newAssignmentVariable: var value: val leftArrow: bool
     ^ NewtonScriptAssignmentNode new variable: var value: val leftArrow: bool
```

Sending a message has to trigger the method lookup algorithm described previously in the method NewtonScriptLike≫methodOwner:. Here we reuse the behavior of the class ImplicitDelegationMethodNode which implements a lookup of method and the use of super. The method ImplicitDelegationMethodNode≫eval: invokes the methodOwner: that is used to execute properly methods in presence of the parent slot. When invoked by the parser, the method newMessageReceiver: rcvr selector: sel arguments: args creates then an ImplicitDelegationMethodNode node.

```
NewtonScriptPNB≫newMessageReceiver: rcvr selector: sel arguments: args
     ^ ImplicitDelegationMethodNode new
```

```
            receiver: rcvr
            selector: sel
            arguments: args
```

Accessing a variable in NewtonScript has the same meaning as in Self: accessing a variable is equivalent to trigger its accessor. Referencing a variable foo is equivalent to sending the message foo to itself. On that point NewtonScript behaves in the same way as in Self. Therefore we reuse the SelfLikeVariableNode behavior as follows:

```
NewtonScriptDelegationPNB≫newVariableName: nameString
      ^SelfLikeVariableNode new name: nameString
```

Variable Assignment. The class NewtonScriptAssignmentNode is a subclass of AssignmentNode. It implements the methods eval: context that defines semantics of assigning a value to a variable in NewtonScript. This method requires a context as argument containing the references to *self* and to the temporary local variables.

```
NewtonScriptAssignmentNode≫eval: context
      | varName val client  owner |
      varName := variable name asSymbol.
      val := value eval: context.
      ^  (context hasLocalVar: varName)
            ifTrue: [ context at: varName put: val ]
            ifFalse:
                  [ client := context at: #self.
                   self inAMethod
                        ifTrue: [ owner := client methodOwnerInParentChain: varName.
                                    owner isNil
                                          ifTrue: [ client addSlot: (variable name, ' = ', val printString) ]
                                          ifFalse: [ owner addSlot: (variable name, ' = ', val printString) ]]
                         ifFalse: [ client globalVar: varName put: val]]

NewtonScriptLike≫methodOwnerInParentAndProtoChain: slotName
       "Return the first frame in the parent hierarchy that has (or one of its proto)
        a slotName defined"
   (slots includesKey: slotName)
       ifTrue: [ ^ self ]
       ifFalse: [
           proto notNil
               ifTrue: [ (self methodLookupInProtoChain: slotName) notNil
                           ifTrue: [ ^ self ]].
           parent isNil
               ifTrue: [ ^ nil ]
               ifFalse: [ ^ parent methodOwnerInParentAndProtoChain: slotName ]]
```

First the right branch of the assignment (designed by the value instance variable) is evaluated using the current context. Then, if the left branch of the assignment, the variable named varName), is already defined in the context, it is a temporary variable. Then its value in it is updated. If not, a new frame variable or a global variable is created regarding if this assignment occurs in a method or not.

If the left branch of the assignment occurs in a method and if varName refers no local or global variable, then varName is related to the state of the current

frame (self). According to the semantic given by NewtonScript a new variable is created in the current frame if none of its prototypes already has a slot named varName. If one of these already defines varName, then its value is updated by the value of the assignment.

This implementation of NewtonScript double inheritance shows how already existing semantical elements provided by the framework can be reused or redefined.

6 Evalutation

Prototalk is ideally suited for teaching concepts of object-oriented languages and has been successfully incorporated into the masters degree program at the University of Paris VI, Montpellier and Berne. It served as a tool to help teaching object-oriented languages and software engineering as well as program interpretation. Here a synthesis of what we learned from these experiments.

Prototalk has first proved to be a very interesting pedagogical tool although we did not perform registered and controlled experiments to evaluate and compare its impact. Firstly, it is an obvious help to explain what prototypes are and to ask students to invent or explain various languages. Secondly, its implementation is an interesting laboratory because it uses important object-oriented concepts and techniques. It is primarily a classical framework made of three main class hierarchies (object models, program nodes and program node builder) conceived to be extended. It also uses the Interpreter Design Pattern and extends it in various ways as such it could be called "Interpreter Family" and still remains to be described. Prototalk explicitly separates the object model from the instructions and their associated semantics in a language. Then it uses class specialization and composition to express the fact that various semantics can, in different languages, be associated to a single syntactic construction. For example each subclass of DelegationMessageNode and their respective eval: methods define one semantics of delegation whose study is an excellent way to present the semantics of delegation in prototype-based languages. Another very interesting point is the study of the AbstractProto class and subclasses. This class defines the structure and behavior of classless objects. The study of that apparent paradox (a class been used to describe and represent classless objects) is a key point for the understanding of language implementations and of reflective languages. Methods defined on AbstractProto are primitives of the classless languages. Within such a methods, an instance of VariableMethodProto for example, is a Smalltalk object but the same physical entity is a prototype in the implemented language.

From a research perspective Prototalk has been used to build the taxonomies and the evaluations presented in [16] [18] [19]. Most of important feature of classless languages have been simulated with Prototalk.

Features and Languages Implemented. In addition to the NewtonScript's double inheritance, we modeled in Prototalk the following language facets.

- The proxy link of ACT-1 [2] and its associated explicit delegation mechanism. Cloning was then specialized to also clone also the object proxy.
- ObjectLisp delegation. ObjectLisp, which was a layer on top of Allegro Lisp and presented as a class language, was underneath a prototype language inspired by Lieberan his. In ObjectLisp sending a message was performed using the construct ask. For example the (ask point (have 'y 4)) sends the message have to the object refer to the variable point. The receiver could be any expression, therefore it should be interpreted in the current context, but the arguments are executed in a new environment in which self is bound to the current receiver. The method ObjectLispAskMessageNode≫eval: context makes clear this semantical point.
- Self's dynamic multiple inheritance. Depending on the versions of Self, an object could have multiple parents and their priority was denoted using *. In the current version of Self (4.1), only one parent is used. In addition, the parent can be changed dynamically.
- Examplars [4] mixed classes and prototypes hierarchies. Exemplars was a prospective language that proposed a dichotomy between a subtyping hierarchy made of classes and a reuse hierarchy made of prototypes. In fact Exemplar is very interesting because its classes were an exact intuition of Java interfaces.

```
ObjectLispAskMessageNode≫eval: context
    "context contains self which is the initial receiver in the current delegation
    chain, that is the client and curRec the current receiver."

    | rec newContext result |
    rec := receiver eval: context.
    newContext := PContext new.
    newContext at: #self put: rec.
    arguments do: [:each | result := each eval: newContext].
    ^ result
```

As we already mentioned in Section 3 and illustrated during the implementation of NewtonScript's double inheritance, Prototalk is a framework that provides predefined abstractions that can be reused, extended and combined to create different languages. At a structural level, three different elements are offered: objects with slots, objects with variables and methods, and objects with parent. In addition, object internal representation can be fixed or modifiable. From a behavioral perspective, Prototalk offers 4 different message passing semantics as we explained before, cloning and object creation with initial values are also offered.

About Influence of the Syntax. Prototalk is a platform supporting the coexistence of several prototype languages. Each language has the ability to execute programs written in this language, to debug programs and to test them. Each language expressed can reuse all the facilities provided by the classical Smalltalk environment. Using the Smalltalk syntax has the following advantages:

- Users can experiment with several conceptually different languages without having to learn several syntaxes.
- By removing all the syntactic decorations, conceptual values are clearly offered without being obfuscated.
- To not deal with lexical and syntactical parser keeps the implementation of Prototalk simple and the implementation of new language simple too. This helps to make its kernel focus only on essential mechanisms.

Code expressed in a prototype language has to have a Smalltalk syntax. This means that computation is described by sending messages where a receiver is explicitly designated. For instance a language where messages implicitly sent to self is implementable but the implementer should use a naming convention to distinguish self sends from implicit self-sends à la Self.

7 Conclusion and Further Work

Thanks to new application domains such as web programming, mobile and distributed computing, various kind of programming languages such as scripting, dynamically typed, or object classless languages are subject to a renewal of interest.

In this paper we have described Prototalk, a Smalltalk framework dedicated to the implementation of prototype-based language interpreters. We have presented it as a research as well as a pedagogical tool. Prototalk has been conceived and used in the nineties to evaluate and operationnaly compare into an uniform environment prototype-based languages such as ACT-1, ObjectLisp, Kevo or Self, considered as possible alternatives to class-based languages somehow and sometimes judged too complex or rigid. All the important facets of prototype-based languages are implemented and usable in a syntactically uniform environment that garanties for easy comparisons and classifications.

The paper precisely describes the Prototalk framework architecture based on an extension of the Interpreter design Pattern and its Smalltalk implementation. It explains the value of using an object-oriented programming language was paying off from a research and pedagogical point of view, as it supported well incremental definition and the possibility of reusing what is already exis-

ting. It demonstrates why using Smalltalk, among other advantages, enabled us to easily implement that design pattern due to its fully integrated parser.

The paper illustrates Prototalk reuse capabilities by presenting the integration of the NewtonScript double inheritance semantics in the framework. It also illustrates why Prototalk has been successfully used as a pedagogical tool to teach object-oriented concepts and technologies.

The perspectives of this environment are numerous. Firstly, we plan to integrate in the platform interpreters for those new languages such as IO, Pic%, Slate or Prothon. Secondly, as Smalltalk only supports single inheritance and since our classifications are not only ontological but also operationnal, we faced the tyranny of the dominant decomposition. Indeed we had to follow a main decomposition and sometimes this forces us to duplicate the behavior. Ideally we would have prefer to have well-identified behaviors that we could freely compose together. Investigating the use of traits [32] could be a way to solve this problem and rely less on inheritance to model the languages. More generally, all new techniques which plug together components or aspects could be of interest to improve that work and enable it to evolve. Finally, it appeared to us that designing a similar environment for another family of languages (*e.g.*, frame languages or component-based languages) should not be difficult and that a generalisation is somehow possible.

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the projects "Tools and Techniques for Decomposing and Composing Software" (SNF Project No. 2000-067855.02) and "Recast: Evolution of Object-Oriented Applications" (SNF 2000-06165-5.00/1). We also thank Gabriela Arévalo and Orla Greevy for their feedback on the paper.

References

[1] A. H. Borning, Classes versus prototypes in object-oriented languages, in: Proceedings of the ACM/IEEE Fall Joint Computer Conference, IEEE Computer Society Press, 1986, pp. 36–40.

[2] H. Lieberman, Using prototypical objects to implement shared behavior in object oriented systems, in: Proceedings OOPSLA '86, Vol. 21, 1986, pp. 214–223.

[3] D. Ungar, R. B. Smith, Self: The power of simplicity, in: Proceedings OOPSLA '87, Vol. 22, 1987, pp. 227–242.

[4] W. R. LaLonde, Designing families of data types using exemplars, Transactions on Programming Languages and Systems 11 (2) (1989) 212–248.

[5] D. Ungar, C. Chambers, B.-W. Chang, U. Holzle, Organizing Programs without Classes, LISP and SYMBOLIC COMPUTATION 4 (3).

[6] P. Steyaert, W. De Meuter, A marriage of class- and object-based inheritance without unwanted children, in: W. Olthoff (Ed.), Proceedings ECOOP '95, Vol. 952 of LNCS, Springer-Verlag, Aarhus, Denmark, 1995, pp. 127–144.

[7] W. De Meuter, Agora: The story of the simplest MOP in the world — or — the sche me of object–orientation, in: Prototype-based Programming, Springer-Verlag, 1998.

[8] A. Taivalsaari, Delegation versus concatenation or cloning is inheritance too, OOPS Messenger 6 (3) (1995) 20–49.

[9] L. Cardelli, A language with distributed scope, Computing Systems 8 (1) (1995) 27–59.

[10] B. Myers, D. Giuse, R. Dannenberg, B. Vander Zanden, D. Kosbie, E. Pervin, A. Mickish, P. Marchal, Garnet: Comprehensive support for graphical highly-interactive user interfaces, IEEE Computer 23 (11) (1990) 71–85.

[11] P. Mulet, J. Malenfant, P. Cointe, Towards a methodology for explicit composition of metaobjects, in: Proceedings of OOPSLA '95, Austin, 1995, pp. 316–330.

[12] H. Lieberman, Delegation and inheritance: Two mechanisms for sharing knowledge in object-oriented systems, Bigre + Globule 48 (1986) 79–89.

[13] W. R.Smith, The newton application architecture, in: Proceedings of the 39th IEEE Computer Society International Conference, 1994, pp. 156–161.

[14] W. Smith, Using a prototype-based language for user interface: The newton project's experience, in: Proceedings of OOPSLA '95, ACM, 1995, pp. 61–73.

[15] L. A. Stein, H. Lieberman, D. Ungar, A shared view of sharing: The treaty of orlando, in: Object-Oriented Concepts, DataBases, and Applications, ACM Press, Addison Wesley, 1989, pp. 31–48.

[16] C. Dony, J. Malenfant, P. Cointe, Prototype-based languages: From a new taxonomy to constructive proposals and their validation, in: Proceedings OOPSLA '92, 1992, pp. 201–217.

[17] J. Malenfant, On the semantic diversity of delegation-based programming languages, in: Proceedings of OOPSLA '95, ACM, Austin, 1995, pp. 215–230.

[18] D. Bardou, C. Dony, Split objects: a disciplined use of delegation within objects, in: Proceedings of OOPSLA '96, 1996, pp. 122–137.

[19] C. Dony, J. Malenfant, D. Bardou, Classification of object-centered languages, in: J. Noble, A. Taivalsaari, I. Moore (Eds.), Prototype-based Programming: Concepts, Languages and Applications, Springer Verlag, 1998, pp. 17–45.

[20] R. Ierusalimschy, L. H. de Figueiredo, W. C. Filho, Lua — an extensible extension language, Software: Practice and Experience 26 (6) (1996) 635–652.

[21] ECMAScript Language Specification, European Computer Machinery Association, 1997.

[22] D. Flanagan, JavaScript: The Definitive Guide, 2nd Edition, O'Reilly & Associates, 1997.

[23] Io home page, http://www.iolanguage.com/.

[24] J. D. Wolfgang De Meuter, Theo D'hondt, Pic% intersecting classes and prototypes, in: Andrei Ershov Fifth International Conference on Perspectives of System Informatics, Siberia, Russia, 2003.

[25] Prothon home page, http://www.prothon.org/.

[26] W. D. M. Jessie Dedecker, Using the prototype-based programming paradigm for structuring mobile applications, in: Workshop: Agent-oriented methodologies. Proceedings of OOPSLA 2002, Seattle, WA USA., 2002.

[27] J.-P. Briot, P. Cointe, Programming with explicit metaclasses in Smalltalk-80, in: Proceedings OOPSLA '89, ACM SIGPLAN Notices, Vol. 24, 1989, pp. 419–432.

[28] P. Cointe, Metaclasses are first class: the objvlisp model, in: Proceedings OOPSLA '87, ACM SIGPLAN Notices, Vol. 22, 1987, pp. 156–167.

[29] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, Reading, Mass., 1995.

[30] J. Gil, D. H. Lorenz, Environmental Acquisition – A new inheritance-like abstraction mechanism, in: Proceedings of OOPSLA'96, 1996, pp. 214–231.

[31] J. Noble, J. Potter, J. Vitek, Flexible alias protection, in: E. Jul (Ed.), Proceedings ECOOP '98, Vol. 1445 of LNCS, Springer-Verlag, Brussels, Belgium, 1998.

[32] N. Schärli, S. Ducasse, O. Nierstrasz, A. Black, Traits: Composable units of behavior, in: Proceedings ECOOP 2003, Vol. 2743 of LNCS, Springer Verlag, 2003, pp. 248–274.

Uniform and Safe Metaclass Composition ⋆

Stéphane Ducasse [a] Nathanael Schärli [a] Roel Wuyts [b]

[a] *Software Composition Group, IAM-Universität Bern, Switzerland*

[b] *Decomp Laboratory, Université Libre de Bruxelles, Belgium*

Abstract

In pure object-oriented languages, classes are objects, instances of other classes called *metaclasses*. In the same way as classes define the properties of their instances, metaclasses define the properties of classes. It is therefore very natural to wish to reuse class properties, utilizing them amongst several classes. However this introduced *metaclass composition problems*, *i.e.*, code fragments applied to one class may break when used on another class due to the inheritance relationship between their respective metaclasses.

Numerous approaches have tried to solve metaclass composition problems, but they always resort to an *ad-hoc* manner of handling conflicting properties, alienating the meta-programmer. We propose a *uniform* approach that represents class properties as *traits*, groups of methods that act as a unit of reuse from which classes are composed. Like all the other classes in the system, metaclasses are composed out of traits. This solution supports the reuse of class properties, and their *safe* and *automatic* composition based on *explicit* conflict resolution. The paper discusses traits and our solution, shows concrete examples implemented in the Smalltalk environment *Squeak*, and compares our approach with existing models for composing class properties.

Key words: Metaclass composition, traits, reflective kernel, reuse, mixins

1 Reusing class properties

In class-based object-oriented programming, classes are used as instance generators and to implement the behavior of objects. In object-oriented languages

⋆ We gratefully acknowledge the financial support of the Swiss National Science Foundation for the projects "Tools and Techniques for Decomposing and Composing Software" (SNF Project No. 2000-067855.02) and "Recast: Evolution of Object-Oriented Applications" (SNF 2000-061655.00/1).

such as CLOS, Smalltalk or Ruby, classes themselves are first-class objects, and instances of so-called *metaclasses* [1,2,?,5,7]. In the same way that classes define the properties for their instances (objects), metaclasses implement the properties for their instances (classes). Examples of class properties are *Singleton*, *Final*, *Abstractness* ... [8].

Treating classes as first-class objects and having metaclasses is important for two main reasons:

- *Uniformity and Control.* In a pure object-oriented language it is natural for classes to be instances of metaclasses. The uniformity defines metaclasses as the natural place to *specify* and *control* object creation and other class behavior.
- *Reuse of Class Behavior.* Since a metaclass is just like any other class, class behavior is reused and conventional reuse and decomposition techniques are applied to the metaclasses [8]. Hence the same techniques that are available for base classes (inheritance and overriding of methods, for example) are applicable at the meta level.

When a language has metaclasses, those metaclasses can be *implicit* or *explicit*. With *implicit* metaclasses the programmer cannot specify the metaclass for a class [9]. As such, implicit metaclasses successfully address the goal of "uniformity and control", but they fall short for achieving "reuse of class behavior". *Explicit metaclasses* avoid this limitation because the programmer can explicitly state from which metaclass his or her classes are instances [1,2,4,5].

Languages without explicit metaclasses suffer from the fact that class properties cannot be reused across classes, and that they cannot be combined. For example, every time one needs a class with the Singleton behavior, the same code needs to be implemented over and over again. With explicit metaclasses the singleton class property can be factored out to a Singleton metaclass, which can then be used to instantiate classes that exhibit the Singleton behavior. However languages with explicit metaclasses suffer from the fact that composition can be *unsafe* [2,10] or are based on *non-uniform* mechanisms *i.e.*, the meta-programmer cannot use the same composition mechanism used for programming at the base level than for programming at the meta level. This is clearly a problem, since metaclasses originate from the wish of uniformity in OOP (see Section 2).

To address these problems we propose to use the *general-purpose* object-oriented language feature *traits* [11]. Traits are composable units of behavior that close the large conceptual gap between a single method and a complete class. Our approach models class properties with traits, and uses trait composition to safely combine and reuse properties in metaclasses. Consequently, metaclass composition (like class composition) enjoys all the conceptual ben-

efits of the traits composition model. In particular, composition conflicts that occur when composing two properties that do not quite fit together are detected automatically and the conflict resolution is *explicit* and under *control* of the composing entity.

As we will show in the rest of the paper, our solution supports the reuse of class properties, their safe and *automatic* composition with *explicit* conflict resolution, and the usage of the same mechanism (traits) for both the base and metalevel. As safety is a broad term we follow the definition of safe metaclass composition as defined in [12] and that we present in the following section. Now we start by identifying precisely

2 Explicit Metaclass Problems

Having explicit metaclasses promotes reuse but introduces several problems summarized in this section and detailed in the rest of the paper.

Unsafe Composition. Some approaches sacrifice the compatibility between the class and the metaclass level [2,10]. Unsafe metaclass composition means that code fragments applied to a class may break when used on another one due to the inheritance relationship between the metaclasses of the classes involved (See section 3).

Ad-Hoc and Non-Uniformity. There are some approaches that are specifically designed to avoid the compatibility problems raised in the first point. Their solutions, however, rely on *ad-hoc* composition mechanisms that are based on automatic code generation and dynamically changing the meta-metaclass [12]. Not only does this make it hard to understand the resulting code, it also leads to problems in case of conflicting properties and results in hierarchies that are fragile with respect to changes. Note that MetaclassTalk by using mixin composition at the metalevel is the only solution that solves this problem [15].

The solutions are not satisfactory from a conceptual point of view either, because the meta level (or meta meta level) does not employ object-oriented techniques (such as inheritance or instantiation) but *ad-hoc* mechanisms only applicable for metaclass composition. This breaks the fundamental idea of reflective programming that uses the *available* features of a language to define and control the behavior of the language itself [4].

Limited Composition. Other approaches used in the specific context of metaobjects use *chain of responsibility* [13] or composite metaobjects [14] to compose metaobjects. The first approach does not provide full control over the composition. The second approach forces the programmer to develop specific metaobjects to compose others, even when the reuse of these composite metaobjects is unclear.

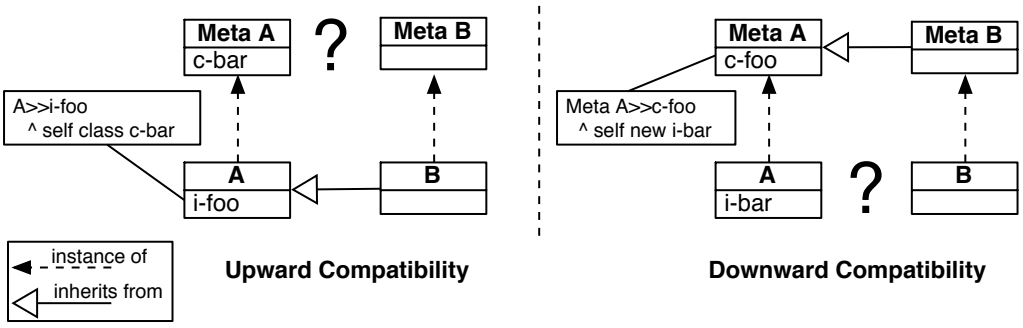


Fig. 1. Left: Upward compatibility - dependencies on the base level need to be addressed at the meta level. Right: Downward compatibility - dependencies on the meta level need to be addressed at the base level.

The ideal metaclass composition solution would make composition be *automatic*. However, as we will discuss in Section 11 a simple solution does not exist in a context where new properties can be defined and composed, and where their semantics can severely conflict. So the solution is a mechanism that is both safe and uniform *i.e.*, one that does not require the developer to make a paradigm shift and where the development of base-level applications and meta-level applications is the same.

3 Qualifying Composition

Offering explicit metaclasses is a way to reuse class properties but it also opens the door for *metaclass compatibility problems* [10]. This section defines criteria by which approaches that solve metaclass composition problems can be characterized and distinguished. We start by listing two criteria that were already identified in [12] (*upward*, *downward* compatibility and *per class property*), and then introduce three new ones that were not previously considered (*property composition*, *property application*, and *control of the composition*), but that qualify the problem in a more detailed way.

Upward Compatibility. The fact that classes are instances of other classes which define their behavior introduces hidden dependencies in the inheritance relationships between the classes and their metaclasses. Careless inheritance at one level (be it the class or metaclass level), can break inter-level communication. N. Bouraqadi et al. [12] refined the metaclass compatibility problems in two precise cases named *upward* and *downward* compatibility.

Let B be a subclass of A, MetaB the metaclass of B, and MetaA the metaclass of A. Upward compatibility is ensured for MetaB and MetaA iff: every possible message that does not lead to an error for any instance of A, will not lead to an error for any instance of B.

Figure 1 left illustrates upward compatibility. When an instance of B receives the message i-foo, the message c-bar is sent to B. The composition of A and B is upward compatible, if B understands the message c-bar, *i.e.*, MetaB should implement it or somehow inherit it from MetaA.

Downward Compatibility. *Let MetaB be a subclass of the metaclass MetaA. Downward compatibility is ensured for two classes B, instance of MetaB and A, instance of MetaA iff: every possible message that does not lead to an error for A, will not lead to an error for B.*

Downward compatibility is illustrated in Figure 1 right. When B receives the message c-foo, the message i-bar is sent to a newly created instance of B. The composition of MetaA and MetaB is downward compatible, if that new instance of B understands the message i-bar, *i.e.*, B should implement it or somehow inherit it from A.

Definition. Metaclass composition is *safe* when it supports downward and upward compatibility.

Per Class Property. Different metaclass properties can be assigned to different classes in an inheritance hierarchy. Some systems such as NeoClasstalk and MetaClasstalk allow one to assign a property to a *single* class without it being inherited by its subclasses [12,15]. The authors of NeoClasstalk and MetaClasstalk, N. Bouraqadi et al. defined *class property propagation* as follows: "A property assigned to a class is automatically propagated to its subclass.". We name this criteria *per class property*. For example it is possible to define that a class is abstract and its subclasses are not abstract and this without having to redefine the property at the subclasses level.

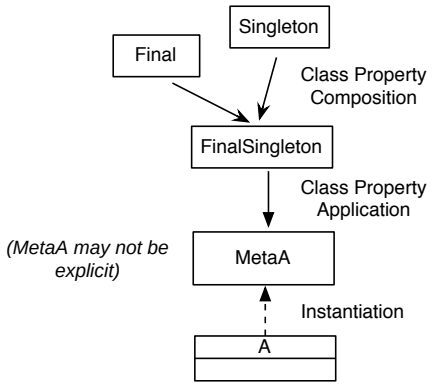


Fig. 2. Property Composition and Property Application: two different stages in the process of reusing class properties.

Property Composition. One of the main motivations for having explicit metaclasses is to *combine class properties*, as shown in Figure 2, so that one class can for example be both a Singleton and Final. Hence a mechanism

is needed that supports such property composition. This can be a general-purpose language mechanism such as multiple inheritance [4,5], mixin composition [15], chain of responsibility [13], or an *ad-hoc* mechanism such as generation of new classes and methods [12].

Property Application. Property application is the mechanism by which the composed properties are applied to classes. As shown in Figure 2 we distinguish the *composition* of properties from the *application* of a property to a specific class because some approaches employ different techniques for these two purposes. As an example, SOM uses ordinary multiple inheritance to compose class properties but it employs a combination of multiple inheritance and code generation to apply a class property to a class.

Control. The mechanism used to apply and combine class properties can be *implicit* or *explicit*. We call the mechanism *implicit* if the system automatically combines or applies the class properties and implicitly resolves conflicts in a way that may or may not be what the programmer intends. We call the mechanism *explicit* if the system gives the programmer explicit control over how the properties are combined and applied. In particular, the programmer should have *explicit control* over how conflicts are resolved. For many approaches, this is not the case because the composition of properties is based on a chain of responsibility which does not provide full control of the composition.

4 Traits in a Nutshell

Traits [11] are an extension of single inheritance with a similar purpose as mixins but avoiding their problems. Traits are essentially groups of methods that serve as building blocks for classes and are primitive units of code reuse. As such, they allow one to factor out common behavior and form an intermediate level of abstraction between single methods and complete classes. A trait consists of *provided methods* that implement its behavior, and of *required methods* that parameterize the provided behavior. Traits cannot specify any instance variables, and the methods provided by traits never directly access instance variables. Instead, required methods can be mapped to state when the trait is used by a class.

With traits, the behavior of a class is specified as the composition of traits and some *glue methods* that are implemented at the level of the class. These glue methods connect the traits together and can serve as accessor for the necessary state. The semantics of such a class is defined by the following three rules:

- *Class methods take precedence over trait methods.* This allows the glue meth-

ods defined in the class to override equally named methods provided by the traits.
- *Flattening property.* A non-overridden method in a trait has the same semantics as the same method implemented in the class.
- *Composition order is irrelevant.* All the traits have the same precedence, and hence conflicting trait methods must be explicitly disambiguated.

Because the composition order is irrelevant, a *conflict* arises if we combine two or more traits that provide identically named methods that do not originate from the same trait. Traits enforce explicit resolution of conflicts by implementing a glue method at the level of the class that overrides the conflicting methods, or by *method exclusion*, which allows one to exclude the conflicting method from all but one trait. In addition traits allow *method aliasing*. The programmer can introduce an additional name for a method provided by a trait to obtain access to a method that would otherwise be unreachable, for example, because it has been overridden. Traits can be composed from subtraits. The composition semantics is the same as explained above with the only difference being that the composite trait plays the role of the class.

5 Using Traits to Reuse and Compose Class Properties

Our approach is based on using traits to compose and reuse class properties within the traditional parallel inheritance schema proposed by Smalltalk (See Figure 8 left). Therefore our approach is safe *i.e.*, it supports downward and upward compatibility. But on top of that it promotes the reuse of class properties. Composition and application of class properties are based on trait composition, which gives the programmer explicit control in a uniform manner.

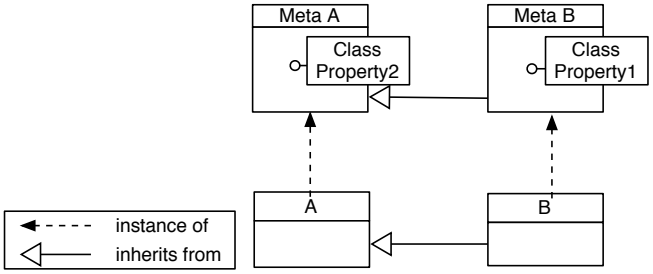


Fig. 3. Metaclasses are composed from traits representing class properties. Traits supports upward and downward compatibility.

We represent class properties as traits, which are then used to compose metaclasses as shown in Figure 3. Since traits have been fully implemented in the open-source Squeak Smalltalk environment [19], we implemented all the examples shown here in Squeak. During our refactoring of Squeak code we

identified the following class properties: TAbstract, TSingleton, TRememberInstances, TCreator, and TFinal which we explain below. We start with a simple example illustrating how a class is composed by reusing a class property, then we look how the traditional Boolean hierarchy [8,12] is re-expressed with traits and finally Section 6 shows that traits provide a good basis to engineer the meta level.

5.1 Singleton

To represent the fact that a class is a Singleton, we define the trait TSingleton. This trait defines the following methods: default which returns the default instance, new which raises an error, and reset which invalidates the current singleton instance. It requires basicNew which returns a newly created instance [1], and the methods uniqueInstance and uniqueInstance:. Note that these accessors methods are needed because traits cannot contain instance variables. Figure 4, left, shows the trait TSingleton.

```
Trait named: #TSingleton uses: {} category: 'Traits-Example'
TSingleton≫default
    self uniqueInstance isNil
        ifTrue: [self uniqueInstance: self basicNew].
    ↑ self uniqueInstance
TSingleton≫new
    self error: 'You should use default'
TSingleton≫reset
    self uniqueInstance: nil
```

As an example, suppose that we want to specify that a certain class WebServer is a Singleton. First of all we define the class WebServer in the traditional Smalltalk way as shown in 4. Then we specify at the metaclass level *i.e.*, in the class WebServer class, that the class is a Singleton by specifying that the class is composed from the trait TSingleton. The metaclass defines state needed to keep an instance around, under the form of the instance variable uniqueInstance. It also defines two glue methods uniqueInstance and uniqueInstance: as accessors methods for the instance variable uniqueInstance. These two glue methods fulfill the required methods with the same name of the trait TSingleton. Note that the required method basicNew is provided by the class Behavior, of which WebServer class, is an indirect subclass (see Figure 4, right).

[1] Using basicNew is the traditional way to implement Singleton in Smalltalk when we want to forbid the use of the new method [20]. basicNew allocates objects without initializing them. It is a Smalltalk idiom to never override methods starting with 'basic' names.

```
Object subclass: #WebServer
  instanceVariableNames: ''
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Traits-Example'

WebServer class uses: {TSingleton }
   instanceVariableNames: 'uniqueInstance'

WebServer class>>uniqueInstance
    ↑ uniqueInstance

WebServer class>>uniqueInstance: anObject
    uniqueInstance := anObject
```

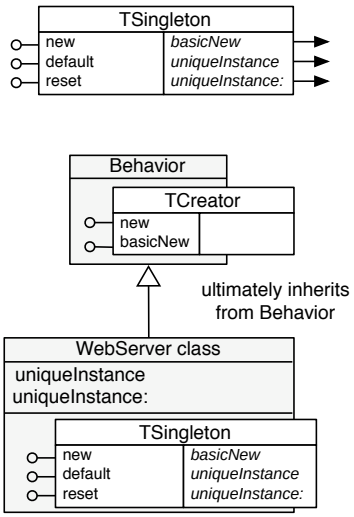


Fig. 4. Left. The trait TSingleton. Right. The class Behavior, the root of metaclasses in Smalltalk, is composed from the trait TCreator and as such provides the method basicNew.

5.2 The Boolean Hierarchy Revisited

The Smalltalk Boolean hierarchy consists of the abstract class Boolean, that has two subclasses True and False that are singleton classes. Traits allow the boolean hierarchy to be refactored as shown in Figure 5. Note that the refactored solution is backwards compatible with the idioms existing in the current Smalltalk implementation and literature [20]. So we assume that a method basicNew is defined on the class Behavior that can always be invoked to allocate instances and that should not be overridden.

Boolean. The class Boolean is an abstract class, so we compose its class Boolean class from the trait TAbstract.

```
Trait named: #TAbstract uses: {} category: 'Traits-Example'
TAbstract≫new
    self error: 'Abstract class. You cannot create instances'
TAbstract≫new: size
    self error: 'Abstract class. You cannot create instances'
```

False and True. The classes False and True are Singletons so their classes False class and True class are composed from the trait TSingleton which is then reused in the two classes.

As mentioned above, the trait TSingleton requires the methods basicNew, uniqueInstance, and uniqueInstance:. Therefore the class False class (resp. True class) has to define an instance variable uniqueInstance and the two associate accessors

methods uniqueInstance and uniqueInstance:. Note that the method basicNew does not have to be redefined locally in the class False or True class as it is inherited ultimately from the class Behavior, the inheritance root of the metaclasses [9] (see Figure 5 right). This example shows that class properties are reused over different classes and that metaclasses are composed from different properties.

```
False class
    uses: {TSingleton }
    instanceVariableNames: 'uniqueInstance'
False class≫uniqueInstance
    ↑ uniqueInstance
False class≫uniqueInstance: anObject
    uniqueInstance := anObject
```

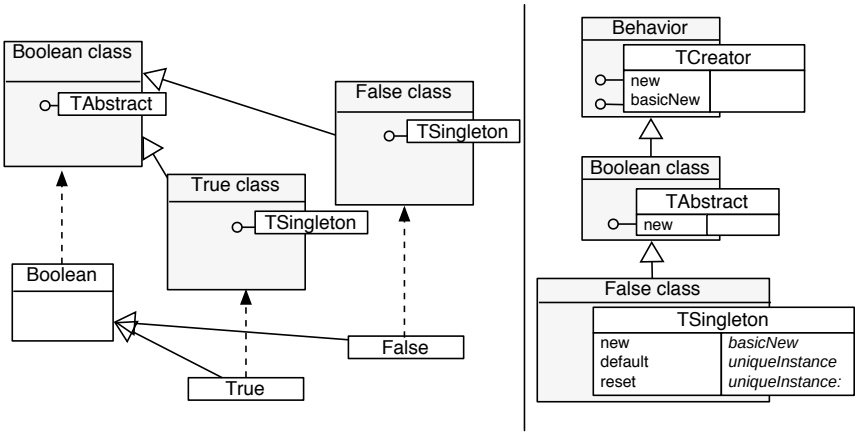


Fig. 5. Left: Boolean hierarchy refactored with traits. Right: The complete picture for the Boolean hierarchy solution.

6 Engineering The Meta Level

So far we presented simple examples that show how traits are well-suited to model class properties, which can then be combined or applied to arbitrary classes. In this section, we show that traits also allow more fine-grained architectures of class properties. We also want to stress that the techniques used here at the meta level are exactly the same as those used at the base level. As such, traits provide a uniform model.

Since many of these properties are related to instance creation, and we perform our experiments in Smalltalk, we first clarify the basic instance creation concept of Smalltalk. In Smalltalk, creation of a new instance involves two different methods, namely basicNew and new [2]. The method basicNew is a low-

[2] Note that there are also the methods basicNew: and new:, which are used to create

level primitive which simply allocates a new instance of the receiver class. The method new stands at a conceptually higher level and its purpose is to return a usable instance of the receiver class. For most classes, new therefore calls basicNew to obtain a new instance and then initializes it with reasonable default values.

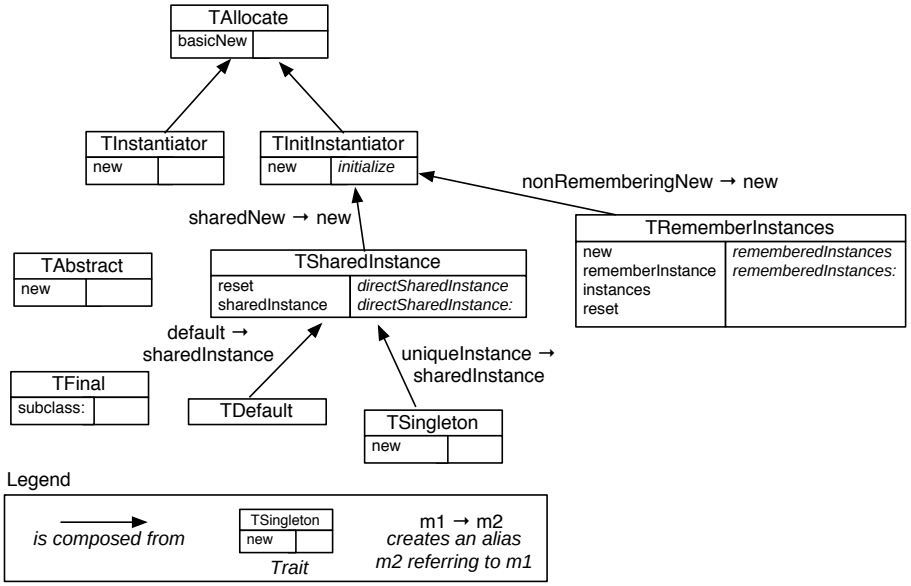


Fig. 6. A fine-grained architecture of class properties based on traits

6.1 Class Properties

Figure 6 gives an overview of the class properties we identified (see Section 10 for a deeper discussion). Note that all of these properties are traits, and that they are therefore composed using trait composition.

Allocation. As indicated by its name, the trait TAllocator provides the behavior to allocate new instances. In our case, this is the standard Smalltalk basicNew method, but of course we could also create another trait with an alternative allocation strategy.

Instantiation. The traits TInstantiator and TInitInstantiator are two class properties for instance creation. The trait TInstantiator uses the trait TAllocator and implements the method new in the traditional Smalltalk manner, which means that it does not initialize the newly created instance. The trait TInitInstantiator uses the trait TAllocator. However, as suggested by its name, it actually ini-

objects with indexed fields (*i.e.,* arrays). For sake of simplicity, we do not take these methods into account here.

tializes the newly created instance by calling the method initialize before the instance is returned.

```
TInstantiator≫new
    ↑self basicNew
TInitInstantiator≫new
    ↑self basicNew initialize
```

Note that the method initialize is called on the new instance, which means that the requirement for initialize in the trait TInitInstantiatior is actually a requirement for the instance side.

Remembering Instances. The trait TRememberInstances represents an instance creation property that remembers all the instances created by a class. It uses the trait TInitInstantiator and aliases the method new of the traits TInitInstantiator which is then available as nonRememberingNew. This aliasing allows one to access the original new method of the trait TInitInstantiator while leaving the option to override the method new in the trait TInitInstantiator. It requires the methods rememberedInstances and rememberedInstances: to access a collection storing the created instances. Then, it implements the methods new, rememberInstance:, instances, and reset as follows:

```
TRememberInstances≫new
    ↑ self rememberInstance: self nonRememberingNew
TRememberInstances≫rememberInstance: anObject
    ↑ self instances add: anObject
TRememberInstances≫instances
    self rememberedInstances ifNil: [self reset].
    ↑ self rememberedInstances
TRememberInstances≫reset
    self rememberedInstances: IdentitySet new
```

Note that another implementation could be to define the methods reset and rememberedInstances: as trait requirements. This would leave the class with the option to use other implementations for keeping track of the created instances.

Default and Singleton. The traits TDefault and TSingleton implement the class properties corresponding to the Default Instance and Singleton design patterns. Whereas a Singleton can only have one single instance, a class adhering to the Default Instance pattern has one default instance but can also have an arbitrary number of other instances.

Since these two properties are very similar, we factored out the common code into the trait TSharedInstance. To get the basic instantiation behavior, this trait uses the property TInitInstantiator and again applies an alias to ensure that the method new is available under the name sharedNew. Then, it implements the methods reset and sharedInstance as follows:

```
TSharedInstance≫reset
    self directSharedInstance: self sharedNew.
TSharedInstance≫sharedInstance
    self directSharedInstance ifNil: [self reset].
    ↑ self directSharedInstance.
```

The property TDefault is then defined as an extension of the trait TSharedInstance that simply introduces the alias default for the method sharedInstance. Similarly, the property TSingleton introduces the alias uniqueInstance for the same method. In addition, TSingleton overrides the method new so that it cannot be used to create a new instance:

```
TSingleton≫new
    self error: 'Cannot create new instances of a Singleton.
        Use uniqueInstance instead'.
```

Another useful class property popularized by Java is the class property TFinal which ensures that a class cannot have subclasses. In Smalltalk, this is achieved by overriding the message subclass: [3] . Note that unlike all the other properties presented in this section, TFinal is not concerned with instance creation and therefore is entirely independent of the other properties. In Section 10 we discuss the relevance of the class properties we presented.

6.2 Advantages for the Programmer

Having an architecture of class properties has many advantages for a programmer. Whenever a new class needs to be created, a choice can be made regarding the creation of instances, and whether or not the class should be final. Besides having the obvious advantage of avoiding code duplication, it also makes the design much more explicit and therefore facilitates understandability of the class. The level of abstraction of the trait design is at the right level: the traits correspond to the class properties, and the class properties can be combined into metaclasses.

In addition, factoring out the properties in such a fine-grained way still gives the user a lot of control about some crucial parts of the system. Suppose for example that at first we would have decided to use the trait TInitInstantiator as the basis for all the other instance creation properties. If later on, we would decide to comply to the Smalltalk standard to create uninitialized instances by default, then we could make this change without modifying any of the involved methods. We would just need to make sure that the traits TRememberInstances

[3] In reality, the method to create a subclass takes more arguments but this is not relevant here

and TSharedInstance use the trait TInstantiator instead of TInitInstantiator.

Explicit Composition Control Power. By providing several different properties that are all related to instance creation behavior, this example also shows why it is so important to have explicit control over composition and application of class properties. In our example, there are many different properties which essentially introduce variants of the method new, and therefore, combining these properties typically leads to conflicts that can only be resolved in a *semantically* correct manner if the user has explicit control over the composition. In case of traits, this is ensured by allowing partially ordered compositions, exclusions, and aliases.

As an example, imagine that we want to combine the properties TDefault and TRememberInstances to get a property that allows both a default instance and also remembers all its instances. With our trait-based approach, we do this by creating a new trait TDefaultAndRememberInstances which uses TRememberInstances and TDefault as follows:

```
Trait named: TDefaultAndRememberInstances
    uses: {  TDefault @ {#defaultReset → #reset}.
             TRememberInstances − {#new}
                 @ {#storeNew → #new.
                    #storeReset → #reset}}

TDefaultAndRememberInstances≫sharedNew
    ↑self storeNew

TDefaultAndRememberInstances≫reset
    self storeReset.
    self defaultReset
```

Since both traits provide a method new, we exclude this method from the trait TRememberInstances when it is composed. As a consequence the trait contains the new method provided by TDefault, which uses sharedNew to create a new instance. Since we want to make sure that each new instance is also stored, we override sharedNew so that it calls storeNew, which is an alias for the new method provided by TRememberInstances.

Because the method reset is also provided by both traits, we use aliasing to make sure that we can access the conflicting methods. Then, we resolve the conflict by overriding the method reset so that it first removes the stored instances (by calling storeReset) and then creates a new default instance (by calling defaultReset). Note that the newly created instance will be remembered as the default instance and will also be stored in the collection with all the instances of the class.

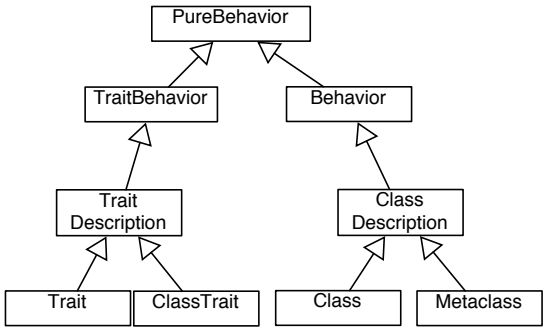


Fig. 7. The hierarchy of the new Smalltalk kernel supporting traits.

7 A New Kernel

In this section we present briefly the key implementation aspects of the new Smalltalk kernel that is bootstrapped with traits. The Figure 7 shows the new class hierarchy that we obtain. As the new kernel with traits is an extension of the traditional Smalltalk kernel, we get the traditional classes: Behavior, ClassDescription, Metaclass, and Class which now deal with the fact that a class may be composed of traits.

To model traits we then follow the previous design of the kernel and mimic the classes Behavior, ClassDescription and Class. Three classes TraitBehavior, TraitDescription, and Trait are introduced. Trait represents a trait and is applied to both the class and instance side. In addition the class Behavior, root of the instantiation graph, uses two important traits: TInstantiator and TInitInstantiator as presented in previous section. The class Behavior in Smalltalk also defines information about the state and behavior related to superclass and instance variables (format). As this is not needed for traits, we introduce a new abstract superclass PureBehavior, which factors out the common code between Behavior and TraitBehavior.

Some Traits. The traits we identified and used in this new kernel are not really remarkable in the sense of new MOP entries. In fact we mainly use traits to reuse code between the classes TraitsDescription and ClassDescription for the following reasons.

- The new kernel is based on the traditional Smalltalk kernel, which uses inheritance as the primary reuse mechanism. A lot of polymorphic methods are used among the classes Behavior and ClassDescription, and the classes Class and Metaclass. As a consequence, there is not much need for introducing traits to share this functionality.
- In the traditional Smalltalk kernel, the class Behavior only defines the minimal state and behavior to support classes as run-time entities. For example Behavior does not define the notion of named instance variables but just

knows the format of the instances the class will create in terms of the number and kind of instance variables [9]. Support for named instance variables is (amongst other things) implemented by ClassDescription, a subclass of the class Behavior. Because of this inheritance structure, some code that is implemented in ClassDescription (and uses state defined in ClassDescription) cannot and should not be reused by pushing it up to PureBehavior. Instead, we use traits to share behavior between ClassDescription and TraitDescription.

The class PureBehavior uses the following traits:

- TBasicCompile supports the compilation and decompilation of methods in a class.
- TTestingSelector supports the testing of selectors of methods (*e.g.*, canUnderstand:, ...).
- TCompiledMethodAccess supports the access into the method dictionary and source code access.
- TMethodIterating supports the iteration over compiled methods.

The class Behavior uses the following traits:

- TInstantiator and TInitInstantiator implement the creation of objects as described in the previous section.
- TFamilyAccess supports the access and enumeration of superclasses and subclasses.
- TInstanceEnumerator supports the enumeration of instances of the class.
- TMethodTesting supports the querying of methods.

The traits resulting from the decomposition of the classes PureBehavior and Behavior are not currently used by any other classes. In contrast, the following traits are used to share behavior between the two classes TraitsDescription and ClassDescription.

- TClassComment supports the management of comments.
- TMethodDictionaryManagement supports the management of methods categories.
- TOrganization supports how methods are sorted into method categories.
- TCodeReformatting supports reformatting of source code.
- TCodeFileOut supports filing-out (saving) of classes.
- TBehaviorCopy supports the copying of methods and their organization.
- TOrganizedCompilation supports the compilation of methods within the context of categories.

It should be noted that in the Smalltalk metaclass kernel, identifying traits that can be reused independently of each others is difficult because the behavior of the kernel is based on inheritance and the code was tightly coupled.

	up	down	per class	application	composition	control
Smalltalk	Yes	Yes	No	No	No	No
CLOS	Yes	Yes	No	multiple inheritance	multiple inheritance	explicit + linearization
SOM	Yes	No	No	multiple inheritance + code generation	multiple inheritance	implicit
NeoClasstalk	Yes	Yes	Yes	inheritance + generation	inheritance + code generation	implicit
MetaclassTalk	Yes	Yes	Yes	inheritance	mixin composition	mixin linearization
Traits	Yes	Yes	No	trait composition	trait composition	explicit

Table 1
Comparison of the models from Section 8 on how they handle the composition problems described in Section 2.

8 Related Work

This section shows how the main approaches that support explicit metaclasses address the problems described in Section 2. We also discuss the solution offered by Smalltalk (although it has implicit metaclasses) since it forms the basis for the NeoClasstalk solution and our own solution. Table 1 summarizes the comparison of these approaches. Note that the table shows the influence of the CLOS approach based on multiple inheritance to support metaclass composition in SOM.

8.1 Metaclass Composition

Smalltalk. In Smalltalk (and more recently in Ruby), metaclasses are *implicit* and created *automatically* when a class is created [9]. Each class is the sole instance of its associated metaclass. This way the two hierarchies are parallel (see Figure 8 left). Hence the architecture is safe as it addresses compatibility issues but completely prevents class property reuse between several hierarchies.

CLOS. CLOS's approach could be summarized as "do it yourself". Indeed by default in CLOS, a class and its subclasses must be instances of the same metaclass, prohibiting classes in the same hierarchy from having different class properties. For example, in Figure 8 right, class B has by default the same metaclass as its superclass A, and this cannot be changed. So class B always has the same class properties as class A. Note that since CLOS has explicit metaclasses, multiple inheritance can be used for composing class properties. For example, in the context described by Figure 2 it is possible to use multiple inheritance to explicitly combine the two properties Final and Singleton expressed as metaclasses into a new class SingletonFinal. Note that such an

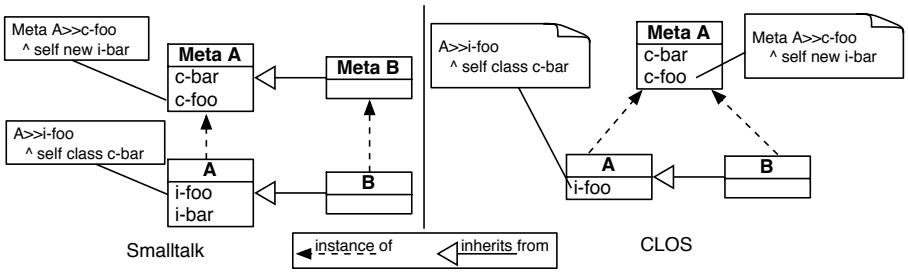


Fig. 8. Left: Smalltalk addresses compatibility issues by preventing reuse using implicit metaclasses and parallel hierarchies. Right: By default CLOS addresses compatibility issues by preventing subclasses to have different metaclasses than their superclasses.

implementation suffers from the same problems as multiple inheritance based on linearization occurring at the base level [16].

The general CLOS rule that a class and its subclasses must be instances of the same metaclass can be circumvented using CLOS's metaobject protocol (MOP). Indeed, the generic function validate-superclass [4] offers a metaprogrammer the possibility to specify that a class and its subclasses can be instances of different classes. However, this comes at a very high price because the CLOS MOP does not provide predefined strategies for avoiding compatibility problems or for dealing with possible conflicts. Hence the semantics of the composition has to be implemented manually, a far from trivial undertaking.

This means that by default CLOS is upward and downward compatible but it prevents usage of different metaclasses within an inheritance hierarchy and reuse of class properties. Both the composition of class properties and the application of properties are done with multiple inheritance. The control of the composition is explicit, because the user has to use multiple inheritance to create a new metaclass. However, since multiple inheritance in CLOS uses implicit linearization, the well-known problems associated with this form of conflict resolution also apply to the meta level [16].

SOM. The solution proposed by SOM (System Object Model) [7] is based on the automatic generation of *derived metaclasses*, that inherit multiply from the metaclasses to compose class properties. When at compile time a class is specified to be an instance of a certain metaclass, SOM automatically determines whether upward compatibility is ensured and if necessary creates a derived metaclass. In Figure 9 left, the class B (originally an instance of MetaB), inheriting from class A (instance of MetaA) finally becomes an instance of a derived metaclass inheriting from MetaA and MetaB. Note that SOM ensures that the existing metaclass MetaB takes precedence over MetaA in case of multiple inheritance ambiguities (since B is a subclass of A).

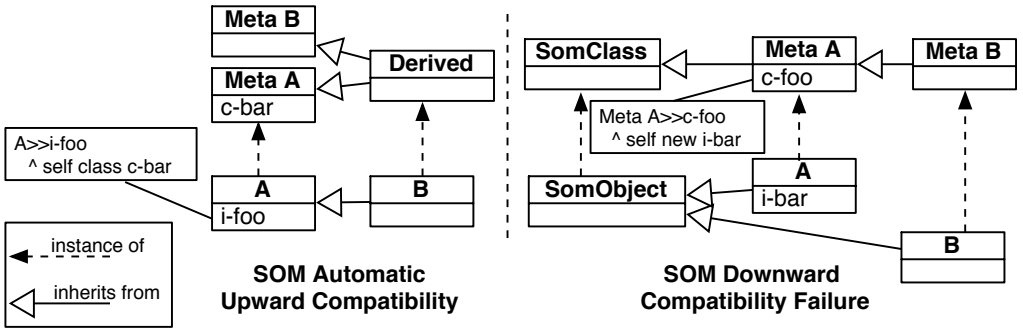


Fig. 9. Left: SOM supports upward compatibility by automatically deriving new metaclasses and changing the class of the inheriting class B. Right: SOM downward compatibility failure example.

While SOM supports upward compatibility as shown in Figure 9 left, it does not support downward compatibility [12] as shown in Figure 9 right. When the class B receives the c-foo message, a run-time error will occur because its instances do not understand the i-bar message. However, in SOM, contrary to CLOS, two distinct classes need not have the same metaclass. But as in CLOS, the composition of class properties is based on multiple inheritance. The application of a class property is done by a combination of multiple inheritance and automatic class generation. This happens at compile time, and the programmer has no explicit control over how possible conflicts are resolved.

NeoClasstalk. NeoClasstalk's approach is interesting since it supports both downward and upward compatibility and enables class property reuse between different hierarchies [17,12,18]. NeoClasstalk uses two techniques to accomplish this: *dynamic change* of classes and the composition of metaclasses by *code generation*. It generalizes the parallel inheritance solution of Smalltalk by enabling class properties reuse, but it also introduces some problems of its own that we discuss in detail after explaining the basic principles.

NeoClasstalk allows properties to be assigned to classes. Figure 10 shows what happens when assigning a property to Meta B. B inherits from class A and is an instance of the class Meta B before the new property is assigned to Meta B. When assigning the property, the system automatically creates a new metaclass Property m + Meta B (called a *property metaclass*), which inherits from the metaclass Meta B and defines the property code. It then *changes the class* of B to be that newly created metaclass. NeoClasstalk supports also per class property, *i.e.*, a property added to a class does not get automatically propagated to its subclasses.

To be able to reuse the property classes, NeoClasstalk stores the class properties in strings on methods of so-called *meta-metaclasses*. The actual metaclasses are then generated from these strings, as shown for our example in Figure 10. For example, the Property m represented by a meta-metaclass is

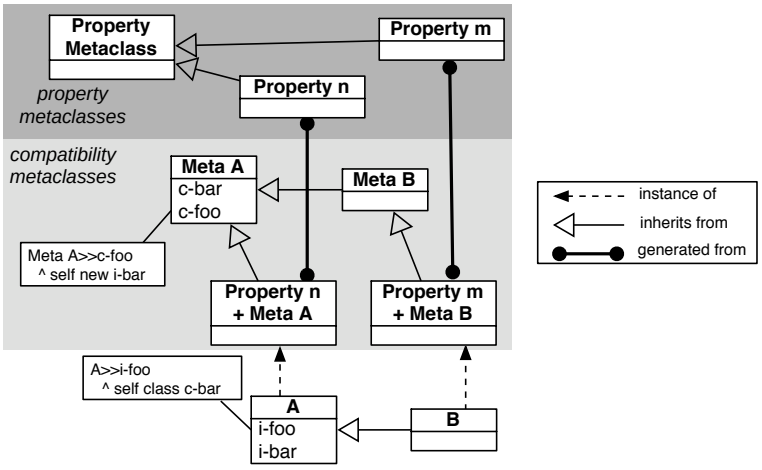


Fig. 10. Assigning the property m to class Meta B and property n to the class Meta A in NeoClasstalk. The light grey area denotes the metaclass area. The dark grey area is the realm of the class properties.

used to generate a new metaclass named Property m + Meta B from the metaclass Meta B and the Property m.

Besides the intrinsic complexity of NeoClasstalk's approach, it has the following drawbacks:

Dynamic class creation and dynamic change of class. The approach relies on the dynamic creation of classes and the dynamic changing of classes. It induces a complex management of meta-metaclass changes that should be propagated to the generated instances. Moreover as programming at the meta meta level is based on manipulating the strings that represent bodies of methods of metaclasses, it is not the same as programming at the metaclass or the base level. Basically, despite the name, the property metaclasses are not really at the meta-metaclass level, but merely storage holders for strings. The relation between the meta-metaclass level and the metaclass level is therefore not instantiation, as one would expect, but code generation. This breaks the uniformity of the model.

Ad-hoc and Implicit Composition. Property metaclasses are composed by code generation and applied implicitly by defining them in an inheritance chain. The composition is based on the assumption that a metaclass is designed to be plugged in this inheritance chain and that other composed behavior can be reached via super invocations. The composite metaclass has only limited control over the composed behavior as it can only invoke overridden behavior but does not have the full composition control.

As a summary, NeoClasstalk provides both downward and upward compatibility, and it allows one to assign class properties on a per-class basis. The composition of class properties is implicit and based on code generation and

chain of responsibility. The application of class properties is based on dynamic class changes and code generation.

MetaclassTalk. MetaclassTalk follows the architecture of NeoClasstalk by offering compatibility and property metaclasses. MetaclassTalk uses mixin composition to compose metaclass properties [15]. This experiment makes MetaclassTalk the closest model to our own approach as it supports both downward and upward compatibility while allowing the reuse of class properties. However MetaclassTalk composition is based on mixin linearization. As such it has the same problems as the ones we present in [11]: the composite entities do not have the full control of the composition, and the glue code is spread over multiple classes. These problems are solved by traits.

8.2 Metaobjects

Other approaches such as CodA [23], Moostrap [13], Iguana/J [24] support the composition and reuse of metaobjects. Such a composition is often based on chain of responsibility [13] *i.e.*, a metaobject is designed to be composed in a chain of metaobjects by invoking the overridden functionality. The problem with chain of responsibility is that it forces all the metaobjects to follow a certain architecture. It more importantly gives the composing metaobject only a very limited control over the composition: it can invoke the rest or do nothing. In contrast, traits composition is automatic when there is no conflict, and when conflicts arise, the composing metaclass has complete control over all the composed class properties.

The authors of Guarana [14] and Reflex [25], introduce *composite metaobjects* *i.e.*, a metaobject that define the composition semantics of several metobjects. This approach works well for coarse-grained composition, such as for making changes to the message passing semantics (broadcast, concurrent dispatch, or remote invocations). However, it is too heavyweight to compose class properties, since it would force the developer to define an explicit *composite metaclass* for all simple conflicts whose reuse is even questionable.

CodA [23] structures the meta-level architecture around several metaobjects responsible for the different actions. However it raises the issue of compatibility between all the metaobjects associated to a given object. The solution is to manually define a semantically coherent configuration of metaobjects implementing the desired semantics [23]. This solution shows again that there is no magic and that composing operations with conflicting semantics cannot be achieved in an automatic fashion.

9 Advantages and Disadvantages

Advantages. Traits support the decomposition of class properties as reusable units of behavior. Since metaclasses are composed of traits and the model is based on the parallel hierarchy of Smalltalk, it is upward and downward compatible and supports the reuse of class properties across different hierarchies.

In addition the proposed model is uniform with respect to the concepts used at the base level and the meta level (like CLOS). Both levels use the same concepts (traits and inheritance). Furthermore, the model is simple, and there is no need for on-the-fly code generation (as in SOM or NeoClasstalk) or for dynamic changes to classes (as in NeoClasstalk).

Class properties can be composed of traits that represent those properties. The application of the properties to an actual metaclass is accomplished by using the appropriate composite trait in the metaclass definition. The composing metaclass has *complete* control of the composition, and possible conflicts are resolved *explicitly* when the property is applied to a metaclass.

Having explicit control over the composition is especially important because it allows a programmer to freely adapt the behavior of the composite metaclass and to compose class properties that may not quite fit together. This means that our approach lets system designers ship their class hierarchies together with a set of *prefabricated* class properties in the form of traits, which can then be used and combined by the programmers. In case some class properties built by different vendors do not quite fit together, the traits model not only indicates the resulting conflicts, but also provides the programmer with the necessary means to resolve the conflicts to achieve the expected semantics.

Disadvantages. Glue methods and state have to be redefined in the metaclass where a property is applied. For example, the instance variable uniqueInstance and the two accessors methods have to be defined in all classes that implement a Singleton. We consider this to be a limit of the traits model and the price to pay to have the minimal mechanism supporting traits composition. Introducing state into traits would solve this but would introduce other problems, such as the well-known *diamond problem* of multiple inheritance [21], where state gets inherited through different paths.

It may happen that instance variables defined in a superclass are not necessary in the subclasses. For example, if the superclass implements a Singleton and the subclasses do not, then the instance variable that holds the Singleton instance as well as the methods to access it will be inherited by the subclasses. However, this problem is not due to traits by itself but is a result of using the inheritance mechanism in general. Table 1 compares the approaches.

10 Class Properties

Traits let us decide if a given functionality is defined as a trait or as a class. When defining functionality as a trait we automatically offer the possibility for future classes to use the identified behavior. One might wonder why we have so few class properties. First of all we chose to reengineer the current implementation of Squeak and not to design a new metaobject protocol. In this article we present the main *class properties* that we identified during our implementation, and we did not invent new ones. Secondly, we deliberately took heavily conflicting class properties, so that we could could clearly show the conflict resolution advantage of traits. Composing non-conflicting properties is trivial. Thirdly, other important efforts to build metaclass libraries, such as SOM [7], present nearly the same set of class properties.

Another point to consider is the role of the classes in the context of a metaobject protocol [4]; we believe that a lot of class properties identified in [8] are due to the fact that the classes were the single entry point in their MOP, while certain responsibilities are definitely the responsibilities of other meta-entities such as methods. It is also out the scope of this paper to present a new metaobject protocol based on traits, even if this is definitely future work.

Class Property Propagation Our approach does not support per class property because we did not want to change the class creation protocol of Smalltalk-80. However, there is nothing in our approach that prevents us to support per class property in a similar way than the compatibility model does [12,15]. Figure 11 shows that the intermediate metaclasses Boolean class + Abstract, False class + Singleton, and True class + Singleton are composed of traits and that properties such abstractness of the class Boolean, are not propagated to its subclasses. Note that metaclasses such as Boolean class could also be composed of traits if the property have to be propagated to the subclasses.

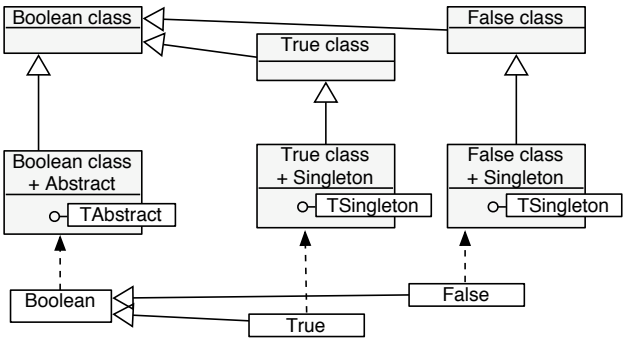


Fig. 11. Controlling class property propagation and trait composition.

11 Automatic Composition

An important difference between traits and most other approaches lies in the fact that traits have automatic conflict *detection* but expect *explicit* conflict resolution controlled by the programmer. Most approaches aim for a completely automatic resolution of conflicts, where possible conflicts are resolved according to some automatic scheme. If such schemes would work in all possible situations and in such a way that the programmer can easily foresee the result of a composition, developing software would be much easier. However, fully automatic resolution of conflicts is no panacea. It is only trivial when the composed semantics are orthogonal, so that conflicts can simply not occur. But it becomes extremely complex or even impossible when the semantics overlaps, which is the case for class properties. Looking in other areas such as multiple inheritance conflicts resolution we see that techniques based on automatic linearization techniques are not always satisfying [16] and often lead to unpredictable method invocations. The same applies here.

Nowadays the problems of composition of services or overlapping aspects is difficult and nearly impossible without the use of meta-data. For example Kienzle and Guerraoui demonstrates that trying to automatically compose transactions with other simpler aspects such as notification is doomed to failure [22]. In a similar vein authentication and encryption composition can only be a success when the encryption is invoked first, but authentication should take precedence over persistency and transactions.

Note that in the context of metaclass composition, the set of metaclass behavior is not predefined and fixed, as such it is possible to load a package in which another meta-programmer has developed new class properties with sensible composition exigence. Therefore any clever composition engine based on meta-information would have to deal with the openness of the set of class properties. Our solution, based on traits, differs from the other approaches since trait composition is automatic as long as there are no conflicts. Conflicts are detected automatically. When there is a conflict, then the traits model offers mechanisms to solve the conflict. This contrasts with the approaches that use an automatic scheme to handle conflicts.

12 Conclusion and Future Work

The need to reuse class properties led to meta-level architectures based on explicit metaclasses [1,2]. While offering reuse of class properties, such models introduced metaclass composition problems [10]. Different approaches exist that try to solve metaclass compositions problems, based on multiple inheri-

tance, code generation or automatically changing metaclasses [5,12,15]. However, the definition, the composition and the application of the class property were not controllable by the developer or meta programmer.

Our solution models class properties with *traits* (first class groups of methods), and uses trait composition to safely combine and reuse them. Using traits to compose class properties first of all solves the metaclass composition problems (upward and downward compatibility is ensured) while supporting the reuse of class properties. In addition, composition and conflict resolution are *explicit* and under *control* of the composing entity. Thirdly, traits is a general-purpose composition mechanism for object-oriented languages that we have already applied successfully at the base level (for example to refactor the Smalltalk collection hierarchy [26]).

We implemented all the examples shown in this article using the Squeak implementation of traits and we started to refactor the kernel of Squeak using traits. Our next step is to use traits to define a new metaobject protocol for Smalltalk.

Acknowledgments. We also like to thank Eric Tanter, Noury Bouraqadi and the anonymous reviewers for their valuable comments and discussions.

References

[1] D. Ingalls, The Smalltalk-76 programming system design and implementation, in: POPL'76, 1976, pp. 9–16.

[2] P. Cointe, Metaclasses are first class: the objvlisp model, in: OOPSLA '87, 1987, pp. 156–167.

[3] J.-P. Briot, P. Cointe, Programming with explicit metaclasses in Smalltalk-80, in: OOPSLA '89, 1989, pp. 419–432.

[4] G. Kiczales, J. des Rivières, D. G. Bobrow, The Art of the Metaobject Protocol, MIT Press, 1991.

[5] S. Danforth, I. R. Forman, Derived metaclass in SOM, in: TOOLS EUROPE '94, 1994, pp. 63–73.

[6] I. R. Forman, S. Danforth, H. Madduri, Composition of before/after metaclasses in SOM, in: OOPSLA '94, 1994, pp. 427–439.

[7] I. R. Forman, S. Danforth, Putting Metaclasses to Work: A New Dimension in Object-Oriented Programming, Addison-Wesley, 1999.

[8] T. Ledoux, P. Cointe, Explicit metaclasses as a tool for improving the design of class libraries, in: ISOTAS '96, LNCS 1049, Springer Verlag, 1996, pp. 38–55.

[9] A. Goldberg, D. Robson, Smalltalk-80: The Language, Addison Wesley, 1989.

[10] N. Graube, Metaclass compatibility, in: OOPSLA '89, 1989, pp. 305–316.

[11] N. Schärli, S. Ducasse, O. Nierstrasz, A. Black, Traits: Composable units of behavior, in: ECOOP 2003, LNCS 2743, Springer Verlag, 2003, pp. 248–274.

[12] N. M. N. Bouraqadi-Saadani, T. Ledoux, F. Rivard, Safe metaclass programming, in: OOPSLA '98, 1998, pp. 84–96.

[13] P. Mulet, J. Malenfant, P. Cointe, Towards a methodology for explicit composition of metaobjects, in: OOPSLA '95, 1995, pp. 316–330.

[14] A. Oliva, L. E. Buzato, The design and implementation of guarana, in: USENIX Conference on Object-Oriented Technologies and Systems (COOTS'99), 1999.

[15] N. Bouraqadi, Safe metaclass composition using mixin-based inheritance, Journal of Computer Languages, Systems and Structures 30 (1-2) (2004) 49–61.

[16] R. Ducournau, M. Habib, M. Huchard, M. Mugnier, Monotonic conflict resolution mechanisms for inheritance, in: OOPSLA '92, 1992, pp. 16–24.

[17] F. Rivard, évolution du comportement des objets dans les langages à classes réflexifs, Ph.D. thesis, Ecole des Mines de Nantes, Université de Nantes, France (1997).

[18] S. Ducasse, Evaluating message passing control techniques in Smalltalk, Journal of Object-Oriented Programming (JOOP) 12 (6) (1999) 39–44.

[19] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, A. Kay, Back to the future: The story of Squeak, A practical Smalltalk written in itself, in: OOPSLA '97, 1997, pp. 318–326.

[20] S. R. Alpert, K. Brown, B. Woolf, The Design Patterns Smalltalk Companion, Addison Wesley, 1998.

[21] A. Snyder, Inheritance and the development of encapsulated software systems, in: Research Directions in Object-Oriented Programming, MIT Press, 1987, pp. 165–188.

[22] J. Kienzle, R. Guerraoui, Aop: Does it make sense? the case of concurrency and failures, in: ECOOP '2002, LNCS 2374, Springer Verlag, 2002.

[23] J. McAffer, Meta-level programming with coda, in: ECOOP '95, LNCS 952, Springer, 1995, pp. 190–214.

[24] B. Redmond, V. Cahill, Supporting unanticipated dynamic adaptation of application behaviour, in: ECOOP 2002, LNCS 2374, Springer, 2002, pp. 205–230.

[25] E. Tanter, N. Bouraqadi, J. Noye, Reflex — towards an open reflective extension of java, in: International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, LNCS 2192, Springer, 2001, pp. 25–43.

[26] A. P. Black, N. Schärli, S. Ducasse, Applying traits to the Smalltalk collection hierarchy, in: OOPSLA '03, 2003, pp. 47–64.

Language support for Adaptive Object-Models using Metaclasses ⋆

Reza Razavi [a] Noury Bouraqadi [b] Joseph Yoder [c]
Jean-François Perrot [d] Ralph Johnson [c]

[a] *Software Engineering Competence Center - University of Luxembourg*
6, rue Richard Coudenhove-Kalergi - Luxembourg, L-1359- Luxembourg
reza.razavi@univ.lu

[b] *Ecole des Mines de Douai - Dépt. G.I.P*
941, rue Charles Bourseul - B.P. 838 - 59508 Douai Cédex - France
bouraqadi@ensm-douai.fr

[c] *Department of Computer Science*
University of Illinois at Urbana-Champaign - Urbana, IL 61801 - USA
yoder@refactory.com - johnson@cs.uiuc.edu

[d] *Laboratoire d'Informatique de Paris VI (LIP6)*
Université Pierre et Marie Curie - CNRS - Paris, 75252 - France
jean-francois.perrot@lip6.fr

Abstract

Adaptive Object Models are a sophisticated way of building object-oriented systems that let non-programmers customize the behavior of the system and that are most useful for businesses that are rapidly changing. Although systems based on an Adaptive Object Model are often much smaller than competitors, they can be difficult to build and to learn. We believe that the problems with Adaptive Object Models are due in part to a mismatch between their design and the languages that are used to build them. This paper describes how to avoid this mismatch by using implicit and explicit metaclasses.

Key words: Adpative Object Model (AOM), TypeObject Pattern, Meta-Object, Implicit Metaclasses, Explicit Metaclasses

⋆ The work communicated in this paper has been conducted while the first author doing his PhD at Laboratoire d'Informatique de Paris 6 (LIP6), Université Paris 6 - CNRS, Paris, France.

1 Introduction

Rapid change in business practice creates the need for software development approaches that permit rapid changes in software. The next generation of software systems must be sufficiently malleable and sensitive to business dynamics to allow businesses to adapt to the shifting operating environments. The Adaptive Object-Model Architectural Style [1] is born from the quest for such software by pioneering business organizations [2–4]. Recent academic researchers have documented the recurring design patterns of those systems [5–11].

Adaptive Object-Models (AOM) are object-oriented applications that use regular objects, i.e. instances, for representing metadata that describes the desired structure and behavior of the software in a given operating environment. For instance, the object-model, business rules [12] and roles relevant to a specific insurance or finance product can be specified at runtime by metadata. Business experts change the metadata using graphical tools to reflect domain changes in the software. The user-generated metadata is interpreted and this leads to "immediate", but controlled, effects on the system interpreting it. This is different from traditional configuration since this architectural style enables runtime and intuitive modification of the class hierarchy of the object-oriented program. Opening the system for intervention of "non-programmers" at runtime is one of the major assets of AOMs. Indeed, as it is observed by Bonnie A. Nardi [13], domain experts have the detailed task knowledge necessary for creating the knowledge-rich applications they want and the motivation to get their work done quickly, to a high standard of accuracy and completeness. AOMs are designed for empowering experts and make it possible:

- To develop and to change software quickly. AOM reduces time-to-market, by giving immediate feedback on what a new application looks like and how it works, and by allowing users experiment with new product types,
- To modify a software in accord with business experts, without calling programmers,
- To avoid shutting down the system in order to adapt it to new or local business needs,
- To reduce the volume of code that programmers should develop and maintain.

AOMs have the potential for providing businesses with highly competitive tools to cope with changes resulting from recurrent merges, alliances, acquisitions, etc. In short, AOMs correspond to an industrial reality, and have the potential to assist organizations in coping efficiently with their business evolution. They represent an important, long-term trend in software engineering. However, AOMs can require more effort to develop. Several reasons that have

been outlined in the literature are:

- An AOM can be hard to implement since it has more complex requirements, e.g. the need for producing, storing and interpreting metadata,
- Since an AOM leads often to a domain-specific language, therefore, as for any language, developers should provide programming tools such as debuggers, version control, and documentation tools,
- Developing database schemas and graphical user interfaces for AOM is harder since the specification of the underlying data changes at runtime,
- The architecture of an AOM can be harder to understand, document and maintain since there are two coexisting object systems; the interpreter written in the object-oriented programming language, and the object-model of the underlying business domain that is interpreted.

In this paper, we explore the use of reflection [14–16] for supporting AOM programming. More precisely, we focus on the use of metaclasses to support class adaptations made by business experts at runtime, while avoiding the mismatch between their design and the languages that are used to build them. So, computer professionals can maintain and evolve code changed by experts. We explored the use of Smalltalk-80 [17] implicit metaclases as an alternative to traditional techniques. Implicit metaclasses come with their drawbacks. Explicit metaclasses experimented with MetaclassTalk [18,19] proved more satisfactory.

The rest of this article is organized as follows. Section 2 describes the context of this research and states the research problem that we address. Section 3 explains how to address that problem using metaclasses. Section 4 presents the related work. Section 5 is devoted to conclusions and perspectives of this work.

2 Background on AOMs and the Problem Statement

The concept of AOM is born recently from the research aiming at discovering and documenting the design principles of a particular class of software with complex behavior that emerged from the industry. AOMs have been also called in the past "User Defined Product architecture" [2], "Active Object-Models" [8,7] and also "Dynamic Object Models" [9].

2.1 AOMs as Metadata interpreters

AOMs are software systems that *interpret an object-oriented representation of some business products and rules*. That representation is described as metadata [8] that specifies the latest object model of the business. Those specifications are stored in a database and loaded when necessary for building up the object model that represents the real business model that is interpreted to provide the desired behavior. So, the architectural style of AOMs emphasizes run-time adaptability by designing business software as a *metadata interpreter*.

Mentioning *metadata* is just saying that if something is going to vary in a predictable way, then you should store the description of the variation in a database so that operating the variation is easy. In other words, if something is going to change a lot, make it easy to change. The problem is that it can be hard to figure out which elements are going to change, and even if you know it then it can be hard to figure out how to describe the variations in your database. Code is powerful, and it can be hard to make your data as powerful as your code without making it as complicated. But when you are able to figure out how to do it right, metadata can be incredibly powerful, and can decrease your maintenance burden considerably. The most difficult part of developing an AOM is of course to figure out the model of the variations. This corresponds to a *metamodel* [20] since its instances represent the object models of the system.

Recent pattern mining effort [7,9,10] has allowed documenting the most fundamental underlying design patterns [21] such as (1) design for runtime-definition of classes, (2) design for runtime-definition of attributes, (3) design for runtime-definition of relationships, and (4) design for runtime-definition of behavior. In the terminology used in this paper, this activity is called *adaptation*, and the entity that results from this activity is also called an *adaptation*. The class that is extended by an adaptation is called an *adaptive class*. Each adaptive class models an evolving element of the changing domain, like Videotape in the example that follows, where Terminator is one of its possible adaptations.

The focus of this paper is only on design for the runtime-definition of a class by a domain expert.

2.2 The Classic AOM Technique for Adapting a Class

The problem of design for runtime class definition in AOMs is a case of the recurrent problem faced by developers of large systems, having to design a class (generically called Component) from which an unknown number of subclasses

should be derived. A standard solution to this problem is documented by the TypeObject pattern [5]. It consists in representing the unknown "subclasses" of Component by simple instances of a class generically called ComponentType. New "subclasses" can be created at run-time by instantiating ComponentType. Instances of these "subclasses" are then created as regular instances of Component with an explicit pointer to the instance of ComponentType that represents their subclass. In this setting, instances of ComponentType are interpreted as *types* for the ordinary instances of Component.

As a simple example of the feature we want to exploit, consider the *Videostore* example of the Type-Object pattern given by Johnson & Woolf [5]. In this case, the business objects are videotapes containing movies (exactly one movie in each tape), which are rented to customers. Terminator n° 20, n° 21, etc. are examples of videotapes of the movie called Terminator. Of course, the store will own and rent several videotapes of the same movie. End-users of the system will deal with existing videotapes, manage their rental to customers and conduct the bookkeeping. Maybe they will sometimes acquire new videotapes for existing films. Domain experts on the contrary will have to introduce new movies, with their title, rental price and rating.

The design problem is that the exact number of available movies can be known only at runtime (our client is not interested in a software that manages a predefined number of movies; it doesn t makes sense nowadays for this business). Applying TypeObject leads to a design with two classes: class Videotape as Component, and class Movie as ComponentType. Each instance of Videotape carries a metaobject (instance of Movie) which represents the movie and contains information shared by all Videotapes of the same title.

It should be noted that class Movie represents here the class model of the interpreter embedded in the AOM. Its instances (generated at run-time) constitute the metadata that represents the different videotapes.

In this example, Videotape should then be an adaptive class, whereas Customer remains an ordinary class (since our domain model does not contemplate its modification: one may suppose that it is fixed by other constraints, such as corresponding to an external database, etc). A usage scenario of this design would be something like:

```
|term term1 term2 joe|
term := Movie title: 'Terminator'.
term1 := Videotape movie: term.
term2 := Videotape movie: term.
joe := Customer new.
term1 rentTo: joe. etc...
```

where title: and movie: are creation methods of classes Movie and Videotape

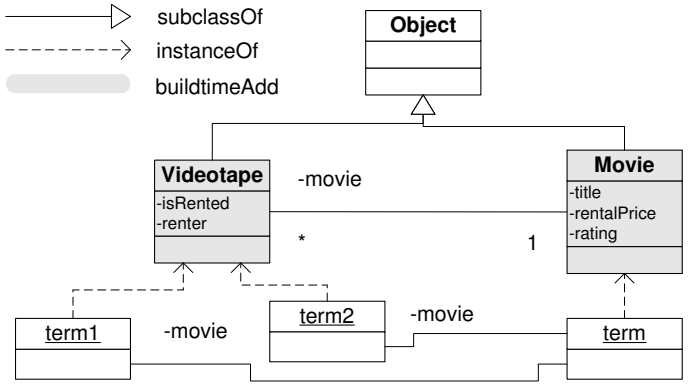


Fig. 1. Class & instance diagram of the Videostore example, with TypeObject design.

respectively. Instance term1 of Movie, representing Terminator, is the meta-object common to instances term1 and term2 of Videotape. Fig. 1 provides the diagram of this example. Classes created on software build-time (i.e. by computer professionals) are grayed.

It should be noted that domain experts execute this scenario or other presented in this paper using a GUI. The code described all along this paper corresponds to what happens behind the scene, from the programmers' point of view.

Here is the relevant fragment of code borrowed from [5], to be compared with the code resulting from approach presented in the following sections.

```
Movie class>>title: aString
   ^self new initTitle: aString

Object ()
   Videotape (movie isRented renter)
   Movie (title rentalPrice rating)

Movie>>newVideotape
   ^Videotape movie: self

Videotape>>initMovie: aMovie
    movie := aMovie

Videotape>>rentTo: aCustomer
    self checkNotRented.
    aCustomer addRental: self.
    self makeRentedTo: aCustomer
```

```
Videotape>>checkNotRented
    isRented ifTrue: [^self error]

Customer>>addRental: aVideotape
    rentals add: aVideotape.
    self chargeForRental: aVideotape rentalPrice

Videotape>>rentalPrice
    ^self movie rentalPrice

Videotape>>movie
    ^movie

Videotape class>>movie: aMovie
   ^self new initMovie: aMovie

Movie>>rentalPrice
    ^rentalPrice

Movie>>initTitle: aString
    title := aString
```

2.3 Business Objects and their MetaObjects: the Mismatch Problem

The following discussion hinges on the distinction between non-terminal and terminal objects (or, respectively, classes and non-classes). A terminal object is not a class; rather it is an instance of a class. The expression "non-terminal object" is equivalent with "class". As an abbreviation, it is often practical to transfer the qualification of instances to their classes: accordingly we shall speak of a *terminal class* to mean a class whose instances are terminal objects, as opposed to a metaclass, whose instances are classes [22].

The central problem in AOM design is the representation of evolving business objects. It follows that a regular set of classes, of which the business objects would be instances, won't be enough. The definition of a business objects must be assigned to specific meta-objects whose role is to represent information about the implementation and the interpretation of the object. As it is described in Dynamic Object Model design pattern (DOM) paper [9], to address this issue AOMs apply currently the TypeObject design technique, where "type" objects represent information about the implementation and the interpretation of associated business objects (which is the role of a meta-object).

As explained above, this pattern introduces two separate classes, known generically as Component and ComponentType. Class Component takes care of the fixed part, and class ComponentType deals with the variable part. Accordingly, a business object will be realized as an instance of Component (the "object") coupled with an instance of ComponentType (the "meta-object"), by means of a pointer. Our analysis is that such a business object will then have two metaobjects, (1) its class, of which it is instance, and (2) the associated component type (instance of ComponentType). For instance, a business object resulting from this design such as the *Spiderman videotape n° 20* will have as metaobjects class Videotape as well as an instance of Movie.

In our running example, Movie is an "ordinary" class, and therefore its instances are non-classes. However, according to TypeObject, each of these instances plays a class-like role for all instances of the class Videotape to which it is related (this role might be described as a "class complement"). This association is created by the initMovie: method in Videotape class, which is itself called by the creation method movie: in the metaclass "Videotape class". In other words, each terminal instance of Movie represents a subclass of Videotape, as was said earlier. However, such a subclass does not exist in reality, and the objects that are considered as being its instances, are in reality instances of Videotape (instances of Movie being terminal, it is impossible to instantiate them).

As an example of the undesirable consequences of this design, when programmers "inspect" the *Spiderman videotape n° 20*, they see that it is an instance of Videotape with an instance variable "type" that refers to a Movie. It would be more helpful for them to see that this videotape is an instance of Spiderman, which would be at the same time a subclass of Videotape and an instance of Movie.

The classic AOM solution thus leads to a conceptual and technical problem from the point of view of AOM programmers and maintainers. The origin of this problem is the mismatch between the semantics of TypeObject and that of the underlying object-oriented language, here Smalltalk-80, in their approach to subclassing. The consequences of this mismatch are well documented in the Dynamic Object-Model design pattern paper as *increased design complexity, increased runtime complexity*, and *new development tools*. The goal of this communication is to present an alternative approach that will avoid this mismatch.

3 Solution by Using Metaclasses

Our proposal is to use as meta-objects those objects that naturally play such a role, namely classes. Of course, this will entail some extensions to the traditional design of classes. This is precisely here that metaclasses [23,22,24] come into play.

Subsection 3.1 gives an overview of metaclass use. Subsections 3.2 and 3.3 describe and illustrate the use of respectively implicit and explicit metaclasses for implementing component types. Subsection 3.4 provides a summary of the approach and its results.

3.1 Overview of Metaclass use

Our design is strongly influenced by Smalltalk and by Pierre Cointe's ObjVlisp model [25,26]. In Smalltalk the idea that classes are indeed objects takes the following strong form: any class may be endowed with a set of instance variables and a dictionary of methods which gives it an individual behavior as an "ordinary" object. In normal practice, this facility is mainly used to define instance creation methods. We propose to make use of the same facility to endow a class X with all the structure and behavior needed to turn it into the metaobject of its instances. As for any object, the way to define those instance variables and methods is to write them in the class of which X is instance. Since X is a class, its class is a metaclass.

To proceed further we need to be more specific about the status of metaclasses. As is well-known, introducing metaclasses leads to a number of non-trivial difficulties for which we have to choose a solution. We shall consider (1) standard Smalltalk (2) an extension of Smalltalk called MetaclassTalk which is the latest development of a long line of research [19,18].

Standard Smalltalk [17] imposes very strict limitations on its metaclasses. With each class C a metaclass called "C class" is automatically associated, of which C is the only instance. "C class" can be edited to receive additional instance variables and methods and hence change the structure and behavior of C. If class B is a subclass of C, then its associated metaclass "B class" is automatically a subclass of metaclass "C class" inheriting the additional structure and behavior that was added to class C and which therefore applies also to B.

The solution *idea* is then to implement *ComponentTypes* as metaclasses. In this context, adding a component type corresponds to adding a new metaclass. Adding a new component means then instantiating the relevant metaclass. To

illustrate this idea, consider again our running example. We propose that *each movie* (call it M) should be represented as a subclass of Videotape, of which those *videotapes* that contain M will be instances. To this end we must endow these classes with the necessary (meta)behavior by means of their metaclasses. A simple way of realizing it in standard Smalltalk is to define the metabehavior in the metaclass "Videotape class", and to take advantage of the parallel inheritance of metaclasses.

The plain Smalltalk-based solution works fine, but since a metaclass can have only one instance the designer cannot reuse the same instance behavior with various meta-behaviors. Also, metaclasses are not treated as ordinary classes, so that there is no possibility of applying the same scheme to a metaclass in order to obtain multiple ontological levels.

In MetaclassTalk things are different. Metaclasses exist in an independent fashion and are created as such. When creating a class, its metaclass can be specified as well as its superclass. It is therefore possible to use both superclass specification for instance behavior, and metaclass specification for class behavior.

3.2 Use of Implicit Metaclasses

The first model of metaclasses that we propose to experiment is that of implicit meta-classes implemented by Smalltalk-80. This model can be sketched as follows:

The metaclass of each class is chosen (created) automatically by the implementing system. Such metaclasses are implicit: they are both anonymous and developers don't handle them directly. The system manages the metaclass inheritance hierarchy and makes it be parallel to the class inheritance hierarchy. Because of these parallel hierarchies, properties (i.e. structure and behavior) of a given class are automatically propagated to sub-classes.

The Smalltalk-80 model of implicit metaclasses supports specialization by means of a *framework* whose extension points are the Object class, the Class metaclass, and the Metaclass metaclass. Object is the root of the hierarchy of classes that describe the structure and behavior of non-classes. Class is the root of metaclasses that describe the structure and behavior of classes. Metaclass is the root of the *meta-metaclass hierarchy* that describe the structure and behavior of implicit metaclasses.

The typical case in Smalltalk is to use Metaclass as the default meta-metaclass. However, it is possible to use a specific meta-metaclass. Figure 2 below provides an example excerpted from VisualWorks version 5i.4 [27]. This system

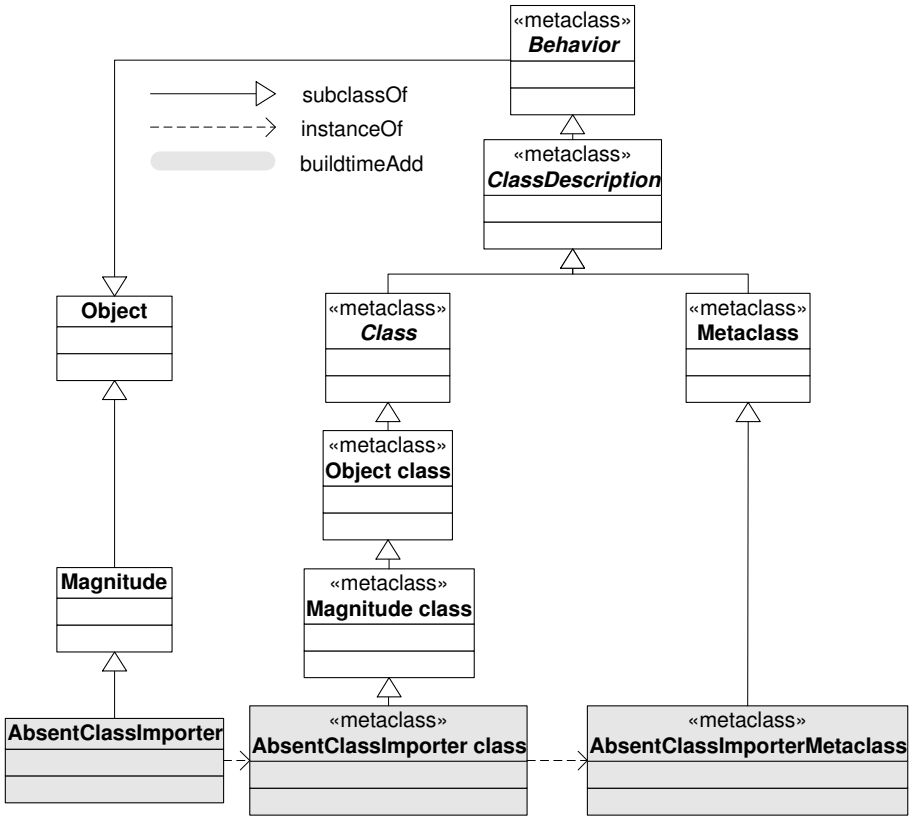


Fig. 2. Example of a meta-metaclass use in Smalltalk-80 (only main instantiation links are drawn).

enables loading a class whose superclass is absent from the system. It introduces new subclasses at the three levels: meta-metaclass AbsentClassImporterMetaclass, metaclass "AbsentClassImporter class" and class AbsentClassImporter.

For having a *ComponentType* created as an implicit metaclass, in the context of rules imposed by this model, programmers should subclass the "Object class" metaclass or one of its sub-metaclasses. Since implicit metaclasses are entirely managed by the Smalltalk system, a *Component* A corresponds in this model always to the sole instance of the *ComponentType* "A class". Definition of an adaptation C of the *Component* A inherits from A itself, while its metaclass "C class" inherits from "A class".

Illustration

Code portions below give the elements of the Smalltalk code corresponding to the implementation of the Videostore example by means of implicit metaclasses. Fig. 3 provides the corresponding class diagram. Classes and metaclasses built par computer professionals appear in gray while those created by

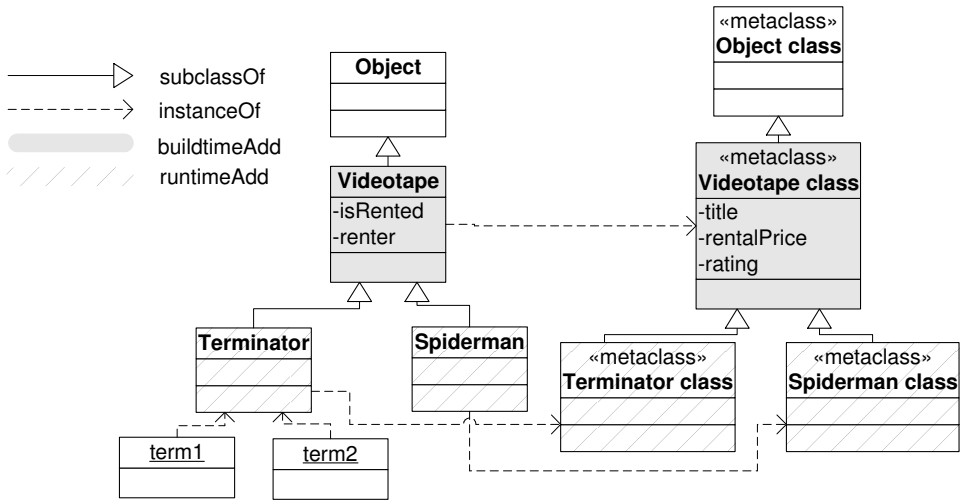


Fig. 3. Class & instance diagram of the Videostore example, with "Implicit Metaclasses" design.

domain experts are hatched.

We first have to define Videotape as a subclass of Object:

```
Object subclass: #Videotape
       instanceVariableNames: 'isRented renter '
       category: 'Videostore'
```

This creates also automatically the implicit metaclass "Videotape class" which is the *ComponentType* metaclass for videotapes. Here "Videotape class" will play the role of Movie in the TypeObject design above. We still need to add the instance variables of Movie, i.e., title, rentalPrice, and rating to "Videotape class". The resulting class hierarchy is as follows:

```
Object class ()
   Videotape class (title rentalPrice rating)
```

The creation method defined in this metaclass produces subclasses of Videotape whose title is received as argument:

```
Videotape class>>title: aString
   |sbcl|
   sbcl := self
       subclass: aString asSymbol
       instanceVariableNames: ''
       category: 'Videostore'.
   sbcl initTitle: aString.
   ^sbcl
```

Which requires the initialization method:

```
Videotape class>>initTitle: aString
    title := aString.
```

Now the access method:

```
Videotape class>>rentalPrice
    ^rentalPrice
```

is needed for

```
Videotape>>rentalPrice
    ^self class rentalPrice
```

The instance variable movie is no longer necessary. This link corresponds now to the instantiation link, managed by the Smalltalk virtual machine (the class message). This ensures better performances for AOMs.

```
Videotape>>movie
    ^self class
```

The remainder of the code is identical.

In the same spirit as the scenario above, the expert can now add new videotapes. The code that is executed behind the scene, since the expert uses a GUI, is as follows:

```
|term term1 term2 joe|
term := Videotape title: 'Terminator'.
```

so that after making several cassettes of this new movie:

```
term1 := term new.
term2 := term new.
```

they can be rented to customers:

```
joe := Customer new.
term1 rentTo: joe. etc...
```

In this setting, the movie Terminator is represented by a subclass of Videotape - the fact that this subclass might be named Terminator is of secondary importance. What is essential is that the metaclass "Terminator class" is automatically a subclass of "Videotape class". Seen as an object, class Terminator inherits the three instance variables defined in Videotape class, i.e. title, rentalPrice, and rating as well as the associated behavior. These instance variables are

set when the subclass is created, by means of the creation method Videotape class≫title: which calls the initializer initTitle:. In this way, class Terminator is fully equipped as a metaobject for the instances it generates, which represent individual cassettes of the movie Terminator. This setting is fully compatible with the object-oriented paradigm both from conceptual and technical points of view.

Now users will have to deal with instances of e.g. class Terminator (renting them to customers), with the creation of instances (buying new cassettes for the store), and with the creation of such classes (introducing new movies into the system). Note that creating a subclass at run-time does not require a different GUI than instantiating an existing class. The code we propose is easily packaged with a GUI which will completely hide the difference in implementation to both end-users and domain experts - but not, of course, to programmers!

The adaptations of Videotape like Spiderman and Terminator are now real subclasses. They can then be edited by programmers using their regular tools. The evolutions of the class hierarchy by experts can then be easily inspected and modified by professional programmers.

3.3 Use of Explicit Metaclasses

An alternative model to Smalltalk-80 implicit classes is that of "explicit" metaclasses. Object-oriented languages that implement this model allow programmers choosing explicitly the properties of their classes [23] by choosing its metaclass upon the creation of each class. This approach has been adopted by several systems, e.g. CLOS, Classtalk [28], and SOM. The prototype that we have developed here uses the MetaclassTalk system [18]. MetaclassTalk [1] is a reflective extension of Smalltalk-80. It results from a series of research on metaclasses and their use for defining class properties, starting with the ObjVlisp model and Classtalk system [28]. MetaclassTalk addresses the metaclass composition and compatibility issues [29]. Smalltalk virtual machine is used for runtime support. Its latest release, that we have used, uses the Squeak flavor.

Figure 4 below illustrates how to implement a specialization in the context of MetaclassTalk explicit metaclasses. The example provided here is a refactoring of the excerpt of VisualWorks provided in Figure 2.

MetaclassTalk provides the class Object and the metaclass StandardClass as extension points. Object is the root of the class hierarchy where properties of

[1] http://csl.ensm-douai.fr/MetaclassTalk

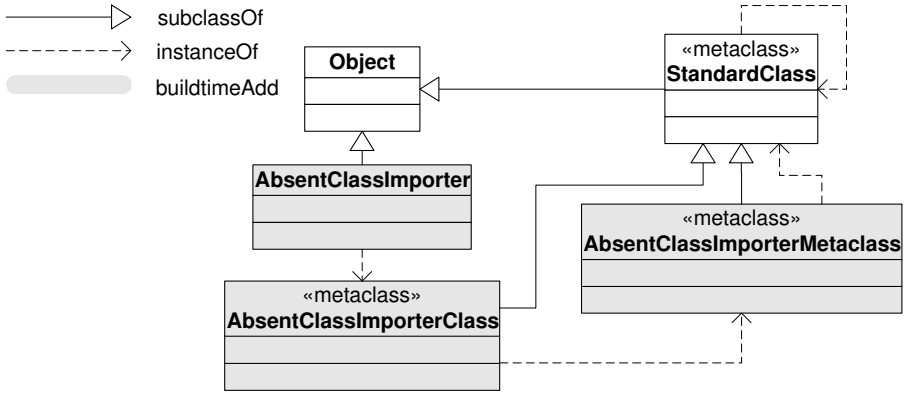


Fig. 4. Example of a specialization using explicit metaclasses.

non-classes are defined. StandardClass is the root of both the metaclass and the meta-metaclass hierarchies. Note that a metaclass is a meta-metaclass if its instances are also metaclasses, i.e. if they inherit (directly or not) from StandardClass. Adding a new abstraction systematically implies the extension of only one of the hierarchies described above. This extension is done *explicitly*. The programmer explicitly designates the superclass of the new abstraction. The programmer also explicitly chooses the type of the created class, by choosing the metaclass of the new class. Support for adapting classes relies then on the metaclass hierarchy.

In this context, to have their component types created as explicit metaclasses, programmers should subclass the StandardClass metaclass or one of its sub-metaclasses. As for the component class, it can be created with any appropriate superclass, e.g., Object. The choice of its metaclass is also only driven by the application needs and can be any metaclass in the system that provides the right behavior.

Illustration

Code portions below give the elements of the MetaclassTalk code corresponding to our running example.

We simply turn the implicit metaclass "Videotape class" of the design with implicit metaclasses into an explicit, free-standing metaclass which we naturally call Movie. Following normal practice in MetaclassTalk, as explained above, Movie is created as an instance and subclass of StandardClass (see Fig. 5).

```
StandardClass subclass: #Movie
    instanceVariableNames: 'title rentalPrice rating'
```

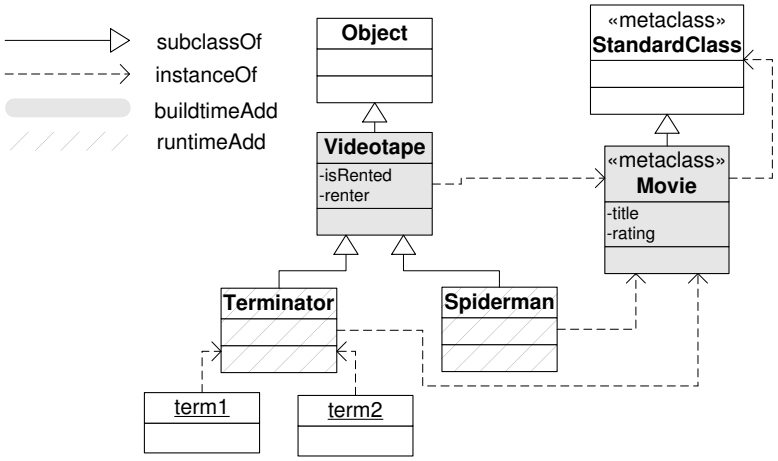


Fig. 5. Class & instance diagram of the Videostore example, with "Explicit Metaclasses" design.

```
    category: 'Videostore1'
    metaclass: StandardClass
```

Accordingly, class Videotape now appears as an instance of metaclass Movie:

```
Object subclass: #Videotape
    instanceVariableNames: 'isRented renter '
    category: 'Videostore1'
    metaclass: Movie
```

The resulting class hierarchy is as follows:

```
Object ()
   StandardClass (...)
       Movie (title rentalPrice rating)
           Videotape (isRented renter)
```

The creation method title: that was previously defined in the metaclass "Videotape class" is now the responsibility of Movie. Note that a typical receiver for this method will be class Videotape.

```
Movie>>title: aString
     |sbcl|
     sbcl := self subclass: aString asSymbol
        instanceVariableNames: ''
        category: 'Videostore1'
        metaclass: self class.
     sbcl initTitle: aString.
     ^sbcl
```

Same transfer for the other methods that were previously defined in "Videotape class":

```
Movie>>rentalPrice
    ^rentalPrice

Movie>>initTitle: aString
    title := aString.
```

The remainder of the code is identical, and the scenario above can be repeated without any change.

The decisive advantage of explicit over implicit metaclasses is the separation of the inheritance and instanciation hierarchies of class Videotape and of metaclass Movie. This allows for arbitrarily complex designs, using standard techniques (e.g. reuse) both for Videotape and for Movie.

This allows also for the construction of several levels of metaclasses, corresponding to the "nested type objects" of Johnson & Woolf. Fig. 6 below sketches the design for one of the extensions they add to the Videostore example, where *movies* may belong to different *movie categories*. In our design, MovieCategory is a meta-metaclass of which metaclass Movie is an instance. Individual *categories* appear as metaclasses that are instances of MovieCategory and subclasses of Movie. Class Videotape remains the same (instance Movie of and subclass of Object). Individual classes representing movies appear now as instances of the various categories (meta-classes) and remain subclasses of Videotape. In the example sketched in Fig. 6, there are two *movie categories* (named First and Second,) and two *movies*, Terminator belonging to category First and Spiderman to category Second. Only two videocassettes are available, both of Terminator. Note that the structural pattern of MovieCategory/Movie/First/Second is identical to that of Movie/Videotape/-Terminator/Spiderman, so that this design is actually simpler than it seems at first sight.

3.4 Discussion

The work communicated here is part of a larger-scale study, of which Razavi's doctoral dissertation [30] was the first version. It aims at designing a framework for AOMs (called in [30] DYCRA [2] for DYnamic Class Refinement Architecture). On the basis of the observations communicated here we plan to improve DYCRA by the systematic use of metaclasses (e.g. metaclass com-

[2] For more information please point to the URL: http://www-poleia.lip6.fr/∼razavi/Dyctalk/.

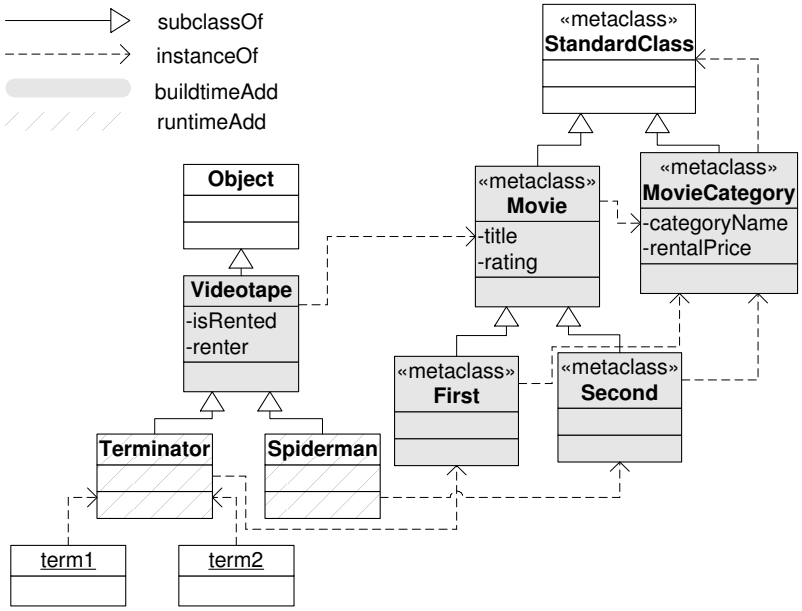


Fig. 6. Class & instance diagram of the extended Videostore example, illustrating the possibilities of the "Explicit Metaclasses" design.

position in the sense of Bouraqadi [19], Ducasse et al. [31]). The following remarks are set in this perspective.

One of the key issues is that AOMs have to undergo phases of refactoring where programmers will have to deal with a system on which experts have made a number of adaptations. If the architecture of the adapted code is complicated, then programmers will have a hard time doing their refactoring job. They will require specific tools and adequate training - and a different training for each new application.

This is why language-level support is essential, language being the common ground for all programmers (generic tools, etc). In Section 2 we pinpointed the mismatch problem due to an *ad hoc* architecture made necessary by the absence of reflective features, and in section 3 we showed how metaclasses can be used to solve this problem. Note that our design ensures that the "dynamic" classes that are created at run-time inherit all their methods from their "static" superclasses, so that the Smalltalk compiler is actually never called.

With our technique, the code remains indeed structured as a set of classes (as opposed to classes mixed with terminal instances in a design based on Type Object) programmers may use any engineering technique, e.g., editing adaptations using their regular editing tools, refactoring, even embedding a micro-workflow architecture [32], etc. Moreover, the code is indeed simpler, since a part of it disappears (the part of an AOM that represents the inter-

preter of the meta-level). The programming language itself is now called to play this role. Programmers of AOMs can then concentrate on modeling the business domain.

With implicit metaclasses (standard Smalltalk, section 3.2), metaclass reuse is limited. This becomes a challenging issue in designing reusable, language-level support for adaptation. That is, if we ant to empower AOM programmers with predefined support for implementing *ComponentTypes* as metaclasses. In this case, programmers should be able to develop their application-specific *ComponentTypes* by extending and reusing existing default metaclasses. For instance, the metabehavior associated to the metaclass "Videotape class" could be provided by the framework, and not implemented by AOM programmers.

As we saw in section 3.3, explicit metaclasses (with MetaclassTalk) eliminate this limitation. Infinite class hierarchies are indeed one of the AOMs fundamental features. MetaclassTalk appears therefore as a good candidate for an implementation language suitable for AOM design.

However, there is a price to pay. Choosing classes instead of arbitrary objects to represent meta-objects involves some limitations on the programmer's freedom. Instances belonging to two different classes will not be able to share the same meta-object. To follow Johnson & Woolf once more, a *Videodisk* and a *Videotape* will not share exactly the same Movie meta-object, since they belong to different classes - but they do not share the same rental price either. Our technique will lead to creating two classes with the same metaclass, e.g. TapeTerminator and DVDTerminator, and in order to express that these two classes are realizations of the same film it will be necessary to provide a supplementary ontological level in the same style as MovieCategory.

Also, changing the meta-object of a business object now involves changing its class, which is a risky operation. This is an important design issue. The gist of our approach is precisely to rely on language mechanisms, and paying the price for it. If the verifications and restrictions that go with it are undesirable, then our method is not the right one!

4 Related Work

Independently from AOMs, tool support for dynamic class specialization has been subject to extensive research since several decades in the field of object-oriented languages. Such a design has been introduced with Smalltalk-76. Extensions to that solution have also been subject to extensive research [25,26,33] [29,18,24]. What is different in the case of AOMs is the need for two different, but compatible, mechanisms for specialization. One is dedicated to program-

mers and the other to business experts.

Another comparable work is that around UML Virtual Machines [34], as well as executable UML models with the UML diagrams being supported by Action Semantics. There are two major issues with these approaches:

(1) the underlying solutions remain properties of private companies and have not been sufficiently documented as standard and reusable solutions; and
(2) they promote programming by UML-based languages, which is far from having gained popularity within the non-programmer business experts. Empirical research in the field of end-user programming shows that business experts are more likely to adopt domain-related solutions [13] and not an object modeling language designed for programmers. AOMs already implement such solutions. We believe that the applicability of UML profiles for building comparable solutions deserves more investigation.

AOMs are comparable with Domain Oriented Programming (DOP) languages, described by Dave Thomas & Brian Berry in [35]. Both DOP languages and AOMs embed a DSL (Domain-Specific Language). They are also designed "to allow knowledgeable end users to succinctly express their needs in the form of an application computation" [35].

Although the Squeak system [36] is not fundamentally designed for building AOMs, it addresses analogous problems since it provides the plain programming IDE as well as the EToy [37] interface. However, the pseudo-compatibility between classes and adaptations (Player class and its subclasses created implicitly during scripting by non-programmers using EToy) is achieved in an *ad hoc* manner.

Finally, this research has been influenced by work on metatool design for facilitating the creation of development tools that implement the "double-metamodeling" approach advocated by the METAGEN group [38]. In this system, one of the metamodels corresponds to a domain-related language for specifying the requirements and the other one corresponds to a technology-related language for expressing the implementation of these specifications. The process of moving from specification models to implementation ones is semi-automated, largely due to model transformation techniques, where NéOpus [39] is used for expressing transformation rules. This experimental work, going on since the early nineties, is closely linked with the more recent Model-Driven Architecture (MDA).

5 Conclusions and Perspectives

AOMs allow building business applications through the collaboration of programmers and business experts. This approach has been chosen by pioneer business organizations as a means to cope rapidly and cost-effectively with the need to adapting their critical software to the business dynamics. Programmers manage of course all technical aspects of the development process. Part of their responsibility is to empower business experts to specialize the "default" class inheritance hierarchy of the system, by dynamically adding new object types using domain-related constructs.

The current approach for creating AOMs (e.g. as documented by the DOM pattern) relies on a non-reflexive programming style. This approach leads to a series of issues that make the creation and maintenance of AOMs hard and costly. In order to help building AOMs while avoiding issues of existing solution, we explored in this paper the use of metaclasses. On the basis of this study, we can conclude that languages offering explicit metaclasses provide better support for adapting classes. MetaclassTalk is an example of such language. Languages that implement implicit metaclasses, like Smalltalk-80, only address part of adaptation support requirements. Indeed, they lead to a series of issues, like the loss of natural inheritance as well as undesirable propagation to a whole inheritance hierarchy of the choice of the metaclass.

Ledoux and Cointe observe that metaclasses are reifications that serve to vary the default semantics of classes [23], and suggest considering each such variation as a class property. Our proposal can then be summarized as "*adaptability should be treated as a class property*".

A last point to discuss is the relative importance of the mismatch problem that is the subject of this paper from the general point of view of AOM design. This mismatch is certainly not the only reason for the difficulties with AOMs. For instance, suppose you want to add a feature to a system. It can be added by the expert using only the tools that the programmer gave him, or it can be added by the programmer herself. The programmer can (and does) use the expert's tools, but the expert cannot use the programmer's tools.

The problem this poses for the expert is that sometimes he wants to do something but can't, and has to ask the programmer for help. The programmer can either build a new tool to let the expert to do it (or change an existing tool) or can just add the feature. Often the programmer does a little of one and a little of the other.

The problem this poses for the programmer is that there are several ways to reach her goal. Suppose that the expert can make a change by finding a set of instances and changing all of them, or the programmer can make a tool

that does it all at once. Suppose the expert can implement a procedure by making a complex workflow and reuse it by copying it and editing it, or the programmer can implement it by adding a new primitive to the system. There is no easy answer to the question "which is better ?". An AOM requires the programmer to be able to think of the system on several levels at once, and most programmers are not educated to do this well.

So, which problems are most important? If mismatch with language features is most important then our framework Dycra should make it measurably easier to build AOMs. But if mismatch is a secondary issue then it might not. We can not figure it out by hard thinking! The only way to tell whether mismatch is the main problem is to try to eliminate it and see what happens. This is the issue that we have addressed in this paper. Now we must experiment with the large-scale framework to figure out its real impact. We plan such experimentation in the course of a new project that aims at providing tool support for building families of adaptive and personalized solutions in the context of Ambient Intelligence [40].

Acknowledgments

The authors gratefully acknowledge the support from the Software Architecture Group (SAG) at UIUC and from the Metafor project at LIP6. This research benefited from a UIUC-CNRS exchange program, directed by Gul Agha and J.-P. Briot.

References

[1] J. W. Yoder, R. Johnson, The adaptive object-model architectural style, in: 3rd IEEE/IFIP Conference on Software Architecture (WICSA3). IFIP Conference Proceedings 224. Jan Bosch, W. Morven Gentleman, Christine Hofmeister, Juha Kuusela (Eds.), Kluwer, 2002, pp. 3–27.

[2] R. Johnson, J. Oakes, The user-defined product framework, (unpublished document).
URL `http://st.cs.uiuc.edu/pub/papers/frameworks/udp`

[3] F. Anderson, R. Johnson, The objectiva telephone billing system, in: MetaData Pattern Mining Workshop), 1998.

[4] M. Tilman, M. Devos, A reflective and repository-based framework. implementing application frameworks, in: Implementing Application Frameworks (M.E. Fayad, D. C. Schmidt, R. E; Johnson ed.), Addison-Wesley, 1999, pp. 29–64.

[5] R. Johnson, B. Woolf, Type object, in: Pattern Languages of Program Design 3, Robert Martin, Dirk Riehle, and Frank Buschmann, eds., Addison-Wesley, 1997, pp. 47–66.

[6] R. Johnson, Dynamic object model, (unpublished document).
URL http://st-www.cs.uiuc.edu/users/johnson/DOM.html

[7] J. W. Yoder, B. Foote, D. Riehle, M. Tilman, Metadata and active object-models, in: Workshop Results Submission OOPSLA'98 Addendum, 1998.

[8] B. Foote, J. Yoder, Metadata and active object-models, in: Proceedings of Plop98. Technical Report wucs-98-25, Washington University Department of Computer Science, 1998.

[9] D. Riehle, M. Tilman, R. Johnson, Dynamic object model, in: Proceedings of the 2000 Conference on Pattern Languages of Programming (PLoP 2000). Technical Report number WUCS-00-29, Washington University Department of Computer Science, 2000.

[10] J. W. Yoder, R. Razavi, Metadata and adaptive object-models, in: ECOOP'2000 Workshop Reader; Lecture Notes in Computer Science, vol. no. 1964, Springer Verlag, 2000.

[11] J. W. Yoder, F. Balaguer, R. Johnson, Architecture and design of adaptive object-models, SIGPLAN Not. 36 (12) (2001) 50–60.
URL http://doi.acm.org/10.1145/583960.583966

[12] A. Arsanjani, Rule object: A pattern language for pluggable and adaptive business rule construction, in: Proceedings of PLoP2000. Technical Report wucs-00-29, Washington University Department of Computer Science, 2000.

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

[14] B. C. Smith, Reflection and semantics in lisp, in: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, ACM Press, 1984, pp. 23–35.

[15] B. Foote, Objects, reflection, and open languages, in: ECOOP'92 Workshop on Object-Oriented Reflection and Metalevel Architectures, 1992.

[16] B. Foote, R. E. Johnson, Reflective facilities in smalltalk-80, in: Conference proceedings on Object-oriented programming systems, languages and applications, ACM Press, 1989, pp. 327–335.
URL http://doi.acm.org/10.1145/74877.74911

[17] A. Goldberg, D. Robson, Smalltalk-80: The Language and its Implementation, Addison-Wesley, 1983.

[18] N. Bouraqadi, T. Ledoux, Aspect-Oriented Software Development, Addison-Wesley, Boston, 2005, Ch. 12 – Supporting AOP Using Reflection, pp. 261–282.

[19] N. Bouraqadi, Safe metaclass composition using mixin-based inheritance, Journal of Computer Languages and Structures 30 (1-2) (2004) 49–61, special issue: Smalltalk Language.

[20] N. Revault, J. W. Yoder, Adaptive object-models and metamodeling techniques, in: Ecoop 2001 Workshop Reader. kos Frohner (ed), LNCS, Springer-Verlag, 2001.

[21] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design patterns: elements of reusable object-oriented software, Addison-Wesley Longman Publishing Co., Inc., 1995.

[22] J. F. Perrot, Objets, classes, hritage : dfinitions, in: In R. Ducournau, J. Euzenat, G. Masini, and A. Napoli, editors, Langages et Modles Objets: Etats des recherches et perspectives, chapter 1, INRIA - Collection Didactique, 1998, pp. 3–31.

[23] T. Ledoux, P. Cointe, Explicit metaclasses as a tool for improving the design of class libraries, in: In Proceedings of ISOTAS'96 - JSSST-JAIST, Springer-Verlag, 1996.

[24] I. Forman, S. Danforth, Putting Metaclasses to Work, Addison-Wesley, 1999.

[25] P. Cointe, Metaclasses are first class: The objvlisp model, in: Conference proceedings on Object-oriented programming systems, languages and applications, ACM Press, 1987, pp. 156–162.
URL `http://doi.acm.org/10.1145/38765.38822`

[26] P. Cointe, The objvlisp kernel: a reflective lisp architecture to define a uniform object-oriented system, in: P. Maes, D. Nardi (Eds.), Workshop on Meta-Level Architecture and Reflection, North Holland Publishing Company, Amsterdam, New York, Oxford, 1988, pp. 155–176.

[27] E. Miranda, Meta-programming in a flexible component architecture, in: Metadata and Dynamic Object-Model Pattern Mining Workshop OOPSLA '98, 1988.

[28] M. H. Ibrahim, Reflection and metalevel architectures in object-oriented programming (workshop session), in: Proceedings of the European conference on Object-oriented programming addendum : systems, languages, and applications, ACM Press, 1991, pp. 73–80.
URL `http://doi.acm.org/10.1145/319016.319050`

[29] N. Bouraqadi, T. Ledoux, F. Rivard, Safe metaclass programming, in: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, ACM Press, 1998, pp. 84–96.
URL `http://doi.acm.org/10.1145/286936.286949`

[30] R. Razavi, Outils pour les langages d'experts — adaptation, refactoring et rflexivit, Thse de doctorat, Universit Pierre et Marie Curie (Paris 6), LIP6, Paris, France (Nov. 2001).

URL
http://www-ftp.lip6.fr/ftp/lip6/reports/2002/lip6.2002.014.pdf

[31] S. Ducasse, N. Schrli, R. Wuyts, Uniform and safe metaclass composition, in: Proceedings of the ESUG Research Track. Also published in a special issue of the Elsevier international journal "Computer Languages, Systems and Structures", 2004, to appear in 2005.

[32] D. A. Manolescu, Workflow enactment with continuation and future objects, in: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, ACM Press, 2002, pp. 40–51.

[33] J. P. Briot, P. Cointe, A uniform model for object-oriented langauges using the class abstraction, in: In Proceedings of the Tenth Internationl Joint Conference on Artificial Intelligence (IJCAI'87), Vol. 1, 1987, pp. 40–43.

[34] D. Riehle, S. Fraleigh, D. Bucka-Lassen, N. Omorogbe, The architecture of a uml virtual machine, SIGPLAN Not. 36 (11) (2001) 327–341.
URL http://doi.acm.org/10.1145/504311.504306

[35] D. Thomas, B. M. Barry, Model driven development: the case for domain oriented programming, in: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, ACM Press, 2003, pp. 2–7.
URL http://doi.acm.org/10.1145/949344.949346

[36] M. Gudizal, K. Rose, Squeak - Open Personal Computing and Multimedia, Prentice Hall, 1999.

[37] B. J. Allen-Conn, K. Rose, Powerful ideas in the classroom: using squeak to enhance math and science learning, Comput. Entertain. 2 (1) (2004) 16–16.
URL http://doi.acm.org/10.1145/973801.973827

[38] N. Revault, H. A. Sahraoui, G. Blain, J. F. Perrot, A metamodeling technique: The metagen system, in: Proceedings of TOOLS 16, 1995.

[39] F. Pachet, J. F. Perrot, Rule firing with metarules, in: Software Engineering and Knowledge Engineering - SEKE '94, Jurmala, Lettonie, Knowledge System Institute, 1994, pp. 322–329.

[40] I. A. Group, Ambient intelligence: from vision to reality - for participation in society & business (2003).
URL http://www.cordis.lu/ist/istag-reports.htm

Classboxes: Controlling Visibility of Class Extensions ⋆

Alexandre Bergel [a] Stéphane Ducasse [a] Oscar Nierstrasz [a]
Roel Wuyts [b]

[a] *Software Composition Group–IAM, Universität Bern, Switzerland*
[b] *Lab for Software Composition and Decomposition, Brussels, Belgium*

Abstract

A *class extension* is a method that is defined in a module, but whose class is defined elsewhere. Class extensions offer a convenient way to incrementally modify existing classes when subclassing is inappropriate. Unfortunately existing approaches suffer from various limitations. Either class extensions have a global impact, with possibly negative effects for unexpected clients, or they have a purely local impact, with negative results for collaborating clients. Furthermore, conflicting class extensions are either disallowed, or resolved by linearization, with consequent negative effects. To solve these problems we present *classboxes*, a module system for object-oriented languages that provides for method addition and replacement. Moreover, the changes made by a classbox are only visible to that classbox (or classboxes that import it), a feature we call *local rebinding*. To validate the model we have implemented it in the Squeak Smalltalk environment, and performed benchmarks.

1 Introduction

It is well-established that object-oriented programming languages gain a great deal of their power and expressiveness from their support for the *open/closed*

⋆ We gratefully acknowledge the financial support of the Swiss National Science Foundation for the projects "Tools and Techniques for Decomposing and Composing Software" (SNF Project No. 2000-067855.02) and "Recast: Evolution of Object-Oriented Applications" (SNF 2000-061655.00/1).

Email addresses: bergel@iam.unibe.ch (Alexandre Bergel), ducasse@iam.unibe.ch (Stéphane Ducasse), nierstrasz@iam.unibe.ch (Oscar Nierstrasz), wuyts@ulb.ac.be (Roel Wuyts).

principle [1]: classes are *closed* in the sense that they can be instantiated, but they are also *open* to incremental modification by inheritance.

Nevertheless, classes and inheritance alone are not adequate for expressing many useful forms of incremental change. For example, most modern object-oriented languages introduce *modules* as a complementary mechanism to structure classes and control visibility of names. Reflection is another example of an increasingly mainstream technique used to modify and adapt behaviour at run-time. Aspect-oriented programming, on the other hand, is a technique to adapt sets of related classes by introducing code that addresses cross-cutting aspects.

In this paper we focus on a particular technique, known as *class extensions*, which addresses the need to extend existing classes with new behaviour. Smalltalk [2], CLOS [3], Objective-C [4], and more recently MultiJava [5] and AspectJ [6] are examples of languages that support class extensions. Class extensions offer a good solution to the dilemma that arises when one would like to modify or extend the behaviour of an existing class, and subclassing is inappropriate because that *specific* class is referred to, but, one cannot modify the source code of the class in question. A class extension can then be applied to that specific class.

Despite the demonstrated utility of class extensions, a number of open problems have limited their widespread acceptance. Briefly, these problems are:

(1) *Globality.* In existing approaches, the effects of a class extension are either global (*i.e.*, visible to all clients), or purely local (*i.e.*, only to specific clients named in the application of the class extension). In the first case, clients that do not require the class extension may be adversely affected. In the second case, *collaborating clients* that are not explicitly named will not see the class extension, even though they should.
(2) *Conflicts.* If two or more class extensions attempt to extend the same class, this may lead to a conflict. In existing approaches, conflicts are either forbidden, or extensions are linearized, possibly leading to unexpected behaviour. In either case, the utility of class extensions is severely impacted.

We propose a modular approach to class extensions that largely solves these two problems by defining an implicit context in which class extensions are visible. A *classbox* is a kind of module with three main characteristics:

- It is a *unit of scoping* in which classes, global variables and methods are defined. Each entity belongs to precisely one classbox, namely the one it which it is first *defined*, but an entity can be made visible to other classboxes by *importing* it. Methods can be defined for any class visible within a classbox, independently of whether that class is defined or imported. Methods defined

(or redefined) for imported classes are called *class extensions*.
- A class extension is *locally visible* to the classbox in which it is defined. This means that the extension is only visible to (i) the extending classbox, and (ii) other classboxes that directly or indirectly import the extended class.
- A class extension supports *local rebinding*. This means that, although extensions are locally visible, their effect extends to all collaborating classes. A classbox thereby determines a namespace *within* which local class extensions behave *as though they were global*. From the perspective of a classbox, the world is flattened.

We have previously introduced classboxes by means of a specialized method lookup algorithm [7] reproduced in Section 5. The main contributions of this paper are: (i) A set-theoretic account of the semantics of classboxes that does not require a special method lookup algorithm and (ii) a detailed description of the prototype implementation in the Squeak Smalltalk system, including performance benchmarks.

The rest of the paper is structured as follows. Section 2 presents the problems in supporting unanticipated changes by giving a motivating example and Section 3 outlines the classbox model. Then a set-theoretic account of classboxes is given in Section 4. We discuss implementation issues arising in our prototype implementation in Section 5. Section 6 contrasts classboxes with related work. Finally, in Section 7 we conclude with some remarks concerning future work.

2 Motivation: Supporting Unanticipated Change

Class extensions provide a mechanism to support *unanticipated changes* in a static setting. Let us first consider a typical scenario, which will enable us to establish some key requirements for class extensions, while highlighting the main problems to be overcome.

A Link-Checker is an application whose purpose is to report a list of the dead links on a web-page at a given URL. One natural way to implement a Link-Checker, depicted in Figure 1, is to download the HTML page from the remote website, and parse it to get an abstract syntax tree of the page composed of various elements representing the HTML tags. Then using a recursive call over the hierarchy, get the list of the links referenced in the page. The liveness of each these links elements is checked by pinging the associated host and trying to obtain the status of the linked page. When a timeout is issued or if the HTTP reply corresponds to an error the link is declared dead.

Based on this example we can identify four properties that a packaging system

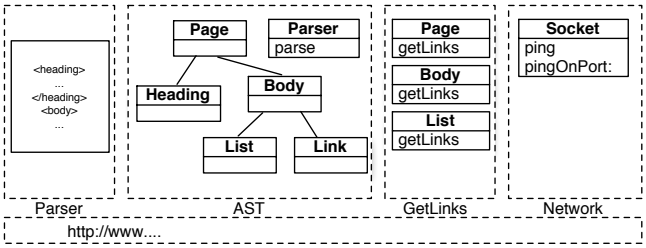


Fig. 1. The conceptual decomposition of the deadlink checker: an HTML parser, an abstract syntax tree for HTML documents, facility to get links from a page, and a network library.

for an object-oriented programming language should support: *class extensions* allowing *redefinition*, *locality* of changes, *propagation* of changes to collaborating clients, and resolution of *conflicts*.

Class Extensions with Redefinition. First, the different elements composing the solution should be packaged so that they can be used in further applications. We can identify the following modules: an HTML scanner and parser, an abstract syntax tree for the HTML elements, a recursive call over these elements to get links contained in a page and some network facilities. One key point is that we have to be able to group together the definitions of the getLinks methods in a module that is different from that of the AST. This means that the GetLinks module has to be able to extend the class definitions of the tree node elements.

Although languages such as CLOS, Smalltalk, MultiJava, and AspectJ offer some solutions, most otherlanguages (including Java), do not allow a class to be extended by a different module or package than the one defining the class. Note that subclassing the tree node elements is not a general solution, since clients that explicitly name the original class will not see the subclass extension.

In our development environment, the default Squeak distribution, the ping method used by the environment does not raise an exception but opens a dialog box when a target host cannot be reached. We therefore not only need the ability to add methods (for packaging the GetLinks module), but also to *redefine* them (to patch existing methods). *We therefore require a module system that supports class extensions with redefinition.*

Locality of Changes. The second key aspect concerns the visibility of changes, *i.e.*, which modules see the extensions made by other modules. In most approaches that support them, class extensions have a global visibility. All clients have a common view of any given class, and any extensions are also seen by all clients. This may lead to unexpected behaviour for some clients.

In the case of the ping method, we only want our redefined version to be visible

within the scope of our application. Other applications may actually rely on the *ad-hoc* behavior provided by ping. Therefore the extensions and changes to the system made by one module should not impact the system as a whole, but only the module introducing the changes and its client modules. *Class extensions should be confined to the module that introduces them.*

Local Rebindings. Even though class extensions should be visible only to the module that introduces them, the actual effect *from the perspective of that module* should be as if the extension were global.

The pingOnPort: method first adjusts the port (value kept in a variable) and then call the ping method. We want that any call to ping made by pingOnPort: triggers the definition brought by our LinkChecker application, even if pingOnPort: is defined in a scope that also contains a previous definition of ping. *Class extensions visible within a module should propagate to collaborating clients.*

Conflicts. Class extensions are useful when, for instance, a library needs to add a particular method to a class provided by the system. Conflicts arise when an application relies on two modules that extend the same method of the same class in different ways.

The ping method provided by Squeak is useful for pinging a remote host. Its default behavior is to display the result in a popup window. The Link-Checker application redefines this method to make it yield a value and to raise an exception if the host is not reachable. Conflicts can arise with other modules that make changes to this method. As a concrete example, Squeak has a SocketICMP module that implements the ICMP network protocol. Amongst other things, this implementation redefines the ping method with an ICMP-based implementation. Using both the Link-Checker and the SocketICMP module therefore leads to a conflict because both redefine the method ping.

There are several ways to handle this conflict: (1) the definition in Link-Checker overrides the definition in SocketICMP's, (2) SocketICMP's definition overrides Link-Checker's, (3) a conflict is detected at composition time and needs to be resolved, or (4) each extension is defined in a different namespace than that of the class.

With Smalltalk, CLOS and Objective-C the result depends on which module is loaded/initialized last which effectively impacts the system. On the other hand, Multijava and Hyper/J detect conflicting situations at compile time. *Selector namespaces*, Smallscript [8] and ModularSmalltalk [9] define the extension in a particular namespace: conflicts are avoided and both extensions are applied to the system within different scopes. *Resolution of conflicting class extensions should take the context of affected clients into account.*

3 Classboxes in a Nutshell

A *classbox* is a module containing *scoped definitions* and *import statements*. Classboxes define classes, methods and variables. Imported declarations may be *extended*, possibly redefining imported methods. When a classbox is instantiated, it yields a *namespace* in which the directly defined, imported and extended entities co-exist with the implicitly imported entities.

Scoped Definitions. A classbox defines *classes*, *methods*, or *variables*. Each class, method or variable *belongs* to precisely one classbox, namely the one in which it is originally defined. Classes and variables defined in a classbox are globally accessible to all methods in the scope of that classbox.

Imports. A classbox may import classes and variables from other classboxes. Imported entities thus become available within the scope of the importing classbox. An imported class may be *extended* with new methods, or methods that redefine existing methods. The extended class is then visible within the scope of the extending classbox, but not in the defining classbox of the extended class.

3.1 Scope of Methods

A method defined on a class in a classbox CB is visible within that classbox, and within other classboxes that import this class from CB. In a given classbox all the methods defined along the chain of import are visible within this classbox.

If several classboxes extend a class with a method with the same name but with different implementations, the implementation chosen during an invocation is the one that is reachable according to the import chain.

A classbox CB that defines a method that already exists in the import chain *hides* its former definition from this classbox CB and other classboxes that may import the extended class from CB.

3.2 The Link-Checker with Classboxes

This section shows how to use classboxes to modularize the Link-Checker example. Because classboxes have been fully implemented in the Squeak [10] environment, code fragments are presented in Smalltalk.

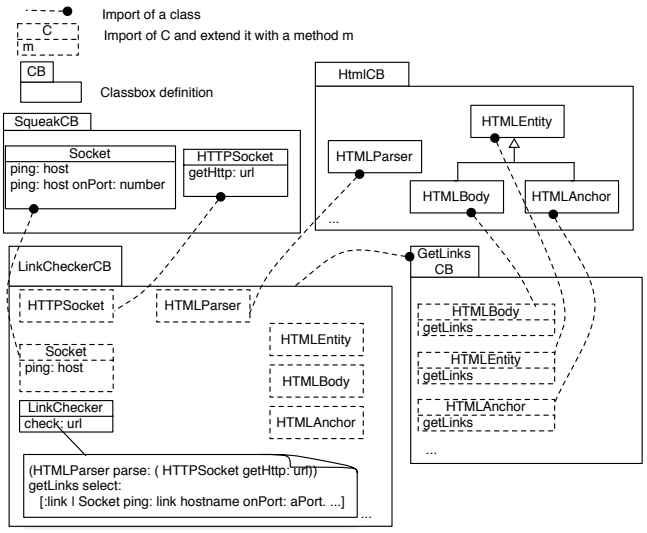


Fig. 2. The dead-link checker modularized with classboxes.

The architecture of the Link-Checker application is depicted in Figure 2. The classbox SqueakCB contains the network facility for checking the existence of a remote host (class Socket with class method ping: host) and for fetching the content associated to a given URL (class HTTPSocket with class method getHttp: url).

The classbox HtmlCB defines the HTML framework facilities. The class HTMLParser is used to parse a text, yielding an abstract syntax tree (AST) composed of nodes such as HTMLEntity (the root of the structure), HTMLBody, HTMLAnchor (representing a link), ...

The classbox GetLinksCB implements the recursive algorithm intended to produce a collection of all the links contained over the AST elements. It imports the relevant nodes from the classbox HtmlCB and *extends* each of the classes representing HTML tag elements by defining the corresponding getLinks methods.

The classbox LinkCheckerCB contains the actual link checker application. It defines the class LinkChecker, containing one method (check: url) which is the entry point of the application. This method first gets the raw content of a page designated by url using the class HTTPSocket). It then parses the page using the class HTMLParser, obtaining an AST of the page. Then it invokes the method getLinks on the root of that AST, obtaining a collection of all the links on the page. Finally it checks the liveness of these links by pinging the hosts mentioned in each link. LinkCheckerCB imports the complete classbox GetLinksCB, so all the extended classes (HTML nodes) are visible within it. As a consequence, within the classbox LinkCheckerCB the AST generated by HTMLParser (class imported from HtmlCB) understands the extensions brought by GetLinksCB. To solve the problem that the method ping: host in the classbox

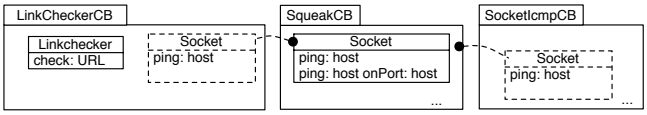


Fig. 3. The method ping is extended by two different classboxes. Conflict is avoided because extensions are confined to their respective classboxes.

SqueakCB displays its results in a dialog box, the classbox LinkCheckerCB redefines it to raise an exception instead.

3.3 Discussion

Locality of Changes. Although the method ping of class Socket is redefined, its visibility is confined to the LinkCheckerCB classbox. Unrelated code in the system relying on the original definition of this method is not affected. This illustrates both *class extensions with redefinition* and *locality of changes*.

Local Rebinding. The classbox SqueakCB defines the class Socket with two methods: ping: host onPort: number and ping: host. The first one calls the second one, and the latter posts a popup menu to display the result of pinging a host. This implementation is not suitable for our application. The classbox LinkCheckerCB imports the class Socket from SqueakCB and extends it by redefining the method ping: host with an implementation that throws an exception when a host is not reachable. Calling ping: host onPort: number within LinkCheckerCB triggers the new implementation of ping: host. This illustrates the *local rebinding* property.

Conflict. The classbox LinkCheckerCB extends the class Socket by redefining the method ping. This extension is local to the classbox. Figure 3 shows another classbox SocketIcmpCB that also imports the class Socket and redefines the same method ping. This class extension is local to SocketIcmpCB. Conflict is avoided because each extension is confined to the classbox that defines it.

4 The Classbox Model

This section presents a set-theoretic model that precisely defines the semantics of classboxes. We abstract away from the operational details of statements and expressions of a given object-oriented language, and instead focus on the key features that interact with classboxes. We start by introducing a basic model of *classes*, *objects* and *namespaces*, where we capture *instantiation*, *message sending*, and *self-* and *super-calls*.

On top of this basic model, we then show how classboxes are defined as a mechanism for introducing class extensions, and for controlling the visibility of class extensions in different namespaces. We show how *locality of changes* and *local rebinding* arise as a consequence of the way that classboxes are composed.

4.1 Environments

We use the basic concept of an extensible *environment* as a mechanism for modeling classes, objects and classboxes.

Definition 1 An *environment* $\epsilon : D \to R^\star$, is a mapping from some domain D to an extended range $R^\star = R \cup \{\bot\}$, such that the inverse image $\epsilon^{-1}(R)$ is finite.

We represent environments as finite sets of bindings, for example: $\epsilon_1 = \{a \mapsto x, b \mapsto y\}$ is an environment that maps a to x and b to y. All other values in the domain of this environment (for example, c) are mapped to $\bot$.

We normally leave out unessential parentheses. Since an environment is a function, we simply invoke it to look up a binding. In this case, $\epsilon_1 a = x$, $\epsilon_1 b = y$ and $\epsilon_1 c = \bot$.

Definition 2 An environment $\epsilon : D \to R^\star$ may *override* another environment ϵ'. We define $\epsilon \rhd \epsilon' : D \to R^\star$ as follows:

$$(\epsilon \rhd \epsilon')x \stackrel{\text{def}}{=} \begin{cases} \epsilon' x & \text{if } \epsilon x = \bot \\ \epsilon x & \text{otherwise} \end{cases}$$

For example, if $\epsilon_2 = \{b \mapsto z, c \mapsto w\}$, then $(\epsilon_1 \rhd \epsilon_2)a = x$, $(\epsilon_1 \rhd \epsilon_2)b = y$, and $(\epsilon_1 \rhd \epsilon_2)c = w$. We employ overriding both for method dictionaries and class namespaces.

4.2 Classes, Namespaces and Objects

The primitive elements of our model are the following disjoint sets: $\mathcal{C}$, a countable set of *class names*, $\mathcal{M}$, a countable set of *messages*, and $\mathcal{B}$, a countable set of *method bodies*.

Definition 3 A *method dictionary*, $\delta \in \mathcal{D}$ is an environment, $\delta : \mathcal{M} \to \mathcal{B}^\star$ that maps a finite set of messages to bodies.

For example, $\delta = \{m_1 \mapsto b_1, m_2 \mapsto b_2\}$ defines a dictionary d that maps message m_1 to body b_1 and m_2 to b_2, and all other messages to $\bot$.

Note that, for the purpose of this paper, we are not concerned with the implementation details of the method bodies. We only consider which kinds of messages are sent in the bodies.

Definition 4 A *class*, $\mathsf{c}\langle\delta, B, \epsilon\rangle$ consists of a method dictionary δ, a superclass name $B \in \mathcal{C} \cup \{\mathsf{nil}\}$, and an environment ϵ, called a *class namespace*, that binds class names to classes.

nil represents an empty class, from which the root of a class hierarchy inherits. By convention, every class namespace is assumed to contain the binding $\mathsf{nil} \mapsto \mathsf{c}\langle\emptyset, \mathsf{nil}, \emptyset\rangle$, which we therefore do not list explicitly.

Definition 5 An *object* $\mathsf{o}\langle c, \phi\rangle$ consists of a class c and an environment ϕ, which is a class namespace (obtained from c) extended with a binding for self.

Note that, for the present purposes, we do not model attributes (instance variables) of objects, aside from the pseudo-variables self and super.

We can send messages to classes and to objects. We use the notation $x[m]$ to send the message m to the class or object x.

Definition 6 We can *instantiate* an object by sending the message new to a class $c = \mathsf{c}\langle\delta, B, \epsilon\rangle$:

$$c[\mathsf{new}] \stackrel{\text{def}}{=} \mu\sigma.\mathsf{o}\langle c, \{\mathsf{self} \mapsto \sigma\} \rhd \epsilon\rangle$$

At this point we recursively bind self to the value of the object itself.

As usual, $\mu x.E$ binds free occurrences of x in E to the value of the recursive expression itself, *i.e.*, $\mu x.E \stackrel{\text{def}}{=} E\{\mu x.E/x\}$, where $E\{y/x\}$ is the usual substitution operation, replacing free occurrences of x in E by y while avoiding name clashes.

Although we do not model the internal details of method bodies here, we must take care to be precise about the environment within which methods are evaluated. As we shall see when we define classboxes, it is precisely the way in which these environments are composed that determines the scope within which class extensions are visible.

Definition 7 A *method closure* $\mathsf{m}\langle b, \phi\rangle$ consists of a method body b and a class namespace ϕ that additionally binds both self and super.

Note that super is bound by methods, not objects, since super-calls are relative to the class in which a method is defined, not the class from which the object is instantiated.

Definition 8 We can *send a message* m to an object $\mathsf{o}\langle c, \phi\rangle$, where $c = \mathsf{c}\langle \delta, B, \epsilon\rangle$ obtaining a method closure:

$$\mathsf{o}\langle c, \phi\rangle[m] \stackrel{\text{def}}{=} \begin{cases} \mathsf{m}\langle \delta m, \{\mathsf{super} \mapsto \mathsf{o}\langle \epsilon B, \phi\rangle\} \rhd \phi\rangle & \text{if } \delta m \neq \bot \\ \mathsf{o}\langle \epsilon B, \phi\rangle[m] & \text{else if } B \neq \mathsf{nil} \\ \bot & \text{otherwise} \end{cases}$$

This definition captures the basic method lookup algorithm of object-oriented programming languages. If the message sent does not correspond to a method defined in the class of the object, the lookup continues in the parent class, and so on. If the method is not found, the message is reported as not being understood ($\bot$). If a suitable method is found, it is evaluated in a context where super is bound to the current object, but from the perspective of the method's superclass. As we can clearly see, super is an object, not a class. Note that according to Definition 4 the superclass B can be nil.

Definition 9 A closure may be evaluated, in which case it may send various messages. Here we are interested in self- and super-sends, and static class references.

$$\mathsf{m}\langle b, \phi\rangle[\![\mathsf{self}\ m]\!] \stackrel{\text{def}}{=} (\phi\ \mathsf{self})[m]$$
$$\mathsf{m}\langle b, \phi\rangle[\![\mathsf{super}\ m]\!] \stackrel{\text{def}}{=} (\phi\ \mathsf{super})[m]$$
$$\mathsf{m}\langle b, \phi\rangle[\![C\ \mathsf{new}]\!] \stackrel{\text{def}}{=} (\phi\ C_\phi)[\mathsf{new}]$$

4.3 Classboxes

A classbox is an *open* entity that provides a number of classes, and which can be extended. When a classbox is *closed*, it yields an ordinary class namespace (Definition 4).

The key point in modeling classboxes is that multiple versions of the same class may be implicitly present within the same classbox. Suppose that we import the class LinkChecker from the classbox LinkCheckerCB, and we locally define a class Socket. Even though LinkChecker collaborates with Socket, ours is a *different* socket class that has nothing to do with the Socket class known to LinkChecker. To capture this aspect we must refine the notion of class names to express the *originating classbox* to which a class belongs:

- $\mathcal{C}$ is the countable set of *raw class names*,
- $\mathcal{X}$ is the set of classbox names,
- $\mathcal{C}^+ = \{C^n | C \in \mathcal{C}, n \in \mathcal{X}\}$ is the set of *decorated class names*.

The *decorated class name* simply encodes the classbox to which the class belongs, *i.e.*, where it was first defined. We call the superscript n of a decorated class name C^n its *origin*.

Definition 10 A raw class name C *matches* a decorated class name B^n if $C = B$:

$$C \sim B^n \textit{ iff } C = B$$

For example, when we use the raw class name Socket, it may not be clear *which* Socket class we are referring to. However the decorated class name $\mathsf{Socket}^{\mathsf{SqueakCB}}$ unambiguously identifies the Socket class first introduced in the SqueakCB classbox.

Note that it is this *same* class that is extended in LinkCheckerCB, since there is no Socket class defined there. There is no $\mathsf{Socket}^{\mathsf{LinkCheckerCB}}$.

Definition 11 A *classbox* $\mathsf{b}\langle n, \alpha\rangle$ consists of an identifier $n \in \mathcal{X}$ (*i.e.* classbox names) and a function α from class namespaces to class namespaces.

The intuition here is that a classbox is *open* because it can always be extended with new class definitions, imports and extensions. As a consequence, we do not yet know the class namespace of the classes it provides. However we can *close* a classbox, thereby fixing the class namespace of all the provided classes.

Definition 12 A *classbox* $\mathsf{b}\langle n, \alpha\rangle$ can be *closed* by sending it the close message, generating a fixpoint: $\mathsf{b}\langle n, \alpha\rangle[\mathsf{close}] \stackrel{\text{def}}{=} \mu\epsilon.\alpha\epsilon$

The resulting class namespace must be closed, *i.e.*, all used class names must be defined. Since α is a function from class namespaces to class namespaces, $\mu\epsilon.\alpha\epsilon$ represents a fixpoint in which all the classes provided by the classbox are made visible to each other.

Definition 13 We may *lookup* the decorated class name C^n corresponding to a raw class name C in a classbox $\mathsf{b}\langle n, \alpha\rangle$:

$$C_\alpha \stackrel{\text{def}}{=} \begin{cases} C^n \text{ if } \exists! n \in \mathcal{X}, (\mathsf{b}\langle n, \alpha\rangle[\mathsf{close}])C^n \neq \bot \\ \bot \quad \text{otherwise} \end{cases}$$

Suppose the LinkCheckerCB classbox is represented by $\mathsf{b}\langle \mathsf{LinkCheckerCB}, \alpha\rangle$. Then Socket_α yields $\mathsf{Socket}^{\mathsf{SqueakCB}}$, since SqueakCB is the origin of Socket in the LinkCheckerCB classbox.

Definition 14 An *empty classbox* with identifier n is: $empty(n) \stackrel{\text{def}}{=} \mathsf{b}\langle n, \lambda\epsilon \to \emptyset\rangle$. Note that $empty(n)[\mathsf{close}] = \emptyset$, *i.e.*, closing an empty classbox yields an empty class namespace.

Definition 15 We can *introduce* to a classbox $\mathsf{b}\langle n, \alpha\rangle$ a new class C that subclasses B (defined in a classbox $\mathsf{b}\langle m, \beta\rangle$) with δ as method dictionary by sending it the message def subclasses with.

$$\mathsf{b}\langle n, \alpha\rangle[\mathsf{def}\ C\ \mathsf{subclasses}\ B^m\ \mathsf{with}\ \delta]$$

$$\stackrel{\text{def}}{=} \begin{cases} \mathsf{b}\langle n, \lambda\epsilon.\{C^n \mapsto \mathsf{c}\langle \delta, B^m, \epsilon\rangle\} \rhd \alpha\epsilon\rangle, \text{ if } C_\alpha = \bot \\ \bot \qquad\qquad\qquad\qquad\qquad\qquad \text{otherwise} \end{cases}$$

Note that the formal parameter ϵ represents the fixpoint we obtain when the classbox is finally closed. We must therefore extend $\alpha\epsilon$ with the new subclass definition, obtaining $\{C^n \mapsto \cdots\} \rhd \alpha\epsilon$. We retain ϵ as a formal parameter so that the classbox remains open (*i.e.*, $\lambda\epsilon.\cdots$). The side condition states that it is an error to introduce a class that is already defined in the classbox. Within a classbox, only decorated class names occur. The newly introduced class has the origin n. We also explicitly identify the origin m of the superclass.

4.4 Importing Classes

Definition 16 A classbox $\mathsf{b}\langle n, \alpha\rangle$ may *import* a raw named class from another, classbox $\mathsf{b}\langle m, \beta\rangle$, by sending it the message import.

$$\mathsf{b}\langle n, \alpha\rangle[\mathsf{import}\ C\ \mathsf{from}\ \mathsf{b}\langle m, \beta\rangle]$$

$$\stackrel{\text{def}}{=} \begin{cases} \mathsf{b}\langle n, \lambda\epsilon.\{C_\beta \mapsto (\mu\phi.\beta(\epsilon \rhd \phi))C_\beta\} \rhd \alpha\epsilon\rangle, \text{ if } C_\alpha = \bot \\ \bot \qquad\qquad\qquad\qquad\qquad\qquad \text{otherwise} \end{cases}$$

Let us call the new classbox we obtain $\mathsf{b}\langle n, \alpha'\rangle$. α' extends α with the imported definition, but we must also take care that the environment of the imported class is properly extended with any pertinent definitions that occur in α'. As before, ϵ represents the class namespace that we obtain when we take the fixpoint of α'. We therefore pass ϵ to α so it is available to all the existing

class definitions in α. We must also look up the correct decorated class name C_β. Finally, we must bind this to the correct definition from β, *extended* with any new definitions from α'.

Suppose we would simply use $C_\beta \mapsto (\mu\phi.\beta\phi)C_\beta$, this would clearly be wrong, because the class we obtain would only see other class definitions from β, and not any definitions that may have already been extended in α. Instead, we create an *intermediate namespace* $\mu\phi.\beta(\epsilon \rhd \phi)$. $\epsilon \rhd \phi$ represents the environment of β *extended with any new definitions from* α'. We then pass this into β to make it available to *all* class definitions in β. Finally we extract this definition, bind it to C_β and use it to extend $\alpha\epsilon$.

Consider, for example, the import relationships in Figure 2. The classbox LinkCheckerCB imports HTMLParser from HtmlCB and HTMLEntity and its subclasses from GetLinksCB. If HTMLParser were naively imported from HtmlCB, it would not see the extensions imported from GetLinksCB. Instead, the import operation is defined so that when HTMLParser is imported, its environment (*i.e.*, ϕ) is extended by all definitions in LinkCheckerCB (*i.e.*, $\epsilon \rhd \phi$). So when HTMLParser is imported, it sees the extended versions of HTMLEntity and its subclasses. This is the *local rebinding* mechanism of classboxes.

Note that it is critical that HTMLEntity imported from GetLinksCB has the same origin as that expected by HTMLParser. If LinkCheckerCB or GetLinksCB were to define a *new* class HTMLEntity, then this would have a different decorated class name from the HTMLEntity originally defined in HtmlCB, and would therefore be invisible to HTMLParser.

4.5 Extending Imported Classes

Definition 17 A classbox $\mathsf{b}\langle n, \alpha\rangle$ may *extend* a raw class named class from another classbox $\mathsf{b}\langle m, \beta\rangle$, by sending it the message extend with.

$$\mathsf{b}\langle n, \alpha\rangle[\mathsf{extend}\ C\ \mathsf{with}\ \delta'\ \mathsf{from}\ \mathsf{b}\langle m, \beta\rangle]$$

$$\stackrel{\text{def}}{=} \begin{cases} \mathsf{b}\langle n, \lambda\epsilon.\{C_\beta \mapsto \delta' \rhd (\mu\phi.\beta(\epsilon \rhd \phi))C_\beta\} \rhd \alpha\epsilon\rangle & \text{if } C_\alpha = \bot \\ \bot & \text{otherwise} \end{cases}$$

where

$$\delta' \rhd \mathsf{c}\langle \delta, B, \epsilon\rangle \stackrel{\text{def}}{=} \mathsf{c}\langle \delta \rhd \delta, B, \epsilon\rangle$$

Extend works just like import, except that the imported class definition is extended with δ'.

As a consequence, importing a class is the same as extending it with a nil extension:

$$\mathsf{b}\langle n, \alpha\rangle[\mathsf{import}\ C\ \mathsf{from}\ \mathsf{b}\langle m, \beta\rangle] \quad \equiv \mathsf{b}\langle n, \alpha\rangle[\mathsf{extend}\ C\ \mathsf{with}\ \emptyset\ \mathsf{from}\ \mathsf{b}\langle m, \beta\rangle]$$

As should be clear from the definition, class extensions are purely local to the classbox making the extension. This guarantees *locality of changes*. Extensions become visible to other classboxes only when they are explicitly imported, or implicitly made visible by the mechanism of local rebinding (as seen in the HTMLParser example discussed above).

Method redefinition is supported since the δ' introduced by a class extension can redefine methods existing in the class being extended. For example, not only can the GetLinksCB classbox extend the HTMLEntity and related classes with a new getLinks method, but the LinkCheckerCB classbox can import Socket from the SqueakCB classbox and *redefine* the ping method.

4.6 Proving Classbox Properties

Proposition 1 A method defined in a classbox is visible within this classbox.

Proof. Because a method is defined either when a class is (i) defined or (ii) imported, this Proof is divided in two parts.

(i) Methods defined at the same time than the class they refer to are visible within the classbox where they are effectively defined. This first part of the proof consists in showing that defining a class C with a method m bound to a compiled method CM makes this method visible within the classbox (*i.e.* invoking m on an instance of C triggers the expected method CM).

Without loss of generality, assume that C has no superclass (*i.e.* it inherits from nil).

$$\begin{aligned}&\mathsf{b}\langle n, \alpha\rangle[\mathsf{def}\ C\ \mathsf{subclasses\ nil\ with}\ \{\mathsf{m} \mapsto \mathsf{CM}\}]\\ &= \mathsf{b}\langle n, \lambda\epsilon.\{C^n \mapsto \mathsf{c}\langle\{\mathsf{m} \mapsto \mathsf{CM}\}, \mathsf{nil}, \epsilon\rangle\} \rhd \alpha\epsilon\rangle \quad = \quad \mathsf{b}\langle n, \alpha'\rangle\end{aligned}$$

Closing this classbox yields:

$$\mathsf{b}\langle n, \alpha'\rangle[\mathsf{close}] = \mu\epsilon.\alpha'\epsilon = \varphi = \{C^n \mapsto \mathsf{c}\langle\{\mathsf{m} \mapsto CM\}, \mathsf{nil}, \varphi\rangle\} \rhd \alpha\varphi$$

Now the instance of this class C^n is $\mathsf{obj} = \mathsf{o}\langle\varphi C^n, \{\mathsf{self} \mapsto \mathsf{obj}\} \rhd \varphi\rangle$. Sending a message m to it yields:

$$\mathsf{obj}[\mathsf{m}] = \mathsf{m}\langle\{\mathsf{m} \mapsto \mathsf{CM}\}\mathsf{m}, \{\mathsf{super} \mapsto \mathsf{c}\langle\emptyset, \mathsf{nil}, \emptyset\rangle\} \rhd \varphi\rangle$$

The implementation identified for the method m is the result of $\{\mathsf{m} \mapsto \mathsf{CM}\}\mathsf{m} = \mathsf{CM}$.

(ii) Methods defined when importing a class are visible within the importing classbox.

$$\mathsf{b}\langle n, \alpha\rangle[\mathsf{extend}\ C\ \mathsf{with}\ \{\mathsf{m} \mapsto \mathsf{CM}\}\ \mathsf{from}\ \mathsf{b}\langle m, \beta\rangle]$$

$$= \mathsf{b}\langle n, \lambda\epsilon.\{C^n \mapsto \mathsf{c}\langle\{\mathsf{m} \mapsto \mathsf{CM}\}, \mathsf{nil}, \epsilon\rangle\} \rhd (\mu\phi.\beta(\epsilon \rhd \phi))C_\beta\} \rhd \alpha\epsilon\rangle$$

Assuming that $C_\beta = C^p$ closing this classbox yields:

$$\mathsf{b}\langle n, \ldots\rangle[\mathsf{close}] \quad = \varphi = \{C^p \mapsto \mathsf{c}\langle\{\mathsf{m} \mapsto CM\}, \mathsf{nil}, \varphi\rangle\}$$

The rest of the proof follows what is already shown in (i).

Proposition 2 Importing a class makes its methods previously defined visible in the importing classbox.

Proof. If $\mathsf{b}\langle m, \beta\rangle C_\beta = \mathsf{c}\langle\{\mathsf{m} \mapsto \mathsf{CM}, \}, B\rangle\epsilon$ then

$$\mathsf{b}\langle n, \alpha\rangle[\mathsf{import}\ C\ \mathsf{from}\ \mathsf{b}\langle m, \beta\rangle]$$

$$= \mathsf{b}\langle n, \lambda\epsilon.\{C_\beta \mapsto (\mu\phi.\beta(\epsilon \rhd \phi))C_\beta\} \rhd \alpha\epsilon\rangle$$

Assuming that $C_\beta = C^p$ closing the resulting classbox yields:

$$\mathsf{b}\langle n, \ldots\rangle[\mathsf{close}] \quad = \varphi = \{C^p \mapsto \beta C^p\} = \{C^p \mapsto \mathsf{c}\langle\{\mathsf{m} \mapsto \mathsf{CM}\}, B, \varphi\rangle\} \rhd \alpha\varphi$$

Then as already shown in the first proof, sending a message m to an instance of φC^p triggers the execution of CM.

Proposition 3 Within a classbox, a method redefinition takes precedence over its former implementation.

Proof. Within a classbox $\mathsf{b}\langle m, \beta\rangle$ a class C has in its method dictionary an entry m bound to a first implementation $\mathsf{CM1}$. This proof consists in showing that importing C in another classbox and redefining m bound to $\mathsf{CM2}$ hides the former implementation.

If $\mathsf{b}\langle m, \beta\rangle C_\beta = \mathsf{c}\langle\{\mathsf{m} \mapsto \mathsf{CM1}\}, B, \epsilon\rangle$ then

$$\mathsf{b}\langle n, \alpha\rangle[\mathsf{extend}\ C\ \mathsf{with}\ \{\mathsf{m} \mapsto \mathsf{CM2}\}\ \mathsf{from}\ \mathsf{b}\langle m, \beta\rangle]$$

$$= \mathsf{b}\langle n, \lambda\epsilon.\{C_\beta \mapsto \{\mathsf{m} \mapsto \mathsf{CM2}\} \rhd (\mu\phi.\beta(\epsilon \rhd \phi))C_\beta\} \rhd \alpha\epsilon\rangle$$

Assuming that $C_\beta = C^p$ closing the resulting classbox yields:

$$\mathsf{b}\langle n, \ldots\rangle[\mathsf{close}] = \varphi = \{C^p \mapsto \{\mathsf{m} \mapsto \mathsf{CM2}\} \rhd \beta C^p\} =$$

$$\{C^p \mapsto \{\mathsf{m} \mapsto \mathsf{CM2}\} \rhd \mathsf{b}\langle \mathsf{m} \mapsto \mathsf{CM1}, B\rangle\varphi\} =$$
$$\{C^p \mapsto \mathsf{c}\langle\{\mathsf{m} \mapsto \mathsf{CM2}\}, B, \varphi\rangle\}$$

The conclusion of this proof follows the end of the very first proof. Instantiating C^p and sending the message m executes the new implementation $\mathsf{CM2}$.

4.7 Resolving Diamond Conflicts

Conflicts are largely avoided. Classes that coincidentally have the same name but are introduced in different classboxes do not conflict because they have separate origins. Contradictions arising from attempts to import the same class from different classboxes of course cannot be resolved automatically. However, an important class of *indirect* conflicts is automatically resolved by the nature of the local rebinding mechanism.

Figure 4 illustrates a diamond pattern arising from two import chains with a common ancestor class. Classbox CB1 defines a class A which provides a method foo returning the value 1. This class is imported by CB2 where the method foo is redefined to return 2. CB2 also defines a subclass of A named B. In a similar way, classbox CB3 imports A from CB1 and redefines foo to return 3. A subclass of A named C is also defined. A fourth classbox CB4 imports B from CB2 and C from CB3. CB4 does not explicitly import class A.

In the context of CB4 invoking foo on an instance of B yields the value 2, whereas invoking foo on an instance of C yields 3. However, if CB4 would explicitly import A from any one of CB1, CB2 or CB3, then that version of A

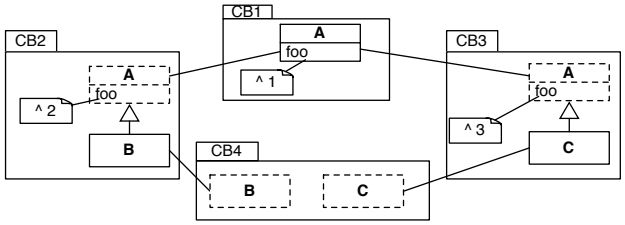


Fig. 4. Resolving Diamond conflicts

would be visible to both B and C. For example, if CB4 would import A from CB1 and redefine foo to return 4, then both instances of B and C would return 4 when foo is invoked.

5 Implementation Issues

Classboxes can be implemented by changing the method lookup algorithm in the virtual machine. This requires a virtual machine that is available for changing, which is why we performed our experiments in Squeak, a Smalltalk environment of which the virtual machine is open source [10,11]. We adapted the method lookup and compiled a new virtual machine that is classbox-aware. In this section we evaluate the impact of this extended method lookup algorithm on performance.

5.1 Method Lookup Description

Encoding the classbox with the method signature makes it possible for different implementations a method to coexist. However, to take advantage of this, the method lookup mechanism has to be changed as well. Figure 5 describes the lookup algorithm we implemented that ensures the local rebinding property.

The algorithm first checks whether the class in the current classbox implements the selector we are looking for (lines 5 to 9). If it is found, the lookup is successful and we return the found method (line 9). If it is not found, we recurse. The algorithm favours imports over inheritance, meaning that first the import chain is traversed (in lines 12 to 18) before considering the inheritance chain (in lines 19 to 30). This last part is the difficult part of the algorithm, since we need to find the classbox where the superclass is defined that is closest to the classbox we started the lookup from. Therefore the algorithm remembers the path while traversing the import chain (line 12), and uses this when determining the classbox for the superclass (line 21).

```
1  lookup: selector class: cls
2              startBox: startbox currentBox: currentbox classboxPath: path
3
4      | parentBox theSuper togoBox newPath |
5      self
6          lookup: selector
7          ofClass: cls
8          inClassbox: currentbox
9          ifPresentDo: [:method | ^ method].
10     parentBox := currentbox providerOf: cls name.
11     ^ parentBox
12         ifNotNil: [path addLast: parentBox.
13                    self
14                       lookup: selector
15                       class: cls
16                       startBox: startbox
17                       currentBox: parentBox
18                       classboxPath: path]
19         ifNil: [theSuper := cls superclass.
20                 theSuper ifNil: [^ cls method: selector notFoundIn: cls].
21                 togoBox := path  detect: [:box | box scopeContains: theSuper].
22                 newPath := togoBox = startbox
23                                ifTrue: [OrderedCollection with: startbox]
24                                ifFalse: [path].
25                 self
26                     lookup: selector
27                     class: theSuper
28                     startBox: startbox
29                     currentBox: togoBox
30                     classboxPath: newPath]
```

Fig. 5. The lookup algorithm that provides the local rebinding.

5.2 Import Takes Precedence Over Inheritance

Figure 5, lines 11-12 shows that if a class is imported (parentBox is not nil) then the lookup pursues in the provider classbox. If this class is not imported (parentBox is nil), as shown at the line 19, then the lookup continues in the superclass.

The lookup in a superclass is done only if it is stated that a class does not provide any implementation for a given message. Within the classbox model this implies that we have to run over the chain of imports to make sure that a classbox does not extend this class with the corresponding method.

Figure 6 illustrates this property of the algorithm by depicting an example. It shows four classboxes: GraphicCB, RoundedWindowCB, DoubleBufferCB and DoubleBufferAndRoundedCB. Each of theses defines extensions or simply imports classes to combine some of the extensions.

GraphicCB defines a hierarchy composed of three classes: Component provides the methods update and paint, and Window and Frame both override the method paint. Window and Frame are imported in RoundedWindowCB. This first class is extended with a new implementation of paint to make corners of windows smooth by rounding them. DoubleBufferCB extends Component, which is imported from GraphicCB, and simply imports Frame from this same

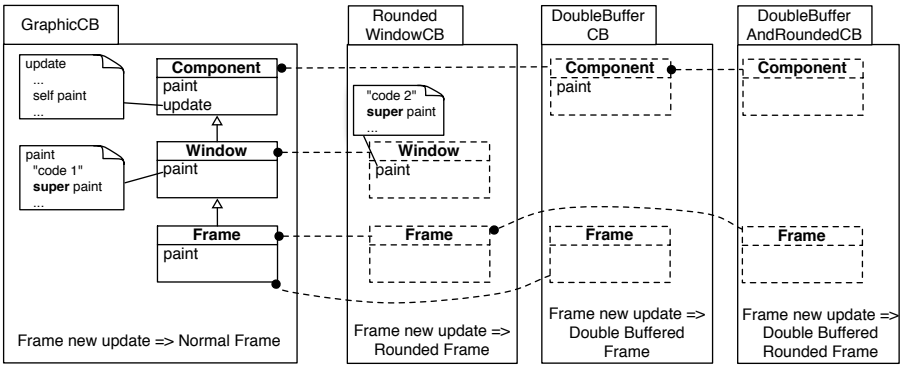


Fig. 6. Import takes precedence over inheritance

classbox. Component is extended with a redefinition of paint to take double buffering facility into account. Finally, DoubleBufferAndRoundedCB combines the two characteristics by importing Component from DoubleBufferAndRoundedCB and by importing Frame from RoundedWindowCB.

In RoundedWindowCB the new implementation of paint does a super paint which executes the paint method in GraphicCB. Evaluating Frame new update in RoundedWindowCB triggers the update method contained in Component and the local definition of paint is executed, the one provided by RoundedWindowCB.

DoubleBufferAndRoundedCB combines the double buffer and the rounded facilities by importing Component from DoubleBufferCB and Frame from RoundedWindowCB. Evaluating Frame new update in DoubleBufferAndRoundedCB triggers update defined in GraphicCB which send the message paint. The implementation taken is the one provided by RoundedWindowCB because Frame is imported from it. This implementation does a super paint, which execute the paint method defined in DoubleBufferCB.

5.3 Method Lookup Performance

Making the overhead related to our new method lookup as low as possible was one of our major concerns. Compared to the description given in [7], our implementation of the model is greatly enhanced: there is no need to modify the VM (due to the message passing control mechanism [12] offered by Squeak) and the cost of the new method lookup greatly reduced (thanks to a cache mechanism).

Classboxes allow you to have several versions of a method to coexist simultaneously. Depending on where this method is called from (*i.e.* from which classbox) the right method implementation is selected according to the method lookup algorithm described previously. When a classbox extends a class it can

either be a method addition or a method redefinition. With our current implementation, calling a method that has been simply added by a classbox does not impose any overhead. However calling a method that has been redefined has an extra cost: the lookup algorithm previously presented is performed. However, this result is cached. Our cache mechanism is based on the following basic assumption: *a redefined method is often called by the same object within the same classbox*. The byte-code of an extended method is transformed to include 5 byte-codes that check if the caller for this method is the one that has been previously cached. For method addition there is no need to use a cache because there is only one version of the method present in the system.

The following table illustrates the cost of the lookup of a redefined method compared with traditional lookup.

6,000,000 calls	**Classbox lookup (ms)**
Over 1 Classbox	5176
Over 2 Classboxes	5126
Over 3 Classboxes	5145
Normal method	1477

The experiment consists in calling 6 millions times a method that is redefined. It shows that there is a constant overhead that does not depend on the graph of import. This overhead is due to the extra few byte-codes added at the beginning of the method. The method used for the benchmarks is composed of one byte-code (simply return a numerical value). The same method that checks if the cache is valid is about 2.5 times ((5176 - 1477) / 1477) slower.

6 Related Work

Selector Namespaces. Languages like ModularSmalltalk [9], Subsystems [13] and Smallscript [8] provide a scoping mechanism called *Selector Namespaces*, in which methods are inserted. As a result, class extension conflicts are avoided, and several applications can bring the same class extension referring to the same class and method without interfering with each other. As a result, class extensions are not globally visible, but confined to a bounded scope. However selector namespaces do not support the *local rebinding* property, since a new definition does not take precedence when original code is called.

Multijava. Multijava [5] is an extension of Java that supports *open classes* and *multiple method dispatch*. An *open class* is a class whose methods are extensible. New methods can be added to an open class. These new methods are visible within the package that provides them and in the packages importing that package. Method redefinitions are not allowed: an open class cannot have

one of its existing methods redefined. On the other hand, two class extensions can define a method on the same class with the same signature. In that case the extensions are scoped separately.

Unit. MZScheme [14] offers an advanced module system where a *unit* is the basic building block. A unit is a packaging entity composed of requirements, definitions and exports. Units have to be instantiated and composed with each other to form a program. The key point of this model is that connections between modules or classes are specified separately from their definitions. This principle allows a module to be instantiated at link time. Reusability and extensibility are expressed by recombining units. An application, made of units, can be recomposed and by aliasing new units can be inserted. Units differ from classboxes since a unit acts as a black box: a class within a unit cannot be extended. Instead a new unit has to be provided and included in a recomposition.

Hyper/J. Hyper/J [15] is based on the notion of *hyperspaces*, and promotes compositions of independent *concerns* at different times. Hyperslices are building blocks containing fragments of class definitions. They are intended to be composed to form larger building blocks (or complete systems) called *hypermodules*. A hyperslice defines methods on classes that are not necessarily defined in that hyperslice. Such methods define a class extension, and classes intended to be extended are known at integration time. However this kind of extension does not allow redefinition and consequently does not help in supporting unanticipated evolution.

Virtual Classes. The specification of a *virtual class* [16] [17] is completely analogous to the specification of a virtual procedure. By introducing a dynamic lookup of a class name in a hierarchy of encapsulating entities (module for Keris [18], collaboration interfaces for Caesar [19] [20], classes for gbeta [21], or teams for Objectteams [22]) it is possible to refine a class within a subentity. One limitation with virtual classes is that the "virtuality" is scoped to a hierarchy: outside this hierarchy a class is not virtual anymore. For instance let us assume C to be a virtual class attribute in a hierarchy H1. In an unrelated hierarchy H2, class C is not virtual anymore and cannot be redefined.

Object-Based Inheritance. By providing *true delegation*, Lava [23] supports dynamic unanticipated changes using class wrappers. By introducing a new language construct, an object a (instance of A) can delegate all non-understood messages it receives to a *delegatee* object b (instance of B). Lava provides a *true-delegation* mechanism whereas the self reference used in the class B refers to the delegating object a. Methods defined in b that are unknown to a are the extensions brought on a. So redefined or new methods are attached to a particular object rather than a class. True delegation provides a way for adding or redefining methods for a particular object whereas

classboxes extend classes.

7 Conclusion and Future Work

Classboxes address the problem that classical module systems do not offer the ability to add or replace a method in a class that is not defined in that module. Classboxes offer a minimal module system for object-oriented languages in which extensions (method addition and replacement) to imported classes are *locally visible*. Essentially, a classbox defines a scope within which certain entities, *i.e.*, classes, methods and variables, are defined. A classbox may *import* entities from other classboxes, and optionally extend them *without impacting the originating* classbox. Concretely, classes may be imported, and methods may be added or redefined, without affecting clients of that class in other classboxes. Local rebinding strictly limits the impact of changes to clients of the extending classbox, leading to better control over changes, while giving the illusion from a local perspective that changes are global.

To see the impact of classboxes on a real-world example we remodularized an existing application (the seaside web server application [24] built upon a web server [25]) with classboxes. The goal is to show the usefulness of class extensions by measuring the proportion of class extension among the defined methods (for more details see the technical report [26]).

We have implemented a proof-of-concept prototype of classboxes in Squeak. In our implementation, the method lookup mechanism in the Squeak virtual machine has been modified to take classboxes into account. This prototype exhibits an overall 10% slowdown in performance for real-world applications.

In the future we will analyze some very large applications developed without any local rebinding facilities in order to identify places where programmers simulated local rebinding.

Currently classboxes function purely as a packaging and scoping mechanism. We intend to investigate various extensions of classboxes. We expect that an integration with traits will be fruitful, as this will enable packaging of collaborating traits [27] (and their associated tests). Presently classboxes lack any notion of a *component model*. We expect that explicit interfaces and composition mechanisms for classboxes will increase their usefulness. In particular, we intend to investigate the application of encapsulation policies [28] to classboxes.

Acknowledgment. We also like to thank Curtis Clifton, Erik Ernst, Günter Kniesel and Peri Tarr for their valuable comments and discussions.

References

[1] B. Meyer, Object-oriented Software Construction, Prentice-Hall, 1988.

[2] A. Goldberg, D. Robson, Smalltalk-80: The Language, Addison Wesley, 1989, book scglib.

[3] A. Paepcke, User-level language crafting, in: Object-Oriented Programming : the CLOS perspective, MIT Press, 1993, pp. 66–99.

[4] L. J. Pinson, R. S. Wiener, Objective-C, Addison Wesley, 1988.

[5] C. Clifton, G. T. Leavens, C. Chambers, T. Millstein, MultiJava: Modular open classes and symmetric multiple dispatch for Java, in: OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2000, pp. 130–145.

[6] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold, An overview of AspectJ, in: Proceeding ECOOP 2001, 2001.

[7] A. Bergel, S. Ducasse, R. Wuyts, Classboxes: A minimal module model supporting local rebinding, in: Proceedings of JMLC 2003 (Joint Modular Languages Conference), Vol. 2789 of LNCS, Springer-Verlag, 2003, pp. 122–131, best paper award.

[8] D. Simmons, Smallscript, http://www.smallscript.com (2002).

[9] A. Wirfs-Brock, B. Wilkerson, An overview of modular Smalltalk, in: Proceedings OOPSLA '88, 1988, pp. 123–134.

[10] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, A. Kay, Back to the future: The story of Squeak, A practical Smalltalk written in itself, in: Proceedings OOPSLA '97, ACM Press, 1997, pp. 318–326.

[11] Squeak home page, http://www.squeak.org/.

[12] S. Ducasse, Evaluating message passing control techniques in Smalltalk, Journal of Object-Oriented Programming (JOOP) 12 (6) (1999) 39–44.

[13] A. Wirfs-Brock, Subsystems — proposal, OOPSLA 1996 Extending Smalltalk Workshop (Oct. 1996).

[14] M. Flatt, M. Felleisen, Units: Cool modules for hot languages, in: Proceedings of PLDI '98 Conference on Programming Language Design and Implementation, ACM Press, 1998, pp. 236–248.

[15] H. Ossher, P. Tarr, Hyper/J: multi-dimensional separation of concerns for java, in: Proceedings of the 22nd international conference on Software engineering, ACM Press, 2000, pp. 734–737.

[16] E. Ernst, Family polymorphism, in: J. L. Knudsen (Ed.), ECOOP 2001, LNCS, Springer Verlag, 2001, pp. 303–326.

[17] O. L. Madsen, B. Moller-Pedersen, Virtual classes: A powerful mechanism in object-oriented programming, in: Proceedings OOPSLA '89, ACM SIGPLAN Notices, Vol. 24, 1989, pp. 397–406.

[18] M. Zenger, Evolving software with extensible modules, in: International Workshop on Unanticipated Software Evolution, Malaga, Spain, 2002.

[19] M. Mezini, K. Ostermann, Conquering aspects with caesar, in: Proceedings of the 2nd international conference on Aspect-oriented software development, ACM Press, 2003, pp. 90–99.

[20] M. Mezini, K. Ostermann, Modules for crosscutting models, in: 8th International Conference on Reliable Software Technologies (Ada-Europe '03), svlncs, 2003.

[21] E. Ernst, gbeta – a language with virtual attributes, block structure, and propagating, dynamic inheritance, Ph.D. thesis, Department of Computer Science, University of Aarhus, Århus, Denmark (1999).

[22] S. Herrmann, Object confinement in Object Teams – reconciling encapsulation and flexible integration, in: 3rd German Workshop on Aspect-Oriented Software Development, SIG Object-Oriented Software Development, German Informatics Society, 2003.

[23] G. Kniesel, Darwin – dynamic object-based inheritance with subtyping, PhD thesis, CS Dept. III, University of Bonn, Germany (2000).

[24] Seaside: Squeak enterprise aubergines server, http://www.beta4.com/seaside2/.

[25] Comanche: a full featured web serving environment for Smalltalk, http://squeaklab.org/comanche.

[26] A. Bergel, S. Ducasse, O. Nierstrasz, R. Wuyts, Classboxes: Controlling visibility of class extensions, Technical Report IAM-04-003, Institut für Informatik, Universität Bern, Switzerland (Jun. 2004).

[27] N. Schärli, S. Ducasse, O. Nierstrasz, A. Black, Traits: Composable units of behavior, in: Proceedings ECOOP 2003 (European Conference on Object-Oriented Programming), Vol. 2743 of LNCS, Springer Verlag, 2003, pp. 248–274.

[28] N. Schärli, S. Ducasse, O. Nierstrasz, R. Wuyts, Composable encapsulation policies, in: Proceedings ECOOP 2004 (European Conference on Object-Oriented Programming), LNCS 3086, Springer Verlag, 2004, pp. 248–274.

Parcels: a Fast and Feature-Rich Binary Deployment Technology

Eliot Miranda [a] David Leibs [b] Roel Wuyts [c]

[a] *Cincom Systems*

[b] *Neometron, Inc.*

[c] *Decomp, Université Libre de Bruxelles, Belgium*

Abstract

While development of a software system is important, it is also very important to have suitable mechanisms for actually deploying code. Current state-of-the-art deployment approaches force the developer to structure the code in such a way that deployment is possible, thereby severely inhibiting reuse and comprehensibility of the system. This paper presents *parcels*, an atomic deployment mechanism for objects and source code that supports *shape changing of classes*, *method addition*, *method replacement*, and *partial loading*. The key to making this deployment mechanism feasible and fast is a *pickling* algorithm that allows the unpickling to be done iteratively instead of with a recursive descent parser. Parcels were developed for VisualWorks Smalltalk, and have been the default deployment mechanism the past years for thousands of customers.

Key words: Code Deployment, Packaging, Pickling, Smalltalk

1 Introduction

This paper considers deployment technologies that are vehicles for storing objects and their behaviour that permit their transportation *between* and importation *into* systems. From this perspective deployment technologies are an essential part of current programming practice. They provide a medium and a mechanism for upgrading, distributing or selling software, a means of physical sharing to reduce disc and memory footprint, and of logical sharing to simplify updating of multiple programs, or incremental updating of a single program.

Such deployment technologies take a number of forms such as source files, binary programs, binary shared and unshared libraries, and, in the OO world, many *pickling* formats [1,2,3,4,5]. A *pickling format* is a recursive grammar

for defining graphs of objects. An object graph is traversed and a stream of tokens in the grammar is produced that describes the graph. This is the *pickled* representation of the graph and is typically stored in a file (but can for example also be used to transfer objects across the network). A parser for the grammar can be used to reconstruct an equivalent graph, *unpickling* the objects therein. A first problem that often occurs is that the grammars encode a straightforward flattening of the elements in the graph, and consequently show a significant parsing overhead when they are read back.

A second problem shared by pickling formats is that at time of use all prerequisites of an object must be present for the object to be successfully restored. For example, a Java class file can only be loaded if its super class is already present and has a definition that matches that expected by the class file. As another example, a shared library linked against other shared libraries requires that those other libraries are available. Because all prerequisites have to be present, the system needs to be decomposable in a tree, whereas it really is a –possibly cyclic– graph. This poses problems during development, where it is sometimes needed to structure the code in such a way that it can be deployed. For example, a visitor in Java cannot be deployed independently from the classes it visits.

Parcels address the problems described above in two ways. First of all parcels use a pickling format that eliminates the need for the recursive descent parser that is normally used when unpickling. As a result the loading times are much faster. Secondly parcels support references to objects outside of the parcel, and, moreover, can be loaded even if not all prerequisites are present.

On top of that parcels can be unloaded (restoring the situation before they were loaded), support partial loading and have sophisticated mechanisms for solving loading and unloading problems, such as support to shape-change classes, method additions, and method replacements.

In short, parcels are a very fast and flexible deployment mechanism that has been enjoyed since the release of VisualWorks 3 by thousands of developers.

The rest of this paper is structured as follows. Section 2 starts by giving an overview of parcels and the features they offer, and introduces the `Color-Editing parcel` example used throughout the paper. Section 3 discusses pickling formats, and the particular one used in parcels to speed-up the loading process. Section 4 shows how parcels support the advanced loading and unloading features such as redefinition of entities, method replacements and partial loading. Section 6 validates the approach. Section 7 discusses problems with the parcels and how we plan to address them in our future work. Section 8 shows how related work handles binary code deployment. Section 9 concludes this paper.

2 Overview of Parcels

Parcels are a binary code deployment technology for VisualWorks Smalltalk that improves upon older technologies by being faster and much more flexible regarding loading and unloading. Parcels store arbitrary Smalltalk objects but are oriented towards the storage of objects that represent the initialized state of Smalltalk code modules. They allow the definition of namespaces, namespace-global variables, classes and methods. Note that methods can be defined on classes defined in that parcel, but can also be defined on classes that are part of the system the parcel is loaded in. When a method is added to an existing class in the system, we call it a *method addition*. When a method from a parcel replaces a method that already exists in the system, we talk about *method replacement*.

Loading a parcel adds the definitions from the parcel to the system, and unloading a parcel removes them. This is not without complications, because of method additions, method replacements and class redefinitions. The system has a sophisticated mechanism to ensure that code that was redefined by the parcel gets restored when the parcel is unloaded.

The parcel system manages class initialization and "uninitialization" automatically. Parcels provide a set of optional configurable actions for performing arbitrary tasks at various times in the load/unload cycle of a parcel. They include a set of prerequisite parcel names which are used to auto-load prerequisite parcels, and information such as a measure of foot-print and several kinds of optional meta-information that can be browsed without loading the parcel (such as version information, developer information or notes).

Parcels exist either as a byte stream, that may persist outside the system (typically in a disc file), or as an object in the system. In the system they may be under development, having never been written out, or loaded, in which case they are also subject to further development. The in-system form of a Parcel is an object that notes which code entities it defines, and various other properties (like the parcel's name, its prerequisites, version, comment, the methods replaced by a parcel on load, etc).

Parcels are accompanied by an optional source file that holds the source code for the binary code in the parcel. This source file is added to a registry of current source files on load, and removed from the registry on unload. This differs markedly from Smalltalk source file-ins where the filed-in source is appended to the system's changed source file, leaving a permanent side effect if the code is removed.

This paper discusses how parcels achieve this set of features. The following section introduces an example that will be used throughout the paper.

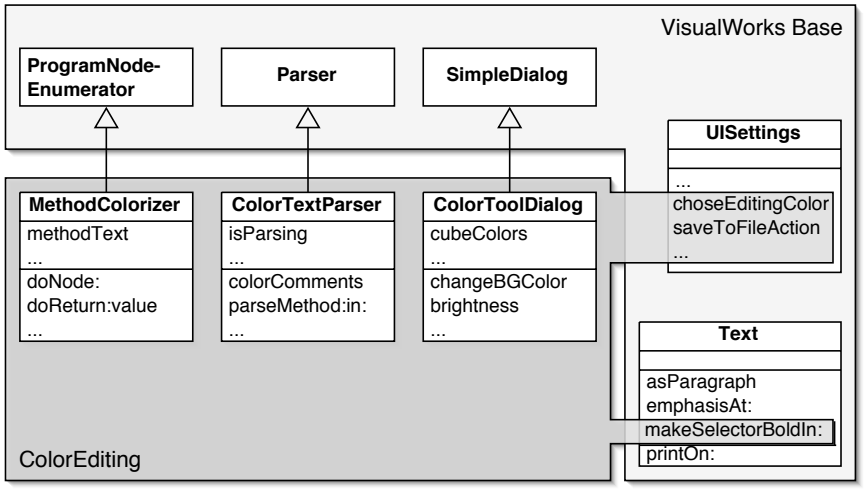


Fig. 1. Figure showing the ColorEditing parcel, that defines three classes (`MethodColorizer`, `ColorTextParser` and `ColorToolDialog`), some method additions on the base class `UISettings` and one method replacement on class `Text`

2.1 Example: The ColorEditing Parcel

Visualworks Smalltalk is distributed as a base image, containing the most commonly used features, in which parcels are loaded to extend this base image. One of these parcels allows code to be displayed in colour, highlighting syntax elements with colours. This changes the default behaviour implemented in the base image, where only the selector of a method is displayed in bold, and the method body is plain.

The ColorEditing feature is implemented as a parcel that provides a number of classes that implement its behaviour, a number of extension methods that integrate the settings a user can make in the existing Settings tool of VisualWorks (allowing colour settings to be customized from a GUI by an end-user) and one method replacement. The method replacement is needed to change the existing method that displays the selector of a method in bold, to now display it in the style the user desires. Figure 1 shows the ColorEditing parcel in a graphical format. This example is used throughout the paper to show concrete examples of parcel features.

3 Parcel Pickling Format

Parcels use a particular format to store an object graph in a file (called *pickling*). This section discusses pickling of objects in general, and then shows the format used by parcels that permits much faster loading.

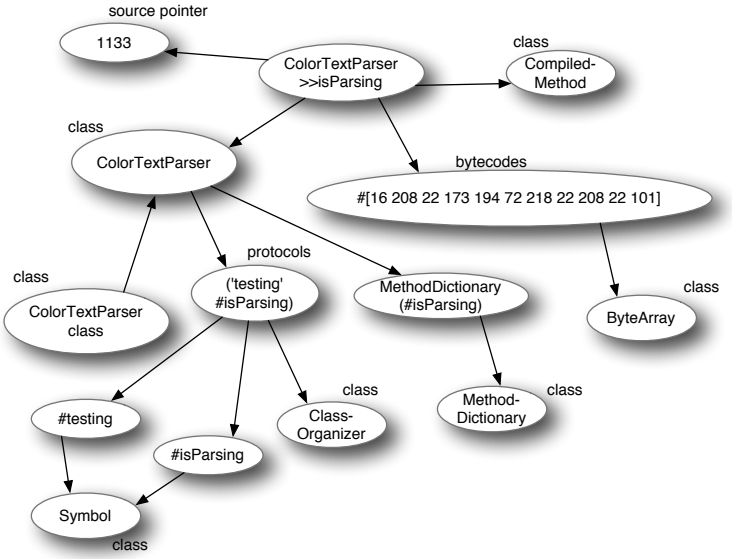


Fig. 2. Part of the object graph for class *ColorTextParser*, with a method *isParsing:*.

3.1 Current Formats for Pickling Objects

The goal of pickling is to store a graph where the nodes are objects, and the arcs are references between the objects. The nodes are typed. Possible node types depend on the language but typically include things like *primitive object types* (integers, floats, characters, strings, etc) and *arbitrary objects* (vectors of slots holding references to other objects in the graph, including their class, etc.). Figure 2 shows a small part of the object graph for the ColorEditing parcel from Figure 1, containing just some of the classes involved, and the method *isParsing:* from class ColorTextParser.

Traditional pickling formats use a stream of bytes organized as a header followed by a byte sequence that encodes an object graph according to a grammar. The grammar is designed to allow unpickling to be done as a form of bytecode interpretation. At each stage the interpreter decodes a portion of the byte sequence to yield a node in the graph. Each type of node has a particular bytecode analogous to a reserved word that introduces a phase in a programming language grammar. At each state in the parse the interpreter dispatches on the bytecode to determine how to interpret the phrase that follows and creates the object encoded by the phrase. The result of the unpickling is the object graph that was stored.

Typical optimizations for pickling formats encode common sub-ranges of the integers with their own bytecodes, etc. These techniques have a lot in common with those used for bytecoded language designs like Smalltalk itself. Although significant in reducing the size of a pickled byte stream they don't help to reduce the significant overhead of interpreting the stream.

The big deficiency with respect to speed of the traditional pickling arrangement is that it uses a recursive grammar. During the parse one may encounter an object whose class has not yet been defined, so the reference to the class will be a class definition phrase. The interpreter is faced with the difficulty of reading ahead to instantiate the class before it can instantiate the original object, and must be written as a recursive descent parser.

3.2 Pickling Objects the Parcel Way

As explained in Section 3.1, the object graph that needs to be stored uses typed nodes. In the case of VisualWorks, the nodes can be of the following type:

- *primitive object:* integer, float, character, string, etc.
- *arbitrary object:* vectors of slots holding references to other objects in the graph, including their class)
- *class definition*
- *method definition*
- *symbolic reference* to objects imported from the loading environment. Note that a symbolic reference to a class not only has the name of that class, but also the complete format of the class, such as the names of its named instance variables.

The pickling format of parcels relies on separating and ordering the descriptions of each node from the description of the references that form the arcs of the graph. Therefore the resulting file is divided into two big sections, preceded by a header: the *objects section* followed by the *references section*. Figure 3 shows how the graph from Figure 2 is stored in a parcel. It omits the header and full content of the pickled format for reasons of space.

Objects Section: The first section comprises a sequence of object descriptions clustered by class. Within the first section objects are present in a specific order. First come the symbolic references used to import the classes of all objects in the parcel whose class is not also defined by the parcel. Then come those literals represented as byte vectors, i.e. byte and two-byte strings, byte and two-byte symbols, byte arrays, floats, doubles, large positive and negative integers, and fixed-point scaled fractions. These are typically the only kinds of byte literals that occur in compiled methods. This is followed by information describing the classes defined by the parcel.

Note that classes are sorted according to their loading prerequisites. By default the loading prerequisite of a class returns the class' super class, but a user can extend this by overriding a method on the class' metaclass to return other prerequisites. A tool computes the transitive closure of the prerequisites

"refs to dependent classes"	**"Classes defined in parcel"**	**"arcs"**
'Kernel.Parser'	Kernel	1
16406	Parser	2
22	'ColorTextParser'	Kernel.ColorTextParser class
'source'	16393	Kernel
'mark'	0	#UISettings
'prevEnd'	9	...
'hereChar'	'superclass'	**"Instances of indexed objects"**
'token'	'methodDict'	OrderedCollection
'tokenType'	'format'	1
...	'subclasses'	5
"strings, bytearrays, floats, ..."	'instanceVariables'	CompiledMethod
'methodDict'	'organization'	141
'Kernel'	'name'	2
'testing'	'classPool'	1
'isParsing:'	'environment'	1
...	16413	...
#[16 208 28 173 ... 208 28 101]	7	**"compiled methods"**
#[16 208 23 173 ... 208 23 101]	29	ColorTextParser>>isParsing:
...	'source'	...
0.86	'mark'	
0.8	'prevEnd'	
...	'hereChar'	
1291792260	'token'	
1292108356	'tokenType'	
...	...	

Fig. 3. Part of the result from pickling the object graph of Figure 2. The first two columns form the objects section, while the third one is the references section. We added comments in bold to split the different parts of the sections.

relationship and does a topological sort on the set of classes to be written. This class ordering can also be done at load time, but it can take as much as one third of the complete load time for a parcel. Therefore we chose to do the computation once and store this information in the parcel itself.

References Section: The second section contains the reference information that encodes the arcs that connect the nodes. The reference information is in the same order as the objects in the object table. The first few references are those for the slots of the first object in the object table, followed by those for the slots of the second object, and so on. The VisualWorks virtual machine uses tagged pointers and encodes a 16-bit character set and a 30-bit signed 2's complement sub-range of the integers directly in object pointers. The reference information is also organized as four-byte tagged pointers that similarly encode either immediate integers, or immediate characters or the index of an object in the object table.

The parcel system achieves the above ordering by making the parcel writing process two-phase. The first phase determines the set of objects to be written. In the case of the example this results in a set containing the classes defined in the parcel (`MethodColorizer`, `ColorTextParser`, `ColorToolDialog`), objects for each of their methods, objects referenced in the methods, etc. The second phase clusters the objects collected in the first phase by class and writes them out in the order described above.

Both phases are implemented as an extension of an existing VisualWorks framework to trace object graphs (the `ObjectTracer`). The `ObjectTracer` uses the Visitor pattern [6], and so the parcel format is open to arbitrary extension. This is typically used for special purposes, for example by methods to collect their source code that should accompany the parcel file, or by `ExternalInterface` classes to include extra information such as the names of the external DLLs they provide interfaces to.

The advantage of this pickling format is that the unpickling can be done in an iterative instead of a recursive way, while still not imposing a hierarchical structure from the outset. At the end of reading the file, the objects are recreated. The following section explains this unpickling process in detail.

3.3 Unpickling a Parcel

The format described above is carefully organized to optimize load times. First of all, the organization ensures that a recursive descent parser is not needed since the class definitions are put before the instantiations of those classes. So the interpreter can batch-up instantiations of all objects of a particular class. As it instantiates objects they are placed in successive elements of an array called the *object table*. This batching provides most benefit for common object types (such as Strings or Symbols) that are the common literals of compiled methods, and for compiled methods themselves.

Once the first section has been parsed the object table is completely populated with the objects that form the nodes of the graph. The graph is knitted together by enumerating over the slots of each object in the object table, resolving its reference information into characters, integers or other objects in the table. The routine that does this operates on a single object. The routine is so simple that it is amenable to implementation as a virtual machine primitive, but performance is so good that we have yet to deploy the primitive! Performance comparisons are given in Section 6.

4 Applying a Parcel

Applying a parcel happens in several consecutive phases: the *preload phase*, the *load phase*, the *install phase* and the *postload phase*:

- *Preload Phase.* The preload phase is where the system is set up to apply a parcel. Prerequisite parcels are loaded, and preload behaviour defined in the parcel is executed.

- *Load Phase.* During the load phase, the objects defined in the parcel are restored. The load phase is done in its own context, so as not to disturb the existing system. Hence at any time prior to installation the load can be aborted and the system will not have been modified in any way.
- *Install Phase* When the load phase finishes successfully, the install phase actually installs the objects defined by the parcel in the system, performs class and global variable initialization and evaluates the parcel's post-load behaviour, if any. The installation phase is guaranteed not to fail, and is effectively atomic.
- *Postload Phase.* This phase allows all loaded parcels to install previously uninstalled method additions and replacements, and classes.

The next sections discuss the phases in detail, and especially the way load problems are handled automatically by the loader.

4.1 Parcel Preload Phase

In the preload phase, it is checked whether the prerequisite parcels of the parcel are already loaded. For each one that is not loaded, the system asks whether it is ok to load the prerequisite, with an option to do so automatically.

The system also has two hooks that allow parcels to execute code before the reading of the parcel starts (the *preread* hook) and before the actual loading starts (the *preload* hook). This fine-grained mechanism allows to test various conditions and to execute code at very specific intervals for those parcels that need it.

4.2 Parcel Load Phase

Loading the parcel means unpickling the contents of the parcel, and adding the resulting objects to a temporary context in the system. During the load phase, problems can occur. For example, the ColorEditing parcel defines the class `ColorTextParser` as a subclass of class `Parser`. When this class is not present, there is a load error. Parcels provide the following advanced features to automatically recover from certain load problems. The features are discussed in more detail afterwards.

(1) The parcel attempts to define a namespace, global, class or static that is already present in the system. This is handled by overrides.
(2) The parcel attempts to load a subclass of a class that has a different shape then that it had when the parcel was written. This is solved by

shape changing classes to effectively update the code defined in the parcel to take the new shape of the class into account.

(3) The parcel attempts to replace a method already present in the system. For this situation parcels support method replacement;

(4) The parcel attempts to use a class not present in the system, either for a method addition or to create a subclass. This is handled by *partial loading*, which boils down to remembering the code that cannot immediately be added.

We now discuss each of the solutions offered by parcels to handle these problems.

Redefinition Support

When a parcel defines a class, namespace or global that already exists, the definition in the system is replaced by the definition from the parcel. Moreover in the case of a class redefinition, existing instances are shape-changed to conform to the parcel's class.

Shape-Changing Classes The shape of a class is the named instance variables it defines. The shape of a class is important in Smalltalk, since Smalltalk methods refer to named instance variables by integer offsets. Hence changing the shape means that compiled methods have to be adjusted to make sure that they still use the correct offsets.

For example, suppose that a new release of the VisualWorks base adds an instance variable to the class `Parser`, and that a user loads the ColorEditing parcel in this new version. The `Parser` class in the system then has a different shape than the `Parser` class against which the methods were compiled. Hence the methods in the parcel need to be updated to make sure they use the correct offsets.

Such a change in definition can be detected because the shape of a class is stored in the parcel file (see the top of the left column in Figure 3 that lists the instance variables of class `Parser`). This information is used to compare the shape for the class import in the parcel against the shape of the class present in the system. When differences are detected the instances of classes that have changed shape are simply remapped, dropping the values of lost instance variables, and setting new variables to the value nil.

Method Replacement The example shows a method that is defined in the ColorEditing parcel that replaces an existing definition of that method in the system: method `makeSelectorBoldIn:` on class `Text`. Method replacement simply replaces the method with the one defined in the parcel. However, the original version of the overridden method is remembered by the system. When a parcel is unloaded it can therefore put the original methods back in place,

as will be discussed in Section 5.

Partial Loading

When deploying code it might happen that a class needed to load a parcel is absent. For example, the class `ColorEditing` adds methods to the class `UISettings`, but a user that does not need a tool for editing system settings could have removed this class. Another example is the class `ColorToolDialog`, that subclasses a class `SimpleDialog`, which again might not exist. Without support for partial loading, the ColorEditing parcel could not be loaded in such a scenario. The only thing that the user could do is to try to figure out the dependencies and create stubs for the needed classes, a tedious and difficult process.

Parcels solve this problem by *partial loading*: the parcel loader constructs a foster home for the code to live in until suitable parentage can be obtained. On encountering an import for a class that is not present the loader raises a warning which the user can respond-to either by aborting the load or by continuing. If the loader continues it creates an instance of a special class (`AbsentClassImport`), stores it at the relevant index in the object table, and initializes the object with all the format information available in the parcel (again using the class import information available in the parcel). Later on in the load attempts may be made to add methods to the `AbsentClassImport` or to subclass it:

Adding methods to a non-existent class. The case of adding methods is handled by the loader collecting these extensions and adding them to the transient properties of the loaded parcel as its uninstalled methods.

Subclassing a non-existent class. When a subclass needs to be made from an absent class, `AbsentClassImport` constructs an impostor super class, an instance of `AbsentClassImporter`, that has all the instance variables defined by the absent import and a few extra to hold information like the binding used to reference the class, and the real name of the absent super class. This impostor then continues to construct the class as for any ordinary class. Class-side code will be shape-changed to account for the extra bookkeeping information (the absent class's binding and name, etc) stored in the impostor. During the install phase of a parcel, this information is used to remember the code in the parcel that could not yet be loaded.

4.3 Parcel Install Phase

When the load phase finishes successfully, the install phase actually installs the objects defined by the parcel in the system: classes are added to their

super class's subclasses sets, globals are added to namespaces, and method additions and replacements are added to existing classes. At the same time uninstalled methods and uninstalled classes are segregated and stored in the parcel's transient properties so that they can be added should these classes become available later on. Once this is complete the system of classes and methods is in a valid state, one that could have been achieved through normal use of the programming environment.

At this point it is safe to perform class and global variable initialization. Although this initialization may cause run-time errors these will now be examples of buggy code, since all installable code has been properly installed, and the system is in a valid state. Hence the programmer is in a position to debug the problems as they would if they were developing unparceled Smalltalk code.

Lastly, the parcel's post-load action, an arbitrary block, is evaluated. Note that messages sent to instances of `AbsentClassImporter` (created during a partial load) are not invoked, since they merely wait to be installed when their prerequisites become available at a later time. The installation phase is guaranteed not to fail, and is effectively atomic.

4.4 Parcel Postload Phase

If a class that was previously absent was made available by a parcel load, then uninstalled methods and classes may become installable. During the post-load phase, all parcels in the system are checked to see whether they can now install previously uninstalled code. Method additions on absent classes are simply added to the now-present versions, making sure to check for any method replacements and adding these to the replaced method set. Uninstalled subclasses of the now-present class are asked to install themselves. The classes re-parent themselves, leaving the `AbsentClassImporters` to be garbage-collected, and their code is once again shape-changed to adjust to the loss of the extra bookkeeping information stored in the `AbsentClassImporter`.

The checking is done by sending a `#postLoad:` message to the parcel, which answers with a boolean indicating whether any uninstalled code was added. The loader continues to send `#postLoad:` to all parcels until all parcels answer false, which tells the loader that no further code was installed, and the system of parcels has reached a fixed-point. In this way the system can handle mutually recursive parcels. Figure 4 illustrates the solution. Four classes (`A`, `B`, `C` and `D`) are involved, that all inherit from eachother. Classes `A` and `C` belong to parcel `P1`, while parcel P2 contains classes `B` and `D`. Loading parcel `P1` installs only class `A` (providing its super class is present). A subsequent load of parcel `P2` will then install classes `B`, after which classes `C` and `D` get installed.

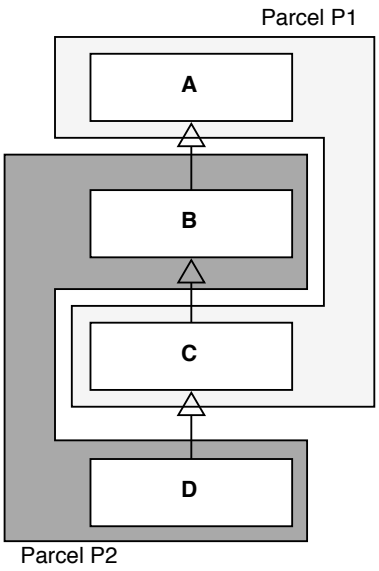


Fig. 4. Two mutually recursive parcels.

5 Unloading

The unloading of a parcel removes the definitions added by the parcel from the system, and removes the source file associated with the parcel (if there was one) from the registry of source files. Parcels tackle several unloading problems:

- When instances exist of classes that are unloaded, *obsolete classes* are created. An obsolete class is the remnant of a removed class. It continues to exist in the image for as long as either at least one compiled method references the removed class or at least one instance of the removed class is reachable. Through the usage of obsolete classes the instances can continue to function. There is one caveat when classes are removed that have subclasses. In this situation the system will prompt for confirmation, since removal of such classes causes problems. For example, removing parcel P2 from the example shown in Figure 4 results in a warning that asks for permission before continuing. If the parcel is then unloaded, problems arise for class `C`, since its super class is removed.
- When a parcel has method replacements (so-called *overrides* in parcel terminology), the code of the original methods is restored. This is possible because the original methods are remembered by the system in a central repository (the class *Override* and its subclasses), and the order in which they were added.

Application	classes/methods/extensions	format	load	write
VW-XML	48 / 543 / 0	chunk	2,615	244
		envy	2,000	900
		parcel	85	211
Soul (v. 2.3)	117 / 1,923 / 18	chunk	7,906	800
		envy	4,000	2,000
		parcel	393	676
RB	187 / 3,327 / 53	chunk	11,554	1,352
		envy	4,000	5,000
		parcel	630	1,920
Jun (v. 4.99.08)	757 / 25,212 / 0	chunk	110,729	8,694
		envy	11,000	14,000
		parcel	2,008	21,979

Table 1
First series of experiments that compare loading and writing times in VisualWorks 3 for the chunk, ENVY/Developer and parcel format (all times in milliseconds - thousands separated by a comma).

6 Validation

Over the past ten years almost all Smalltalk environments have been extended with various code storage mechanisms that have improved upon source files. This section first shows two series of benchmarks to compare the speed for reading and writing of parcels to other approaches. In a first series of experiments with VisualWorks 3, the ENVY/Developer ormat is compared to the chunk and parcel formats. A second series of experiments compares VisualWorks 7.2 parcels with the XML-chunk, VisualWorks Store, Squeak's Monticello and Dolphin's PAC format. Both series of experiments were performed on the same machine (AMD 3000+, 768 Mb of memory, Windows XP SP1). The section then finishes with an enumeration of some user benefits that show how code deployment is facilitated by the advanced loading features offered by parcels.

6.1 VisualWorks 3 Experiments

Table 1 shows the results of the first series of experiments that compares parcels with the following two formats:

- *envy* The binary format of ENVY/Developer *dat* files), the multi-user development add-on for VisualWorks [7]. We worked on a locally installed repository to eliminate network access, and took the times loading from and saving to the repository (not the importing of the code in the repository).
- *chunk format* The traditional Smalltalk textual chunk format [8].

Note that we only had access to ENVY/Developer, and that this version requires the virtual machine of VisualWorks 3. Therefore we used VisualWorks 3 for this whole first series of experiments, so that we can make a direct comparison between the numbers. We chose the following 4 applications of different sizes to do the comparisons:

- *VM-XML* [1] was a project to come to a dialect independent, XML-based file-out format. It is the smallest application we tested, having only 48 classes.
- *Soul* [9] [2], a logic programming language that lives in symbiosis with Smalltalk. We used the older version 2.3 for these tests, since it was the last version for VisualWorks 3.
- *Refactoring Browser* [10] [3] the browser that pioneered the integration of refactoring operations in a development environment.
- *Jun* [4] is a 3-D graphics framework that maps to OpenGL. It is the largest application we experimented with (consisting of 757 classes).

As can be seen in Table 1, parcels have the fastest loading speed of the formats tested. The chunk format is by far the slowest, since it is an ASCII format that is parsed on reading, adding the code to the system as the file is read. It is not a mechanism to store objects. The ENVY/Developer ormat is a bit faster than the chunk format, especially for larger applications. Regarding writing speed it can be noticed that for bigger applications the creation of parcels is slower than using "simple" formats such as the chink format. This is to be expected, since the writing process is two-phase and has to do more work.

6.2 VisualWorks 7.2 Experiments

In the following series of experiments we compared the reading and writing speed of parcels with more recent techniques from VisualWorks and other Smalltalk environments.

---

[1] version for *VW5i*, http://wiki.cs.uiuc.edu/VisualWorks/VisualWorks+XML+Framework

[2] version 2.3, http://prog.vub.ac.be/research/DMP/soul/soul2.html

[3] version for *VW30*, http://st-www.cs.uiuc.edu/users/brant/RefactoringBrowser/

[4] version 4.99.08, http://www.sra.co.jp/people/aoki/Jun/htmls/Download_e.html

Application	classes/methods/extensions	format	load	write
Classifications2	6 / 116 / 2	Store	2,194	1,095
		mcz	376	600
		PAC	238	787
		parcel	189	139
Soul (v. 3.2)	146 / 2,081 / 61	Store	17,586	16,690
	(131 / 1,834 / 51; no init)	mzc	6,799	5,000
	(129 / 1812 / 51; no init)	PAC	4,684	455
		parcel	1,514	1,677
Swazoo	101 / 6,646 / 4	Store	61,721	43,501
	(no initialization)	mcz	3,420	2,100
	(no initialization)	PAC	1,667	734
		parcel	1,764	49,479
SmallWiki	119 / 1,613 / 13	Store	12,959	8,664
	(no initialization)	mcz	5,776	4,000
	(117 / 1,539 / 8; no init)	PAC	3,016	1,095
		parcel	4,229	1,015
	(117 / 1,539 / 8; no init)	parcel	1,903	983
SIXX	37 / 271 / 100	Store	3,352	2,822
	(no initialization)	mcz	1,741	900
	(55 / 573 / 112)	PAC	1,485	4,031
		parcel	2,331	353

Table 2
Comparing loading and writing times in VisualWorks 7.2 for the chunk, Store, Monticello, PAC and parcel format (all times in milliseconds - thousands separated by a comma).

- *VisualWorks Store* Store is the multi-user development add-on for VisualWorks. Since it did not exist for VisualWorks 3, we compared it here with the other non-VisualWorks mechanisms. We installed a local *PostgreSQL* database, and used that for the experiments so that network latency was avoided. We experimented using the commonly used textual format.
- *Squeak Monticello mcz* Monticello is a distributed concurrent versioning system based on a declarative representation of Squeak source code. The *mcz* files are basically zip files that contain a manifest file giving meta-

information and a textual chunk file.
- *Dolphin PAC files* PAC files are the default mechanism for packaging code in the Dolphin environment. They are a chunk-like textual format.

For performing the comparison we took a number of applications that we could run in these different environments. Note that we had to use other applications (or other versions of applications) due to the differences between VisualWorks 3 and VisualWorks 7, such as the addition of namespaces in the latter. Note that we did not port the applications: we merely made sure we could the source code and perform measurements.

- *Classifications2* [11] [5]: The classifications model is the domain model for the StarBrowser. We used it because it is small and runs identical between Squeak and VisualWorks.
- *Soul*: A more recent version (3.2) of the language also used in the other series of experiments.
- *Swazoo* [6]: An open source Smalltalk HTTP server with resource and web request resolution framework.
- *SmallWiki* [7]: An implementation of a wiki-wiki server in Smalltalk.
- *SIXX* [8]: A XML framework for Squeak, Dolphin and VisualWorks.

The results of the experiments are shown in Table 2. Parcels and Dolphin's PAC format are the fastest mechanism for reading. The very good performance of PAC files is quite surprising, since it is basically a chunk format. Upon investigating the issue we found out that the reason is that we removed not only methods and classes that gave compatibility problems for loading (such as DLL/CC classes, that are used by VisualWorks to link with external C and C++ libraries), but also removed the initialization code that would normally be launched after filing in. The Store, chunk and parcel formats therefore do much more work: the code is loaded and then executed. To give an example about the difference this makes, we created a (defunct) SmallWiki parcel with exactly the same classes and methods as used by the Dolphin version, and removed the initialization code. The loading time then falls from 4,229 milliseconds down to only 1,903 milliseconds (versus 3,016 milliseconds for PAC files). Note that this does not explain the difference in loading speed for the SIXX parcel, but for SIXX we had completely different implementations for parcels and PAC.

Surprising is that the parcel writing is also quite fast, about on par with the writing of PAC and chunk files. We are investigating the issue of why the writing of the Swazoo parcel takes such a very long time.

[5] version 0.63, Cincom Public Store
[6] version 0.9.76-bb10, Cincom Public Store
[7] version 0.9.51, Cincom Public Store
[8] version 0.1h, http://www.mars.dti.ne.jp/~umejava/smalltalk/sixx/

Our experience has shown that the method replacement and partial loading facilities contribute significantly to improved programmer-productivity when componentising, and to the flexible configuration of deployed systems. Users of the Smalltalk environment *Visual Studio* that use SLL files and that switched to parcels reported that Parcels are much more convenient to use, because they allow for flexible packaging of the code. VisualWorks 3.0 was the first release of VisualWorks to support parcel source, method replacements and partial loading. It was also the first release to deliver all add-on modules in the form of parcels. Method replacements and partial loading were instrumental in being able to decompose the entire system and deliver it on schedule. Later released added minor enhancements.

The method replacement facility enables parcels to perform necessary base modifications on load and enables parcels that perform base modifications to be written such that they unload. Unloading makes it much easier for a programmer to try-out a parcel, and to use an extraneous tool in their development context temporarily, e.g. temporarily loading the mail tool into a development image.

The partial loading facility reduces the number of parcels that must be developed and maintained. They allow the programmer to maintain a single logical entity instead of a proliferation of physical parcels. In certain cases it may be more convenient to deploy a single parcel that in some circumstances only partially loads. The warning suppression block can be used to suppress absent class import warnings in a deployment context, allowing the parcel to load cleanly without alarming the user.

What makes the parcel system so usable is the combination of these features with the added speed benefits over other formats. This explains the popularity of parcels over other VisualWorks mechanisms as soon as they were released.

7 Future Work

The parcel system in its current state has been the code deployment mechanisms for VisualWorks since VisualWorks 3. It also served as the basis for the multi-user development system of VisualWorks (called *Store*). However, parcels have some drawbacks that we are planning to tackle in new releases.

Extending Shape-changing. The Parcel system currently has very rudimentary support for shape-change of instances. We plan to add an extensible

framework by which objects can intervene to augment the simple re-mapping shape-change done at present. Furthermore we plan to add a scheme for adding code to the current system that is used to shape-change instances in old parcels. Ted Kaehler has done a powerful scheme for Squeak's loadable component system [9]. But this scheme requires the user to provide shape-changing code at load time. While this is appropriate and useful in the hands of skilled Smalltalk developers it seems inappropriate for most deployment contexts.

Parcel Security Framework. Parcels have been used to implement a Smalltalk applet framework where code is loaded across the internet a la Java applets. However, no suitable security framework has yet been developed. We find the Java sandbox approach limiting, in that it works by restricting the full power of the system. Instead, we are investigating higher-level mechanisms, in particular proof-carrying code. Proof-carrying code approaches would require a type system such as [12]. Type inference systems are also useful when attempting to determine the boundaries of components [13]. We expect the combination of the Parcel System and a suitable type system to be very synergistic.

No lazy Loading. The standard parcel system does not support lazy loading, *à la* Java class files [14]. Currently the performance of the parcel system is so good that we have not needed to use lazy loading to improve load times, but more experience, for example in a web setting, may cause us to reconsider.

Building the underlying mechanism for a lazy-loading scheme is relatively straightforward in VisualWorks since all references to classes are through variable bindings. Therefore we can simply add and use subclasses of class `VariableBinding` that search for and load the class before returning it. However, a major issue for lazy loading is the design of the registry that maps requests for classes to the parcel to load. The current loader provides a user-specifiable set of directories in which to search for parcels when finding prerequisite parcels. But the simple set of parcel directories, while convenient in many contexts is inadequate when parcels are used to represent applets loaded over the net. Were parcel headers extended with a list of the classes they define then a policy of searching the path for the first parcel that defines the required class might suffice.

Ordering Method Additions When the loader installs method additions and method replacements, it installs all additions before any replacements, since replacements may attempt to invoke additions. One can construct artificial cases where this ordering is inadequate. While so far this policy has proven adequate in practice, it needs to be made full proof.

[9] Private communication. The results can be seen in the Squeak Smalltalk environment found at `http://www.squeak.org/`

8 Related Work

There is some older, undocumented related work. First of all, OTI extended *ENVY/Developer* (used in our benchmarks) with method loading. The resulting system is called *ENVY/App*. There were also some companies that implemented binary application loading themselves (such as Quallcomm). Since we were unable to find information on these systems, let alone obtain running systems, we were unable to compare them or benchmark them with parcels.

Java has support for pickling objects [5]. It uses a recursive approach to accomplish this. As noted in [5], "the current recursive traversal is suitable for only modest size graphs and will need to be extended to accommodate very large graphs." A similar approach is used in Python. It could be interesting to implement the pickling scheme proposed in this paper in Java or Python, and then compare the recursive approach with the non-recursive approach.

Regardless of the speed claims of the pickling format used, we do not know of other packaging mechanisms that have the loading and unloading features that parcels offer. The mechanism that comes closest is probably the *Classbox* system [15], a module system for object-oriented languages that allows *local rebinding*. Local rebinding means that a classbox (module) can make method additions and method replacements that are visible only to the classbox that defines them, without impacting the rest of the system. Such a feature is not available with classes. However, classboxes lack the meta information found in parcels (such as developer name, timestamps, etc.), storing arbitrary objects and last but not least, partial loading.

9 Conclusions

We have described a deployment technology for storing objects and their behaviour that permit their transportation between and importation into systems. The system is novel in that its binary format supports extremely fast loading and its provision of method replacements and partial loading frees the programmer from maintenance tasks required by less flexible technologies. This technology has proven itself in industrial use and underpins the product architecture of VisualWorks 3.0 and 4.0.

We have described some deficiencies of the parcel system and a number of avenues for further work to resolve these issues.

Acknowledgments

We would like to thank Alexandre Bergel, Gilad Bracha, Steve Dahl, Stéphane Ducasse, Brian Foote, and Ralph Johnson for reviewing drafts of this paper. Thanks are also due to the members of the VisualWorks development team who were and continue to be instrumental in the design and implementation of the Parcel system, and to many brave customers who have uncovered the bugs.

References

[1] S. R. Vegdahl, Moving structures between Smalltalk images, in: Proceedings OOPSLA '86, ACM SIGPLAN Notices, Vol. 21, 1986, pp. 466–471.

[2] Parcplace systems, objectworks reference guide, smalltalk-80, version 2.5, chapter 36, parcPlace Systems (1989).

[3] G. Nelson, Systems Programming With Modula-3, Prentice Hall Series in Innovative Technology, 1991.

[4] D. Ungar, Annotating objects for transport to other worlds, in: Proceedings OOPSLA '95, 1985, pp. 73–87.

[5] R. Riggs, J. Waldo, A. Wollrath, K. Bharat, Pickling state in the Java system, Computing Systems 9 (4) (1996) 291–312.
URL `http://citeseer.nj.nec.com/riggs96pickling.html`

[6] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, Reading, Mass., 1995.

[7] J. Pelrine, A. Knight, Mastering ENVY/Developer, Cambridge University Press, 2001.

[8] G. Krasner, Smalltalk-80: Bits of History, Words of Advice, Addison Wesley, Reading, Mass., 1983.

[9] R. Wuyts, A logic meta-programming approach to support the co-evolution of object-oriented design and implementation, Ph.D. thesis, Vrije Universiteit Brussel (2001).
URL `http://www.iam.unibe.ch/~scg/Archive/PhD/Wuyts-phd.pdf`

[10] D. Roberts, J. Brant, R. E. Johnson, A refactoring tool for Smalltalk, Theory and Practice of Object Systems (TAPOS) 3 (4) (1997) 253–263.

[11] R. Wuyts, S. Ducasse, Unanticipated integration of development tools using the classification model, Journal of Computer Languages, Systems and Structures 30 (1-2) (2004) 63–77.
URL `http://www.iam.unibe.ch/~scg/Archive/Papers/Wuyt03cClassifications.pdf`

[12] G. Bracha, D. Griswold, Strongtalk: Typechecking Smalltalk in a production environment, in: Proceedings OOPSLA '93, ACM SIGPLAN Notices, Vol. 28, 1993, pp. 215–230.

[13] O. Agesen, D. Ungar, Sifting out the gold — delivering compact applications from an exploratory object-oriented programming environment, in: Proceedings OOPSLA '94, LNCS, Springer-Verlag, 1994.

[14] S. Liang, G. Bracha, Dynamic class loading in the Java virtual machine, in: Proceedings of OOPSLA '98, 1998.

[15] A. Bergel, S. Ducasse, R. Wuyts, Classboxes: A minimal module model supporting local rebinding, in: Proceedings of JMLC 2003 (Joint Modular Languages Conference), Vol. 2789 of LNCS, Springer-Verlag, 2003, pp. 122–131, best paper award.
URL
`http://www.iam.unibe.ch/~scg/Archive/Papers/Berg03aClassboxes.pdf`

Seaside – A Multiple Control Flow Web Application Framework ⋆

Stéphane Ducasse [a] Adrian Lienhard [b] Lukas Renggli [b]

[a] *Software Composition Group*
Institut für Informatik und angewandte Mathematik
Universität Bern, Switzerland

[b] *netstyle.ch GmbH*
Bern, Switzerland

Abstract

Developing web applications is difficult since (1) the client-server relationship is asymmetric: the server cannot update clients but only responds to client requests and (2) the navigation facilities of web browsers lead to a situation where servers cannot control the state of the clients. Page-centric web application frameworks fail to offer adequate solutions to model control flow at a high-level of abstraction. Developers have to work manually around the shortcomings of the HTTP protocol. Some approaches offer better abstractions by composing an application out of components, however they still fail to offer modeling control flow at a high level. Continuation-based approaches solve this problem by providing the facilities to model a control flow over several pages with one piece of code. However combining multiple flows inside the same page is difficult.

This article presents Seaside. Seaside is a framework which combines an object-oriented approach with a continuation-based one. A Seaside application is built out of components (*i.e.,* objects) and the logic of the application benefits from the continuation-based program flow infrastructure. Seaside offers a unique way to have *multiple control flows* on a page, one for each component. This enables the developer to write components that are highly reusable and that can be used to compose complex web applications with higher quality in less time.

⋆ We gratefully acknowledge the financial support of the Swiss National Science Foundation for the projects "Recast: Evolution of Object-Oriented Applications" (SNF 2000-061655.00/1) and netstyle.ch.

Email addresses: `ducasse@iam.unibe.ch` (Stéphane Ducasse), `alienhard@netstyle.ch` (Adrian Lienhard), `renggli@netstyle.ch` (Lukas Renggli).

1 Introduction

With the victory of the World Wide Web as an information platform the Internet has become one of the most important platforms for applications. When the WWW was invented in 1989 its intended purpose was to link static documents. But soon simple forms processed by cgi-scripts [1] enabled the user to enter data: Web applications were born. However many of the web applications that exist today are much more complex than those based on the original form technology.

One of the major advantages of web applications, compared to conventional client-server applications, is that the user does not need to install a special program for each application he likes to use. Furthermore web applications are easier to deploy – they just have to be installed on a single server. A web application is an application running on a web server which interacts with web browsers over the Internet. HTTP is the protocol used for communication: An interaction between server and client over HTTP is a sequence of requests from the client and responses from server. The server usually responds with a HTML document that is then rendered in the web browser on the client side. Since HTTP is stateless, the server cannot identify the request associated with the response and the server can only respond to requests from the client but cannot update the client spontaneously.

A major difficulty of web applications is that they are based on the asymmetric design of the HTTP Protocol and the fact that it is stateless [2]: The server is unable to send updates to the client and has to wait for incoming requests. Moreover, the web server is unable to control the navigation facilities in web browsers, like the back- and forward-buttons or the capability to open new windows of the same page. These navigation facilities lead to synchronization problems with the state of the server and its clients. This means that the server has to deal with the fact that for one question asked (*e.g.,* filling in an order form) there may be more than one answer. This happens if the user uses the back button or clones a window and submits a form a second time. Thus a user can follow several paths of interaction in a session at the same time.

Another major problem arises from the architecture of web pages: Each HTML link or form-action encodes in its URL the file which handles the request – and additional parameters that can be passed with the query string. Whereas this scheme is appropriate for static documents it is not ideal for programming complex applications. Without an appropriate abstraction it leads to the problem that control flow logic which would ideally be implemented in a single piece of code has to be split into different parts – one for each request sent by the client. From this point of view URLs with their query strings are procedure calls that do never return: control flow has to be defined in a goto-like

manner which leads to poor designs [3] [4]. This problem remained unsolved since the existence of cgi-scripts – even by widely used page-centric frameworks or newer technologies such as WebObjects [5] or ASP.NET [6]. Several frameworks like Jakarta Struts [7], JWIG [8,9] or RIFE [10] have proposed solutions that model control flow explicitly on a higher level of abstraction.

Continuation-based frameworks propose a more innovative approach to web-serving [11] [12] [13] [14] [15] [2]. Using continuations enables the web application server to offer a procedural view to the programmer that makes it very convenient to model a flow of pages using a single piece of code. Continuations increase application control flow abstraction. However, current continuation-based frameworks make it difficult to combine several control flows into a single page. This hampers the design of truly reusable and composable components from which web application could be built.

Seaside [16] is a mature framework which combines an object-oriented approach with a continuation-based one. A Seaside application is constructed out components (*i.e.,* objects) and the logic of the application benefits from the power provided by continuation-based program flow infrastructure. This combination enables a framework in which applications are built out component each having its *own* control flow. This unique *multiple control* flow enables the definition of highly reusable components that can be even used multiple times on the same page and the definition of control flow at a high level of abstraction. Seaside is developed in Squeak [17], an open-source platform and development environment for Smalltalk, by *Avi Bryant* and *Julian Fitzell*. The last two authors are using Seaside professionally.

The contributions of the paper are: (1) a description of the most common problems in web application development, (2) the description of the Seaside framework a combination of object-oriented application composition with continuation-based flow and (3) the description of the multiple control flow provided by this combination. Before presenting the core concepts of Seaside and going into some details of its implementation, we describe the problems that a web application has to face. As a motivating example and for illustration purposes we use a simple web shop example throughout the rest of the article.

2 A Web Shop as a Motivating Example

Our shop that comes as an example application with Seaside, selling sushis, is composed of various elements: it provides a search on the left, a batched list of products with a detail view of a selected product in the middle, and a shopping cart on the right (see Figure 1). The title bar, the search and the

shopping cart are displayed on every page.

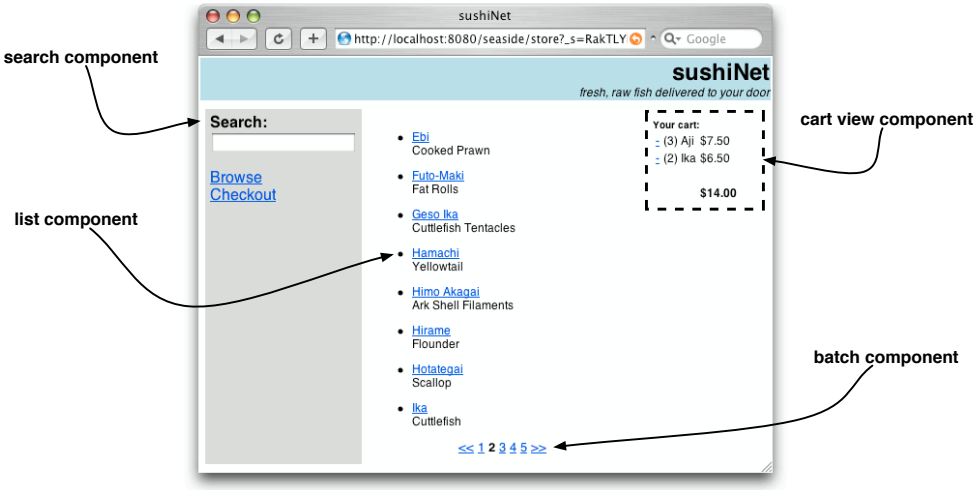


Fig. 1. The Sushi Web Shop and its components.

The shop application also defines several control flows that the user can follow. For example the checkout process, which is the following (see Figure 2): When the user wants to check out, he first has to confirm the contents of the cart, and if he agrees, he is asked for the shipping address. Subsequently, a dialog asks him if he wants to have a separate billing address. If he answers with yes, an additional address dialog is displayed. After having entered the payment information the order is finished and a confirmation page is displayed. Between each of these steps there is a validation logic that may decide to redisplay the previous dialog with an error message.

3 Current Limits of Web Application Development

There are basically two kinds of problems: the ones related to the control flow logic and the ones concerned with the state which has to be remembered during user interaction.

3.1 Control Flow Problems

Many of today's frameworks (such as Servlets/JSP [18] [19], PHP [20], ASP [21], JSP [19] or Zope [22]) fail to provide a high-level abstraction over how pages are linked. Indeed, a web application has to model *control flow*.

The control flow logic (as illustrated by the check-out in the shop example) would ideally be implemented in one single piece of code with common pro-

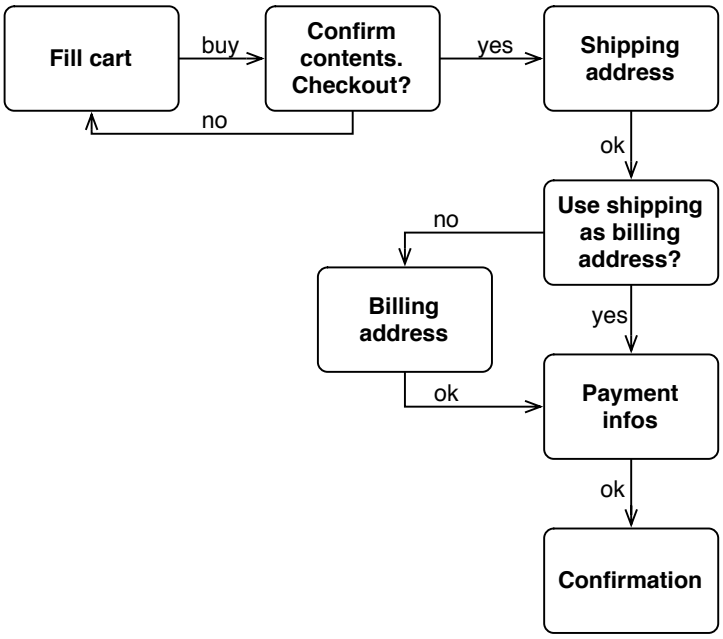


Fig. 2. The checkout process in our shop.

gram statements. Unfortunately, this way of modeling web applications is inverse to what the HTTP protocol with its request/response model implies. As long as the framework does not provide the abstractions, the developer is forced to use unintuitive workarounds. Here is a list of the most important problems related to control flow.

Mixing Application Logic and Component Logic. The user session in a in a web application can be seen as two repeating tasks that take place: The first is generating the page and the second is processing data when the user submits it. In many frameworks these two parts are disconnected and the processing takes place in two separate executions of the server program. The first part produces the HTML representation of the page. The latter processes the data and starts the generation of the next page, *e.g.,* by doing a validation and by deciding what to do next *i.e.,* which rendering code to execute to produce the next page. It might be the same page with the previously entered values and additional error messages in case of a validation failure *or* the next step in the sequence of control flow. Thus, the decision on "what to show next" is coupled with the processing of the data of the last page rather than being defined in a control flow logic at a higher level. In page-centric frameworks plugging together components of the application and defining how they should interact with each other (application logic) has to be done in each component itself (component logic): each page in a sequence of pages has a hardcoded pointer to the next one. This is comparable to programming with goto-statements which never return.

Difficult Composition of Control Flows. Continuation-based web servers were a big step forward to support a better abstraction of application control flow [12] [14] [15]. However, they do not allow an easy composition of multiple control flows coming from the different components that composed the application within the same page. For example, the user can be browsing a detailed description of an item while on the same page, it can get prompted to know whether the numbers of item he selected is correct, and he does not have to be forced to answer the question but can perform all kinds of other tasks before and in parallel. Not being able to easily combine multiple control flows hampers the definition of truly reusable and composable components.

Controlling Program Flow. In some situations we have to strictly control which requests are valid and which should not be processed anymore. Because the user can clone browser windows and go back in the history of pages he is able to send the same or different requests more than once. In our shop, the user should not be able to add additional items to his shopping cart after having payed for it.

3.2 State Problems

A web application typically has to deal with the following kind of states: (1) user interface state (*e.g.,* remember which is the current page number in a batched list) (2) domain model state which has to make persistent before it gets stored in a database (*e.g.,* the billing address in the shop checkout process) and (3) state related to the user session (*e.g.,* customer id of the user). But handling state in the web's client-server context is hard because the control flow is not linear [23] [24]. This is mainly caused by the fact that HTTP is stateless and by the capabilities of web browsers: The users are used to going back in the browser or cloning windows to undo their last step or to interact with the application with several windows in parallel: A form in a web application can be submitted more than once and maybe even at a later time. Since the server cannot update clients, the page shown in the browser may be out of date.

What makes handling state even more complicated is the fact that the facilities to pass state from the client to the server or vice versa are very limited: Information can be encoded into URLs, submitted in hidden form fields or stored in cookies. Cookies only hold information per session and not per browser window and are therefore not suited to store state that is individual for each path of user interaction.

Instead of passing information back and forth from the client to the server, storing state on the server in the session object would be an obvious solution.

However, this approach fails because there can be more than one path of user interaction in parallel in the same session. Thus, especially in page-centric frameworks the developer is forced to encode state in the responses sent to the client by manipulating query strings or using hidden form fields. Those values then have to be decoded from the next request of the user. However, this solution is not only cumbersome, it also leads to the following problems:

Encoding State in Pages. When domain model state should exist over a sequence of pages, each page has to pass the information from all previous pages to the next page. This makes code almost not reusable since one page depends on the previous ones.

Name Clashes. There can be name clashes in URLs or in form fields. The programmer has to take care that identifiers are unique across each page. This gets especially bad if we like to reuse code. For example, it is not easily possible to write reusable user interface components such as batched lists or tab widgets and use them more than once on the same page.

Mixing Presentation and Domain Logic. To encode state in the pages, the developer has to mix the generation of HTML with domain logic using string concatenation or templates. This is cumbersome and leads to unreadable code.

To summarize, developing web-applications face the following problems: it is difficult to (1) define *reusable* components with their *own* logic, (2) compose the logic of an application out of component logic, and (3) represent the state of an application.

4 Seaside Main Building Blocks: Components

The main entities in a Seaside application are objects, but called components in Seaside parlance. They are responsible of defining the user interface and the control flow of application part. A component is an instance of a user-defined subclass of Component that defines the look and the behavior of a portion of a page. Components therefore can be seen as views and controllers of the MVC triad [25]. Note that contrary to file-based frameworks component instances often exist during the whole lifetime of a user session, *e.g.,* the component displaying the cart in our shop. On each request the session lets the components evaluate their *callbacks* and *render* their current state by writing onto the response stream as presented hereafter.

Rendering. Each component which is visible in an application gets its hook method renderContentOn: invoked to render itself, *i.e.,* to generate an XHTML representation of itself. The component's method renderContentOn: is invoked with as argument an instance of HtmlRenderer, named by convention *html*.

Such an instance is a stream-like object that understands different messages to conveniently create most of the XHTML tags [26].

The following example shows the shop's root component render method which defines a table with one row containing the main title and the subtitle, both embedded in div-tags with specific CSS classes (see the result in Figure 1). Nesting of XHTML tags (*e.g.,* table:) is done by using Smalltalk blocks. The last call to html passes the root component's task which defines the body of the application. This is discussed in the following paragraphs.

```
Store ≫ renderContentOn: html
    html cssId: 'banner'.
    html table: [
        html tableRowWith: [
            html divNamed: 'title' with: self title.
            html divNamed: 'subtitle' with: self subtitle ] ].
    html divNamed: 'body' with: self task.
```

The example shows how XHTML code is generated programmatically. This is very convenient because control- and loop-statements can be defined uniformly without the need to switch between Smalltalk code and HTML definitions.

Embedding Components. To compose an application out of different components, components can be embedded into each other. This is what the last line of the previous example is doing. The method divNamed:with: – as well as most of the other methods of HtmlRenderer – takes as second argument a component or any other object that can be rendered. In our example it is a *task* that the store component holds as an instance variable. A task is a special kind of component that only defines control flow. It will be discussed in more detail in Section 5.

Action Callbacks. So far we only discussed how a component renders itself. Components can react to actions performed by the user by means of *action callbacks*. Action callbacks are defined on buttons and anchors as well as on form elements such as select boxes, text input fields etc.

Action callbacks are defined using blocks: For buttons and anchors blocks without an argument are used, for form fields blocks are evaluated with one argument, the current value of the element. The following code snipped (from StorePaymentEditor) shows the definition of a select box which lets the user choose the credit card he likes to pay with:

```
    html
        selectFromList: self cardTypes
        selected: self cardType
        callback: [ :value | self cardType: value ]
```

```
labels: [ :each | each abbreviation ]
```

The first argument passed to the method selectFromList:selected:callbacl:labels: is a collection of classes, one for each supported credit card type. The second argument defines which object, if any, of the previous list should be selected. StorePaymentEditor holds an instance variable cardType to remember the selection and defines the accessors cardType and cardType:. The third argument is the action callback block which itself takes one argument. When the user submits the form which holds the selected value, the block is evaluated with the selected value. The value is one of the objects of the provided list, in our case a class. In the example the value is stored for later usage. The last argument specifies how the items are labeled in the select box, in this example each class from the list of possible card types responds to abbreviation returning a description string.

Most of the time action callbacks to anchors or buttons call methods that define component or application control flow. This is a very central mechanism in Seaside: the specification of *control flow* by means of temporarily passing control from one component to another in a non-goto like manner (*i.e.,* in a procedural or method invocation manner). This facility is provided on component level which enables to define multiple control flows independently from each other. This is the subject of the following sections.

5 Multiple Control Flows in Seaside

As each component defines its own control flow independently of the other components displayed on the same page and a component can be composed of multiple other components, a component or an application has multiple control flows. Whenever a new page is requested by hitting a link or a button, one of the components is able to go one step further in its own flow, while the other remain in the same state. Furthermore, control flow in Seaside does not have to be sequential: control statements, loops, function calls and domain code might be mixed with messages to display new web pages. All this is done simply by writing *plain* Smalltalk code, there is no need to build state machines like in Struts [7], JWIG [8] or RIFE [10].

In the following we present how a component defines its own control flow, then we describe how a component can pass its control to another component and finally how multiple components can be assembled together to create page with multiple control flows.

Each component can have its own control flow that may describe simple widget logic or more advanced control flow. The method StoreTask ≫ go below defines the control flow of the StoreTask component. It specifies the central logic of our shop: the sequence of pages which are shown when browsing through the shop and performing the checkout. It models the process shown in Figure 2 precisely and it almost reads like a piece of pseudo-code. There are different helper-methods called and each of them displays information, offers choices or collects data from the user. Users are able to browse products and put them into their shopping cart: this subprocess is implemented by another component which is invoked by fillCart:. Finally the user orders the products by providing a shipping and, if necessary, a billing address and payment information. When everything has been completed correctly, the method ship:to:billTo:payWith: executes the final ordering and displayConfirmation confirms the order to the customer.

```
StoreTask ≫ go
    | cart shipping billing creditCard |
    cart := StoreCart new.
    [   self fillCart: cart
        self confirmContentsOfCart: cart ] whileFalse.
    shipping := self getShippingAddress.
    billing := (self useAsBillingAddress: shipping)
        ifFalse: [ self getBillingAddress ]
        ifTrue: [ shipping ].
    creditCard := self getPaymentInfo.
    self ship: cart to: shipping billTo: billing payWith: creditCard.
    self displayConfirmation.
```

This method models the shop control flow at a high level of abstraction: It defines how parts of the application are composed. Information from one part can be:

(1) passed to the next part, *e.g.,* for example the instance of StoreCart, cart is passed to be filled, then passed to the shipping part,
(2) used to decide what to do next, *e.g.,* useAsBillingAddress: asks the user if he wants to use the shipping address as billing address) or
(3) stored in a temporary variable for later use, *e.g.,* shipping.

There is neither a need to pass information from one page to another, so that it is available later one, nor the need to model logic that encodes which component to display next in a called component.

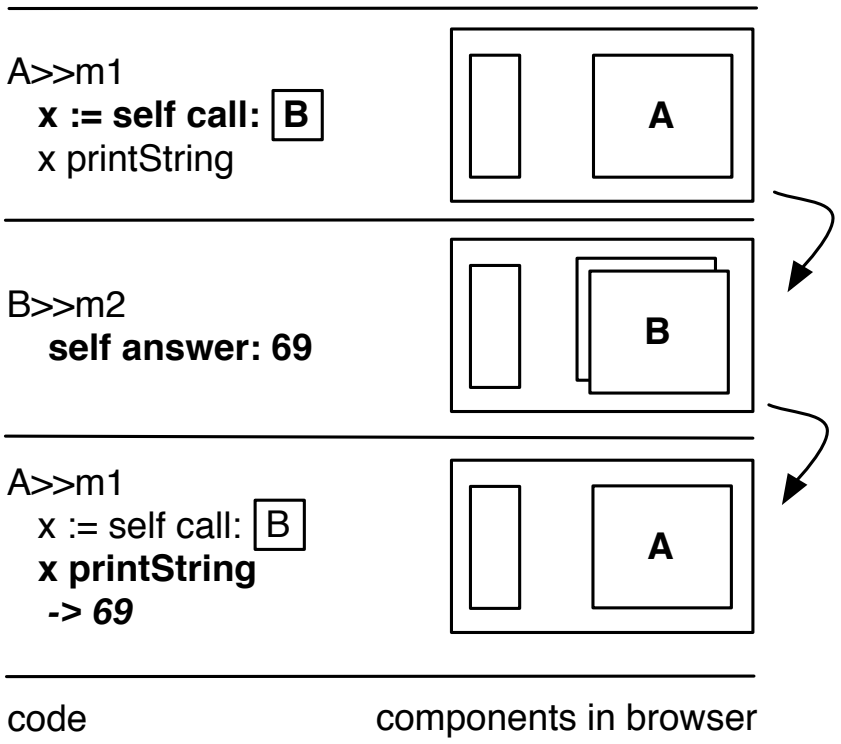


Fig. 3. call:/answer: the basic elements of control flow in Seaside. The framed B in the method m1 is a graphical object displayed as the window B in the web browser. m2 is a method that is invoked in a callback *i.e.,* when an action on the component B is invoked such as a button pressed or a link clicked.

5.2 Call and Answer: Passing Control to Another Component

In Seaside the control flow is based on the interplay between the methods call: and answer: (see Figure 3). Several other methods are built on top of these two messages, such as inform: to display an information dialog or request: to ask the user for a string and make the developer's work more convenient.

Call. At any time a component can pass control to another one. During this time it is temporarily *replaced* by the other component. This is achieved by sending the message call: with the new component as parameter to the component that should be replaced. In Figure 3 sending the method call: to the currently displayed component with the component B installs the component B and passes it the control. Other components elsewhere on that page stay functional and can be used independently of the new component.

The sushi listing in our web shop illustrates call: use. The link's action handler of each sushi item is implemented as shown in the following piece of code. The effect of this code is to replace the current main component (the batched list)

by a component displaying detailed information about the chosen product.

```
StoreFillCart ≫ displayItem: anItem
   main call: (StoreItemView new
                        item: anItem;
                        cart: cart;
                        yourself)
```

Answer. At any time a component can give back control to the component from which it was called using the method answer:. Every call in Seaside eventually returns at some point and it is even able to return a value to the caller. This makes it possible to pass resulting objects from called components back into the control flow and avoids the necessity to pass around strings with requests and responses.

To pass the control back, a called component should send to itself the message answer: aValue. In Figure 3, the expression self answer: 69 makes the component B return the number 69 to component A. As argument of the answer: message, any object can be given and this object will be handed back to the caller of the method call:. For example after the expression x := self call: B, the value of x is the value passed as argument in the expression self answer: 69 of the method m2. For convenience, if there is no return value needed, one might also call answer that is implemented as self answer: nil.

It is then possible to collect information by calling a component which will return an object. In case of a confirmation dialog this might just be a boolean – but it can also be a business object as the following example of our shop demonstrates: Whenever a user has finished selecting his sushis, he has to provide valid payment information. Within the component StoreTask the method getPaymentInfo calls StorePaymentEditor and returns the result of this message send.

```
StoreTask ≫ getPaymentInfo
   ^self call: StorePaymentEditor new.

StorePaymentEditor ≫ ok
   self answer: (cardType new
                        name: name;
                        number: cardNumber;
                        expiry: (Date newDay: 1 month: month year: year)).
```

The message call: replaces the current component instance of StoreTask with a new instance of StorePaymentEditor and stops the execution until the user has provided valid payment information. The class StorePaymentEditor implements a method called ok, which is evaluated when the user is hitting the okay button. The method ok creates a new instance of the selected cardType and passes the

information collected in the dialog (the name, cardNumber, month and year) to the newly created object. The method answer: returns this object to its caller (StoreTask ≫ getPaymentInfo) where Seaside resumes the control flow which has been previously stopped at this position.

The real power of the call and answer mechanism relies in the capability to build a flow of components which embed several complex user interactions. Seaside allows one to call different components one after the other, using control-statements such as loops and conditional clauses, or other non web related code in-between these calls. An important point is that passing control to another component is done with normal message sending: A method returns and the execution continues from this point – even if this is at an undefined point in the future. This prevents the goto-like definition of control flow without the possibility to return.

5.3 Composing Components: Multiple Control Flow

The examples discussed so far, illustrate one flow of control at a time: StoreTask ≫ go (see Section 5.1) models the control flow of the application at the highest level. This control flow is defined at the top component, however with Seaside it is possible to let each sub-component define its own control flow. In the context of our web shop example, this enables the user to interact *at the same time* with the sushi list (*i.e.,* getting a detail description of the sushi, browsing the sushi list) *and* with the shopping cart.

Figure 4 shows this process in the web browser. It presents four states of the webshop user interface. First on the left we see the description of a sushi (Chuboro Magoro) which can be added to the cart displayed on the right. The card contains five California Rolls and three Chuboro Magoro. In the cart, the user can change his order by pressing the minus sign in front of the line. This is what the user did, and the right component is replaced by a dialog box that checks if the user really wants to remove some of the delicious sushi from the cart (1). The user is hesitating and quits the description of the Chuboro Magoro sushi to browse the sushi list (2). Finally he decides to remove all the Chuboro Magoro sushi from his order (3).

The following code describes how the available sushi list and the cart are plugged together. The two components are both stored in instance variables of the component StoreFillCart and are placed inside a HTML table to be rendered next to each other on the same page.

```
StoreFillCart ≫ renderContentOn: html
   html table: [
      html tableRow: [
```

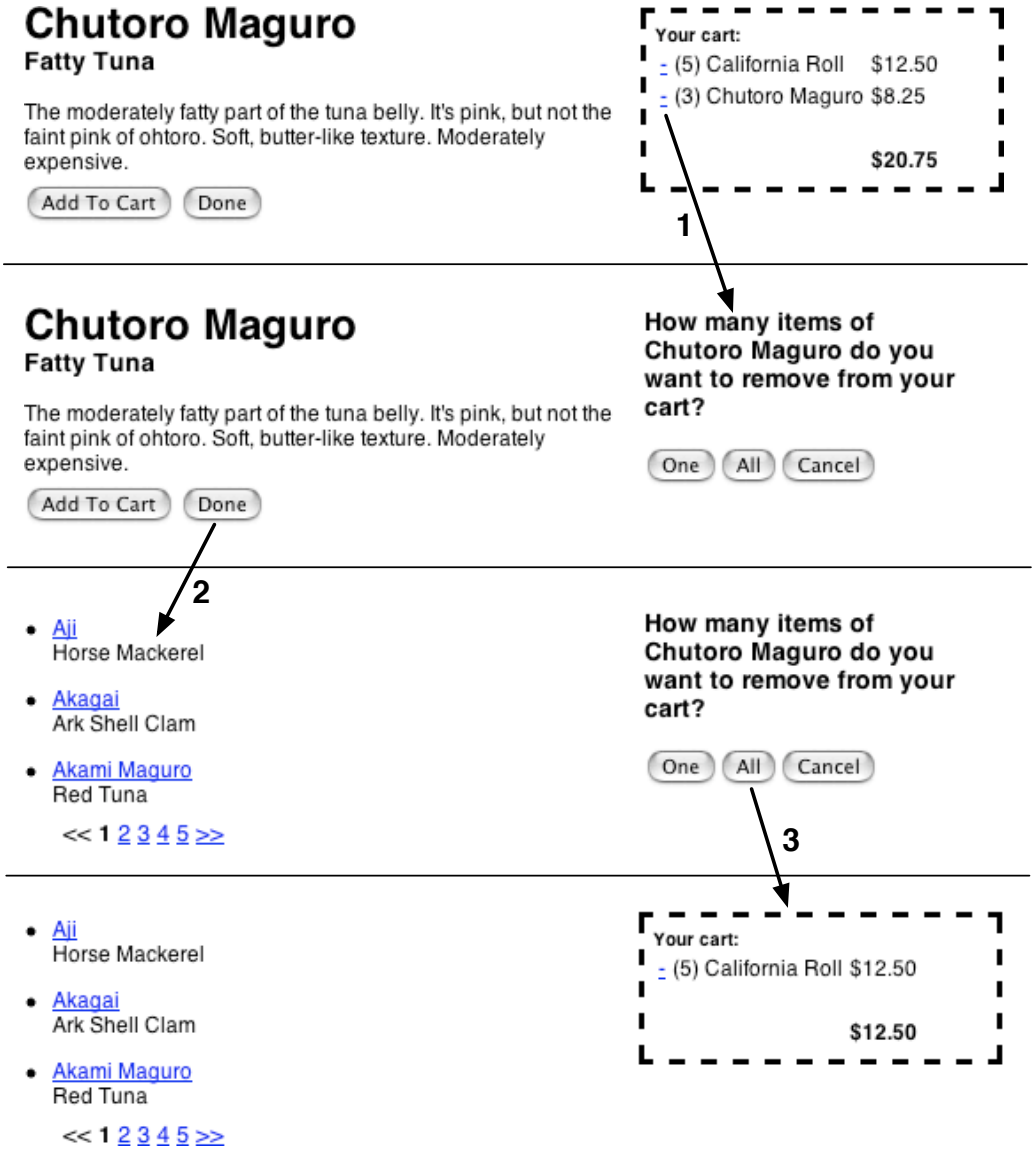


Fig. 4. Multiple flows on the same page in the web shop. The user can interact freely with different components and their flows of the sushi shop.

```
html tableData: productList.
html tableData: cartView ] ]
```

Each component can have its own control flow. Such a flow is either bound to the component itself (if our component is a subclass of Task) or defined in actions connected with user events (if our component is a subclass of Component). Whenever there is an interaction with a component and there is either a message send to call: or answer: a different component is shown instead of the old one and the flow of that particular component is able to advance. All the other components on the page however stay in the same state until there

is an user interaction allowing them either to return or to call another one.

6 Managing Non-Linear Control Flow

The control flow capabilities of Seaside provide a non-linear control flow implicitly – from the view of the developer the code is linear and does not need any additional logic to handle backtracking in the control flow. However there are still some problems related to the application state. Seaside handles backtracking in the control flow by resuming computation at the right place. But since the user interacting with the application modifies state, we also have to ensure that it is correctly handled. As discussed in the problems Section (see Section 3) we have to deal with user interface and domain model state. It is important to handle the latter correctly as it is crucial for security reasons.

In the case of our shop, when the user has checked out and already payed the products, it should not be possible for him to go back and modify the cart business object by adding additional sushi to it. Similarly, the user would not be happy if he were to accidently submit the credit billing page a second time, and be charged twice as a result.

This leads to two different kinds of situations: In the first situation we would like to support the user to backtrack. This is solved by backtracking user interface and domain model state so that when the user goes back, the old state is restored. The second situation is when the user is forbidden to backtrack. This is solved by specifying the conditions in which a request should not be processed anymore. We present these two situations now.

6.1 Backtracking State

Each component in Seaside has its own state which is stored in instance variables. For example, a batched list remembers its current page number or the search component stores its last search string and result etc. Since the user interactions share the same component instances (because the resumed computation is always the same) their instance variables may not represent what the user sees when he goes back, or takes several paths in parallel.

To solve this problem, Seaside offers a mechanism to register an object to be backtracked (Session ≫ registerObjectForBacktracking:). After each response sent to the client, Seaside snapshots the registered objects by creating a copy and putting them into a cache. The session stores the registry as a temporary variable in the computation which sends the response to the client and which

receives the next request. Since the continuation (which at this point composes the current request/response loop) restores the context when resuming, the registry is made persistent to the future point when requests, originated from this response, are handled. Before processing the request, the registry restores the registered objects.

This ensures that when processing a request, the values are the same as when the previous response was created. For example the batched list's current page number will be the same as the one shown to the user when a request from this page is processed.

6.2 Transactions

In complex applications it is often the case that we must ensure that the user is prevented from going back a sequence of pages to make modifications. This is applicable in the case of our checkout process, where the user should not be able to change anything after having paid. Controlling the control flow is implemented by the method Component ≫ isolate: which takes a block as argument. It treats the control flow defined in the block as a transaction. The transaction makes sure that the user can move forward and backward as he likes *within* the transaction. But as soon as he completed the transaction, he cannot backtrack anymore.

The following method shows the shop process enhanced with transactions. By surrounding the filling of the cart and its confirmation by an isolate: invocation, we allow backtracking freedom within that part of the application, while protecting a completed order from being changed. Similarly we are making sure that after the confirmation has been displayed, the user is unable to go back and change shipping and payment information.

```
StoreTask ≫ go
    | cart shipping billing creditCard |
    cart := StoreCart new.
    self isolate: [
        self fillCart: cart
        self confirmContentsOfCart: cart ] whileFalse ].
    self isolate: [
        shipping := self getShippingAddress.
        billing := (self useAsBillingAddress: shipping)
            ifFalse: [ self getBillingAddress ]
            ifTrue: [ shipping ].
        creditCard := self getPaymentInfo.
        self ship: cart to: shipping billTo: billing payWith: creditCard ].
    self displayConfirmation.
```

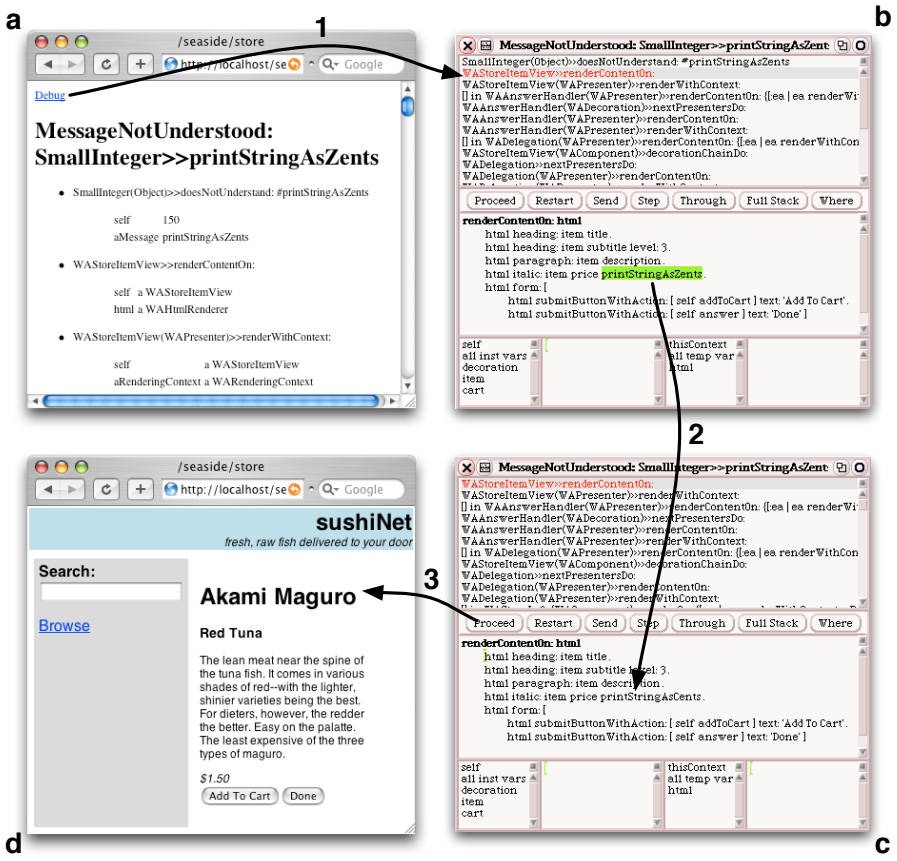


Fig. 5. Debugging with Seaside – a continuous loop fixing a bug without restarting the application.

7 Development Environment

As Seaside is written in Smalltalk it is based on a very powerful, fully object-oriented language and development environment. In addition to being able to use the tools provided by the environment, Seaside integrates them seamlessly with the web. This makes the platform a versatile and productive environment for web application development. We start by looking at the debugging facilities before presenting the Seaside specific tools.

Incremental Programming. Smalltalk's philosophy of incremental programming in an interactive environment is supported by Seaside. Code can be added and edited while the web application is running and there is neither the need to manually recompile the code nor to restart the server. In many cases this makes it possible to update a system in production on the fly without any outage and without the need to set-up a temporary backup server.

Debugging. Most of today's frameworks do not support debugging of web applications well. Most display the error and the line number in the web

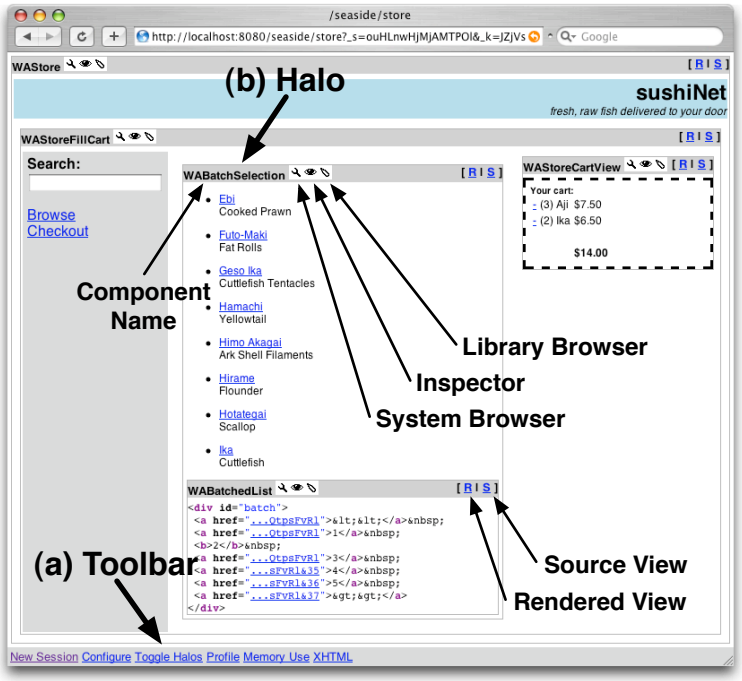


Fig. 6. The Sushi-Shop as seen in Figure 1 in development mode, with toolbar (a) and halos (b) enabled.

browser only, which makes it very inconvenient to find and fix bugs.

Seaside has unique debugging capabilities: When an unhandled exception occurs as seen in Figure 5, a stack trace is shown in the web browser (a) with a link called debug. Clicking this link (1), the developer activates a debugger (b) within the development environment which lets him inspect variables and even modify the code on the fly. In the given example the message printStringAsCents, that is automatically highlighted in the debugger (b), has been spelled wrongly and is fixed (2) by the developer. The debugger now displays the recompiled method (c). During this time, the web browser keeps waiting for the response of the server. When hitting proceed (3), the processing of the request which had caused the error is resumed and the resulting page is finally displayed in the web browser (d).

This feature makes debugging web applications very powerful: There is no manual recompilation and restarting of the web server required. The developer is put right back into the questionable page where he is able to see if he fixed the error properly and is able to continue the testing session.

Toolbar. A toolbar that is shown at the bottom of the web-application during the development phase (Figure 6) enables the programmer to access additional tools from within the web. Of course, all these tools have been written in Seaside itself:

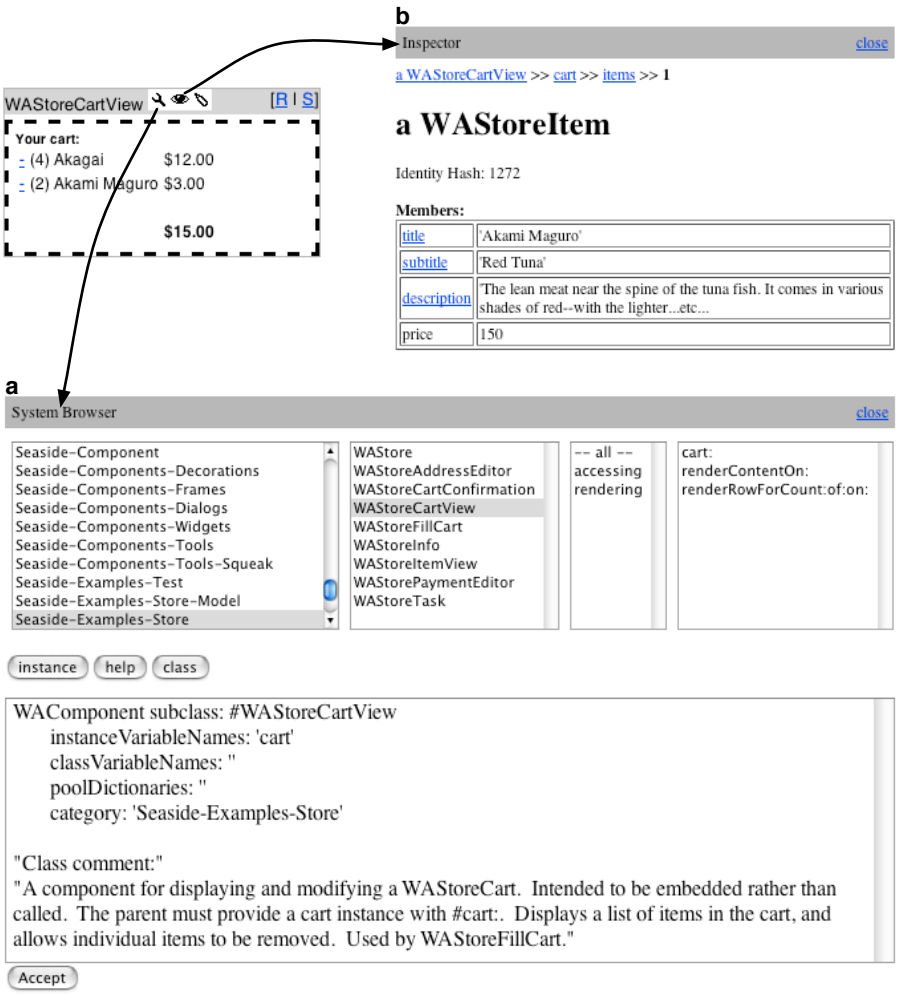


Fig. 7. Browsing (a) and inspecting (b) the CardView component of the running Sushi-Shop.

- New Session starts the application within a new session.
- Configure opens a dialog letting the user configure the settings of the application. This includes properties such as where to start a new session, what should happen in case of an exception or if the development toolbar is displayed or not.
- Toggle Halos shows or hides the halos, which are discussed in detail in the next paragraph.
- Profile shows a detailed report on the computation time that has been consumed while building this page.
- Memory Use displays a detailed report on the amount of memory consumed by this application.
- XHTML starts an external XML validator on this page.

Halos. When enabling the halos, every component gets surrounded by a thin grey line and a header giving the class name of the component and a set of buttons to run tools and to change the viewing mode (Figure 6).

- System Browser opens an editor on the current component and lets the developer modify its class and all the methods from within the web, while the application is still running in the background (Figure 7, a). When closing the browser-view the application immediately runs with the new code without having to restart the session.
- Inspector opens a view on the component, so that the developer can browse the internal structure of this object (Figure 7, b). It presents the names of the instance variables and the current values, whereas the user is able to dive into the referenced objects by clicking on the links. In Figure 7, b, the first item of the items of the cart instance variable of the WAStoreItem instance is displayed.
- Library Browser opens an editor that lets a UI designer tweak the associated CSS-Stylesheets. This makes it very convenient to try out different layouts directly in the web-browser without leaving the running application.
- Source View provides a pretty-printed and syntax-highlighted XHTML view onto the source code of the current component. Like this the developer is able to observe the generation of XHTML while still being able to interact with the application by clicking on its links.

8 Evaluation

Based on continuations, Seaside transparently manages the request/response loop and the handling of the necessary URLs, query strings and hidden form fields. This prevents any name clashes and frees the developer from manually encoding information in the response and decode it later again. This is the foundation to model the control flow explicitly at a higher level of abstraction rather than having to hardcode the next step of flow in each component itself.

Seaside components are responsible for rendering themselves and handling input by action callbacks. Action callbacks are block closures that are bound to user interface elements such as input fields or submit buttons. When a request is processed, the applying action callbacks are evaluated. This enables the processing of form fields or the execution of embedded control flow directly in the component in a natural way.

In Seaside, each component can run its own control flow independently of the others. This makes possible to compose complex applications out of small and reusable components without having the problem of composing the individual control flows. This component composition makes possible to have multiple control flows within the same page naturally. In addition in Seaside, a transaction allows one to specify how to group a control flow part and to ensure that the user cannot go back into it after he left it. This is a powerful solution to define security independently form the involved components. Again, this is

crucial for reusability.

Seaside's control flow with its unique call/answer semantics offers passing around business objects between components. In contrast to other frameworks this avoids the need to pass state from one component/page to the next over the client. To synchronize state with the current page displayed in the user's browsers, Seaside offers backtracking of objects. This makes the back button to be a fully supported navigation facility of the application.

On the negative side, in the current version all the continuations have to be kept in memory which consumes resources. Continuations could be stored in database but this solution implies to be able to serialize continuations which is a non trivial task [15] [2].

9 Related Work

The idea of modeling sessions as a continuous piece of code has been popping up independently in multiple places in the past [14] [12] [15].

DrScheme. The *Scheme Web Servlets* library included with DrScheme [11] was one of the first frameworks to support continuation-based web development. However it does not provide a simple solution to provide multiple control flows feasible. Furthermore DrScheme does not provide a collection of components that can be composed easily and reused with different applications.

CocoonFlow. Most of todays mainstream programming languages unfortunately do not support continuations, therefore the authors of CocoonFlow [27] decided to enhance their JavaScript runtime to support this concept. However CocoonFlow does not provide such a high level abstraction over the HTTP protocol as Seaside does, the function sendPageAndWait has to be called to suspend the execution after the page has be generated and sent to the client. This mechanism does not allow to have multiple flows on the same page easily.

RIFE. Java Servlets/JSP [19] is lacking the possibility to model control flow in a clever way. RIFE [10] provides a declarative way to define application flow based on state machines. For simple Web applications, this model works. However, RIFE is facing a well-known problem with state machines: the number of states and transitions grow fast and it becomes hard to understand what is happening in the application. Multiple control flows are not supported.

Imposter. Python is lacking support for continuations as well. Imposter [28] provides an abstraction over the session handling by saving the whole internal state of the applications between two requests. However, as there is only one

snapshot stored in the memory, using the back button is not supported and the state of the application and the web browser window cannot be synchronized properly.

WebObjects. Apple Web Objects [5] provides a component-based framework to reuse and compose components which offers solution to the back button problems, however it is lacking the possibility to describe a flow of pages as a continuous piece of code and multiple control flow.

ASP.NET. Microsoft ASP.NET [6] is a web application framework running on the .NET platform. Its web controls are comparable to Seaside's components. However, web controls are not capable of modeling a continuous flow of user interaction in one piece of code. Moving to another page is still done in a goto-like manner by redirecting the user to the next page. Reusability and flexibility suffers and the problems of the back button and cloning of windows have to be addressed by the developer by implementing workarounds.

Struts. Struts [7] purpose is to bring the MVC pattern to the J2EE platform. The Struts architecture acts as a wrapper for Java applications and divides its code into a Model, View and Controller. Although Struts manages to add a layer of abstraction to model control flows, the costs are high. Compared to Seaside a form validation requires rewriting several parts of the application: the form class, an XML file and the JSP code to display validation error messages. Adding validation to an existing application even requires changing the inheritance trees. In Seaside a validation can be done by adding a decorator around the component.

10 Conclusion

Building web applications raised specific problems due to the disconnected flow between clients and servers. Application control flow and state management made web application development difficult and cumbersome. Programmers are often forced to program using goto-style. The solutions can be roughly classified into two categories: the ones that use objects to model pages and applications, and the ones that use the power of continuations to provide an advanced control flow. The first ones to not address well the problem related to control flow. While more advanced the second approaches do not support well the composition of independent control flows.

Seaside, the framework presented in this article, combines both approaches: a web application is composed of components, each having its own control flow and been able to pass the control to other component. Applications are then a composition of components whose control flow is combined and can run

independently. This enables Seaside to offer a better reuse and composition of predefined components. With Seaside, the programmer is able to write business application logic at a high-level of abstractions.

Acknowledgment. We thank Avi Bryant for his feedback and encouragements, Shriram Krishnamurthi for the discussions on continuations and Orla Greevy and Alexandre Bergel for their feedback.

References

[1] CGI, The Common Gateway Interface, http://hoohoo.ncsa.uiuc.edu/cgi/.

[2] C. Queinnec, Continuations and web servers, Higher-Order and Symbolic Computation: an International Journal (2004) 1–16.

[3] E. W. Dijkstra, Go To statement considered harmful, Comm. ACM 11 (3) (1968) 147–148, letter to the Editor.

[4] O.-J. Dahl, E. W. Dijkstra, C. Hoare, Structured Programming, Academic Press, 1972.

[5] WebObjects, http://www.apple.com/webobjects/.

[6] ASP.NET, http://www.microsoft.com/net/.

[7] The Apache Struts web application framework, http://jakarta.apache.org/struts/.

[8] JWIG, Java Extensions for High-Level Web Service Development, http://www.brics.dk/JWIG/.

[9] A. S. Christensen, A. Moller, M. I. Schwartzbach, Extending java for highlevel web service construction, ACM Transaction on Programming Languages and Systems 25 (6) (2003) 814–875.

[10] RIFE, https://rife.dev.java.net.

[11] DrScheme, http://www.drscheme.org.

[12] J. Hughes, Generalising monads to arrows, Science of Computer Programming 37 (2000) 67–111.

[13] P. Graham, Beating the averages, http://www.paulgraham.com/avg.html.

[14] C. Queinnec, The influence of browsers on evaluators or, continuations to program web servers, in: ACM SIGPLAN International Conference on Functional Programming, 2000, pp. 23–33.

[15] P. Graunke, S. Krishnamurthi, S. Van Der Hoeven, M. Felleisen, Programming the Web with high-level programming languages, in: Proceedings of ESOP 2001, Vol. 2028 of Lecture Notes in Computer Science, 2001, pp. 122–136.

[16] Seaside: Squeak enterprise aubergines server, http://www.beta4.com/seaside2/.

[17] Squeak home page, http://www.squeak.org/.

[18] D. Coward, Java servlet specification version 2.3, http://java.sun.com/products/servlet/ (2000).

[19] Java Server Pages, http://java.sun.com/products/jsp/.

[20] PHP: Hypertext Preprocessor, http://www.php.net/.

[21] ASP, Microsoft Active Server Pages, http://msdn.microsoft.com/nhp/?contentid=28000522.

[22] Zope, http://www.zope.org.

[23] P. Graunke, R. B. Findler, S. K. and Matthias Felleisen, Automatically restructuring programs for the web, in: International Conference on Automated Software Engineering, 2001.

[24] J. Matthews, R. B. Findler, P. Graunke, S. Krishnamurthi, M. Felleisen, Automatically restructuring programs for the web, Automated Software Engineering: An International Journal .

[25] G. E. Krasner, S. T. Pope, A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80, Journal of Object-Oriented Programming 1 (3) (1988) 26–49.

[26] N. Kurt, Using lisp as a markup language the LAML approach, european Lisp User Group Meeting (1999).

[27] Apache Cocoon, The Apache Cocoon Project, http://cocoon.apache.org/.

[28] Imposter, http://csoki.ki.iif.hu/∼vitezg/impostor/.