Serge Demeyer and Jean-François Perrot (Ed.)

Proceedings of the

# International Conference on Dynamic Languages

organised in conjunction with the

15th International Smalltalk Joint Conference 2007

Lugano, Switzerland, 25-31 August 2007
http://www.esug.org/conferences/2007

# Preface

Although Smalltalk is one of the oldest object-oriented programming languages, its conception and programming environment can still be considered as a design pearl and as a beacon in the realm of programming languages and programming environments. Other dynamic languages (such as Lisp, Scheme, Self,...) were similarly influential in expanding what software engineers can express with their programs. With the rising popularity of languages like Ruby, Python, Javascript, PHP, and with the growing challenges of aspect-orientation, pervasive computing, mobile code, and context-aware computing, dynamic languages are a worthy topic for further research. Therefore, ESUG decided to broaden the scope of the formerly "Smalltalk only" research track of its yearly meeting in order to enable cross-fertilization with research conducted using other dynamic languages. This way we hope to obtain more significant scientific results on various aspects of dynamicity in programming languages.

This volume holds the papers that were presented during the 2007 edition of the conference which was held in Lugano, Switzerland. After careful reviewing by at least three reviewers we selected 11 papers out of 16 for inclusion in the conference. We took great care to avoid conflicts of interest by ensuring that reviewers did not have any formal connections to one of the authors of the papers they reviewed, namely (a) working in the same institution, university, or research group, (b) having written joint papers, (c) supervised earlier work, (d) had family ties, (e) or otherwise felt uncomfortable reviewing the work. The papers were reviewed using the typical academic standards: (a) present sound scientific work (a relevant problem, a convincing solution described in sufficient detail to allow replication, a sound validation, cite related work), (b) help the community (have something interesting to say to researchers working on dynamic programming languages in general and Smalltalk in particular), (c) reports something worthwhile for further reference (other researchers will cite this work in the future).

We hope you enjoy the proceedings of this conference and that the papers included add to the growing body of knowledge in our research field.

*Serge Demeyer (University of Antwerp)*
*and Jean-François Perrot (Université Pierre et Marie Curie, Paris)*
August 2007

## Program Committee:

- Noury Bouraqadi
- Nathanael Schaerli
- Serge Stinckwich
- Alan Knight
- Christophe Roche
- Maja D'Hondt
- Dave Thomas
- Gilad Bracha
- Michel Tillman
- Tudor Girba
- Wolfgang De Meuter
- Laurence Tratt
- Pascal Costanza
- Philippe Mougin
- Damien Pollet

# Table of Contents

# Part I

# Change

# Change-Oriented Software Engineering

Peter Ebraert[1], Jorge Vallejos[1], Pascal Costanza[1],
Ellen Van Paesschen[2], Theo D'Hondt[1]

[1] Programming Technology Lab – Vrije Universiteit Brussel
Pleinlaan 2 – B-1050 Brussel – Belgium
{pebraert,jvallejo,pascal.costanza,tjdhondt} @vub.ac.be
[2] Laboratoire d'Informatique Fondamentale de Lille – University of Lille 1
59655 Villeneuve d'Ascq – France
ellen.vanpaesschen@lifl.fr

**Abstract.** We propose a first-class change model for Change-Oriented Software Engineering (COSE). Based on an evolution scenario, we identify a lack of support in current Interactive Development Environments (IDEs) to apply COSE. We introduce a set of five extensions to an existing model of first-class changes and describe the desired behaviour of change-oriented IDEs to support COSE. With the help of an evolution scenario, we show why those extensions are required. Finally we describe ChEOPS: a prototypical implementation of a change-oriented IDE on top of VisualWorks and illustrate how it supports the extended first-class change model. ChEOPS is finally used to validate COSE as a solution for the shortcomings of existing IDEs.

## 1 Introduction

The evolution history of a software system describes a series of changes made to a software program in response to changes of requirements. The changes are integrated with the base functionality of the program as new versions are released. To keep track of the evolution of the program, developers use file-comparison and versioning tools [1]. However, none of these tools store changes explicitly and the information about changes has to be recovered by reasoning over two subsequent versions of the system. This makes the program history difficult to understand and reason about [2].

In [3], Robbes and Lanza argue that to obtain more accurate information about the evolution of a program, changes should be considered first-class entities, i.e. entities that can be referenced, queried and passed along in a program. First-class change entities, modelled as objects in Robbes's and Lanza's approach, represent the behaviour of the different kinds of changes required for a program (for example to add, remove, and modify classes). The classes of those objects represent the means to create, apply and undo the change objects. However, those change objects only represent modifications to class and method signatures which may be insufficient to reveal the programmer's intention for the changes, for example if the programmer changes the body of a method or performs higher-level changes such as refactorings.

In this work, we propose to deepen the idea of treating changes as first-class entities in the program environments. We argue the need for Change-Oriented Software Engineering in which the program history is represented as a list of first-class changes rather than just a set of resulting versions. The changes should (1) have different levels of granularity to better describe the programmers' intentions, (2) provide an expressive way of representing a program's history and (3) provide better ways of exploring the evolution history of a program. We validate our proposal by implementing the model of first-class changes and by integrating it in the VisualWorks Smalltalk Interactive Development Environment as an extension to its ChangeList tool.

This paper is structured as follows. In Section 2 the use of a first-class change model is motivated based on the shortcomings of ChangeLists change management system. To identify the shortcomings, that section uses an evolution scenario of a simple chat application. Section 3 discusses the requirements to support Change-Oriented Software Engineering. It lists five extensions to the first-class change model of ChangeList and ways in which IDE's could exploit that model to support COSE. ChEOPS, a prototypical tooi for COSE is presented in Section 4. It is used to validate how the extended first-class change model overcomes the shortcomings of the ChangeList tool. A discussion on COSE and its related work can be found respectively in Section 5 and 6. Before drawing some conclusions in Section 8 some tracks of future work are enumerated in Section 7.

## 2 Motivation

Most Smalltalk interactive development environments (IDEs) include a tool which is called *ChangeList* [4, 5] to manage the changes applied to a software system. Using this tool, programmers can inspect, compare, edit and merge changes performed on classes or methods. Each Smalltalk image[3] holds one single change list which records all the performed changes on that image. Hence the user may recover from a crash by backtracking to the most recent non-erroneous state of the image and reapplying changes listed by the ChangeList tool.

Changes are modelled as normal Smalltalk objects in the ChangeList tool and as such they can be referenced, queried and passed along. Every time a system is modified, the Smalltalk IDE logs this modification by creating a change object for it, linking this object to the project, and adding the object to the list handled by ChangeList.

Change objects of the ChangeList tool suit properly the notion of first-class changes for software evolution defined by Robes and Lanza [3]. According to them, change objects provide more accurate information about the history of a program than *file*-based and *snapshot*-based versioning change management

[3] Most Smalltalk systems represent the application code (for example classes) together with the application state (objects) in a single memory region called *image*. Images can be saved and loaded by the Smalltalk environment as a snapshot of the current code and state of a program.

systems. Change objects also better represent the incremental and entity-based way in which the evolution of a program occurs (changes are expressed in terms of system entities and not over text). However, change objects still suffer from a number of shortcomings which we illustrate in the following scenario.

## 2.1 Evolution Scenario

Consider the scenario of software evolution for the chat application depicted in Figure 1. This application originally consists of two classes, `User` and `Chatroom`, which respectively maintain a reference `cr` and `users` to one another. A user can subscribe to a chatroom using the `register` method and exchange messages with the rest of users of the chatroom using the `send` and `receive` methods. Messages sent to the chatroom are propagated to all the registered users.

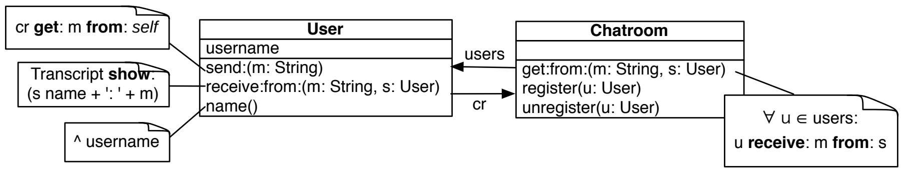

cr **get**: m **from**: *self*
Transcript **show**: (s name + ': ' + m)
^ username
User
username
send:(m: String)
receive:from:(m: String, s: User)
name()
users
cr
Chatroom
get:from:(m: String, s: User)
register(u: User)
unregister(u: User)
∀ u ∈ users: u **receive**: m **from**: s

**Fig. 1.** Class diagram of the chat application

Assume that there are two developers working simultaneously on new features for this application. The first developer is responsible to introduce *types* of users so that it can be possible, for instance, to differentiate between registered and guest users. The second developer is responsible to ensure the privacy of the users, which in this case corresponds to encrypting and decrypting the messages when they are sent and received respectively.

For the first change the developer adds two subclasses of `User` to the application program, `RegisteredUser` and `Guest`. In this example, the only difference between these two types of users is that the registered users can be identified by their name in the chat room whereas the guests cannot: Accordingly, the `username` attribute of `User` class is moved to the `RegisteredUser` class. Figure 2 shows this first feature added to the application.

To ensure user privacy, the second developer adds two methods to the `User` class, `encrypt` and `decrypt`, which are called from the `send` and `receive` methods respectively. Figure 3 shows this second feature added to the application.

After implementing these two new features in the application, the developers merge the two changes resulting the class diagram showed below (Figure 4).

Figure 5 illustrates how these three steps are registered by the ChangeList tool in Smalltalk. Each line in this list corresponds to a change required for the implementation and merging of the two new features described above. Each of

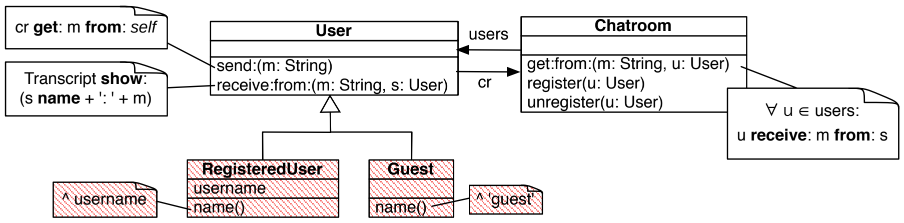
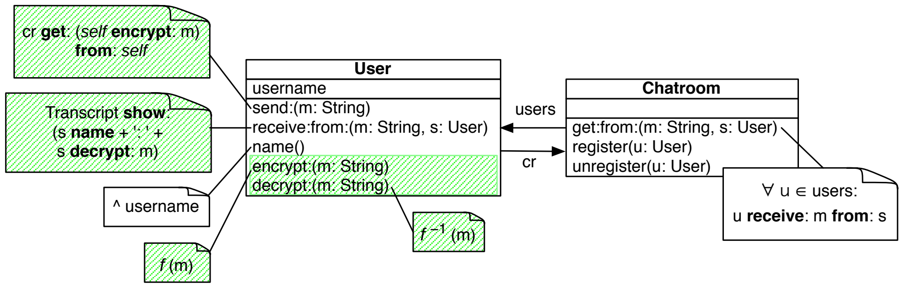


**Fig. 2.** First change: differentiating users



cr get: (self encrypt: m) from: self
Transcript show: (s name + ': ' + s decrypt: m)
^ username
f (m)
User
username
send:(m: String)
receive:from:(m: String, s: User)
name()
encrypt:(m: String)
decrypt:(m: String)
f −1 (m)
users
cr
Chatroom
get:from:(m: String, s: User)
register(u: User)
unregister(u: User)
∀ u ∈ users: u receive: m from: s

**Fig. 3.** Second change: ensuring user privacy

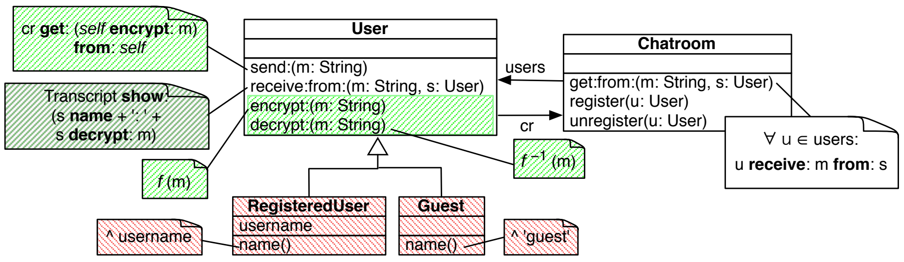


**Fig. 4.** Merging the changes

these changes is stored in a change object. We observe a series of problems in this representation of the evolution of the chat application:

```
 1    Created package ChatApp
 2     define User
 3     doIt User organisation addCategory:#messaging
 4    User receive:from: (change)
 5     define User
 6     define User
 7    User send: (change)
 8     doIt User organisation addCategory:#accessing
 9    User name (change)
10     define Chatroom
11     define Chatroom
12     doIt Chatroom organisation addCategory:#messaging
13    Chatroom get:from: (change)
14     doit Chatroom organisation addCataegory:#registering
15    Chatroom register: (change)
16    Chatroom unregister: (change)
17
18    define Guest
19    doIt Guest organisation addCategory:#messaging
20   Guest name (change)
21    define RegisteredUser
22   RegisteredUser name (change)
23   User name (remove)
24    define User
25    define RegisteredUser
26
27    doIt User organisation addCategory:#encryption
28   User encrypt: (change)
29   User decrypt: (change)
30   User send: (change)
31   User receive:from: (change)
```

**Fig. 5.** Evolution Scenario *user privacy*: Change list by ChangeList

**Restricted level of granularity** The entities contained in the change list are restricted to classes and methods. Additions or removals of attributes are lost and thus they can only be detected by differentiating different versions of the program. This is what happens, for instance, with the `send:` method which is first defined and then modified. As changes within methods are not logged, these two changes result in two occurrences of `User send: (change)` on lines 7 and 30.

**Term overloading** The definition of a change object is in some cases ad hoc and inconsistent. The same kind of Smalltalk change object can represent several kinds of modifications. For instance, a `ClassDefinitionChange` object is required to add a class to the program (for example to add the `User` class) but also to add or remove attributes (for example to add the `username` attribute in the `User` class). As a result of this term overloading, the change `define User` appears four times in the change list (lines 2, 5, 6 and 24). This hinders the understanding of the changes in the list.

**Lack of high-level changes** The ChangeList tool does not allow the explicit monitoring of high-level changes which better represent the intention of the developers. In this evolution scenario, the two intentions of the developers (introducing user kinds on lines 18-25 and ensuring user privacy on lines 27-31) are concatenated in the final change list. This may become a problem if the developers need to change the way in which the two features are merged, for instance, to only enable registered users benefit from encrypted communication (see Figure 6). The developers have to manually recompose their implementations from the change list.

**No exploration facilities** The three shortcomings described above illustrate how difficult the change management can become. After making several modifications to a program, the change list can contain large amounts of change objects. In such a case, the exploration of the change lists is cumbersome and error-prone.

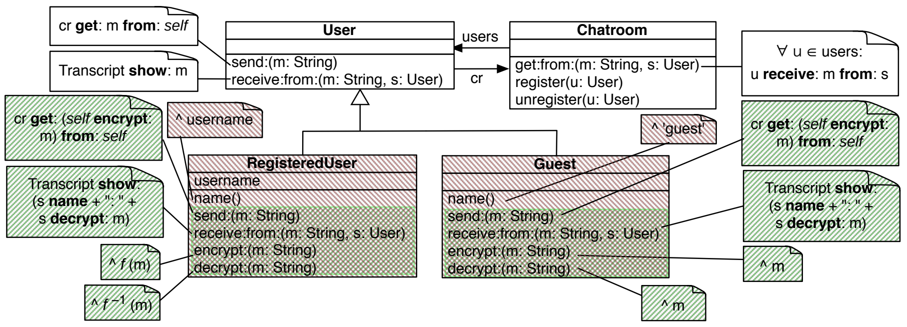


**Fig. 6.** New way of merging the changes

In summary, the first-class change objects featured by the Smalltalk ChangeList tool provide information about the history of each entity of a program. However, this information has a restricted level of granularity, overloads terminology, lacks high-level changes and does not facilitate exploration. In the next section, we present a model that extends first-class changes and overcomes these issues.

## 3 A First-Class Change Model for Change-Oriented Software Engineering

To address the issues discussed in the previous section, we now introduce a first-class change model for software evolution. This model is an extension to the underlying change model of ChangeList.

### 3.1 Fine-grained First-Class Changes

In the ChangeList tool, the different kinds of changes are structured in a hierarchy of change classes. This hierarchy eases extending the set of changes, stimulates reuse and improves maintainability. Therefore we preserve the idea of structuring the types of changes in an inheritance hierarchy.

The first-class changes of the ChangeList tool express changes about packages, classes, attributes or methods. The statements of a method body are not explicitly considered as a subject of change. The dissection of a method body, however, always reveals more detailed information that can be used to study the evolution of the concerned program. The fact that a method statement includes an invocation of another method, implies that there is a relation between both methods, and that the former method could be affected when the latter is changed.

Another example of such a restriction of granularity can be found in the modification of attributes. Changes to method parameters are also not explicitly modeled by the the ChangeList tool. Assume a method call that has a complex expression as an argument. This expression can contain method invocations which reveal links between the caller and the callee.

In our model, we introduce dedicated change objects for all possible associations between program entities. An `AddInvocation` change for instance, describes the addition of an invocation of a specific method and maintains a reference to the method it invokes. Keeping this reference avoids the need to recover it later (which is even not always possible due to dynamic typing). Logging changes at this level of granularity enables the change list to contain the information which can be used to recover the invocations of a method.

### 3.2 Composable First-Class Changes

In order to introduce a new feature in a software system, a programmer usually needs to apply several changes. For example, ensuring privacy for users incorporates four changes (lines 28-31 in Figure 5). Every one of those four changes has the same *Raison d'Être*: ensuring user privacy. As such, they could be grouped together based on their common Raison d'Être. We distinguish between two kinds of changes: atomic and composite changes, as depicted in Figure 7.

*Atomic changes* are operations that manipulate the abstract syntax tree of a particular program. These operations consist of adding or removing an entity (for instance a class) as well as changing the properties of an entity (for instance the name of a class). Each atomic change operation is applicable which makes it possible to reproduce each development stage that a program went through during its evolution. This can be done by iterating over the complete list of changes.

*Composite changes* consist of multiple changes which can in turn be composite or atomic changes and which are carried out as a transaction whenever the composite change is executed. They group some changes in a composition, which can be more expressive than the atomic changes on their own. A `ChangeMethod`, for instance, is a composition of a `RemoveMethod` and an `AddMethod` change. First removing a method is not allowed, as it would violate the system invariant which states that a method can not be called if there is no definition for it. As such, all the invocations of that method would have to be removed before

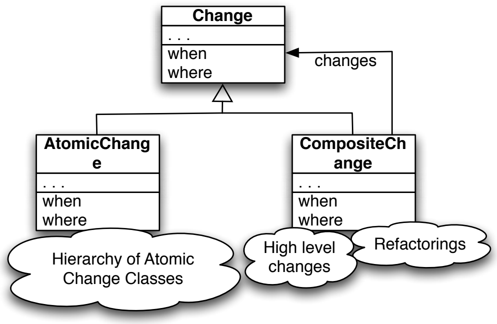


**Fig. 7.** Composable First-Class Changes Design

applying the `RemoveMethod` and re-added after applying the `AddMethod`. First adding the new method would also be prohibited as this would violate the system invariant that a class can only contain one method with a certain signature. When a `RemoveMethod` and an `AddMethod` are applied in as a `ChangeMethod` composition, the invocations of that method do not need to be altered.

Composing changes also improves the understandability of the change list. A system history consisting of only atomic operations leads to an enormous and poorly organized amount of information. Therefore atomic operations are composable: they can be grouped together into higher-level operations with a more abstract meaning.

A final application of composite changes can be found in domain-specific changes. Imagine that we envision the addition of lots of new kinds of users to our Chat application. In that case, it probably would be better to define an `AddUserClass` change, which will in turn add a subclass to the `User` hierarchy and add methods for instanciating that new class. As such, the level of abstraction is raised, improving the change list's understandability.

### 3.3 Dependent First-Class Changes

A change is always applied on a building block, which we refer to as the *subject* of change. *Creational changes* are changes which have as subject a new entity that they produce. In general, a change `c1` is said to *depend* on a change `c2` if that is the creational change of the subject of `c1`.

In our model, every change has a set of preconditions that should be satisfied before a change is applied. Such preconditions are related to system invariants imposed by the programming language (usually defined by the meta-model of the language), for example, methods can only be added to existing classes. Preconditions enable expressing dependency relationships between changes. In the scenario of Section 2, for instance, the change that adds the method `name` to the `RegisteredUser` class depends on the change that added the `RegisteredUser` class to the system, as the latter is the creational change for the subject of the former.

## 3.4 Intensional First-Class Changes

A set of changes can be specified extensionally – by listing them – or intensionally – by expressing them declaratively. Assume a change in which we rename the `name` method of the users of Figure 6 to `username`. This evolution step can be implemented by the following two algorithms 1 and 2.

**Algorithm 1** Change method body: extension

```
Change the name of method name in class RegisteredUser to username
Change the name of method name in class Guest to username
Change the invocation s name in method receive:from: from User to s username
```

**Algorithm 2** Change method body: intension

```
Change the name of method name in all subclasses of the class User to username
Change every invocation of method name to an invocation of method username
```

Both algorithms 1 and 2 represent the same modification to the system: a change name refactoring to method `name`. There is an important difference between both, however. Combining this change with the addition of another invocation of method `name` would result in an inconsistency when the extensional list is applied – the other invocation of `name` would not be changed). As the intensional change finds *every* invocation of `name` by definition (Algorithm 2), such conflicts are avoided. Such changes are called intensional changes. Note that we need a way for expressing qualifiers like *every* for specifying intensional changes. A logic-based declarative language is proven to be suited for such purpose [6].

## 3.5 Change-Oriented Interactive Development Environment

In order to support COSE, an Interactive Development Environment (IDE) is necessary which allows developers to develop (and evolve) their software by means of changes. In fact, from this point of view, developing is not different from evolving software. Concretely, a change-oriented IDE should:

- incorporate an extensible change hierarchy allowing new kinds of changes,
- support the verification of pre-conditions of changes,
- allow the specification of composed atomic changes and log their common raison d'être,
- support the declaration of intensional changes,

– maintain the references between the program entities and their creational changes,
– include ways for developers to evolve their software by means of changes [4].

## 4 Validation: ChEOPS

As a proof of concept, we have developed ChEOPS: the Change- & Evolution-Oriented Programming Support which is written as a plugin for the Smalltalk VisualWorks IDE. It supports COSE with a focus on object-oriented programs. As such, the subjects of the ChEOPS changes correspond to the building blocks of an object-oriented design. As a meta-model for those object-oriented designs, the *Famix* model [8] was chosen. It offers a language-independent expression of class-based object-oriented designs and splits up between arguments(for instance parameters), entities (for instance classes), associations (for instance invocations) and models (which are not taken into account by our implementation) [9]. Famix Objects have some properties which also depicted in Figure 8.

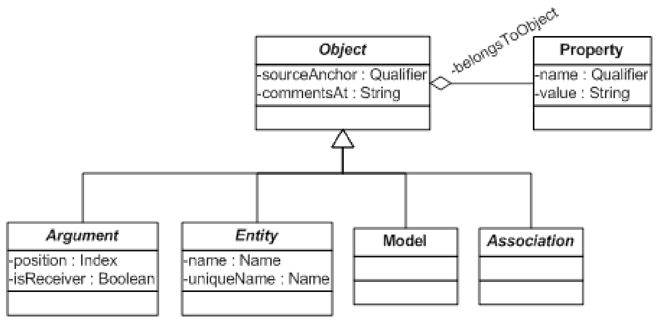


**Fig. 8.** Famix Objects

Change types for all the Famix building blocks were defined and implemented in ChEOPS. Figure 9 highlights the part of the ChEOPS change class hierarchy which have entities as their subject. We refer to [10] for a more complete and detailed explanation of the change classes.

The following goes back to the evolution scenario from Section 2.1 and shows how the support provided by ChEOPS overcomes the identified four issues. Figure 10 shows the list of changes of the basic 2-class chat application, as they are

[4] In fact, some IDE's already provide some support to that regard. Eclipse [7] and VisualWorks [4] for instance, both provide an interactive way of adding a new class to a system. Both do this by means of graphical dialogs which request the desired information from the developers. Support to apply other kinds of changes, like adding methods or more fine-grained changes, however, is not included in those IDE's.

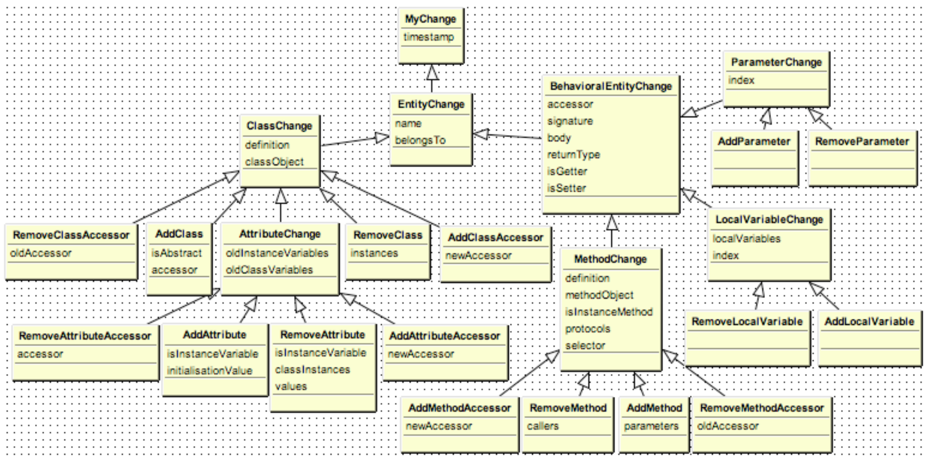


**Fig. 9.** Changes on the Famix metamodel

logged by ChEOPS. Figures 11 and 12 respectively show the changes of adding different user kinds and ensuring user privacy in the way that ChEOPS logged them.

As opposed to Figure 5, the ChEOPS change list allows a clear separation between an addition of a new class (line 2 of Figure 10) and the addition or removal of instance variables to a class (lines 5 and 6 of Figure 10). This is a consequence of *not overloading* the `AddClass` change, but separating every change in a different kind of change class. This improves the understandability of the change list.

```
1   Changes on ChatApp package:
2     Add new class "User"
3     Add new instance method "receive: m from: s" to class "User"
4                -> Invocation tree added
5     Add new instance variable "cr" to class "User"
6     Add new instance variable "username" to class "User"
7     Add new instance method "send: m" to class "User"
8               -> Invocation tree added
9     Add new instance method "name" to class "User"
10              -> Add Read Access
11    Add new class "ChatRoom"
12    Add new instance variable "users" to class "ChatRoom"
13    Add new instance method "get: m from: s" to class "ChatRoom"
14              -> Invocation tree added
15    Add new instance method "register: u" to class "ChatRoom"
16              -> Invocation tree added
17    Add new instance method "unregister: u" to class "ChatRoom"
18         Add invocation "users remove: u"
```

**Fig. 10.** Chat Application: change list by ChEOPS

In the ChEOPS change list, every modification on the method level is not only represented by a statement like `User send: (change)` (lines 7, 30 of Figure 5). Instead, ChEOPS distinguishes between the addition (line 7 of Figure 10) and the removal of a method (line 10 of Figure 12). Next to that, ChEOPS does not only log changes on the method level, but also logs more *fine-grained changes* on the statement level of the method bodies. This overcomes the problem of *the restricted granularity* which was identified in Section 2.

```
Changes on ChatApp package:
  Add new class "Guest"
  Add new instance method "name" to class "Guest"
           -> Invocation tree Added
  Add new class "RegisteredUser"
  Add new empty instance method "name" to class "RegisteredUser"
  Add read access to behavioral entity "name" >> return value of  variable "username"
  Remove instance method "name" from class "User"
           -> Invocation tree Removed
```

**Fig. 11.** Adding Different Users: change list by ChEOPS

```
 1   Changes on ChatApp package:
 2     Add new instance method "encrypt: m" to class "User"
 3              -> Invocation tree Added
 4     Add new instance method "decrypt: m" to class "User"
 5              -> Invocation tree Added
 6     Remove instance method "receive: m from: s" from class "User"
 7              -> Invocation tree Removed
 8     Add new instance method "receive: m from: s" to class "User"
 9              -> Invocation tree Added
10     Remove instance method "send: m" from class "User"
11              -> Invocation tree Removed
12     Add new instance method "send: m" to class "User"
13              -> Invocation tree Added
```

**Fig. 12.** Adding user privacy: change list by ChEOPS

When both extensions need to be merged in order to only allow registered users to send and receive encrypted messages, we actually want to obtain a change list like in Figure 13. This shows that just concatenating the changes of both programmers does not do the job. In fact, the four changes of the second programmer which were originally applied on the single class `User` now need to be applied on both the `RegisteredUser` class and the `Guest` class. As such, we want to group these changes by their intention and parameterize them with the class on which they need to be applied. Additionally, the two classes differ in what kind of encryption and decryption functions are required. Consequently these functions have to be expressed as additional parameters to this composition of changes. The change list of Figure 14 shows the same ChEOPS change list as Figure 13, but expressed by the high-level `AddEncryptionChange`.

This does not only solve the merging problem of the evolution scenario, but also brings along improved support for reusing changes. This high-level change can now be applied on all kinds of classes that understand the `name` message and have access to an instance variable `cr` which behaves like a `Chatroom`. Next to that, it also improves the readability of the change list, as the extensive list of composite changes is abstracted away behind the high-level change.

```
Changes on ChatApp package:
  Add new instance method "encrypt: m" to class "RegisteredUser"
            -> Invocation tree Added
  Add new instance method "decrypt: m" to class "RegisteredUser"
            -> Invocation tree Added
  Remove instance method "send: m" from class "RegisteredUser"
            -> Invocation tree Removed
  Add new instance method "send: m" to class "RegisteredUser"
            -> Invocation tree Added
  Add new instance method "encrypt: m" to class "Guest"
            -> Invocation tree Added
  Add new instance method "decrypt: m" to class "Guest"
            -> Invocation tree Added
  Add new instance method "send: m" to class "Guest"
            -> Invocation tree Added
  Add new instance method "receive: m" to class "Guest"
            -> Invocation tree Added
```

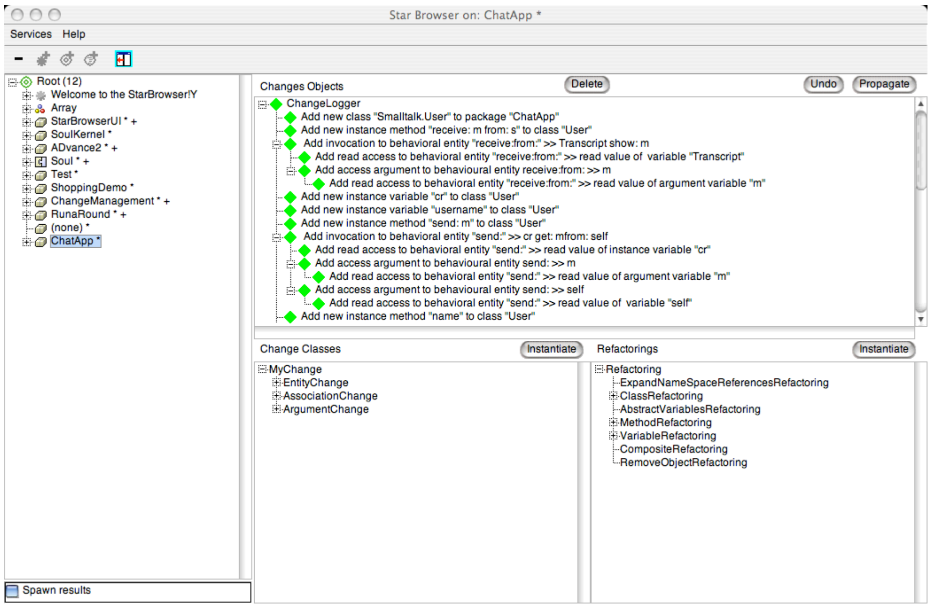
**Fig. 13.** Adding user privacy correctly: change list by ChEOPS

```
Changes on ChatApp package:
  Add encryption for class "RegisteredUser" with function "f(x)= encrypt(x)" and
   "f(x)= decrypt(x)"
            -> composite changes
  Add encryption for class "Guest" with function "f(x)=x" and "f(x)=x"
            -> composite changes
```

**Fig. 14.** Adding user privacy correctly: (compositional) change list by ChEOPS

Exploring the change list of ChEOPS is easier and more user-friendly than in the traditional ChangeList tool. This is a consequence of exploiting the *improved exploration support* brought by the first-class change model behind COSE. In ChEOPS, changes can be looked at from three perspectives: ordered on time, grouped by affected entity, grouped by intension.

- The first view (depicted in Figure 15) is similar to the traditional ChangeList approach, with the difference that ChEOPS organises the changes in a tree structure where the fine-grained changes beyond the method-level are hidden in the branches of the method-changes.
- Figure 16 shows how the second view groups the changes by the entity they affect. This entity can be any of the building blocks of the Famix meta-model:



```
ChangeLogger
  Add new class "Smalltalk.User" to package "ChatApp"
  Add new instance method "receive: m from: s" to class "User"
  Add invocation to behavioral entity "receive:from:" >> Transcript show: m
    Add read access to behavioral entity "receive:from:" >> read value of  variable "Transcript"
    Add access argument to behavioural entity receive:from: >> m
      Add read access to behavioral entity "receive:from:" >> read value of argument variable "m"
  Add new instance variable "cr" to class "User"
  Add new instance variable "username" to class "User"
  Add new instance method "send: m" to class "User"
  Add invocation to behavioral entity "send:" >> cr get: mfrom: self
    Add read access to behavioral entity "send:" >> read value of instance variable "cr"
    Add access argument to behavioural entity send: >> m
      Add read access to behavioral entity "send:" >> read value of argument variable "m"
    Add access argument to behavioural entity send: >> self
      Add read access to behavioral entity "send:" >> read value of  variable "self"
  Add new instance method "name" to class "User"
```

```
MyChange
  EntityChange
  AssociationChange
  ArgumentChange
```

```
Refactoring
  ExpandNameSpaceReferencesRefactoring
  ClassRefactoring
  AbstractVariablesRefactoring
  MethodRefactoring
  VariableRefactoring
  CompositeRefactoring
  RemoveObjectRefactoring
```

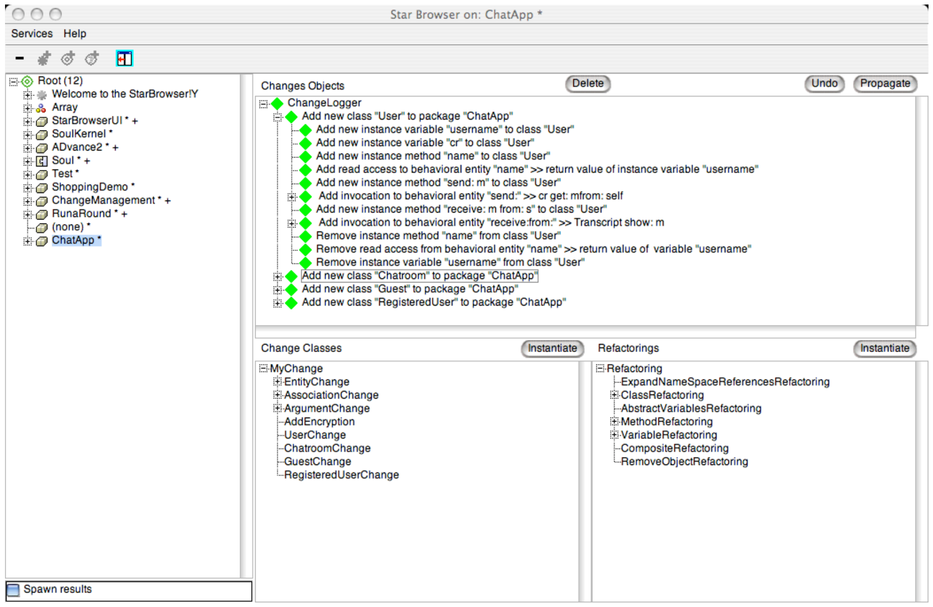
**Fig. 15.** ChEOPS view on change list (ordered by time)

class, method, attribute, etc. This view also contains a tree of changes, where the dependent changes are hidden in the branches of the creational change of their subject. For example, the dependent changes of the creational change of `User`, are structured in a branch below that change.

– A third view which is included in ChEOPS groups the changes by their composition. Figure 17 shows the composite `AddAncryptionChange`, and the atomic changes it contains. This view dramatically decreases the number of changes to be displayed.
– Yet another view could group the changes by their common intension. this is however, not implemented yet in ChEOPS. The nodes of this tree view should contain intensional changes, while the branches contain the corresponding extensions. Note that, when an intensional node is expanded, first it's intension is evaluated against the current application and second the corresponding extension is produced and listed in the view. As such, the extension corresponding to an intension depends on the other changes in the change history.

## 5 Discussion

Table 1 shows an overview of the four problems we identified in Section 2. The vertical axis shows the extensions that we propose to the first-class change model (Section 3). Cells containing an `x` denote that the extension of the cell's row helps

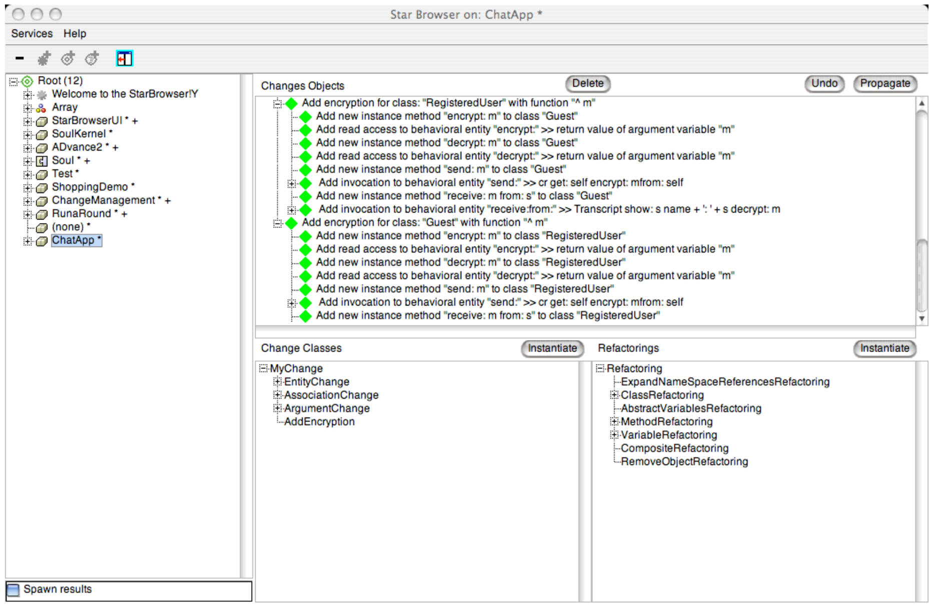


```
ChangeLogger
  Add new class "User" to package "ChatApp"
    Add new instance variable "username" to class "User"
    Add new instance variable "cr" to class "User"
    Add new instance method "name" to class "User"
    Add read access to behavioral entity "name" >> return value of instance variable "username"
    Add new instance method "send: m" to class "User"
     Add invocation to behavioral entity "send:" >> cr get: mfrom: self
    Add new instance method "receive: m from: s" to class "User"
     Add invocation to behavioral entity "receive:from:" >> Transcript show: m
    Remove instance method "name" from class "User"
    Remove read access from behavioral entity "name" >> return value of  variable "username"
    Remove instance variable "username" from class "User"
  Add new class "Chatroom" to package "ChatApp"
  Add new class "Guest" to package "ChatApp"
  Add new class "RegisteredUser" to package "ChatApp"
```

```
MyChange
  EntityChange
  AssociationChange
  ArgumentChange
  AddEncryption
  UserChange
  ChatroomChange
  GuestChange
  RegisteredUserChange
```

```
Refactoring
  ExpandNameSpaceReferencesRefactoring
  ClassRefactoring
  AbstractVariablesRefactoring
  MethodRefactoring
  VariableRefactoring
  CompositeRefactoring
  RemoveObjectRefactoring
```

**Fig. 16.** ChEOPS view on change list (ordered by affected entity)



```
  Add encryption for class: "RegisteredUser" with function "^ m"
    Add new instance method "encrypt: m" to class "Guest"
    Add read access to behavioral entity "encrypt:" >> return value of argument variable "m"
    Add new instance method "decrypt: m" to class "Guest"
    Add read access to behavioral entity "decrypt:" >> return value of argument variable "m"
    Add new instance method "send: m" to class "Guest"
     Add invocation to behavioral entity "send:" >> cr get: self encrypt: mfrom: self
    Add new instance method "receive: m from: s" to class "Guest"
     Add invocation to behavioral entity "receive:from:" >> Transcript show: s name + ': ' + s decrypt: m
  Add encryption for class: "Guest" with function "^ m"
    Add new instance method "encrypt: m" to class "RegisteredUser"
    Add read access to behavioral entity "encrypt:" >> return value of argument variable "m"
    Add new instance method "decrypt: m" to class "RegisteredUser"
    Add read access to behavioral entity "decrypt:" >> return value of argument variable "m"
    Add new instance method "send: m" to class "RegisteredUser"
     Add invocation to behavioral entity "send:" >> cr get: self encrypt: mfrom: self
    Add new instance method "receive: m from: s" to class "RegisteredUser"
```

```
MyChange
  EntityChange
  AssociationChange
  ArgumentChange
  AddEncryption
```

```
Refactoring
  ExpandNameSpaceReferencesRefactoring
  ClassRefactoring
  AbstractVariablesRefactoring
  MethodRefactoring
  VariableRefactoring
  CompositeRefactoring
  RemoveObjectRefactoring
```

**Fig. 17.** ChEOPS view on change list (ordered by composition)

to overcome the problem of the cell's column. The rest of this section discusses how this is achieved:

| Extension | Restricted granularity | Term overloading | Lack of high-level changes | No exploration support |
|---|---|---|---|---|
| Fine-grained changes | x | x | | |
| Composable changes | x | | x | x |
| Dependent changes | | | | x |
| Intensional changes | | | x | x |
| IDE support | | | x | x |

**Table 1.** Problems handled by extensions to first-class change model

**Restricted level of granularity** The restricted level of granularity is noticeable by the lack of both very fine-grained and very coarse-grained changes. Our model includes not only coarse-grained changes (such as the addition of classes or methods), but also more *fine-grained changes* such as method invocations and accessors. The possibility to compose changes into *composite changes*, allows the definition of more coarse-grained changes, which can be used to abstract away from the fine-grained level. These extensions provide the granularity required to reason about and understand the evolution history of programs.

**Term overloading** Our model provides a different change for every building block of the Famix meta-model. As such, every change on such a building block is captured by a specific change. This avoids overloading change classes to capture different kinds of changes. We can conclude that in our model, the definition of a change is unique.

**Lack of high-level changes** Our model enables to define high-level changes that better represent the developers' intentions. The model is extensible so that developers can define their own domain-specific changes. This is achieved (1) by the use of an inheritance hierarchy in which all the change types reside and (2) by the composite design pattern (distinguishing between atomic changes and *composite changes*). High-level changes can also be defined as *intensional changes*, which describe a pattern of change. The application of high-level changes is conditioned by the fulfillment of their preconditions. This is supported by an *IDE*, which guides the programmers in defining new kinds of changes, applying their changes, undoing changes and verifying the preconditions to ensure the application consistency.

**No exploration facilities** All the notions of first-class change described in this work are implemented in the *ChEOPS IDE*, which supports program exploration by providing different views on the changes. The views structure the changes based on the *dependencies* between them, on their *composition*, on their *intension* or on the time on which the changes were made. Different views on changes can be used for different goals. While the dependency view, seems interesting to undo and redo changes, the intensional view seems more interesting for understanding the changes.

## 6 Related Work

Change-Oriented Software Engineering centralises changes. Consequently, managing changes is a crucial part of it. In this paper, we focussed on a change management system which is entity-based and works incrementally. This section broadens up this restriction to the entire scope of change management systems, and explains why ChangeList's model of first-class changes was chosen as a starting point of this paper. Afterwards, some change-oriented approaches are explained which are similar to the approach we identified.

### 6.1 Change Management Systems

Change management systems are used to capture, store and reuse changes of evolving software systems. Those changes are then managed at a central repository where each involved party has access to in order to retrieve/publish the desired changes. Hence, change management systems contain useful information about the evolution of the managed software systems (for example created classes during the lifetime of a system) and they are a valuable source to provide information about the evolution of software. Change management systems can be classified into two dimensions: time (when are changes stored) and structure (how are changes stored).

The first dimension differentiates between the snapshot-based and incremental approaches. In a snapshot-based setting, developers explicitly store changes to software systems at the central repository of their change management system. Hence, each developer carries the responsibility of storing these changes on a regular basis, called commits. In an incremental setting, it is the Interactive Development Environment (IDE) which is responsible of logging the changes whenever they are performed. As such, in the latter approach, every separate change is automatically committed ensuring that changes are logged as fine-grained as possible and in the order in which they were applied.

The second dimension separates the file-based from the entity-based approaches. While the former approaches store and manage changes as differences on textual files, the latter treat changes on program entities. Program entities are the building blocks of programs. The exerted programming paradigm decides which kind of building blocks must be considered. The file-based approaches require parsing and comparing files to recover information about the changes. This makes it hard to extend the available spectrum of changes as the comparison algorithms would have to be adapted to incorporate new patterns of changes.

Table 2 shows the possible combinations of the two dimensions which were explained above, and presents at least one example per combination. Preference goes out to the incremental entity-based change management systems.

| | File Based | Entity Based |
|---|---|---|
| Snapshot Based | CVS, SNV | CatchUp!, StORE |
| Incremental | AJC ActiveBackup<br>FileHamster | ChangeList<br>Spyware |

**Table 2.** Change Management Systems

**CVS** or Concurrent Versions System is a version control system that enables the recording of the history of source-files and documents. CVS uses a client-server architecture and allows multiple connections from different locations. It is developed to organize and maintain a collection of source files which are stored at the explicit request of team members. Instead of storing each committed file separately, CVS uses an optimized technique. It stores all the committed versions of a file in one single file by only containing the textual differences between them ($\Delta$V). For more information about CVS we refer the reader to [11].

**SVN** or Subversion is an advanced, open source version control system. Its main goal is to help you track the changes to directories of files under version control. Subversion also uses a client-server architecture allowing multiple connections from different locations. Developers can commit revisions anytime. Each revision has its own root which is used to access its contents. Subversion maintains for each file a reference to its most recent version. We refer interested readers to [12] for more details.

**CatchUp!** Henkel and Diwan propose a lightweight approach for recording and replaying refactorings [13]. Their proof-of-concept tool CatchUp! is implemented as an Eclipse extension. After a change is committed by a user, a corresponding Java object is created and stored in an XML trace file. The captured refactorings can be replayed manually by the user or automatically by the provided CatchUp! tool that recreates the refactoring based on the trace file.

**StORE** is the version control system used by the VisualWorks for Smalltalk environment [4]. It is based on a client-server architecture: It uses a centralized server with a database acting as the central repository. Developers have the possibility to publish (commit) packages which will be versioned by StORE. Instead of versioning files, StORE works on a granularity-level of program entities (for instance a class or method) which facilitates for example the merging of source code of different developers.

**AJC Active Backup** is an automatic revision control system that continuously monitors changed files [14]. The user may configure which folders and files are monitored by specifying wildcards. Every time changes are saved to a monitored file, it is revised in a compact archive that acts as a local repository. The incremental approach implemented by this tool results in a complete record of what the user has been doing. AJC Active Backup also offers the possibility of comparing monitored files and showing the differences between them. Instead of storing each changed file separately, AJC Active Backup compresses the archived files by only storing the changes to files.

**FileHamster** is a version tracking application focused on meeting the needs of content creators [15]. It functions in a similar way as AJC Active Backup: It continuously monitors user-specified files and automatically creates incremental backups whenever those files are modified. FileHamster allows the annotation of changes with notes for a detailed overview or quickly locating specific revisions. It also provides the possibility of viewing the differences between two file revisions. The core of FileHamster stores each changed file separately but the tool supports a multitude of plugins for extra functionalities (for example compression).

**Envy** is a software engineering environment that supports the configuration, history and change management required for the development and maintenance of large software systems [18]. Envy contains a snapshot- and entity-based change management system which requires developers to commit their code to an Envy repository maintained on an Envy server. Consequently, it does not provide the accurate information required from a change management system [3].

**ChangeList** is tool which is included in the mainstream Smalltalk IDE's. It maintains changes applied to a Smalltalk program as first-class entities and stores change information geared towards specific Smalltalk program entities. The model of first-class changes behind ChangeList, however, has some shortcomings which show up in the tool's frontend. Section 2 elaborates on these shortcomings.

**SpyWare** is a change-based software repository which was proposed by Robbes and Lanza as the solution for the shortcomings introduced by using snapshot- or file-based change management systems for analyzing software evolution [4, 5]. SpyWare [3] is an IDE plug-in for the Squeak Smalltalk environment. In Spyware, a system history is viewed as the sequence of changes applied to that system. Each change is capable of reconstructing its successive state of source code, expressed by an abstract syntax tree (AST). A system is thus represented by an evolving abstract syntax tree. That abstract syntax tree is composed of program entities specific for the Smalltalk language. As such, SpyWare changes have only Smalltalk specific subjects, implying that SpyWare does not support language independent reasoning. Currently, Robbes and Lanza are working on porting SpyWare to the Java/Eclipse platform by isolating common concepts between Java and Smalltalk.

## 6.2 Other Related Change Tools and Approaches

There are already some approaches which partially support COSE. This section explains those approaches and relates them to COSE.

**Refactoring Browser** The refactoring browser [16] is a powerful Smalltalk browser which allows the programmer to perform various automated refactorings on Smalltalk source code such as renaming variables and methods. In this case the focus is on rapidly and automatically performing a set of standard refactorings without introducing errors. The changes which are applied by the Refactoring Browser, however, are not stored as composite changes, but in as an extensive list of atomic changes.

**MolhadoRef** This Eclipse plugin is described in [17]. It is a version control system that never loses the history of refactored elements, by tracking the evolution history at a fine level of granularity. MolhadoRef is aware of the program elements and treats refactorings as first-class changes. The difference between ChEOPS and MolhadoRef lies in the granularity of the preserved first-class changes. While MolhadoRef only treats refactorings as first-class, ChEOPS also treats the more fine-grained changes as first-class.

**IDE support** Many environments such as Eclipse [7] or VisualWorks [4] already initiate interactive dialogues to add a class. Some of them provide interactive support for refactorings [19, 16] To the best of our knowledge, there are no environments that provide dialogues to maintain all kinds of changes (for instance adding a method, defining a new kind of change, undoing a refactoring, etc).

**Conditional transformation** In [20] a new kind of refactoring tools is proposed, which allow users to create, edit and compose refactorings. The challenge in this work is the computation of the precondition of the composite refactoring from the preconditions of the composed refactorings. For this purpose, a formal model for automatic, program-independent composition of conditional program transformations, is introduced. These techniques can be used to derive the preconditions of composite changes.

## 7 Future work

Change-Oriented Software Engineering lifts up the level of abstraction of the development process. Instead of expressing the programs in a programming language, they are expressed in terms of changes to the building blocks of a chosen programming style. In this paper, we focussed on class-based object-oriented programming, and more specifically on the Famix model.

This work is situated in the domain of software evolution in which component-based programming is frequently asserted. We envision an implementation of COSE for a component-based meta-model, which consists of components, provided services, required services, ports to those services and connectors to connect the ports. Primary experimental results show that COSE is also applicable in a component-based context. More experiments need to be conducted in order to validate its applicability and to find out the opportunities that this brings along.

Another point of future work lies in the modification of the change semantics. Change lists contain a list of changes, which can be applied in order to produce a software system. To apply the changes of a change list, semantics need to be defined for every kind of change from that list. These semantics can be modified in order to tweak the generation of the software system. Possible applications of modification of semantics include platform-specific code-generation, automatic generation of uninstall programs and the propagation of changes to a running system.

## 8 Conclusion

In this paper, we envision Change-Oriented Software Engineering (COSE): a programming paradigm which targets software evolution and which centralises changes as the main entity in the development process. The subject of the change refers to the building block of a programming language in which the program is being developed. As such, COSE builds on top of another programming paradigm in which those building blocks are written. In this paper, we build on the object-oriented programming and take the Famix meta-model as a model for describing the programs that are to be changed.

We show why first-class changes are desired to program in a change-oriented way. We then identify four problems with respect to the model of first-class changes, as it is presented in a related approach. The restricted level of granularity in the different types of changes, the overloading of change types, the lack of high-level changes and the lack of program exploration facilities hinder good software evolution support. This explains the need to extend the model of first-class changes in such a way that these problems are overcome.

Four extensions to the model of first-class changes are presented: fine-grained, composable, dependable and intensional changes. These extensions overcome the problems that are identified in the existing model of first-class changes. ChEOPS, a Smalltalk implementation of COSE is explained. It is based on the existing implementation of first-class changes, but extended with solutions for the four extra requirements that were identified. Experiments in ChEOPS made a validation of the extensions to the model possible. In ChEOPS, we have implemented an evolution scenario in which we show how COSE manages to overcome the four identified problems of the change model.

## References

1. Estublier, J.: Software configuration management: a roadmap. In: ICSE - Future of Software Engineering Track. (2000) 279–289
2. Robbes, R., Lanza, M.: Versioning systems for evolution research. In: Proceedings of Eighth International Workshop on Principles of Software Evolution, IEEE Computer Society (2005) 155–164
3. Robbes, R., Lanza, M.: A change-based approach to software evolution. Electronic Notes in Theoretical Computer Science (2007) 93–109
4. Howard, T., Goldberg, A.: VisualWorks - Application Developer's Guide. Cincom Systems (1993-2005)
5. University of Illinois at Urbana-Champaign: Visualworks: Change list tool. http://wiki.cs.uiuc.edu/VisualWorks/Change+List+Tool (2007)
6. Mens, K., Michiels, I., Wuyts, R.: Supporting software development through declaratively codified programming patterns. In: Journal on Expert Systems with Applications. Volume 23., Elsevier Publications (2002) 405–413

7. The Eclipse Corporation: Eclipse. http://eclipse.org (2007)
8. Demeyer, S., Tichelaar, S., Steyaert, P.: FAMIX 2.0 - the FAMOOS information exchange model. Technical report, University of Berne (1999)
9. Demeyer, S., Ducasse, S., Tichelaar, S.: Why famix and not uml? uml shortcomings for coping with round-trip engineering. UML'99 Conference Proceedings (1999)
10. Ebraert, P., Mens, T., D'Hondt, T.: Enabling dynamic software evolution through automatic refactorings. In: Proceedings of the Workshop on Software Evolution Transformations (SET2004), Delft, Netherlands (2004)
11. Price, D.R.: Cvs - open source version control. http://www.nongnu.org/cvs/ (2006)
12. CollabNet: Subversion. http://subversion.tigris.org/ (2006)
13. Henkel, J., Diwan, A.: Catchup!: capturing and replaying refactorings to support api evolution. In: ICSE '05: Proceedings of the 27th international conference on Software engineering. (2005) 274–283
14. AJC Software: Ajc active backup. http://www.ajcsoft.com/AJCActBk.php (2007)
15. Mogware: Filehamster - a personal revision control solution for content creators. http://www.mogware.com/FileHamster/ (2006) [Last accessed 30 May 2007].
16. Brant, J., Roberts, D.: Refactoring browser. Technical report, http://wiki.cs.uiuc.edu/RefactoringBrowser (1999)
17. Dig, D., Nguyen, T.N., Manzoor, K., Johnson, R.: Molhadoref: a refactoring-aware software configuration management tool. In: OOPSLA'06 Companion, Portland (2006)
18. Pelrine, J. , Knight, A., Cho A. Mastering Envy/Developer Cambridge University Press (2001)
19. Ekman, T., Asklund, U.: Refactoring-aware versioning in eclipse. Electr. Notes Theor. Comput. Sci. **107** (2004) 57–69
20. Kniesel, G., Koch, H.: Static composition of refactorings. Science of Computer Programming **52**(1-3) (2004) 9–51

# Encapsulating and Exploiting Change with Changeboxes

Marcus Denker, Tudor Gîrba, Adrian Lienhard,
Oscar Nierstrasz, Lukas Renggli, Pascal Zumkehr

Software Composition Group, University of Bern
`www.iam.unibe.ch/∼scg`

**Abstract.** Real world software systems change continuously to meet new demands. Most programming languages and development environments, however, are more concerned with limiting the effects of change rather than enabling and exploiting change. Various techniques and technologies to exploit change have been developed over the years, but there exists no common support for these approaches. We propose Changeboxes as a general-purpose mechanism for encapsulating change as a first-class entity in a running software system. Changeboxes support multiple, concurrent and possibly inconsistent views of software artifacts within the same running system. Since Changeboxes are first-class, they can be manipulated to control the scope of change in a running system. Furthermore, Changeboxes capture the semantics of change. Changeboxes can be used, for example, to encapsulate refactorings, or to replay or analyze the history of changes. In this paper we introduce Changeboxes by means of a prototype implementation. We illustrate the benefits that Changeboxes offer for evolving software systems, and we present the results of a preliminary performance evaluation that assesses the costs associated with Changeboxes while suggesting possible strategies for improvement.

## 1 Introduction

It is well-established that so-called *E-type* systems, *i.e.,*, real-world applications that are *embedded* in the environment where they are used, *must* change continuously or else become less useful over time [1].

Oddly enough, most programming languages and development environments have traditionally invested more effort into providing mechanisms that *limit* change than into those that enable or exploit change [2, 3]. Some typical symptoms of this phenomenon include:

– A name for a software artifact, such as a class, a type or a module, is generally assumed to have a globally consistent meaning within a single running system. Different versions of the same artifact cannot be simultaneously active within the same system, *e.g.,*, a single virtual machine (VM).
– Whenever an artifact evolves, this must be done in such a way that existing clients are not adversely affected. Although interfaces may be *deprecated*, it can be hard or impossible to definitely remove them.

– Frameworks provide not only black-box but also white-box components. Refactoring may preserve the public interface of a subsystem while breaking implicit contracts visible only to subclasses in client code [4]. As a consequence, developers may be reluctant to modify framework components that may break client application code.

The fact that software evolution needs to be effectively managed can be observed by the range of tools and techniques developed over the years. Consider some of the following examples. Versioning systems are used to keep track of changes made to a software system, and configuration management systems manage the different versions that need to be deployed [5]. In Model-Driven Engineering, model transformations are used to ensure that different views of a software system remain synchronized when changes to requirements occur [6]. Refactoring operations can be mined from versioning repositories or stored when they happen, and then replayed to ease migration at multiple client sites [7, 8]. Adaptations of third-party software can be conveniently expressed as locally visible extensions [9]. OpenModules [10] allow clients to adapt a module by providing advice code for methods and pointcuts declared in the interface. Each of these techniques, however, manages and exploits change from its own perspective and at different levels of abstractions, and little effort is made to benefit from a common infrastructure. In each case we have thoroughly different mechanisms to express and manage change. Furthermore, these approaches generally focus on managing change to static software artifacts, rather than on enabling change to a running system.

Although change is fundamental to software, we are lacking the means to encapsulate and express change in such a way that it can be effectively controlled and exploited. Essentially we need to address the following three points:

– *Encapsulating change.* In order to fully exploit the history of changes, we need a mechanism to *capture change as it occurs.* In this way the full history of a system is accessible as it evolves. Furthermore, the semantics rather than simply the syntactic effects of change need to be captured so that we can reason about change and manipulate it in a meaningful way. Depending on the application, a change may be encapsulated as, for example, an edit, a refactoring, or a more general kind of transformation.
– *Scoping change.* Different versions of the same software may be active simultaneously, even within the same virtual machine (VM). Therefore we need means to control the scope of change within a running system.
– *Deploying change.* Running systems evolve, therefore we need means to merge and deploy changes on the fly and without restarting the system.

We propose Changeboxes as a mechanism to address these issues. A Changebox is a first-class entity that encapsulates change while providing an execution scope for controlling the visibility of changes to the running system. A Changebox provides a unifying mechanism for various kinds of change. Multiple Changeboxes can be simultaneously active *within the same running system.* Since Changeboxes capture the entire history of change, it is possible to exploit these historical views at run-time. Furthermore Changeboxes can be composed by merging. A fine degree of control over the semantics of merging is possible — even necessary — to allow Changeboxes to be fully exploited.

*Structure of this paper:* In the following section we present a running application of Changeboxes that illustrates several key scenarios for encapsulating and exploiting change. In Section 2 we provide a capsule summary of the Changebox model. Section 4 presents a more detailed description of the prototype implementation of the Changebox model. In particular, we explain how change is encapsulated as Changeboxes, how Changeboxes provide a scope for execution, and how different strategies for merging are supported. We evaluate the prototype in Section 5 with the help of a number of benchmarks. In Section 5 we discuss new avenues that are opened by this work. We discuss related work in Section 7, and we conclude in Section 8 with some remarks about ongoing and future work.

## 2 Motivating example

In this section we present different scenarios that hamper evolution when using traditional static versioning systems such as CVS or SVN. As an example we have chosen Pier [11], an industrial strength Content Management System (CMS) built on top of the Seaside web application framework [12]. Web applications offer a good example of a domain in which changes occur frequently, yet the applications need to be up and running virtually all the time.

In Figure 1 we see the evolution of a Pier application along three development branches. The arrows point backwards in time to their respective ancestor versions. Each snapshot of the system is defined by a commit to the versioning system of all the sources of this particular version.

### 2.1 Traditional Versioning System

We see that the code has been split from the release branch and deployed to create a customized version (1) for a customer. A third development branch leads to a defective version (2) where the current development for next version is going on.

To fix a bug (3) in the released branch the developer needs to either update his working copy to this particular version or check out a new working copy at a different location. Either way, he is required to recompile the system and restart the development server, which can take a significant amount of time for large applications. Moreover, if developers want to have different versions of the same application running at the same time, they need a complicated setup, presumably with different back-ends and multiple server entry points.

Merging and deploying the bug fix (4) on the server is difficult. Since a downtime of the application is often not acceptable, the developer has to install the new version separately. Then the server configuration has to be changed so

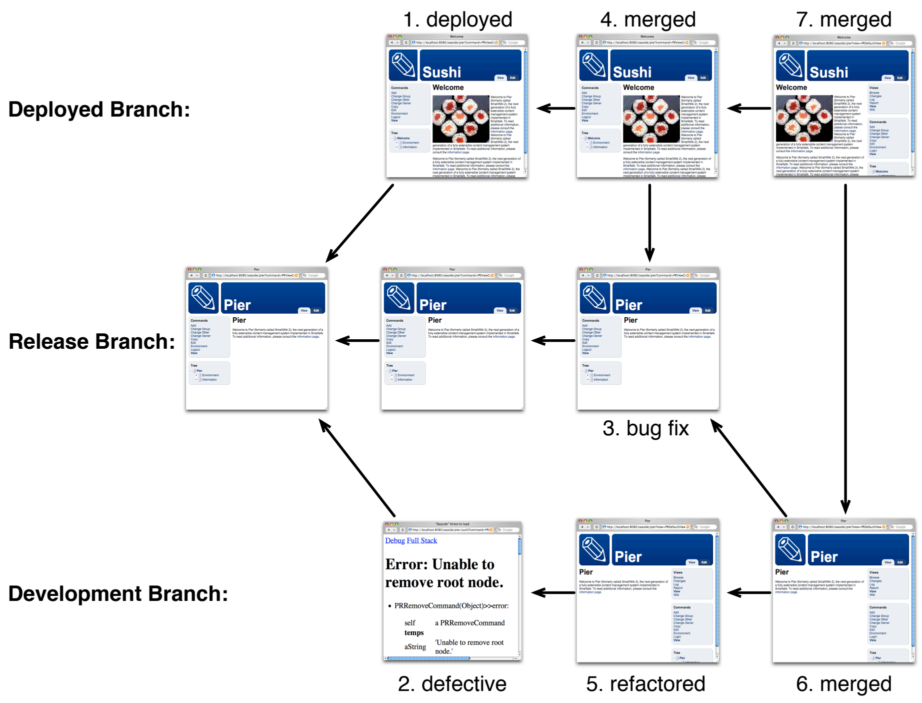


**Fig. 1.** The evolution of a web application: every screenshot represents the running system in the context of its history. The arrows points back to the ancestor versions.

that new user session will open in the new server instance. Old sessions continue to live in the old version or, if possible, are redirected and migrated to the new one.

In the meantime some heavy refactorings (5) — like class-renames affecting many different places in the code — were applied to the development branch. Merging of the bug fix into the other two branches is not trivial anymore. Unfortunately the versioning system does not know what refactorings were performed and is therefore unable to automatically transform the code to be merged (6, 7). The developer is required to manually perform the refactorings.

## 2.2 Advantages when using Changeboxes

We have also developed the previously described scenario within a Changeboxes-aware system. While doing so, the drawbacks described above are overcome. As Changeboxes keep track of different versions of the source code available within the development environment there is no need for manual checkout and recompilation. Moreover, Changeboxes also define the mechanisms to execute any version of this code at any time, so it is just a matter of swapping the scope by selecting the version to be used. Every process of the system runs within its own scope defining a specific view on source code, classes and executable methods.

When deploying an update of the application on a specific server (4, 7), there is no need to set up a backup server. New code can be directly loaded and compiled in the running system without affecting any currently existing user session. Within the scope of the new code, developers can even run unit- and functional-tests on the server itself while the productive part is still using the old code. When the developer is confident about deploying the revised code, it can be activated for new sessions. Existing sessions will run to completion with the old code, since every session references the specific version it has been started with. Unless a developer decides to change this reference manually, existing sessions will stick with the same code even though changes have been applied to the system in the meantime.

Although it may seem highly unusual to develop, deploy and test within the same environment, numerous advantages can be gained if this is done in a disciplined way. Having the possibility to work on the same machine for development, testing and deployment eliminates the need to set up multiple machines with similar but not necessarily identical environments. Problems the customer might experience can be investigated and possibly fixed directly on the server and do not have to be reproduced in a different environment, thus speeding up corrective maintenance tasks.

Refactorings such as class renamings affect many places in the code. The Changeboxes system tracks what refactorings were performed and transforms the code to be merged as well, so that the fix can be added to the 1.1 branch without any difficulty (6). Thus, backwards compatibility is rendered irrelevant.

Evolution of the Changebox-aware Pier application benefits from the following points:

- Instead of checking out code and manually recompiling the system, all versions are always available in an executable form. Every process can define its own view on the system and therefore multiple versions can be running on the same VM at the same time.
- Instead of having to reproduce bugs within a different development environment, they can be quickly fixed and tested on the running server. Users of the application benefit by immediately profiting from fixes without even having to login again.
- Instead of having the versions available in a textual form like in traditional versioning systems, the versions are available in an executable form. By running tests in different versions developers can determine exactly what change caused a test to fail.
- Refactorings performed in one branch and encapsulated as Changeboxes can be replayed in another development branch by merging the corresponding Changeboxes.

## 3 Changeboxes in a nutshell

In Figure 2 we show the core of the Changebox metamodel (at the bottom) and several versions of a Changebox-aware application conforming to this metamodel (on top).

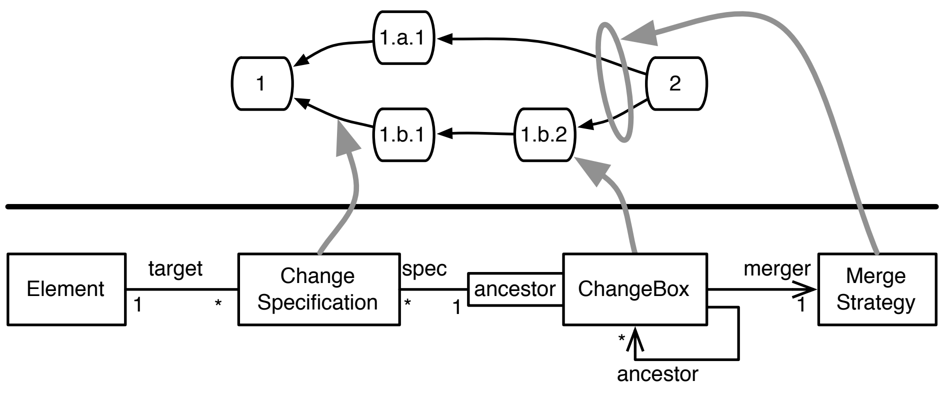


**Fig. 2.** The Changebox metamodel and an example of Changebox dependencies.

A software system is composed of various *elements*, such as classes, methods and packages. Many different versions of a system may exist and may be run at the same time. Each of these versions consists of incremental changes to a previous *ancestor* version.

A Changebox is an immutable entity that defines a snapshot of a system by:

– encapsulating a set of change specifications,
– specifying a set of ancestor Changeboxes to which these changes apply, and
– providing *a scope for dynamic execution*.

A *change specification* describes how one version of an element may be transformed into another version of that element. The system can only be changed by creating a new Changebox that encapsulates a change specification.

In Figure 2, Changebox 1 is empty. Changebox 1.a.1 encapsulates a single change specification, thus defining a new version of the system and a new scope for dynamic execution. Changebox 2 has two change specifications which are applied to elements found in ancestors 1.a.1 and 1.b.2. Changebox 2 performs a merge. As merges can introduce possible conflicts, a strategy is used to implement the different ways of solving the conflicts.

In Figure 2 we also introduce the core of a Changebox-system (a more detailed diagram can be found in Figure 3). An element represents a structural entity in the programming language (*e.g.*,, package, class, method). A Changebox has several ancestors and encapsulates a set of ChangeSpecifications. The possible conflicts that appear due to merging are resolved with a MergeStrategy.

At first glance, a Changebox-system may appear to be yet another versioning system. However the key difference is that a Changebox-system maintains versions of a running system, not just static software artifacts. Furthermore, any and all versions may be executable at the same time in the same VM, by providing for each Changebox a scope for dynamic execution.

Each process is executed within the scope of a Changebox which defines a consistent, flat view of the system. This view is determined by the changes of the selected Changebox and all its ancestors. Rather than compiling the whole system for a specific version to be executed, the actual version of a method or a class is looked up at runtime. This allows for compiled methods to be shared between different versions of a system. Furthermore, since methods are compiled incrementally, execution of a new version starts instantaneously.

## 4 Implementing Changeboxes in Smalltalk

The Changebox prototype is implemented in Squeak[1], an open source Smalltalk dialect.

We decided to use the reflective features of Smalltalk to implement Changeboxes rather than attempting to modify the underlying VM. Although an implementation at the level of the VM would certainly have performance advantages, we felt that the development effort would be too great for a first prototype. This strategy enabled us to quickly obtain a proof-of-concept prototype of Changeboxes with adequate performance to carry out real experiments. At the same time we were able to gain insights into the implications of various implementation approaches.

To implement Changeboxes in Smalltalk, the following problems must be solved:

– The Changebox metamodel needs to be implemented and the Smalltalk reflective kernel needs to be extended to capture structural changes to software entities as they occur (Section 4.1).
– The Smalltalk runtime environment needs to be extended with execution scopes so that different versions of the same program may execute concurrently (Section 4.2).
– A flexible approach is needed to merging Changeboxes that can easily be adapted to the needs of different applications (Section 4.3).

We summarize our approach in the rest of this section. Full details are to be found in Zumkehr's Masters thesis [13].

### 4.1 Modeling and capturing changes

Figure 3 illustrates the key classes of the Changebox implementation. In addition to the three key classes `ChangeBox`, `ChangeSpecification` and `Element` we see:

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

– the concrete element and change specification subclasses,
– the system classes `Class` and `CompiledMethod` representing specific versions of classes and methods used by the VM,
– the class `MergeStrategy` to define the merge semantics,
– the class `WorkSession` which tracks a sequence of changes of a user, and
– the class `Scope`, the common superclass of `ChangeBox` and `WorkSession`.

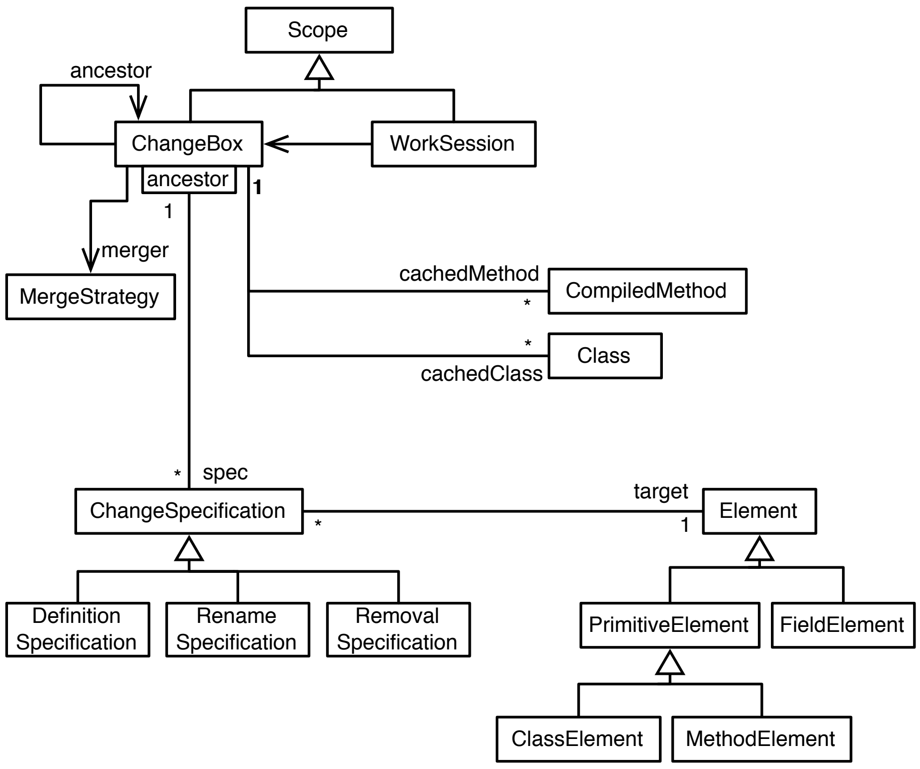


**Fig. 3.** Changebox implementation class diagram.

Elements represent the historical perspective of static entities of the programming language, such as classes, methods, and fields. Only within a specific scope can a version of an actual software entity be produced from an element (*e.g.*,, `CompiledMethod`).

Since Smalltalk is a reflective system implemented in itself, instances of the system classes `Class` and `CompiledMethod` are the meta-objects used by the VM as it executes. They provide the necessary information for the VM to instantiate new objects (the class format) and execute methods (bytecode). Fields, however, are not reified in the Smalltalk runtime, so only classes and methods are considered to be primitive elements. (`ClassElement` and `MethodElement` inherit from `PrimitiveElement`, but `FieldElement` does not.) As a consequence, renaming a field will entail a change to the class that defines it.

The actual changes between two points in the evolution of a system are defined by change specifications which are encapsulated in Changeboxes. A change specification implements the process to change the form of an element (*e.g.,*, (re-)definition, rename, or removal) within a certain system snapshot. For example, a definition specification holds the new properties of the changed element (as the source code of a method, or the superclass of a class), and a rename specification specifies the new name.

The changed element is stored in the Changebox which encapsulates the change (*i.e.,*, in the form of a `Class` or a `CompiledMethod`). In this way the right version of an element can be efficiently retrieved when it is looked up in a specific execution scope. Lookup is discussed in detail in Section 4.2 and merging is discussed in Section 4.3.

In the remaining part of this section we discuss how changes are captured, how work sessions keep track of a sequence of changes and how specifications are generated from changes.

**Capturing changes.** In a conventional Smalltalk system, there exists exactly one version of each method and class at a time. Changes, *i.e.,*, the recompilation of methods and classes, are performed through the reflective kernel; the previous version of a compiled method or class is replaced with the new one and is eventually garbage collected.

In the Changebox model, the system can always be viewed exactly as it existed at an arbitrary moment in the past. Therefore, a Changebox system needs to keep track of all changes. Our implementation modifies the appropriate structural reflection entry points of the kernel to capture changes and to install the appropriate mechanisms to make execution aware of scopes (see Section 4.2).

A *work session* manages a sequence of changes being produced by a developer with a Changebox-aware development tool. While a developer is modifying the system within a particular work session, the work session keeps track of the most recent Changebox in effect. A work session, like a Changebox, can be used to define an execution scope. The difference is that the Changebox defines a fixed execution scope, while a work session takes the most recently produced Changebox to define its scope.

A work session is often used as a common execution scope of a set of IDE tools. A change produced in one of the tools is then automatically reflected in the other tools running within the same scope. Until now, the main tools that have been made aware of Changebox are the code browser and its various helpers (to browse senders, implementors, variables and so on), the debugger and the test runner. Additionally, the version control system was adapted to be able to load code from and into specific work sessions. A work session browser manages the various work sessions present in the system.

As each new change is captured, a change specification is generated. Each change specification encapsulates the process rather than the result of the change. Change specifications are generated by comparing the resulting `Class` or `CompiledMethod` instances of a change to their previous version.

The advantage of encapsulating the change process is that the semantics of a change can be preserved at a more finely grained level. For example, renaming a field in Smalltalk results in a change of its class and yields a new `Class` instance. A change specification to rename a field element, however, captures the intent of renaming the field (rather than redefining a whole class) by only encapsulating the field element and its new name.

This *bottom-up* approach captures all structural changes at a fine grain of detail independent of their high level intent.

**Capturing higher-level changes.** The approach to capturing changes described above captures each *single* change separately. In the case of changes performed by a refactoring tool, however, a single high-level change may entail a large number of individual finer-grained changes. A simple example is renaming a method. This change results in the removal of the method and its re-compilation under a new name.

As Changeboxes are to capture the semantics of changes as precisely as possible, we complement the bottom-up approach with a *top-down* mechanism to capture changes. The top-down mechanism hooks into the IDE tools and captures a change at the higher level where the change semantics are expressed. Tools can directly instantiate change specifications and pass them to the current work session.

For example in the case of a method rename, a `RenameSpecification` operating on a `MethodElement` can be created, specifying the new method name. As a result, only one Changebox is created to encapsulate this specification. Apart from the advantage that there is only one Changebox generated, this strategy also allows for capturing refactorings which may have different effects depending on the system snapshot they are applied in. This is possible because the purpose rather than the effect is captured at a more abstract level. A method rename specification, for example, can then be applied to different snapshots to appropriately update the senders of the renamed method (*e.g.*,, library refactorings applied to different client applications).

## 4.2 Scoping execution

Processes are always executed within the scope of a Changebox. The Changebox in effect is determined by dynamic variable scoping, that is, it is accessed by searching down from the top of the execution stack for the most recent definition. The active Changebox can be set at run-time for a block closure which then executes its code in this new scope.

During execution, each message send and each access to a class reference results in a lookup of the most recent version of the element (as described below).

While the versions of methods and of class references depend on the active Changebox, the version of an object's class is never changed. The consequence is that an object instantiated in the execution scope of one Changebox cannot be moved to the scope of another Changebox in which the shape of its class is different, *i.e.,*, where fields were added or removed.

An early version of our prototype made use of Smalltalk's exception mechanism to search down the stack for the active Changebox. To improve performance we then enhanced the implementation to cache the active Changebox within the executing process. We added an instance variable to the system class `Process` to store the active execution scope. This variable is changed when starting to execute a block for which a new Changebox is defined. On leaving the execution of the block the previous value is restored. This approach avoids expensive stack lookups while perserving the semantics.

We now provide the details of how the standard method lookup and access to class references are modified to reflect the runtime scope.

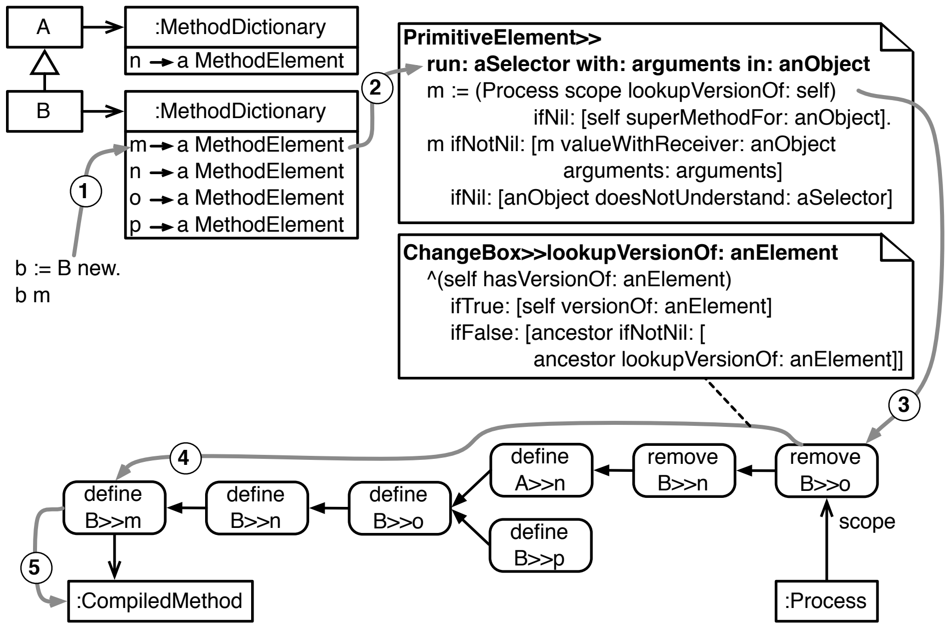
**Method lookup.** Since multiple versions of a given method may be simultaneously active in a Changebox-aware system, in our prototype implementation we modify the method lookup to find the right version in the currently active Changebox.

In Smalltalk, several techniques for message passing control exist [14]. Our implementation is based on method substitution [15], thus making use of the following reflective capability of the VM. The method dictionary of a class associates selector names to compiled methods. If a selector maps to an object which is not an instance of the class `CompiledMethod`, then the VM sends another special message to this object (*i.e.,*, `#run:with:in:`). We use this mechanism to dispatch the message send to the actual version of a method which depends on the active execution scope.

The dispatching process is illustrated by Figure 4. As an example, let's consider the classes `A` and `B` and a set of Changeboxes which define and remove several methods. The message `#m` sent to an instance of class `B` now triggers the following sequence of events.

1. The VM performs a normal method lookup and finds the associated object of selector `#m` in the method dictionary of `B`. The object is a method element for `B»m`.
2. Since the associated object is not of type `CompiledMethod`, the VM sends the message `#run:with:in:` to this object (the method selector `#m`, an empty collection for the arguments, and the original receiver instance `b` are passed as arguments).
3. The method element gets the active scope from the currently executing process. Within this execution scope a lookup of the Changebox which last changed the element is started.
4. The matching Changebox is the one that has a definition change specification for the method element `B»m`.

5. This Changebox then returns the actual compiled method, which is eventually executed on the instance of `B`.



```
PrimitiveElement>>
  run: aSelector with: arguments in: anObject
  m := (Process scope lookupVersionOf: self)
          ifNil: [self superMethodFor: anObject].
  m ifNotNil: [m valueWithReceiver: anObject
               arguments: arguments]
    ifNil: [anObject doesNotUnderstand: aSelector]
```

```
ChangeBox>>lookupVersionOf: anElement
  ^(self hasVersionOf: anElement)
    ifTrue: [self versionOf: anElement]
    ifFalse: [ancestor ifNotNil: [
              ancestor lookupVersionOf: anElement]]
```

**Fig. 4.** Scoped Method Lookup.

If in step 4 no Changebox was found or if the method was not found, the lookup would continue in the superclass. The message `#n` sent to an instance of `B` is an example of this case. Since method `#n` was deleted from `B` in the second last Changebox, the method `#lookupVersionOf:` returns nil, which forces a second lookup in the superclass `A`. This lookup of the method element `A»n` then succeeds.

If the superclass is nil a `DoesNotUnderstand` exception is raised, simulating the normal behavior of the VM. This case is exemplified by sending the message `#o` to an instance of `B`. Since the element `B»o` was removed and there is no definition in any superclass, the message `#doesNotUnderstand:` is sent to the instance of `B`.

Sending the message `#p` results in an exception as well because the method for `#p` is defined in a different branch and cannot be found in the current execution scope.

To improve performance, in each Changebox all previous lookup results are cached. On each subsequent request for the same element in the same execution scope, the value from the cache can be returned directly. This leads to a dispatching overhead which does not depend on the number of Changeboxes (*i.e.*,, number of versions in the system). As with a compilation phase, the caches for a specific execution scope can be filled up front. Benchmarks are discussed in Section 5.

**Class reference resolution.** In Smalltalk, class references in the source code are resolved at compile time and are stored directly in the `CompiledMethod` instance. Whenever a class is rebuilt, all references to this class are updated by the VM using object identity swapping. Additionally, all classes are stored in the global `SystemDictionary` instance.

The global identity swapping mechanism conflicts with the requirement of having different versions of the same class simultaneously and selecting one depending on the active execution scope. Hence, class references should not be resolved at compile time, but rather at runtime.

Our implementation solves this problem by modifying the compiler. The bytecode output for class references is changed to perform a lookup instead of pushing the class reference from the literal frame of the method onto the stack. With this level of indirection the reference to the appropriate instance of the class can be obtained by sending a message to the global `SystemDictionary` instance which then returns the appropriate version of the class.

Furthermore, the system dictionary is modified to store elements instead of classes in case they are under control of Changeboxes. Those elements then determine the actual version of the class using the same lookup mechanism as for the method lookup discussed above.

### 4.3 Merging

A key element of Changeboxes is that different strategies for merging system snapshots can be provided, depending on the application domain.

Several Changeboxes can be joined together, combining all the changes they contain. This may lead to various conflicts about which version of an element to use in the merged snapshot. A conflict occurs when each of two merged branches contains a Changebox with a different change specification but for the same element. Additionally, removal specifications for certain elements can conflict with specifications for dependent elements. For example, the removal of a class conflicts with the definition of a new method for that class.

There is no universal strategy for conflict resolution that would be appropriate for all possible applications of Changeboxes. Various approaches for conflict resolution in merge processes have been proposed for different application domains. For example:

– A wide variety of popular software revision control systems such as RCS [16], CVS [17] or Subversion [18] work with line-based three-way merging. These tools do not consider any semantical information [19], neither of the versions to be merged nor of the changes that led to these versions. Furthermore, they are entirely text-based and therefore do not take any objects into account.

– A different approach is operation-based merging, which is designed for object systems and works on the information about the changes performed on the objects to be merged [20–22]. Operation-based merging works on arbitrary object types and allows one to detect semantical conflicts unrecognized by textual algorithms [19].
– Finally, depending on the application, very specific conflict resolution strategies can be demanded as well. Maybe the specifications from one branch should always overwrite the ones from the other, or the change that was performed later in time should be chosen.

Since the merge strategy to be used will depend on the application domain, we have kept the implementation of Changeboxes open to any possible algorithm. Different algorithms can be implemented by subclassing `MergeStrategy` (see Figure 3). Changeboxes that merge two or more ancestors delegate the conflict resolution to their instance of merge strategy. A merge strategy also orders the selected specifications following the concept of operation-based merging taking the dependencies between specifications into account. For example, class definitions are applied before the definitions of their methods.

As a proof of concept we implemented three different merge strategies. Two simple ones which run without user interaction; one based on the specification order of the system snapshots to merge, the other based on the point in time a change was performed. A third, more sophisticated strategy asks the user for interaction by presenting a list of conflicting changes to be resolved manually.

## 5 Performance evaluation

Although the Changebox model offers attractive possibilities for managing software evolution, the runtime cost must be acceptable or it will never be used in practice. The proof-of-concept prototype presented in the previous section adopted a very straightforward implementation strategy based on the existing Smalltalk runtime architecture. The only optimization adopted is to cache lookup results, thus dramatically improving the cost of repeated lookups.

The goal of this section is to investigate further the actual cost of our simple implementation strategy, and thus to identify areas where performance improvements would be needed for a practical implementation of Changeboxes.

We report on two kinds of benchmarks. First we evaluate the runtime overhead of two different kinds of real application by comparing runtime in plain Smalltalk versus Changebox-aware Smalltalk. Second, we carry out micro benchmarks to identify more precisely the overhead of method lookups and class reference resolution with two different implementation strategies.

### 5.1 Benchmarking Real Applications

We benchmarked[2] two applications, Hessian and Pier. In each case we loaded all available versions of these applications into a Changebox-aware Smalltalk image,

[2] All benchmarks were performed on an Apple MacBook Pro, 2 GHz Intel Core Duo

thus generating a large sequence of Changeboxes representing the development trail of the applications. We then compared the performance of these applications in the Changebox-aware Smalltalk against that in a regular image.

**Hessian.** Hessian is a binary web service protocol application with 28 classes and 468 methods. Hessian comes with 107 test cases, which we used to compare performance with and without Changeboxes. In a plain Smalltalk image, these tests run in 3.85 seconds.

The source code repository contained 13 versions for this project, which generated a total of 570 Changeboxes. In order to evaluate the cost of lookup through different numbers of Changeboxes, we also evaluated the performance of several different versions of Hessian with additional, dummy Changeboxes introduced in between the Changeboxes representing the real changes to the application. The results of the experiment are shown in Table 1.

| **Number of Changeboxes** | **1st execution** | | **2nd execution** | |
|---|---|---|---|---|
| | **Time** | **Ratio** | **Time** | **Ratio** |
| 570 | 4.33 s | 1.12 | 3.96 s | 1.03 |
| 29070 | 67.73 s | 17.57 | 4.09 s | 1.06 |
| 57570 | 112.13 s | 29.09 | 3.99 s | 1.03 |
| 86070 | 173.12 s | 44.92 | 4.02 s | 1.04 |

**Table 1.** Hessian: runtime in seconds of 107 tests with 570 Changeboxes and artificially added Changeboxes. Runtime without Changeboxes: 3.85 s.

The first row shows the values for the original 570 Changeboxes. The first time the tests are run they take 4.33 seconds, representing an overhead of 12% (*i.e.*,, a ratio of 1.12 compared to the time they take in an unaltered image). The second time the tests are run, the message and class lookup results have been cached, so the overhead drops to just 3% (*i.e.*,, a ratio of 1.03).

In the subsequent rows 50, 100 and 150 dummy Changeboxes have been inserted in between each connected pair, respectively. When we add large numbers of dummy Changeboxes, the ancestry becomes larger, and the lookup has to go deeper and thus takes a longer time to complete. The time increases linearly with the number of Changeboxes in the ancestry and becomes up to 45 times slower (*i.e.*,, 173.12 s) with over 86000 Changeboxes in place. For the second and subsequent runs, however, the overhead is negligible, between 3% and 6%.

**Pier.** The second case study is Pier, the content management system we use as motivating example in Section 2. We loaded 115 versions of Pier into a Changebox-aware Smalltalk image, the most recent of which consisted of 194 classes and 1883 methods. In total, 6283 Changeboxes were created from these versions. As test bed we used 1057 tests, which run in a total of 1.01 seconds in a plain Smalltalk image (average time over 100 runs).

As with the Hessian case study, we inserted dummy Changeboxes to stress test the lookup. The values in Table 2 show similar results as for the Hessian implementation: a linear growth of the running time for the first execution and a constant time once the caches are filled. The overhead for running the Pier tests with a cache is much bigger with a ratio of about 4.9 compared to a system running without Changeboxes.

| **Number of Changeboxes** | **1st execution** | | **2nd execution** | |
|---|---|---|---|---|
| | **Time** | **Ratio** | **Time** | **Ratio** |
| 6283 | 24.28 s | 23.96 | 4.87 s | 4.83 |
| 56547 | 305.40 s | 301.37 | 5.00 s | 4.94 |
| 106811 | 489.63 s | 483.18 | 5.02 s | 4.95 |

**Table 2.** Pier: runtime in seconds of 1057 tests with 6283 Changeboxes and artificially added intermediate boxes. Runtime without Changeboxes: 1.01 s

As a second performance evaluation we examined how many requests a Pier application server can handle with and without Changeboxes. The benchmark was performed with the tool ApacheBench to request 1000 times in sequence the same page. In a first run we requested a small page, in a second run a large page (see Table 3).

| **Page size** | **without Changeboxes** | **with Changeboxes** | **Ratio** |
|---|---|---|---|
| 8 kB | 10.36 | 4.83 | 2.14 |
| 179 kB | 4.60 | 1.82 | 2.53 |

**Table 3.** Pier: requests/second with and without Changeboxes, average over 1000 requests.

Compared to the unit tests benchmarks (Table 2) the slowdown for this benchmark considerably smaller (*i.e.*,, 2.53 compared to 4.83). The reason is that only part of the execution time is effected by Changeboxes, as the web server and web application framework Pier is based on are not under control by Changeboxes.

**Comparing Hessian and Pier.** The difference in overhead between the two applications is quite striking, but can be easily explained. Only methods and classes belonging to the loaded application are encapsulated in Changeboxes. Those belonging to the Smalltalk base system itself are unaffected, and experience no overhead. The Hessian protocol implementation makes heavy use of standard Smalltalk classes (*e.g.*,, `Array`). As a consequence relatively few messages are sent during the test run to objects affected by Changebox-based lookup.

Pier, on the other hand, is a much larger framework, and the test runs cause a much larger proportion of messages to be sent to objects coming from Pier itself.

As a result, a far greater proportion of message sends must be resolved with the Changebox-based lookup, and the overhead is consequently much higher.

## 5.2 Micro Benchmarks

In a conventional Smalltalk image, message sends are directly performed by the VM, and class references are resolved at compile time, resulting in no run time costs (apart from pushing the class on the stack). In our prototype, both lookups are performed at runtime, without special support from the VM. We performed several micro benchmarks to assess the cost of message sending and class lookups in a Changebox-aware image using two different implementation strategies.

The benchmarks are shown in Table 4. We compared the times for message sending and class lookup (i) in the global scope (*i.e.,*, without Changeboxes), (ii) using the exception handling mechanism for the execution scope lookup (*i.e.,*, dynamic variables), and (iii) with the scope cached in an instance variable of the current Smalltalk Process object.

| **Operation** | **Global Scope** | **Dynamic Variables** | | **Process Variable** | |
|---|---|---|---|---|---|
| | **Time** | **Time** | **Ratio** | **Time** | **Ratio** |
| $10^6$ sends | 100 ms | 14023 ms | 140.0 | 6510 ms | 65.1 |
| $10^6$ lookups | 2030 ms | 13724 ms | 6.8 | 6053 ms | 3.0 |

**Table 4.** Benchmarks for message sending and class lookup, average over three runs.

The micro benchmarks are all done with only one Changebox and with a filled cache (*i.e.,*, corresponding to the second execution) since we wanted to measure the pure overhead of the basic mechanisms used for the message send and class access. As all benchmarks are run in a Changebox-aware Smalltalk, class reference resolution is always performed at run-time. This causes the high class lookup time measured in the global scope.

The major difference between the execution times is not surprising since it merely reflects the difference between executing compiled basic operations and interpreting them. More interesting is the difference between the two implementation strategies. There is a factor of two difference between using dynamic variables and the Process instance variable.

We have further investigated this difference by also varying the stack depth. Table 5 shows the access times for the scope value based on different method context stack depths. This is important because dynamic variables have to search the stack upwards for the definition of their value.

The difference of the two approaches is even more significant when measured separately. Even without intermediate method calls, the access of the `Process` instance variable is 29 times faster than the access of the dynamic variable. With growing method context stacks, dynamic variables get increasingly slower, while the access to the `Process` instance variable remains at a stable value. The results of these experiments led us to the decision to use the Process instance variable for the prototype implementation of Changeboxes.

| **Scope access** | **Dynamic Variables** | **Process Variable** |
|---|---:|---:|
| Stack depth 0 | 7455 ms | 258 ms |
| Stack depth 100 | 7967 ms | 258 ms |
| Stack depth 1000 | 17883 ms | 255 ms |

**Table 5.** Benchmarks for scope access ($10^6$ times each), average over three runs.

As shown in Table 4, $10^6$ message sends take 6510 ms to run with the Process instance variable. How expensive are the different parts of the modified method lookup as described in Section 4.2? We split up the lookup into the following three measurable steps:

1. Dispatch the message send on the object in the method dictionary.
2. Determine the scope with Process instance variable strategy.
3. Lookup the version of the method in the scope and execute it.

The first step only takes a fraction of the total time with 146 ms (2.2%). The scope lookup in the second step takes 258 ms (4.0%) as shown in Table 5. The last step uses the most time with 6106 ms (93.8%).

The first step is fast because it uses a reflective capability of the VM. The second step only accesses instance variables and is not very time consuming as well. The third step, in contrast, is much slower because it consists of many message sends including a dictionary lookup implemented in Smalltalk. We discuss possible ways to further improve performance in Section 5.

## 6 Discussion

First, we consider possible directions for applying Changeboxes, and then we discuss some of the problems and challenges for developing Changeboxes into a truly effective tool for managing software evolution.

We have already seen how Changeboxes can be used to manage multiple development branches of an application within a single running application. A Changebox can therefore package bug fixes and refactorings as well as new features. Furthermore, Changeboxes can be dynamically deployed, thus avoiding the need to shut down and restart the application. In some cases, non-invasive changes can be applied to running sessions, thus affecting the behaviour of existing objects, not just newly created ones. As such, Changeboxes are intended to serve as a general-purpose mechanism for controlling the scope of software evolution.

Another application for Changeboxes would be to package extension to third-party software. Class extensions, for example, add or override methods to existing classes. A potentially serious problem with class extensions is that the newly defined behaviour may break existing clients. A Changebox tackles this by providing a well-defined execution scope within which these extensions are visible. (See also the discussion of Classboxes in Section 7.)

A Changebox does not simply specify a set of software elements, but rather how some existing software elements from an existing system snapshot are *changed* to yield a new snapshot. How exactly these changes are specified and how sets of changes are merged to resolve eventual conflicts will depend on the application domain. One particularly interesting track is to consider change specifications as general *transformations* which may be applied to a variety of software elements. By capturing high-level changes directly in the tools that produce them, we are able to to encapsulate general, reusable transformations rather than low-level edits. With appropriate tool support, such change specifications could be composed, merged and replayed to yield new system snapshots. This can be used for example to ease framework refactoring: all refactoring transformations would be stored as high-level change specifications. Clients that want to migrate to a new version can use these change specifications to transform their own code.

In the current implementation, the Changebox which provides the execution scope for method lookup is set explicitly. This may be done deliberately by a developer, as outlined in the scenario of Section 2. Alternatively, the current Changebox could be set implicitly depending on properties of the current runtime context. Such a mechanism would enable applications to adapt their behaviour at runtime. This would be useful in particular for implementing context-aware, mobile applications, but it would also be interesting for other domains. Consider, for example, collaborative work applications where the same objects may exhibit different behaviour or features depending on who is interacting with them. For each user there would be a Changebox providing those features that should be in effect when that person is using the system [23].

The current implementation is performant enough to serve as a proof-of-concept prototype, and to offer a platform for realistic experiments. A real implementation, however, would need to address several problems:

- To improve performance to meet the requirements for real usage and for a system in which also library classes are under control by Changeboxes would require a scope-aware execution at the VM level.
- The current implementation keeps all Changeboxes in memory, even though they may not all be used. Having a policy to transparently swap-out unused Changeboxes and flush lookup caches would heavily reduce the memory footprint.
- Currently, the class of an object remains that from which it was instantiated in its execution scope. Changeboxes do not offer any mechanism for migrating to new internal representations or new classes. Such support would be necessary to fully enable dynamic adaptation of running objects as they are accessed from the execution scope of different Changeboxes.

One problem which we face with the Smalltalk implementation is that the notion of *execution scope* is not explicit, so it must be simulated by modifying the method lookup. An implementation based on an explicit and efficient scoping mechanism would lead to a more natural implementation strategy. The performance analysis (Section 5) has shown that most time is spent in image level Smalltalk code. We plan to experiment with moving the lookup logic into the VM. This alone should provide a substantial improvement in performance. Other ideas to explore are more elaborate caching schemes, for example one cache per send side that takes the current execution scope into account, similar to a *Polymorphic Inline Cache* (PIC) [24].

Current versioning systems support collaboration by allowing a programmer to checkout a snapshot from the server, perform changes locally and then commit them. In agile environments, programmers are suggested to perform frequent commits to stay as much in contact with the overall development as possible. In a Changebox-aware system, we can envision a server which is the development platform for all developers, and eventually even the deployment platform. In such a system, programmers transparently and concurrently perform changes. They can effectively see who is changing the system when and where, and can choose to integrate at any point.

The above example would pose a challenge to the way we should navigate and even think about Changebox-based systems. We currently have difficulty managing large information spaces — the space to navigate becomes much larger if we have access to every possible version of software elements. Changeboxes should make it easier rather than more difficulty to navigate this space, but we are currently lacking good metaphors for representing and navigating the version space.

Having the entire history of changes provides us with new kinds of data for understanding how systems evolve. For example, currently, a large body of work is invested into recovering the meaning of the past changes (*e.g.,*, refactorings). However, given that the meaning of a change is encapsulated in Changeboxes we can concentrate more effort on understanding why the changes have happened. We can encapsulate in Changeboxes the actions of a programmer to create the changes (*e.g.,*, clicks, menu activations, keys pressed). Such information can reveal which tools are used for development tasks and how they are used.

## 7 Related work

Changeboxes superficially resemble Classboxes [25, 9, 26], a module system that provides scoped access to class extensions. Within the scope of a Classbox, new classes may be defined, or classes may be imported from other Classboxes and extended with new or overridden methods. Extensions are only visible from within a Classbox, or another Classbox that imports classes from it. As a consequence, within a single running system, different versions of the same class may be active. Classboxes only support addition and overriding of methods; they do not support removal and thus cannot model changes. Classboxes also do not support high-level change specifications — only new method definitions. They also do not support any general merging operations — method extension simply override existing methods of the same name.

Virtual classes [27, 28] allow class names to be looked up dynamically. Virtuality of classes, however, is associated with a hierarchy of encapsulating entities, rather than with a particular version of the system as it evolves.

Piccola [29] is a language for specifying applications as compositions of software components. The key mechanism in Piccola is the notion of a first-class namespace (or *form*) which is used to encapsulate the services of a component [30]. Forms also serve as the execution context for scripts. In particular, within a single running application, different execution contexts can be simultaneously active. Like Changeboxes, forms are immutable. Piccola does not provide any special support for encapsulating or merging changes.

PIE [31–34] was an experiment to extend the Smalltalk object model with the notion of *views* coming from frame-based languages. PIE is implemented in itself, and therefore, the code does not consist of regular Smalltalk classes, but of PIE *nodes*. PIE provides code with multiple views, *i.e.,*, representing design decisions from the perpectives of different developers. PIE does not support the possibility of multiple views to be simultaneously active. Before execution, code is flattended to regular Smalltalk classes.

ContextL [35] is a language to support Context-Oriented Programming (COP). The language provides a notion of *layers*, which package context-dependent behavioural variations. In practice, the variations consist of method definitions, mixins and *before* and *after* specifications. Layers are dynamically enabled or disabled based on the current execution context. ContextL does not support a more general notion of change specification.

Us [36] is a system based on Self that supports subjective programming. Message lookup depends not only on the receiver of a message, but also on a second object, called the *perspective*. The perspective allows for layer activation similar to ContexL and does not provide a first-class representation of change.

Aspect-Oriented Programming (AOP) [37] provides a general model for modularising cross cutting concerns. *Join points* define points in the execution of a program that trigger the execution of additional cross-cutting code called *advice*. Join points can be defined on the runtime model (*i.e.,*, dependent on control flow). Although AOP is used to effect changes to software systems, the focus is on cross-cutting concerns, rather than on software changes in general. The availability of control flow based pointcuts enables different executions to execute different code, but it is normally not used to express versioning.

Gemstone [38] provides the concept of class versions. Classes are automatically versioned, but existing instances keep the class (shape and behavior) of the original definition. Instances can be migrated at any time. Gemstone provides (database) transaction semantics, thus state can be rolled back should the migration fail.

In Java, new class definitions can be loaded using a class loader [39]. Class loaders define namespaces, a class type is defined by the name of the class and it's class loader. Thus the type system will prohibit references between namespaces defined by two different loaders. Class loaders can be used to load new versions of code and allow for these versions to coexist at runtime, but they do not provide a first-class model of change.

Dynamic software updating has seen a lot of research over the last years [40–43]. The focus here is updating a system at runtime, both code and data. All of these systems are not concerned with providing a first-class model of change.

In Erlang [44, 45] two different versions of the same software artifact can be active at the same time. When code is loaded in the running system, it retains both the old and new version. Calling conventions define which code is called. This allows for a module to continue to execute old code until it is restarted. There are at most two versions active at any time. If a third version is loaded, all processes executing the oldest code are killed. Erlang focuses on on providing a robust model for dynamic code loading. It does not try to model change.

CLOS [46] provides a protocol for changing class definitions at runtime: whenever a class definition is changed, all existing instances are prepared to be updated to the new version when they are accessed subsequently. The process of updating can be customized by the programmer. In both systems, the focus is not on encapsulating change, nor on providing multiple execution contexts within the same running application.

DOORS and its Smalltalk prototype [47] enable dynamic object evolution. Depending on a condition, objects can be altered or extended at runtime. The usefulness of this feature is shown for modeling domain objects, it does not provide a general model of change.

## 8 Concluding remarks

Changeboxes offer a simple and uniform mechanism for encapsulating change specifications. They provide a consistent execution scope for running applications, which means that different versions of the same software elements can be simultaneously active within one software system. Changeboxes are immutable, so they can be safely combined and merged to form new Changeboxes without affecting existing ones.

Changeboxes have many potential applications for managing software evolution. Our prototype implementation illustrates how bug fixes, new features and refactorings can be safely integrated into a running system without impacting active sessions. Although the prototype is intended only as a proof-of-concept for demonstration purposes, its performance is more than adequate to illustrate the potential benefits of the approach.

Future directions include:

- developing a cleaner and more efficient implementation approach based on first-class execution scopes,
- providing support for expressing higher-level, composable change specifications,
- developing a broader spectrum of merging operations for changes,
- support for migrating the representation of running objects,
- mechanisms to support context-aware applications by automatically enabling Changeboxes based on properties of the current context,

– experimentation with new metaphors for developers to navigate the space of changes.

**Acknowledgments.** We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Analyzing, capturing and taming software change" (SNF Project No. 200020-113342, Oct. 2006 - Sept. 2008). We would also like to express our thanks to Alexandre Bergel, Pascal Costanza, Stéphane Ducasse, Orla Greevy and Roel Wuyts for their help in reviewing various drafts of this paper.

## References

1. Lehman, M., Belady, L.: Program Evolution: Processes of Software Change. London Academic Press, London (1985)
2. Nierstrasz, O., Bergel, A., Denker, M., Ducasse, S., Gaelli, M., Wuyts, R.: On the revival of dynamic languages. In Gschwind, T., Aßmann, U., eds.: Proceedings of Software Composition 2005. Volume 3628., LNCS 3628 (2005) 1–13 Invited paper.
3. Nierstrasz, O., Denker, M., Gîrba, T., Lienhard, A.: Analyzing, capturing and taming software change. In: Proceedings of the Workshop on Revival of Dynamic Languages (co-located with ECOOP'06). (July 2006)
4. Steyaert, P., Lucas, C., Mens, K., D'Hondt, T.: Reuse Contracts: Managing the Evolution of Reusable Assets. In: Proceedings of OOPSLA '96 (International Conference on Object-Oriented Programming, Systems, Languages, and Applications), ACM Press (1996) 268–285
5. Nguyen, T., Munson, E., Boyland, J.: An infrastructure for development of object-oriented, multi-level configuration management services. In: Internationl Conference on Software Engineering (ICSE 2005), ACM Press (2005) 215–224
6. Hearnden, D., Lawley, M., Raymond, K.: Incremental model transformation for the evolution of model-driven systems. In: International Conference on Model Driven Engineering Languages and Systems (Models/UML 2006). Volume 4199 of LNCS., Berlin, Germany, Springer-Verlag (2006) 321–335
7. Dig, D., Johnson, R.: The role of refactorings in API evolution. In: Proceedings of 21st International Conference on Software Maintenance (ICSM 2005). (September 2005) 389–398
8. Dig, D., Johnson, R.: How do APIs evolve? a story of refactoring. Journal of Software Maintenance and Evolution: Research and Practice (JSME) **18**(2) (April 2006) 83–107
9. Bergel, A., Ducasse, S., Nierstrasz, O., Wuyts, R.: Classboxes: Controlling visibility of class extensions. Computer Languages, Systems and Structures **31**(3-4) (December 2005) 107–126
10. Aldrich, J.: Open modules: Modular reasoning about advice. In: Proceedings ECOOP 2005. Volume 3586 of LNCS., Glasgow, UK, Springer Verlag (July 2005) 144–168
11. Renggli, L.: Magritte – meta-described web application development. Master's thesis, University of Bern (June 2006)
12. Ducasse, S., Lienhard, A., Renggli, L.: Seaside — a multiple control flow web application framework. In: Proceedings of 12th International Smalltalk Conference (ISC'04). (September 2004) 231–257

13. Zumkehr, P.: Changeboxes — modeling change as a first-class entity. Master's thesis, University of Bern (February 2007)
14. Ducasse, S.: Evaluating message passing control techniques in Smalltalk. Journal of Object-Oriented Programming (JOOP) **12**(6) (June 1999) 39–44
15. Bergel, A., Denker, M.: Prototyping languages, related constructs and tools with Squeak. In: Proceedings of the Workshop on Revival of Dynamic Languages (co-located with ECOOP'06). (July 2006)
16. Tichy, W.F.: RCS — a system for version control. Software Practice and Experience **15**(7) (July 1985) 637–654
17. Berlin, L.M.: When objects collide: Experiences with reusing multiple class hierarchies. In: Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices. Volume 25. (October 1990) 181–193
18. Collins-Sussman, B., Fitzpatrick, B.W., Pilato, C.M.: Version Control with Subversion. O'Reilly & Associates, Inc. (2004)
19. Mens, T.: A state-of-the-art survey on software merging. IEEE Transactions on Software Engineering **28**(5) (May 2002) 449–462
20. Feather, M.S.: Detecting interference when merging specification evolutions. In: Proc. Fifth International Workshop on Software Specification and Design. (1989) 169–176
21. Lippe, E., van Oosterom, N.: Operation-based merging. In: SDE 5: Proceedings of the fifth ACM SIGSOFT symposium on Software development environments, New York, NY, USA, ACM Press (1992) 78–87
22. Munson, J.P., Dewan, P.: A flexible object merging framework. Proceedings of the 1994 ACM conference on Computer supported cooperative work (1994) 231–242
23. Göker, A., Myrhaug, H.I.: User context and personalisation. In: ECCBR Workshop on Case Based Reasoning and Personalisation, Aberdeen, UK (2002) invited paper.
24. Hölzle, U., Chambers, C., Ungar, D.: Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In America, P., ed.: Proceedings ECOOP '91. Volume 512 of LNCS., Geneva, Switzerland, Springer-Verlag (July 1991) 21–38
25. Bergel, A.: Classboxes — Controlling Visibility of Class Extensions. PhD thesis, University of Berne (November 2005)
26. Bergel, A., Ducasse, S., Wuyts, R.: Classboxes: A minimal module model supporting local rebinding. In: Proceedings of Joint Modular Languages Conference (JMLC'03). Volume 2789 of LNCS., Springer-Verlag (2003) 122–131
27. Ernst, E.: Propagating class and method combination. In Guerraoui, R., ed.: Proceedings ECOOP '99. Volume 1628 of LNCS., Lisbon, Portugal, Springer-Verlag (June 1999) 67–91
28. Mezini, M., Ostermann, K.: Conquering aspects with Caesar. In: Proceedings of the 2nd international conference on Aspect-oriented software development, ACM Press (2003) 90–99
29. Achermann, F., Nierstrasz, O.: Applications = components + scripts — a tour of Piccola. In Aksit, M., ed.: Software Architectures and Component Technology. Kluwer (2001) 261–292
30. Achermann, F., Nierstrasz, O.: Explicit namespaces. In Gutknecht, J., Weck, W., eds.: Modular Programming Languages, Proceedings of JMLC 2000 (Joint Modular Languages Conference). Volume 1897 of LNCS., Zürich, Switzerland, Springer-Verlag (September 2000) 77–89
31. Bobrow, D.G., Goldstein, I.P.: Representing design alternatives. In: Proceedings of the Conference on Artificial Intelligence and the Simulation of Behavior. (July 1980)

32. Goldstein, I.P., Bobrow, D.G.: Descriptions for a programming environment. In: Proceedings of the First Annual Conference of the National Association for Artificial Intelligence. (August 1980)
33. Goldstein, I.P., Bobrow, D.G.: Extending object-oriented programming in Smalltalk. In: Proceedings of the Lisp Conference. (August 1980) 75–81
34. Goldstein, I.P., Bobrow, D.G.: A layered approach to software design. Technical Report CSL-80-5, Xerox PARC (December 1980)
35. Costanza, P., Hirschfeld, R.: Language constructs for context-oriented programming: An overview of ContextL. In: Proceedings of the Dynamic Languages Symposium (DLS) '05, co-organized with OOPSLA'05, New York, NY, USA, ACM Press (October 2005)
36. Smith, R.B., Ungar, D.: A simple and unifying approach to subjective objects. TAPOS special issue on Subjectivity in Object-Oriented Systems **2**(3) (1996) 161–178
37. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-Oriented Programming. In Aksit, M., Matsuoka, S., eds.: Proceedings ECOOP '97. Volume 1241 of LNCS., Jyvaskyla, Finland, Springer-Verlag (June 1997) 220–242
38. Otis, A., Butterworth, P., Stein, J.: The GemStone object database management systems. Communications of the ACM **34**(10) (October 1991) 64–77
39. Liang, S., Bracha, G.: Dynamic class loading in the Java virtual machine. In: Proceedings of OOPSLA '98, ACM SIGPLAN Notices. (1998) 36–44
40. Duggan, D.: Type-based hot swapping of running modules. In: Intl. Conf. on Functional Programming. (2001) 62–73
41. Hicks, M., Nettles, S.: Dynamic software updating. ACM Transactions on Programming Languages and Systems **27**(6) (nov 2005) 1049–1096
42. Oriol, M.: An Approach to the Dynamic Evolution of Software Systems. Ph.D. thesis, Centre Universitaire d'Informatique, University of Geneva (April 2004)
43. Kramer, J., Magee, J., Finkelstein, A.: A constructive approach to the design of distributed systems. In: Proc 10th Intl Conf on Distributed Computing Systems. IEEE (June 1990) 580–587
44. Armstrong, J., Virding, R., Wikström, C., Williams, M.: Concurrent Programming in Erlang. Prentice Hall (1996)
45. Armstrong, J.: Making reliable distributed systems in the presence of software errors. PhD thesis, The Royal Institute of Technology Stockholm (2003)
46. Bobrow, D., DeMichiel, L., Gabriel, R., Keene, S., Kiczales, G., Moon, D.: Common lisp object system specification, x3j13. Technical Report 88-003, (ANSI COMMON LISP) (1988)
47. Mezini, M.: Dynamic object evolution without name collisions. In: Proceedings ECOOP '97, Springer-Verlag (June 1997) 190–219

# Redesigning with Traits: the Nile Stream trait-based Library

Damien Cassou[1], Stéphane Ducasse[1], and Roel Wuyts[2]

[1] LISTIC, University of Savoie, France
[2] IMEC, Leuven and Université Libre de Bruxelles

**Abstract.** Recently, traits have been proposed as a single inheritance backward compatible solution in which the composing entity has the control over the trait composition. Traits are fine-grained units used to compose classes, while avoiding many of the problems of multiple inheritance and mixin-based approaches.
To evaluate the expressiveness of traits, some hierarchies were *refactored*, showing code reuse. However, such large refactorings, while valuable, may not be facing all the problems, since the hierarchies were previously expressed within single inheritance and following certain patterns. We wanted to evaluate how traits enable reuse, and what problems could be encountered when building a library using traits *from scratch*, taking into account that traits are units of reuse. This paper presents our work on designing a new stream library named Nile. We present the reuse that we attained using traits, and the problems we encountered.
**Keywords.** Object-Oriented Programming, Inheritance, Refactoring, Traits, Code Reuse, Smalltalk

## 1 Introduction

Multiple inheritance has been the focus of a large amount of work and research efforts. Recently, traits proposed a solution in which the composite entity has the control and which can be flattened away, *i.e.,* traits do not affect the runtime semantics [1, 2]. Traits are fine-grained units that can be used to compose classes. Like any solution to multiple inheritance, the design of traits is the result of a set of trade-offs. Traits favor simplicity and fine-grained composition. Traits are meant for single inheritance languages. Trait composition conflicts are automatically detected but the composer has the control to resolve these conflicts explicitly. Traits claim to avoid many of the problems of multiple inheritance and mixin-based approaches that mainly favor linearization where conflicts never arise explicitly and are solved implicitly by ordering.

Note that there exist different trait models. In the original trait model, *Stateless traits* [1, 2], traits only define methods, but not instance variables. *Stateful traits* [3] extends this model and lets traits also define state. *Freezable traits* [4] extend stateless traits with a visibility mechanism. In the context of this paper when we use *trait* we mean *Stateless trait.*

Previous research evaluated the usefulness of traits by refactoring the Smalltalk collection and stream libraries, which showed up to 12% gain in terms of code reuse [5]. Other research tried to semi-automatically identify traits in existing libraries [6]. While these are valuable results, they are all refactoring scenarios that investigated the applicability of traits using *existing* systems as input. Usability and reuse of traits when developing a *new* system has not been assessed. Implementing a stream library from scratch is an important experience to test the expressiveness of traits. By doing so we may face problems that may have been hidden in previous experiences and also face a large scheme of trait composition problems.

The goal of this paper is to experimentally verify the original claims of simplicity and reuse of traits in the context of a forward engineering scenario. More specifically, our experiments want to get answers to the following questions that quickly arise when using traits in practice:

– Trait granularity. We want to assess the granularity of traits that maximize their reusability and composition.
– Trait reusability. We want to understand how much code can be reused.
– Can we define traits as composable building units?
– Can we identify guideline to assess when trait composition should be preferred over inheritance?
– To what extent can we fix the problems identified in the current stream hierarchy?
– What trait limits and problems do we encounter?
– Does the use of trait imply an execution cost?

Our approach is based on designing and implementing a non-trivial library from scratch using traits. We decided to build a stream collection library (called *Nile*) that follows the ANSI Smalltalk standard [7] yet remains compatible with the current Smalltalk implementations. The choice for a stream library was motivated by a number of reasons:

– streams exhibit problems linked to the fact that they are naturally modeled using multiple inheritance. In presence of single inheritance the implementors are reduced to duplicated code and other tricks such as canceling methods;
– N. Schärli [5] and A. Lienhard [6] already refactored the Stream library using traits so we can compare with their results;
– streams are an important abstraction of computer language libraries;
– several constraints are imposed by the ANSI Smalltalk standard and the need to remain usable in existing Smalltalk dialects.

Nile is structured around three core traits and a set of libraries. During the definition of the libraries, the core traits proved to have a good granularity: it was easy to obtain each desired functionality composition using the adequate part of the core. Nile has 18% less methods and 15% less bytecodes than the corresponding Squeak collection-based stream library. Moreover, Nile has neither canceled method nor method implemented too high in the hierarchy. There are only three overrides compared to the fourteen of Squeak.

The contributions of the paper are: (1) the design of *Nile*, a new stream library made of composable units, (2) the assessment that traits are good building units for defining libraries and that they enable clean design and reuse through composibility, and (3) the identification of problems when using the traits.

We start by presenting the existing Squeak Stream hierarchy limits and the ANSI Smalltalk standard protocols (Section 2). Section 3 presents an overview of Nile and the core of the library around its three most important traits. Section 4 and Section 5 detail the implementation of the collection-based and file-based stream libraries, respectively. Two other libraries will be presented in Section 6. Section 7 compares our approach with the one of N. Schärli [5]. It analyses the reuse offered by traits as well as performance issues and optimization solutions. Finally, Section 8 presents the problems we identify due to the use of traits.

## 2 Analyses

In this section, we analyze the existing stream hierarchy of Squeak the open-source Smalltalk [8]. We highlight the key problems and present the ANSI Smalltalk standard.

### 2.1 Analysis of the Squeak stream hierarchy

Squeak [8], like all Smalltalk environments, has its own implementation of a stream hierarchy. Figure 1 presents the core of this implementation, which is solely based on single inheritance and does not use traits. Note that most Smalltalk dialects reimplemented streams and therefore have similar yet different implementation. For example, even though Squeak and VisualWorks are both direct descendants from the original Smalltalk-80, their stream hierarchies are different since the one in VisualWorks was completely reimplemented.

The existing single-inheritance implementation has different problems that we detail.

*Methods implemented too high in the hierarchy.* A common technique to avoid duplicating code consists in implementing a method in the topmost common superclass of all classes which need this method. Even if efficient, this technique pollutes the interface of classes which do not want this method. For example, Stream defines nextPutAll: which calls nextPut:

```
Stream>>nextPutAll: aCollection
  "Append the elements of aCollection to the sequence of objects
   accessible by the receiver. Answer aCollection."

  aCollection do: [:v | self nextPut: v].
  ^ aCollection.
```

The method nextPutAll: writes all elements of the parameter aCollection to the stream by iterating over the collection and calling nextPut: for each element.

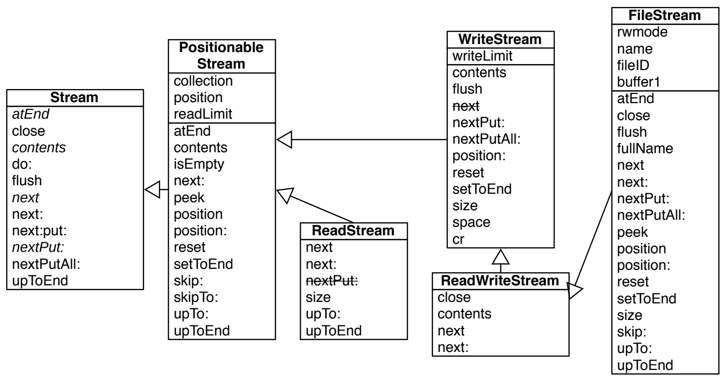


**Fig. 1.** The Squeak core Stream hierarchy. Only the most important methods are shown.

The method nextPut: is abstract and must be implemented in subclasses, and even if Stream defines methods to write to the stream, some subclasses are used for read-only purposes, like ReadStream. Those classes must then cancel explicitly the methods they don't want.[3] This approach, even if it was probably the best available solution at the time of the first implementation, has some drawbacks. First of all the class Stream and its subclasses are polluted with a number of methods that are not available in the end. This complicates the task of understanding the hierarchy and extending it. It also makes it more difficult to add new subclasses. To add a new subclass, a developer must analyze all of the methods implemented in the superclasses and cancel all unwanted ones.

*Unused superclass state.* The class FileStream is a subclass of ReadWriteStream and an indirect subclass of PositionableStream which is explicitly made to stream over collections (see Figure 1). Then, the instance variables collection, position and readLimit inherited from the PositionableStream and writeLimit inherited from WriteStream are not used for FileStream and all its subclasses.

*Simulating multiple inheritance by copying.* ReadWriteStream is conceptually both a ReadStream and a WriteStream. However, Smalltalk is a single inheritance-based language, so ReadWriteStream can only subclass one of these. The behaviour from the other one has to be copied, leading to code duplication and all of its related maintenance problems.

The designers of the Squeak stream hierarchy decided to subclass WriteStream to implement ReadWriteStream, and then copy the methods related to reading from ReadStream.

[3] In Smalltalk, canceling a method is done by reimplementing the method in the subclass and calling shouldNotImplement from it.

One of the copied methods is next, which reads and returns the next element in the stream. This leads to a strange situation where next is being canceled out in WriteStream (because it should not be doing any reading), only to be reintroduced by ReadWriteStream. The reason for this particular situation is due to the combination of next defined too high in the hierarchy and single inheritance.

*Reimplementation.* In Figure 1, one can see that next: is implemented five times. Not a single implementation uses super which means that each class completely reimplements the method logic instead of specializing it. But this statement should be tempered because often in the Squeak stream hierarchy, methods override other methods to improve speed execution: this is because in subclasses, the methods have more knowledge and, thus, can do a faster job. However, a method reimplemented in nearly all of the classes in a hierarchy suggests inheritance hierarchy anomalies.

## 2.2 The ANSI Smalltalk standard

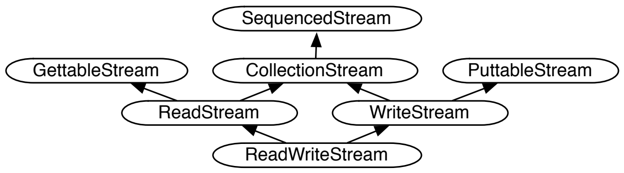


**Fig. 2.** The ANSI Smalltalk standard stream protocol hierarchy.

Figure 2 shows that even if Smalltalk is a single inheritance language, the ANSI Smalltalk standard [7] defines the different protocols using multiple inheritance. In the standard, streams are based on the notion of sequence values. Each stream has past and future sequence values. The ANSI Smalltalk standard defines a decomposition of stream behavior around three main protocols: GettableStream, SequencedStream and PuttableStream. Table 1 and Table 2 summarize the protocol contents.

The ANSI Smalltalk standard provides a useful starting point for an implementation even if a lot of useful methods are not described. We therefore chose to adopt it for Nile.

*About GettableStream>>peekFor:.* The standard proposes a definition of peekFor: that most Smalltalk implementations do not follow. The ANSI Smalltalk standard is equivalent to an equality test between the peeked object and the parameter:

| SequencedStream | |
|---|---|
| close | Disassociate a stream from its backing store. |
| contents | Returns a collection containing the receiver's past and future sequence values in order. |
| isEmpty | Returns a boolean indicating whether there are any sequence values in the receiver. |
| position | Returns the number of sequence values in the receiver's past sequence values. |
| position: | Sets the number of sequence values in the receiver's past sequence values to be the parameter. |
| reset | Resets the position of the receiver to be at the beginning of the stream of values. |
| setToEnd | Set the position of the stream to its end. |

| PuttableStream | |
|---|---|
| flush | Upon return, if the receiver is a write-back stream, the state of the stream backing store must be consistent with the current state of the receiver. |
| nextPut: | Writes the argument to the stream. |
| nextPutAll: | Enumerate the argument, adding each element to the receiver. |

**Table 1.** The SequencedStream and PuttableStream protocols defined by the ANSI Smalltalk standard.

| GettableStream | |
|---|---|
| atEnd | Returns true if and only if the receiver has no future sequence values available for reading. |
| do: | Evaluates the argument with each receiver future sequence value. |
| next | The first object is removed from the receiver's future sequence values and appended to the end of the receiver's past sequence values. The object is returned. |
| next: | Does next a certain amount of time and returns a collection of the objects returned by next. |
| nextMatchFor: | Reads the next object from the stream and returns true if and only if the object is equivalent to the argument. |
| peek | Returns the next object in the receiver's future sequence values without advancing the receiver's position. |
| peekFor: | Peek at the next object in the stream and returns true if and only if it matches the argument. |
| skip: | Skip a given amount of object in he receiver's future sequence values. |
| skipTo: | Sets the stream just after the next occurrence of the argument and returns true if it's found before the end of the stream. |
| upTo: | Returns a collection of all the objects in the receiver up to, but not including the next occurrence of the argument. |

**Table 2.** GettableStream protocol defined by the ANSI Smalltalk standard.

```
GettableStream>>peekFor: anObject
  ^ self peek = anObject
```

Most Smalltalk implementations (including Dolphin, GemStone, Squeak, VisualAge, VisualSmalltalk, VisualWorks, Smalltalk-X and GNU Smalltalk) do not only test the equality but also increment the position in case of equality as shown by the following implementation.

```
peekFor: anObject
  "Answer false and do not move over the next element if it is not equal
   to the  argument, anObject, or if the receiver is at  the end. Answer
   true and  increment the position, if the  next  element  is  equal to
   anObject."

  ^ (self atEnd not and: [self peek = anObject])
       ifTrue: [self next. true]
       ifFalse: [false]
```

This definition lets the following code parse '145', ' 145' and '-145' without problem:

```
stream := ReadStream on: '- 145'.
negative := stream peekFor: $-.
stream peekFor: Character space.
number := stream upToEnd.
```

*Regarding the name of SequencedStream.* The name SequencedStream is not well chosen, since this protocol provides absolute positioning in the stream. A name evoking this would have been better.

## 3 Nile overview and core

Nile is designed around a core of traits offering base functionality reflecting the ANSI Smalltalk standard. The core consists of only three traits and it is then used in several libraries that we discuss in detail throughout the paper. File-based streams and collection-based streams are among the most prominent libraries. Other libraries we discuss are support for writing character constants and decoders (streams that can be chained). Figure 3 presents an overview of Nile.

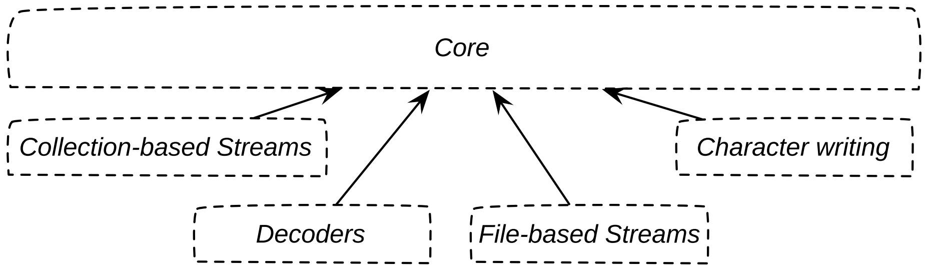


**Fig. 3.** Overview of Nile: the core and its different libraries.

We designed Nile around three independent traits, reflecting the ANSI Smalltalk standard: TPositionableStream, TGettableStream and TPuttableStream. They are shown in Figure 4.

*TGettableStream.* The trait TGettableStream is meant for all streams used to read elements of any kind. The trait requires 4 methods: atEnd, next, peek and outputCollectionClass. The method peek returns the following element without moving the stream whereas next reads and returns the following element and moves the stream. The method TGettableStream>>outputCollectionClass is used to determine the type of collection which is used when returning collection of elements as with next: and upTo:.

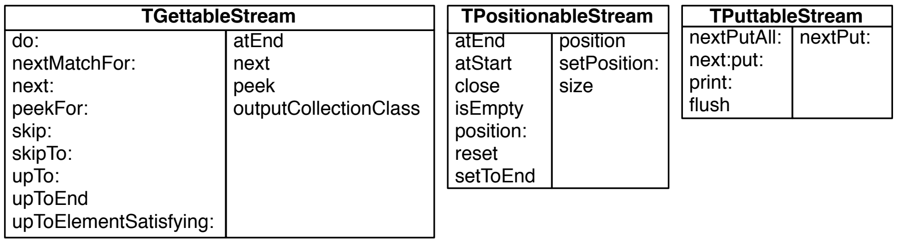


**Fig. 4.** The Nile core traits.

*TPositionableStream.* The trait TPositionableStream allows for the creation of streams that are positioned in absolute manner. It corresponds to the ANSI Smalltalk standard SequencedStream protocol; we thought the name TPositionableStream made more sense. The only required methods are size and two accessors for a position variable. We decided to implement the bound verification of the method position: in the trait itself: the parameter must be between zero and the stream size. This means that two methods have to be implemented: a pure accessor, named setPosition: here, and the real public accessor named position: which verifies its parameter value.

*TPuttableStream.* This trait is the simplest of the Nile library. It provides nextPutAll:, next:put:, print: and flush and requires nextPut:. By default, flush does nothing. It is used for ensuring that everything has been written. Buffer-based streams should have their own implementations.

## 4 Collection-based streams

To support streaming over collections we implemented a set of dedicated traits and what we call *trait factories* that define their creation protocols. Note that, in contrast to the default Squeak implementation and like in VisualWorks, our implementation actually works with any sequenceable collection, not just Arrays and Strings.

### 4.1 The traits

The traits TCollectionStream, TReadableCollectionStream and TWriteableCollectionStream implement the collection-based functionalities (as shown in Figure 5 — Note that in the figures traits have their name in bold whereas classes not). They provide all necessary methods required by the core traits, while only requiring 4 new accessors.

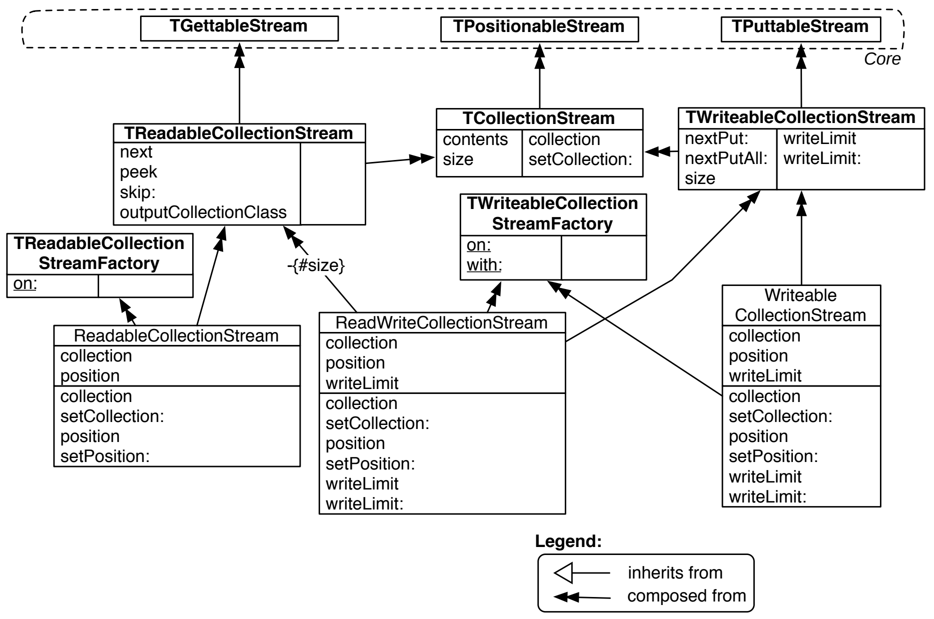


**Fig. 5.** The collection-based stream library. We use a UML-based notation to represent traits: methods on the right are *required* and methods on the left are *provided*.

*TReadableCollectionStream.* The trait TReadableCollectionStream helps creating classes which streams over readable collections. It implements the required methods of TGettableStream: next, outputCollectionClass, and peek. It also redefines skip: for efficiency reasons. The required method TGettableStream>>atEnd is provided by TPositionableStream and thus, does not require further work.

*TCollectionStream.* This trait is inspired by the ANSI Smalltalk standard. It is used for every stream that needs to read from or write to a collection. This trait defines contents and size in terms of two new methods: collection and setCollection:. The former must return the internal collection and the latter provides a setter for this collection. The method size returns the size of the collection.

*TWriteableCollectionStream.* The trait TWriteableCollectionStream depends on a new instance variable accessible through two accessors writeLimit and writeLimit:. This variable allows the internal collection to be bigger than the number of characters in the stream. This is a common technique used to avoid creation of a new collection each time an object is written to the stream. The TWriteableCollectionStream>>size returns the value of writeLimit and nextPut: writes its parameter at the right position in the collection. The trait also reimplements nextPutAll: for efficiency reasons.

## 4.2 Trait factories

The ANSI Smalltalk standard defines ReadStreamFactory>>on: and WriteStream-Factory>> with: to create new streams. Basically there are three places where the stream instance creation methods can be defined. The most two natural ones are on the traits TReadableCollectionStream and TWriteableCollectionStream or directly in the classes. Each solution has advantages and disadvantages. Adding the instance creation methods in the two traits helps their reuse. However, this forces all classes interested in these traits to have those same methods, even if they don't need them. If the instance creation methods are implemented in the classes, there will be duplication amongst the different classes.

We chose a third solution and implement the instance creation methods in separate traits. We named those traits "factories" because they support new stream creation.

We developed two factories: TReadableCollectionStreamFactory and TWriteableCollectionStreamFactory. The former implements on: and the latter implements on: and with:. Even if the ANSI Smalltalk standard does not define on: for writeable streams, we decided to implement it following the Squeak and VisualWorks implementations.

## 4.3 Classes

Traits alone are not enough to create a library. Classes are required to compose and create new instances. The original Squeak hierarchy provides three classes for collection-based streams: ReadStream, WriteStream and ReadWriteStream. Our implementation has equivalent classes with more explicit names: ReadableCollectionStream, WriteableCollectionStream and ReadWriteCollectionStream.

Those classes have nothing more to do than declaring the use of already defined traits, declaring some instance variables and implementing the required accessors.

The only difficulty arises with ReadWriteCollectionStream which has a conflict with the method size. The method size is implemented in both TReadableCollectionStream, obtained from TCollectionStream, and TWriteableCollectionStream. The first implementation reflects the size of the collection whereas the other takes care of the variable writeLimit and the efficient implementation in TWriteableCollectionStream. That's why ReadWriteCollectionStream has to use the implementation of TWriteableCollectionStream. To do this, the class removes the implementation of size coming from TReadableCollectionStream. This can be seen in Figure 5 on the arrow going from ReadWriteCollectionStream to TReadableCollectionStream[4].

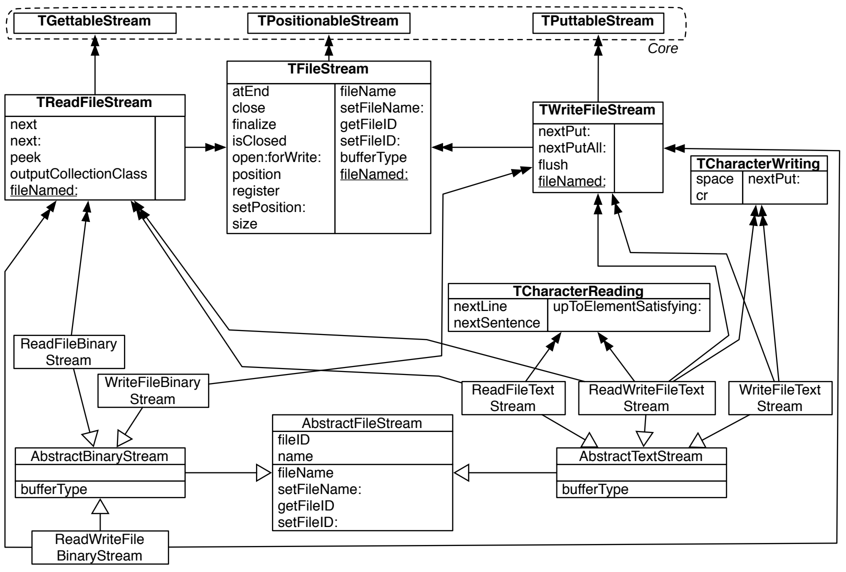


**Fig. 6.** FileStream implementation.

## 5 File-based streams

Nile includes a file-based stream library, shown in Figure 6. As with other file-based streams, it allows one to work with both binary and text files, supporting three access modes for each (read, write, and readwrite).

Each kind of file access is represented by a different class: the developer must explicitly choose the class based on what she wants to do with the file: reading, writing or both, in a binary or a text file. That way, the user has only the methods she can send in the interface of the stream. Note that this is a library design choice and it does not impact the way we decompose the behavior into traits.

Each file-based stream should be positionable, that's why the trait TPositionableStream is used by TFileStream. TFileStream is the common trait for all file-based streams. It implements base functionalities for file access and requires four accessors, a bufferType method and an instance creation method fileNamed. The private method bufferType is used to differentiate binary from text files.

The traits TReadFileStream and TWriteFileStream use the reusable traits TGettableStream and TPuttableStream from the core, respectively. They implement the required methods of these traits. Having implemented the reading and writing methods in separate traits instead of classes really helps here. This way,

[4] The trait model gives the composer the possibility to remove methods through the minus (-) operator.

our file-based streams only get the desired methods, not all methods like in the Squeak hierarchy.

At the very bottom of Figure 6, we defined the abstract classes AbstractFileStream, AbstractBinaryStream and AbstractTextStream to factorize instance variables definition and accessors. These abstract classes allow the definition of the six concrete classes with no more work.

Text-based streams use the traits TCharacterReading and TCharacterWriting depending on the type of file access. Even if simple, these two traits help defining methods only where they are needed and in all places where they are needed.

## 6 Other libraries

In this section we show how traits support reuse by presenting two libraries. We first present how character-related writing methods can be factored out in a trait. And then we describe the trait TDecoder that implements stream composition. Note that Nile offers several other libraries which are summarized later in Table 3.

### 6.1 Writing characters

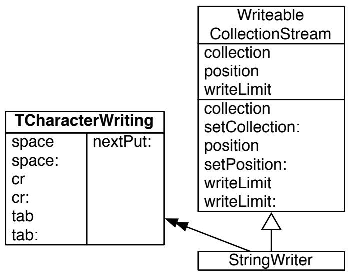


**Fig. 7.** Writing characters to a string.

In the Squeak hierarchy, the class WriteStream contains methods like space, cr and tab to write specific characters. These methods are only useful in case the user wants to write characters in her stream. If she wants to write binary data then those methods are useless and even pollute the interface of the stream. That's why we chose to implement the character-writing methods in a specific trait TCharacterWriting. Another advantage of using a specific trait is that Nile is then able to give those methods to any class which can write characters such as StringWriter in Figure 7 (a collection-based write stream which is writing characters) or WriteFileTextStream and ReadWriteFileTextStream in Figure 6.

## 6.2 Decoders

Developers often want to chain several streams. They want to use them like pipes that are connected together. For example, a developer may want a stream to read from a file and another stream which decompresses the first one on-the-fly. We generalized a mechanism which was already available in Squeak for classes like ZipWriteStream and have implemented a trait to support the composition of such decoders. We first present a scenario for such decoders and then describe our implementation.

A decoder is a GettableStream which reads its data from another GettableStream called its input stream. This way decoders can be chained. The decoder can do whatever it wants with the contents of its input stream: for example, it can ignore some elements, it can convert characters to numbers, it can compress or decompress. . .

*Selective number reading.* Imagine you have a string, or a file, containing space separated numbers. We can get all even numbers as presented in the code below. Here the developer composes three elementary streams which are subclasses of Decoder which uses the trait TDecoder.

```
| stream |
stream := ReadableCollectionStream on: '123 12 142 25'.
stream := NumberReader inputStream: stream.
stream := SelectStream selectBlock: [:each | each even] inputStream: stream.

stream peek.      ==> 12
stream next.      ==> 12
stream atEnd.     ==> false
stream next.      ==> 142
stream atEnd.     ==> true
```

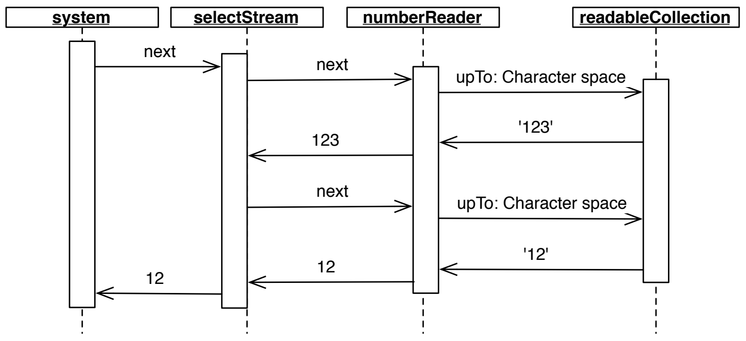


**Fig. 8.** Chaining streams

Figure 8 illustrates the stream connection. NumberReader transforms a character based stream in a number-based stream. SelectStream ignores all elements in the input stream for which the select block does not answer true.

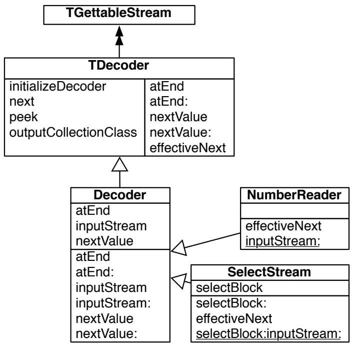


**Fig. 9.** The decoder and two possible clients.

*The trait TDecoder.* Figure 9 shows the decoder hierarchy. A decoder is basically a GettableStream, that's why TDecoder uses the trait TGettableStream. We chose to implement the decoding methods in a trait to let developers incorporate its functionalities into their own hierarchies.

TDecoder provides implementations for all required methods of TGettableStream (see Figure 4) but atEnd and it requires four accessors (including atEnd) and the method effectiveNext. This method effectiveNext is where all the work happens. It should read its input stream and return a new element. The method TDecoder>>next calls effectiveNext and catches StreamAtEndErrors for setting the atEnd variable.

Factoring the Nile core in traits proved again to be useful. If we had implemented it using single inheritance we would have been forced to choose a superclass between class Stream, which provides writing methods we don't want, or class ReadStream which only streams over collections, which is not what we want to do with decoders.

## 7 Discussions

This section compares Nile with other stream implementations, analyzes its performance and discusses where traits did and did not help us.

## 7.1 Comparison with previous work

There is no previous work building a library from scratch using traits. However, Schärli *et al.* [5] were the first to refactor the collection and stream hierarchies using traits.

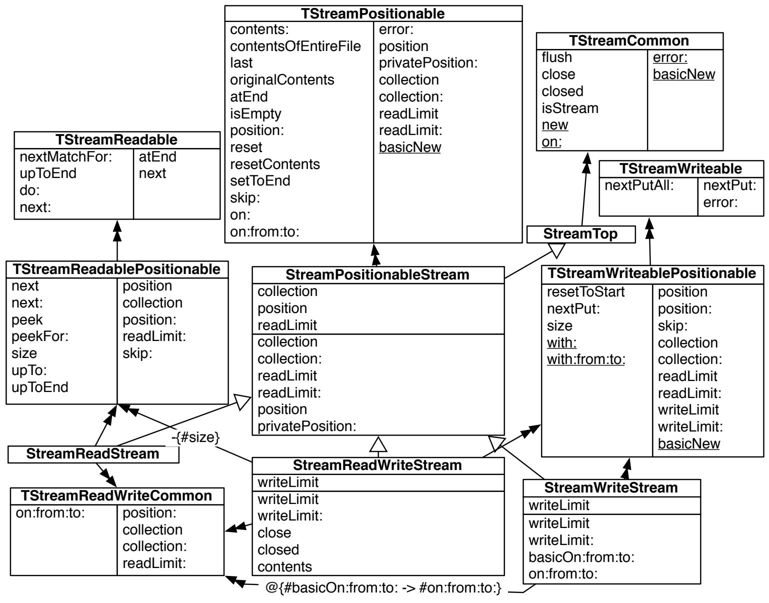


**Fig. 10.** Schärli's refactored stream hierarchy.

Figure 10 shows Schärli's stream hierarchy. Their work is a refactoring, where they took the original Squeak stream hierarchy and extracted the existing behavior into traits. This was a valuable experience that showed how a non-trivial implementation could be replaced with a cleaner implementation that was backwards compatible. While valuable, the backward compatibility constraint forces the result to be linked to the original implementation. Therefore it exhibits a number of problems:

– The positioning methods for a stream have to be based on collections because the methods position:, atEnd and setToEnd are all defined in the trait TStreamPositionable which depends on collection and collection:. Therefore it can not be used with files for example.
– The method TStreamReadablePositionable>>peek is dependent of the existence of methods collection and position but it shouldn't be.
– The granularity of the traits is big which hampers their reuse. For example, if we would like to have a skip: method, which is provided by TStreamPosi-

tionable, we would get many more methods and, worse, we have to provide many collection-related methods.

### 7.2 Nile Analysis

*The factories.* Having implemented the factories in two separate traits complicates the hierarchy. Another solution would have been to define ReadWriteCollectionStream as a subclass of WriteableCollectionStream to inherit both instance creation methods directly. However we believe that having explicit traits is better, since they are potentially reusable.

*Using an abstract superclass.* Nile defines three concrete classes to stream over collections: ReadableCollectionStream, WriteableCollectionStream and ReadWriteCollectionStream. They all define the same instance variables and the same instance creation methods. To simplify the implementation of these classes, we could have implemented an abstract superclass for all of these classes with two common instance variables position and collection and their accessors. This is what we chose for the file-based streams (see Figure 11).

*Classes vs. Traits.* One of the key questions when building a system with traits is to decide when to use classes and when to use traits. In certain situations as illustrated by the Squeak stream hierarchy (see Section 2), defining a class or inheriting from a class does not make sense since some of its state is not used or its behavior should be canceled. This is a clear indication for using traits.

Most of the time however the decision is not that easy to take and the designer has to assess whether potential clients may benefit from the traits, *i.e.,* if the defined behavior can be reused in another hierarchy. In a lot of situations this means that traits are favored, since the price to pay to use traits is very low compared with the benefits one gets.

*Reuse at Work.* Figure 11 offers an overview of the core and some libraries of Nile. The fact that we based our implementation on traits rather than on inheritance and that we completely rethought the stream hierarchy leads to several advantages.

With Nile comes some really reusable traits which can be plugged in any other hierarchy. For example, implementing socket-based streams would only require socket manipulation work whereas utility methods like nextPutAll:, skip:, upToEnd are offered to the developer. Using the trait TGettableStream, a developer can easily implement a Random class which is basically a stream over random numbers. Table 3 presents the current clients we implemented in Nile using traits as well as the number of implemented methods to get the desired behavior.

Table 4 presents how much our core traits are reused. It presents for each traits the number of clients, the number of required methods and the number of methods that the trait provides. We see a good ratio provided/required for

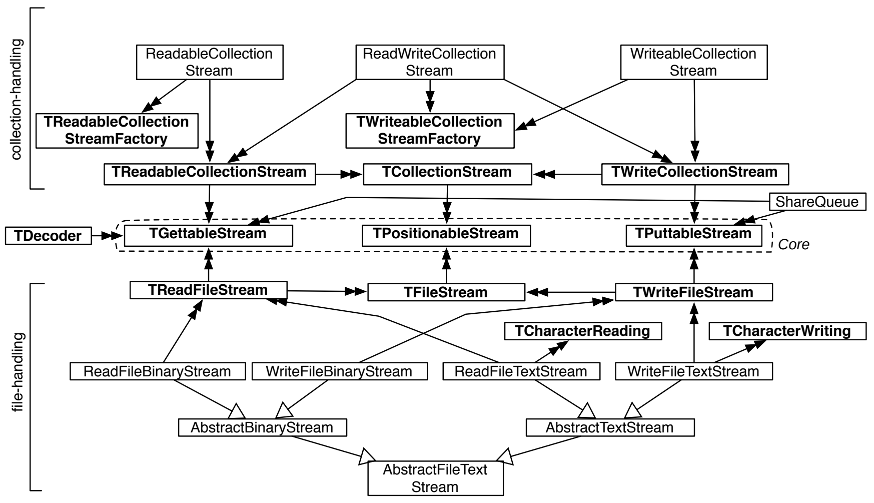


**Fig. 11.** An overview of Nile first clients

| client name | superclass and trait used | met. | description |
|---|---|---|---|
| Random | TGettableStream | 4 | generate random numbers. |
| LinkedListStream | TGettableStream<br>TPuttableStream | 5 | stream over linked elements. |
| History | TReadableCollectionStream<br>TWriteableCollectionStream | 7 | manage do and undo of command objects. |
| SharedQueue | TGettableStream<br>TPuttableStream | 5 | concurrent access on a queue. |
| StringReader | ReadableCollectionStream<br>TCharacterReading | 0 | add character-based reading methods. |
| StringWriter | WriteableCollectionStream<br>TCharacterWriting | 0 | add character-based writing methods. |
| CompositionStream | Decoder | 1 | multiplexer for input streams. |
| Tee | Decoder | 1 | fork the input stream (like the Unix *tee* command). |
| Buffer | Decoder | 1 | add a buffer to any kind of input stream. |
| NumberReader | Decoder | 1 | read numbers from a character based input stream. |
| SelectStream | Decoder | 1 | select elements from an input stream. |
| PipeEntry | TGettableStream<br>TPuttableStream | 7 | allow data to be manually put into a pipe. |

**Table 3.** Nile clients

most of the traits. The ratio may still improve if additional behavior based on the core functionality is introduced.

Table 5 presents some metrics which compares the same functionalities in the Squeak implementation and in Nile for the collection-based streams. The first two metrics show that Nile uses a lot of traits and only a few classes. This is because Nile is designed to have fine grained and reusable units. The next two (number of methods and number of bytes) are more interesting and show that the amount of code is really smaller in Nile than in Squeak. Nile has 18% less

| Trait | client classes | required met. | provided met. | $\frac{provided}{required}$ |
|---|---|---|---|---|
| TGettableStream | 22 | 4 | 11 | 275% |
| TPositionableStream | 20 | 3 | 9 | 300% |
| TPuttableStream | 13 | 1 | 4 | 400% |
| TReadableCollectionStream | 6 | 4 | 26 | 650% |
| TCollectionStream | 12 | 4 | 11 | 275% |
| TWriteableCollectionStream | 6 | 6 | 23 | 383% |
| TCharacterReading | 3 | 2 | 1 | 50% |
| TCharacterWriting | 3 | 1 | 8 | 800% |
| TByteReading | 3 | 3 | 6 | 200% |
| TByteWriting | 3 | 2 | 5 | 250% |
| TDecoder | 7 | 6 | 14 | 233% |

**Table 4.** Nile-trait reusability.

| | Squeak | Nile | $\frac{Squeak-Nile}{Squeak}$ |
|---|---|---|---|
| Number of Classes And Traits | 5 | 13 | -160% |
| Number of Classes | 5 | 4 | 20% |
| Number of Methods | 53 | 43 | 18% |
| Number of Bytes | 1725 | 1459 | 15% |
| Number of Cancelled Methods | 2 | 0 | 100% |
| Number of Reimplemented Methods | 14 | 3 | 78% |
| Number of Methods Implemented Too High | 10 | 0 | 100% |

**Table 5.** Some metrics for the collection-based streams

methods and 15% less bytecodes than the corresponding Squeak collection-based stream library. Finally, we can deduce from the last metrics that the design of Nile is better: there is no cancelled method nor method implemented too high and there are only four methods reimplemented for speed reason compared to the fourteen of the Squeak version.

*About Trait Composition.* During trait composition it is possible that required methods of a trait are fulfilled by the provided methods of another traits. When this happens the developer does not have to do any extra work and benefits from the composition result. We can see this at work for the method atEnd that is required in TGettableStream and provided by TPositionableStream. The trait TReadableCollectionStream doesn't have any work to get the implementation of atEnd. However, such a situation is rare and based on the decomposition of traits using a compatible behavior and vocabulary.

However, it is sometimes better or necessary to override a method coming from a trait. It is because the new implementation have more knowledge than the overridden one and thus can do a better job. For example, the method TReadableCollectionStream>>skip: overrides the method TGettableStream>>skip:. The new method is more efficient because the stream is positioned directly, needing only a small bound computation:

```
TReadableCollectionStream>>skip: amount
```

```
"Moves relatively in the stream. Go forward amount elements. Go backward if amount is
negative. Overrides TGettableStream>>skip: for efficiency and backward possibility."

 self position: ((self position + amount) min: self size max: 0)
```

Moreover, skip: is now able to go backward if amount is negative, which was not the case in the implementation of TGettableStream.

## 7.3 Performance optimization

One of the key challenge of Nile in terms of performance is to be able to iterate over any kind of collection while at the same time be as efficient as the squeak implementation for Arrays and Strings. We present our solution to this challenge.

Contrary to the Squeak class WriteStream, Nile's TWriteableCollectionStream is able to iterate over any kind of SequenceableCollection. In Squeak the method WriteStream>>nextPutAll: directly manipulates its internal collection using a primitive call to replaceFrom:to:with:startingAt: implemented in String and Array[5], Nile has more work.

The idea is to propose a dedicated set of classes working specifically on Array and String. We first reimplemented the method nextPutAll: in TWriteableCollectionStream to take care of any kind of collection. This proved to be slow when iterating over Arrays and Strings compared to Squeak. Benchmarking shows that too much time was lost into calling methods. We have then implemented an optimized version (*i.e.,* using the primitive mentioned above) of nextPutAll: directly into the classes ReadWriteArrayStream and WriteableArrayStream in which we are sure that the underlying collection is an Array (as shown on the left side of Figure 12).

*Accessor-use impact.* Traits cannot define state, which must be accessed via accessor methods. As Squeak does not have a JIT compiler, using accessors instead of direct instance variable access has a cost. Table 6 shows that using accessors in the context of stream on strings and arrays is 41% times slower than direct instance variable access. To optimize our library as much as possible we used direct accesses, *i.e.,* as shown on the left of Figure 12 we redefined nextPutAll:. However this has as impact that we have to duplicate the optimized implementation of nextPutAll into WriteableArrayStream and ReadWriteArrayStream.

The right of Figure 12 presents another solution we implemented using an extra trait to share the optimized method for the two classes (*i.e.,* calling the primitive). However we discarded this solution since it is slower because traits forced us to use accessors.

[5] While replaceFrom:to:with:startingAt: is implemented for all kinds of SequenceableCollections, it does not work for OrderedCollection.

|  | execution (per second) | $\frac{nile}{squeak}$ |
|---|---|---|
| Squeak implementation | 126 |  |
| Nile with direct variable access | 138 | 110% |
| Nile with accessors | 81 | 64% |

**Table 6.** Nile performances. Without accessors, Nile is faster than Squeak. But using them makes it slower.

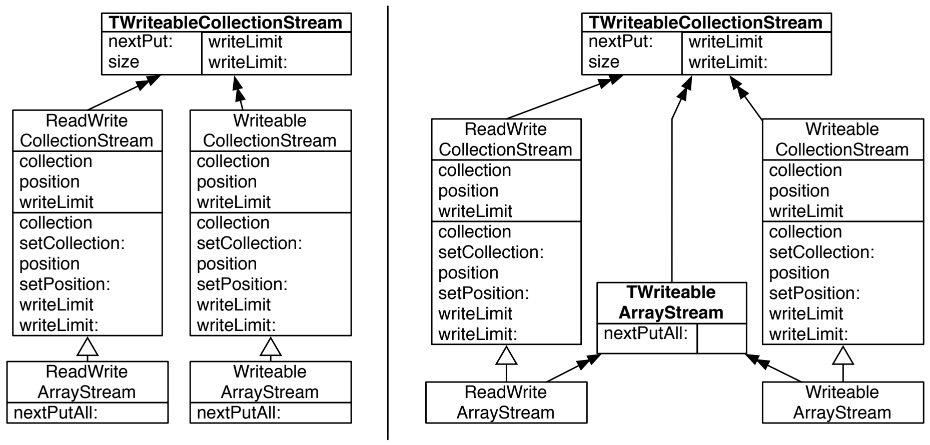


**Fig. 12.** Two solutions to optimize nextPutAll:.

## 8 Problems with traits

Since one of the original goal of the work presented in this paper is to identify potential problems with traits, we now report the problems we faced while developing Nile. Note that some problems are not trait specific but due to Smalltalk's lack of visibility controls. In addition it should be noted that we did not encounter problem with aliasing in the context of recursive calls which is a known problem of traits.

### 8.1 Interface pollution

In this section we present some problems due to class interface extension.

*Required accessors.* With stateless traits, it is not possible to add state, *i.e.,* instance variables, to traits. Instead, the developer must add required accessors to its trait and the classes will implement those required accessors and the instance variable. This is a problem because the accessors are then part of the interface of the classes and this adds a burden to the class developers. However this would be solved if Smalltalk would have method access control. Stateful traits

[3] solve this problem by allowing traits to contain private state. For example, if we had used stateful traits the methods TPositionableStream>>setPosition: and TWriteableCollectionStream>>writeLimit would not have been required.

However, developing Nile showed that stateful traits would not have been of great help. If we examine the trait TCollectionStream in Figure 5, we can see that implementing an instance variable collection here would have been interesting because classes would not have needed to define it. But, methods collection and setCollection: would still need to be in the interface because TReadableCollectionStream>>outputCollectionClass and TReadableCollectionStreamFactory>>on: need them.

We believe that stateful traits are not as interesting as what a first impression might tell.

*Lazily initialized variable.* There are basically three ways of initializing an instance variable giving it a first value: initializing lazily the variable in the accessor, using an initialize method, or initializing the variable in the instance creation method through an accessor.

Lazy initialization is a common programming pattern. Here is an example in Smalltalk which returns the value of the variable checked if it has been set or sets it to false and returns false:

```
checked
  ^ checked ifNil: [checked := false]
```

Now, imagine a trait needs a variable and a default value. Since traits can't contain state in their standard implementation, accessors must be required methods. But where do you lazily initialize the variable? Two solutions are possible: you can force users of the trait to initialize the variable or you can initialize in the trait and use another method for accessing the variable. Here is an example of the later possibility:

```
checked
  ^ self getChecked ifNil: [self checked: false. false].

checked: aBoolean
  self explicitRequirement.

getChecked
  self explicitRequirement.
```

This solution pollutes the trait interface with an unnecessary method getChecked. The other solution consists of letting the trait user initialize the variable. This solution does not pollute the interface but gives more responsibility to other developers and may produce code duplication or bugs.

The same problem appears when you want to do some checking before assigning to a variable as shown in TPositionableStream>>position: for example:

```
TPositionableStream>>position: newPosition
  "Sets the number of elements before the position to be the parameter
   newPosition. 0 for the start of the stream. Throws an error if the
   parameter is lesser than 0 or greater than the size."

   (self isInBounds: newPosition) ifFalse: [InvalidArgumentError signal].
   self setPosition: newPosition.
```

This setter needs an additional method setPosition: which really modifies the variable and which is a required method of the trait.

*Initializing a trait.* In a class, when a developer wants to initialize a newly created object, he can use an initialize method:

```
initialize
  super initialize.
  color := Color transparent.
```

This can be done in a trait too provided that the developer uses an accessor instead of a direct reference to the variable. Problems arise when a class or a trait uses multiple traits, each defining its own initialize method. In this case, there will be conflicts and those conflict can only be resolved by aliasing. This brings lots of pollution in the class interface and require a lot of work.

Another solution would be to use a specific name for each method initialize. For example, it the trait TPositionableStream needs an initialize method, the developer can name it initializePositionableStream. Each user of the trait now needs to define its own initialize method which calls initializePositionableStream. We believe this is still clunky and requires too much work from the developer.

*Initializing in the instance creation method.* Instance creation methods can be used to initialize variables. This is what we did for Nile:

```
TWriteableCollectionStreamFactory>>on: aCollection
  ^ self basicNew
      initialize;
      setCollection: aCollection;
      writeLimit: 0;
      reset;
      yourself
```

Smalltalk is made such that this requires that setters are available in the interface of the class. It also puts more responsibility on the instance creation method which now needs more knowledge over the class it instantiates.

## 8.2 Methods silently ignored

Sometimes, modifying a trait does not modify the users of this trait in the same way because of name overriding. Note that this problem is not trait specific but it is a problem of object-oriented programming as shown by Figure 13.

Figure 13 shows a part of our test hierarchy for Nile. The test hierarchy is very similar to the Nile hierarchy: for each model trait or class, there is a test trait or class. The method nextPutAll: is tested in two different places: in the methods TPuttableStreamTest>>testNextPutAll1 and TWriteableCollection-StreamTest>>testNextPutAll2. If a tester adds a new test named testNextPutAll2 in the trait TPuttableStreamTest, then the test is silently ignored and will never be launched.

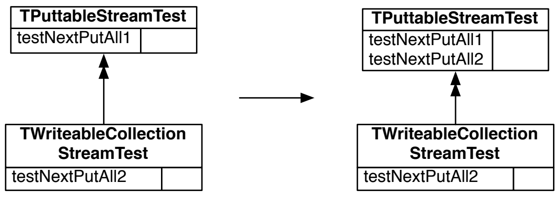


**Fig. 13.** If the tester implements TPuttableStreamTest>>testNextPutAll2, the test will never be launched because TWriteableCollectionStreamTest>>testNextPutAll2 hides it.

## 9 Related work

We already compared our approach with the few work refactoring existing code using traits [5, 6]. We now present the approaches that automatically transform existing libraries using Formal Concept Analysis (FCA) or other techniques. FCA was used in different ways.

Godin [9] developed incremental FCA algorithms to infer implementation and interface hierarchies guaranteed to have no redundancy. To assess their solutions from a point of view of complexity and maintainability they propose a set of structural metrics. They analyze the Smalltalk Collection hierarchy. One important limitation is that they consider each method declaration as a different method and thus cannot identify code duplication. Moreover their approach serves rather as a help for program understanding than reengineering since the resulting hierarchies cannot be implemented in Smalltalk because of single inheritance.

In C, Snelting and Tip analyze a class hierarchy making the relationship between class members and variables explicit [10]. By analyzing the *usage* of the hierarchy by a set of client programs they are able to detect design anomalies such as class members that are redundant or that can be moved into a derived class. Taking into account a set of client programs, Streckenbach infer improved hierarchies in Java with FCA [11]. Their proposed refactoring can then be used for further manual refactoring. The tool proposes the reengineer to move methods up in the hierarchy to work around multiple inheritance situations generated by the generated lattice. The work of Streckenbach is based on the analysis of the usage of the hierarchy by client programs. The resulting refactoring is behavior preserving (only) with respect to the analyzed client programs.

Lienhard *et al.* applied Formal Concept Analysis to semi-automatically identify traits [6]. We cannot really compare their resulting hierarchy with ours since the information about the respective traits is no longer available. However we can conclude that the resulting hierarchy was limited and resulted only from a refactoring effort and not from a new design.

Interfaces and specifications of the Smalltalk collection hierarchy are also analyzed by Cook [12]. He also takes method cancellation into account to detect protocols. By manual analysis and development of specifications of the Smalltalk collection hierarchy he proposes a better protocol hierarchy. Protocol hierarchies explicitly represent similarities between classes based on their provided methods. Thus, compared to our approach, protocol hierarchies present a *client* view of the library rather than one of the *implementor*.

Moore [13] proposes automatic refactoring of Self inheritance hierarchies. Moore focuses on factoring out common expressions in methods. In the resulting hierarchies none of the methods and none of the expressions that can be factored out are duplicated. Moore's factoring creates methods with meaningless names which is a problem if the code should be read. The approach is more optimizing method reuse than creating coherent composable groups of methods. Moore's analysis finds some of the same problems with inheritance that we have described in this paper, and also notes that sometimes it is necessary to manually move a method higher in the hierarchy to obtain maximal reuse.

Casais uses an automatic structuring algorithm to reorganize Eiffel class hierarchies using decomposition and factorization [14]. In his approach, he increases the number of classes in the new refactored class hierarchy. Dicky *et al.* propose a new algorithm to insert classes into a hierarchy that takes into account overridden and overloaded methods [15].

The key difference from our results is that all the work on hierarchy reorganization focuses on transforming hierarchies using inheritance as the only tool. In contrast, we are interested in exploring other mechanisms, such as explicit composition mechanisms like traits composition in the context of mixin-like languages. Another important difference is that we do rely on algorithms. This is important since we want to be able to use our result to compare it with the result of future approach extracting traits automatically, so the Nile library may serve as a reference point.

## 10 Conclusion

Traits are units of reuse that can be used to compose classes. This paper is an experience report. Even if other experiences have been made to test traits, they were always refactoring an existing hierarchy, moving methods from classes to traits. Our work however presents a brand new implementation. We started from the textual description from the ANSI Smalltalk standard and from existing implementations of stream libraries in Squeak and VisualWorks. Our result is a completely new implementation, named Nile, of the stream hierarchy which does not share any code with previous implementations.

Our experience shows that traits are good building blocks which favor reuse across different hierarchies. In the present implementation of Nile we get up to 15% less code than the corresponding Squeak code. Core traits are reused by numerous clients. We also presented the problems we faced during the experience and believe that Nile can be used in the future as a reference point for comparing future trait enhancement.

This experience shows that well defined traits can naturally fit into lots of different clients which can benefit from methods offered by the trait for a relatively low cost.

**Acknowledgment**

We gratefully acknowledge the financial support of the Agence Nationale pour la Recherche (ANR) for the project "Cook: Rearchitecting object-oriented applications" (2005-2008).

## References

1. Schärli, N., Ducasse, S., Nierstrasz, O., Black, A.: Traits: Composable units of behavior. In: Proceedings of European Conference on Object-Oriented Programming (ECOOP'03). Volume 2743 of LNCS., Springer Verlag (July 2003) 248–274
2. Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., Black, A.: Traits: A mechanism for fine-grained reuse. ACM Transactions on Programming Languages and Systems (TOPLAS) **28**(2) (March 2006) 331–388
3. Bergel, A., Ducasse, S., Nierstrasz, O., Wuyts, R.: Stateful traits. In: Advances in Smalltalk — Proceedings of 14th International Smalltalk Conference (ISC 2006). Volume 4406 of LNCS., Springer (2007) 66–90
4. Ducasse, S., Wuyts, R., Bergel, A., Nierstrasz, O.: User-changeable visibility: Resolving unanticipated name clashes in traits. In: Proceedings of 22nd International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'07), New York, NY, USA, ACM Press (2007) To appear.
5. Schärli, N., Ducasse, S., Nierstrasz, O., Wuyts, R.: Composable encapsulation policies. In: Proceedings of European Conference on Object-Oriented Programming (ECOOP'04). LNCS 3086, Springer Verlag (June 2004) 26–50
6. Lienhard, A., Ducasse, S., Arévalo, G.: Identifying traits with formal concept analysis. In: Proceedings of 20th Conference on Automated Software Engineering (ASE'05), IEEE Computer Society (November 2005) 66–75
7. ANSI New York: American National Standard for Information Systems - Programming Languages - Smalltalk, ANSI/INCITS 319-1998. (1998) http://wiki.squeak.org/squeak/uploads/172/standard_v1_9-indexed.pdf.
8. Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., Kay, A.: Back to the future: The story of Squeak, A practical Smalltalk written in itself. In: Proceedings OOPSLA '97, ACM SIGPLAN Notices, ACM Press (November 1997) 318–326

9. Godin, R., Mili, H., Mineau, G.W., Missaoui, R., Arfi, A., Chau, T.T.: Design of Class Hierarchies based on Concept (Galois) Lattices. Theory and Application of Object Systems **4**(2) (1998) 117–134
10. Snelting, G., Tip, F.: Reengineering Class Hierarchies using Concept Analysis. In: ACM Trans. Programming Languages and Systems. (1998)
11. Streckenbach, M., Snelting, G.: Refactoring class hierarchies with KABA. In: OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications, New York, NY, USA, ACM Press (2004) 315–330
12. Cook, W.R.: Interfaces and Specifications for the Smalltalk-80 Collection Classes. In: Proceedings of OOPSLA '92 (7th Conference on Object-Oriented Programming Systems, Languages and Applications). Volume 27., ACM Press (October 1992) 1–15
13. Moore, I.: Automatic Inheritance Hierarchy Restructuring and Method Refactoring. In: Proceedings of OOPSLA '96 (11th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications), ACM Press (1996) 235–250
14. Casais, E.: Automatic reorganization of object-oriented hierarchies: A case study. Object-Oriented Systems **1**(2) (December 1994) 95–115
15. Dicky, H., Dony, C., Huchard, M., Libourel, T.: On Automatic Class Insertion with Overloading. In: Proceedings of OOPSLA '96 (11th ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications), ACM Press (1996) 251–267

# Part II

# Development Tools

# Feature Driven Browsing

David Röthlisberger[1], Orla Greevy[1], and Oscar Nierstrasz[1]

Software Composition Group, University of Bern – Switzerland

**Abstract.** Development environments typically present the software engineer with a structural perspective of an object-oriented system in terms of packages, classes and methods. From a structural perspective it is difficult to gain an understanding of how source entities participate in a system's features at runtime, especially when using dynamic languages such as Smalltalk. In this paper we evaluate the usefulness of offering an alternative, complementary feature-centric perspective of a software system when performing maintenance activities. We present a feature-centric environment combining interactive visual representations of features with a source code browser displaying only the classes and methods participating in a feature under investigation. To validate the usefulness of our feature-centric view, we conducted a controlled empirical experiment where we measured and compared the performance of subjects when correcting two defects in an unfamiliar software system with a traditional development environment and with our feature-centric environment. We evaluate both quantitative and qualitative data to draw conclusions about the usefulness of a feature-centric perspective to support program comprehension during maintenance activities.

## 1 Introduction

System comprehension is a prerequisite for software maintenance but it is a time-consuming activity. Studies show that 50-60% of software engineering effort is spent trying to understand source code [1, 2]. Furthermore, object-oriented language characteristics such as inheritance and polymorphism make it difficult to understand runtime behavior purely by inspecting source code [3–5]. Especially in dynamic languages it is nearly impossible to get a complete understanding of a system by just looking at its static source code. The task of understanding a software system is further exacerbated by a best practice in object-oriented programming to scatter behavior in many small methods, often in deep inheritance hierarchies [6].

The problems of understanding object-oriented software are poorly addressed by current development tools, since these tools typically focus only on a structural perspective of a software system by displaying static source artifacts such as packages, classes and methods. This is also true for modern Smalltalk dialects and development environments such as Squeak [7] or Cincom VisualWorks [8].

Understanding how a system's features are implemented is a prerequisite for system maintenance, as maintenance requests are usually expressed in terms of features [9]. Thus, a software engineer typically needs to maintain a mental map between features and how they are implemented as classes and methods. As features are not explicitly represented in the source code, it is not easy to identify and manipulate them. In this paper, we adopt the definition of a feature as being a unit of observable behavior of a system [10].

As traditional development environments offer the software engineer a purely structural perspective of object-oriented software, they make no provision for the representation of features. To tackle this shortcoming, we propose to support the task of understanding during maintenance by providing a feature perspective of a software system. We present a novel feature-centric environment providing support for visual representation, interactive exploration, navigation and maintenance of a system's features. We are motivated by the following questions:

– How useful is a feature-centric development environment for understanding and maintaining software?
– How do we quantitatively measure the usefulness of a feature-centric development environment?
– How do software engineers subjectively rate the usefulness of a feature-centric perspective of a system to perform their maintenance tasks?

The fundamental question we seek to answer is if software engineers can indeed better understand and maintain a software system by exploiting a feature-centric perspective in a dedicated feature-centric development environment. We want to determine if a feature-centric perspective is superior to a structural perspective to support program comprehension. To address this question, we implemented a feature-centric development environment in Squeak [7], a dialect of Smalltalk. Our feature-centric development environment acts as a proof of concept for the technical feasibility of our approach and as a tool which we can actually validate with real software engineers.

The key contributions of this paper are: (1) we present our feature-centric development environment, and (2) we provide empirical evidence to show its usefulness to support comprehension and maintenance activities as compared with the structural views provided by a traditional development environment.

**Paper structure.** In the next section we expand on the problem of feature comprehension and provide a motivating example. Based on this, we formulate our hypotheses of the usefulness of a feature-centric perspective for performing maintenance tasks. In Section 3 we introduce our *feature browser* proof of concept tool, allowing a developer to work in a feature-centric development environment. We validate the usefulness of our feature-centric development environment by conducting an empirical study in Section 4. We present the results and evidence of our study in Section Section 5. We report on related work in Section 6 and finally we conclude in Section 7.

## 2 Problem of Feature Maintenance

It is a generally accepted *best practice* of object-oriented programming that functionalities or *features* are implemented as a number of small methods [4]. This, in addition to the added complexity of inheritance and polymorphism in object-oriented software, means that a software engineer often needs to browse a long chain of small methods to understand how a feature is implemented. In Figure 1 we illustrate this with an example from Pier [11], the system we chose as a basis for our experimentation. Pier is a web content management system encompassing a Wiki application [11] implemented in Squeak Smalltalk [7]. Figure 1 shows a small part of the class hierarchy of Pier and an excerpt of a call tree, generated by exercising the *copy page* feature. The call tree reveals that the feature implementation crosscuts the structural boundaries of the Pier system.

A software engineer, faced with the task of maintaining the *copy page* feature, first needs to locate the relevant classes and methods, and then browses back and forth in the call chain to establish her mental map of the relevant parts of the code and to gain an understanding of how the feature is implemented. This is a cumbersome and time-consuming activity and often the software engineer loses time and focus browsing irrelevant code.

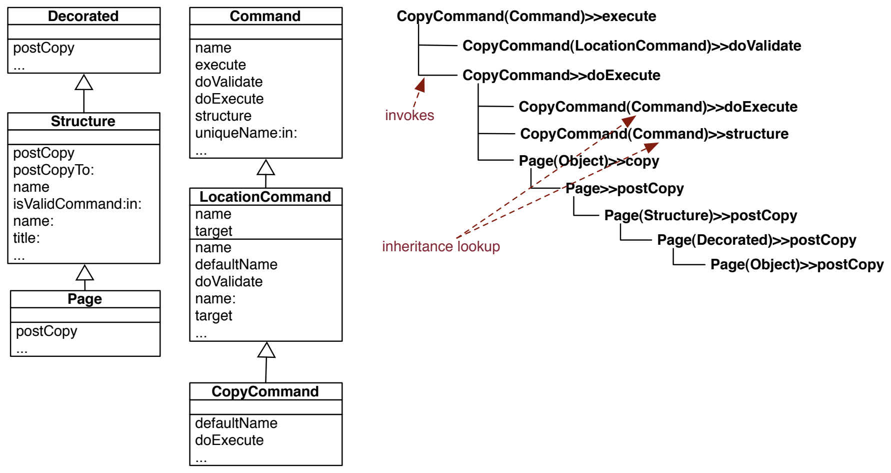


**Fig. 1.** The relevant Pier Class Hierarchies for the *copy page* Feature and its Call Graph

## 2.1 Making Features Explicit in the Development Environment

From the software engineer's perspective, a feature consists of a set of all methods executed to perform a certain task in a software system. Thus features denote units of behavior of a system. The relationships between the methods of a feature are dynamic in nature and thus are not explicit in the structural representation of the software [12]. We represent these dynamic units of behavior (*i.e.,* features ) in terms of their particpating methods. The goal is to support the software engineer when maintaining or fixing a defect in a feature. As described in the work of Licata et. al. , for our experimentation we assume that for a given software system, test cases exist that align with a system's features [13].

Our premise is that by explicitly representing features in the development environment, we support maintenance activities by providing the software engineer with an explicit map between features and source entities that implement the feature. By focusing the attention of the maintainer on only relevant source entities of a given feature or set of features, we improve the understanding and the ease with which she can carry out maintenance tasks.

We state our hypotheses as:

- *A feature-centric development environment decreases the time a software engineer has to spend to maintain a software system (*e.g., *to correct a bug) compared to a traditional development environment which provides only a structural perspective of the code*
- *A feature-centric development environment improves and enriches the understanding of how the features of a software system are implemented*

We refine our hypotheses in Section 4, when we describe the details of our empirical study. Our qualitative and quantitative evaluation of the findings of our experimentation reveal that these hypotheses do indeed hold.

## 3 Proposing a Solution: A Feature-centric Environment

The foundation of our approach is our *feature browser* tool which we embed in the software engineer's integrated development environment (IDE). The purpose of the *feature browser* is to augment an IDE with a feature perspective of a software. We implemented our prototype *feature browser* in Squeak Smalltalk [7]. The *feature browser* complements the traditional structural and purely textual representation of source code in a browser by presenting the developer with interactive, navigable visualizations of features in three distinct but complementary views. These views are enriched with metrics to provide the software engineer with additional information about the relevancy of source artifacts (*i.e.,* classes and methods ) to features.

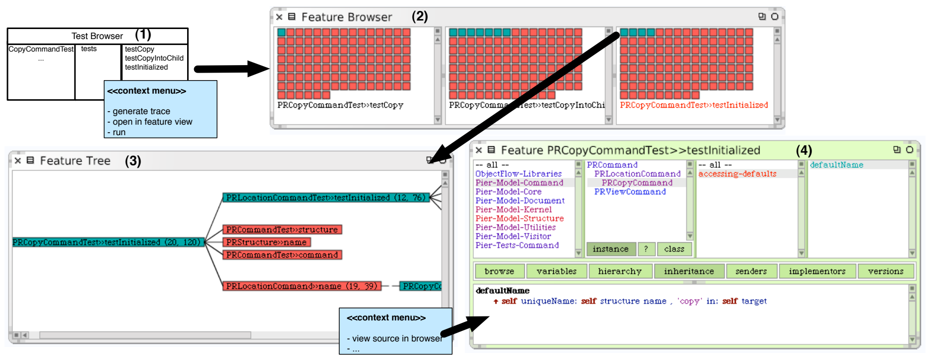
Initially, we introduce the key elements of our feature-centric environment . Subsequently, we describe how feature-centric environment promotes software engineer's comprehension of scattered code in object-oriented programming while performing maintenance tasks on a system's features.

### 3.1 Feature Affinity in a Nutshell.

In previous work [14], we defined a *Feature Affinity* measure to assign a relevancy scale to methods in the context of a set of features. Feature Affinity defines an ordinal scale corresponding to increasing levels of participation of a source artifact (*e.g.,* a method) in the set of features that have been exercised. For our feature-centric environment we consider four Feature Affinity values: (1) a *singleFeature* method participates in only one feature, (2) a *lowGroupFeature* method participates in less than 50% of the features, (3) a *highGroupFeature* method participates in 50% or more of the features and (4) an *infrastructuralFeature* method participates in all of the features.

We exploit the semantics of *Feature Affinity* to guide and support the software engineer during the navigation and understanding of one or many features. We assign to the visual representation of a method a color that represents its *Feature Affinity* value. Our choice of colors corresponds to a heat map (*i.e.*, a cyan method implies *singleFeature* and red implies *infrastructuralFeature*, *i.e.*, used by all the features we are currently investigating).

## 3.2 Elements of the feature-centric environment



```
defaultName
    ↑ self uniqueName: self structure name , 'copy' in: self target
```

**Fig. 2.** The Elements of our Feature Browser Environment

The feature-centric environment contributes three different visualizations for one and the same feature: (1) the compact feature overview, (2) the feature tree view and (3) the feature artifact browser.

*Compact Feature Overview*

The Compact Feature Overview presents a visualization of two or more features represented in a compacted form. The Compact Feature view represents a feature as a collection of all methods used in the feature as a result of capturing its execution trace. Each method is displayed as a small colored box; the color represents the *Feature Affinity* value. The methods are sorted according to their *Feature Affinity* value. The software engineer decides how many features she wants to visualize at the same time (see Figure 2 (2)). Clicking on a method box in the Compact Feature View opens the Feature Tree View, which depicts a call tree of the execution trace. This visualization reveals the method names and order of execution. All occurrences of the method selected in the Compact Feature View are highlighted in the call tree.

*Feature Tree*

This view presents the method call tree, captured as a result of exercising one feature (see Figure 2 (3)). The first method executed for a feature (*e.g.,* the "main" method) forms the root of this tree. Methods invoked in this root node form the first level of the tree, hence the nodes represent methods and the edges are message sends from a sender to a receiver. As with the Compact Feature Overview, the nodes of the tree are colored according to their *Feature Affinity* value.

The key challenge of dynamic analysis is how to deal with the large amount of data. For our experimentation we chose Pier [11], a web-based content management system implemented in Smalltalk. We obtained large traces of more than 15'000 methods. We discovered that it is nearly impossible to visualize that amount of data without losing the overview and focus, but still conveying useful information. To overcome this we applied two techniques: First, we compressed the execution traces and the corresponding visual representation as a feature tree as much as possible without loss of information about order of execution of method sends. Second, we opted to execute test cases of a software system rather than interactively trigger the features directly from the user interface. For instance, instead of looking at the entire *copy page* feature initiated by a user action in the user interface, we analyze the *copy page* feature by looking at the test cases that were implemented to test this feature. As stated in the work of Licata [13], features are often encoded in dedicated unit test cases or in functional tests encoded within several unit test cases. In the case of a software system that includes a comprehensive test suite, it is appropriate to interpret the execution traces of such test cases as feature execution traces.

Furthermore, we tackle the problem of large execution traces by compressing feature trace with two different algorithms: First we remove common subexpressions in the tree and subsequently we remove sequences of recurring method calls as a result of loops.

*Common Subexpression Removal*

A subexpression in a tree is a branch which occurs more than once. If, for example, a pattern "method c invokes methods a and b" occurs several times in a call tree, we identify this pattern as a common subexpression. Our analysis reveals that the execution traces of features typically contain many common subexpressions. By compacting the representation of these subexpressions, we reduce the tree by up to 30% on average. Our visualization still includes an expandable root node of a common subexpression branch in the tree, the subexpression can be opened in a pop-up window by the software engineer. Figure 3 (1) shows a schematic representation of how we display common subexpressions in our feature tree view.

To perform the removal of common subexpressions, we applied the algorithm presented in [15].

*Sequence Removal*

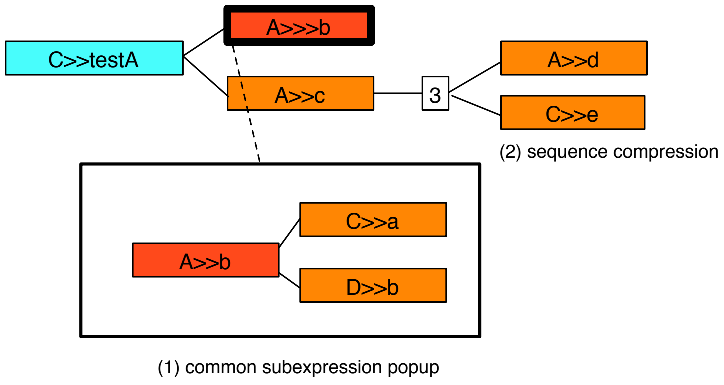


**Fig. 3.** The Common Subexpression and Sequence Compression of the Feature Tree

Often a feature trace contains several sequences, *e.g.,* methods invoked in a loop. It is straightforward to compress these nodes included in a loop by only presenting them once. Furthermore, we indicate how often the statements of a loop are repeated. If, for example, the methods d and e are executed three times in a loop, we add an *artificial numeric* node labeled with a '3' to the tree and link the nodes for d and e. to this node ( as shown in Figure 3 (2)). To detect and compact sequences, we implemented a variation of the algorithm presented in [16].

Despite having applied these techniques, the feature tree is still complete and is easily read and interpreted by the software engineer. No method calls occurring in the feature are omitted.

Initially a feature tree is displayed collapsed to the first two levels. Every node can be expanded and collapsed again. In this way, the software engineer can conveniently navigate even large feature trees.

When a user selects a method node in the compact feature overview , all the occurrences of this method are highlighted in the feature tree. Also the tree is automatically expanded to the first occurrence of the selected method and the feature tree window is centered on that node. The user can navigate through all occurrences of the desired method by repeatedly clicking on the corresponding node in the compact feature overview . By opening a method node in the feature tree the engineer is able to follow the complete chain of method calls from the root node to the current opened method node. No intermediate calls of methods belonging to the software system under study are omitted.

Every node of the tree provides a button to query the source code of the corresponding method in the feature artifact browser.

*Feature Artifact Browser*

The source artifacts of an individual feature are presented as text in the *feature artifact browser* (see Figure 2 (4)). It exclusively displays the classes and methods actually used in the feature. This makes it much easier for the user to focus on a single feature of the software. Our *feature artifact browser* is an adapted version of a standard class browser available in the Squeak environment. It containing packages, classes, method categories and methods in four panes on the top, while the lower pane contains the source code of the selected entity. This version of the class browser not only presents static source artifacts, but also the feature affinity metric values by coloring the names of classes and methods accordingly.

The three distinct visualizations provided by the feature browser are tightly interconnected so that a software engineer does not lose the overview when performing a maintenance task. For instance, the user selects a method in the compact feature overview , the tree opens with all occurrences of the selected method. From the tree perspective, the software engineer can choose to view a method as source code in the feature artifact browser by double-clicking on a node that represents the method of interest in the feature tree.

To initiate a maintenance session with the *feature browser*, the software engineer selects test cases that exercise features relevant to her specific maintenance task. To ease the collection of features traces, we extended the standard class browser of Smalltalk (called OmniBrowser) to seamlessly execute instrumented test cases and capture feature traces as shown in Figure 2 (1). Thus, once she has executed the instrumented test cases, the software engineer launches a feature browser with an initial number of features.

## 3.3 Solving Maintenance Tasks with the Feature Environment

The feature-centric environment supports program comprehension and feature understanding for maintenance tasks in three ways:

Firstly, by comparing several features with each other in the compact feature overview , software engineers gain knowledge about how different selected features are related in terms of the methods they have in common. Since the features included in this compact feature overview can be arbitrarily selected, a developer can compare any features with each other. However, when performing a specific maintenance task, it makes sense to select related test cases to *e.g.,* compare failing tests with similar, non-failing tests to determine which parts of a software system may be most likely responsible for a defect. If, for example, only one test method of a test class exercising the *copy command* feature of a software system is failing, then we compare this test method to all test methods exercising the *copy command* feature. We assume that by looking first at the methods that are only used in the single failing test method, we are more likely to be faster at discovering the defect, as the chances are high that one of the *singleFeature* methods (*i.e.,* methods unique to one compact feature view) are responsible for that defect. The aim of the compact feature overview is to support the quick identification and rejection of candidate methods that may contain a defect.

Secondly, the feature tree provides an insight into the dynamic structure of a feature, an orthogonal dimension compared to the static structure visible in the source code. The nodes in the tree are also colored according to the Feature Affinity metric which guides the software engineer to identify faulty methods. In the example described above where a single feature is failing the most likely candidate methods responsible for the defect are colored in cyan in the feature tree. Since this tree is complete, *i.e.,* it contains all method calls for that specific feature, chances are high that the engineer discovers the source of the defect in one of these single methods that are easy to locate in the feature tree due to their coloring.

The software engineer can also navigate and browse the tree to obtain a deeper understanding of the implementation of the feature and the relationships between the different methods used in the feature. For every method of a feature, the software engineer can easily navigate to all occurrences of this method in the feature tree to find out how and in which context the given method is used. This helps one to discover the location of a defect and the reason why it occurs. The feature tree transforms and improves the understanding of the dynamic structure of a feature and reveals where and how methods are used. For every node in the feature tree, the developer can view the source code of that method in the feature artifact browser .

Thirdly, the feature artifact browser helps the developer to focus only on entities effectively used in a feature. The number of methods that might be responsible for bugs is thus reduced to a small subset of all existing methods in a class or a package. Since only packages, classes and methods are presented to the developer in the feature artifact browser , it is much easier for her to find information relevant for a defect or another maintenance task, *i.e.,* classes or methods. Hence the feature artifact browser helps one to focus on relevant source artifacts and to not lose track and context.

## 4 Validation

To obtain a measure of the usefulness of our feature-centric environment and its concepts in practice, we conducted an empirical experiment with subjects using and working with the feature-centric environment . The goals of the experiment were to gain insight into the strengths and shortcomings of our current implementation of the feature-centric environment , to obtain user feedback about possible improvements and enhancements, and to assess the practical potential of the feature-centric environment . Our primary goal was to gather quantitative data that would indicate how beneficial was the effect of using the feature-centric environment as compared with the standard structural and textual representations of a traditional development environment. We introduce and describe the experiment in this section, formulate the hypotheses we address and describe precisely the study design. Finally, we present the results we obtained from the experiment.

### 4.1 Introducing the Experiment

To validate our feature-centric environment we asked twelve subjects (computer science graduate students) to perform two equally complex maintenance tasks in a software system, one task performed in the feature-centric environment and the other in the standard environment of Squeak Smalltalk (*i.e.,* using Omni-Browser). As a maintenance task, we assigned the subjects the correction of a defect in the software system. The presence of the defect is revealed by the fact that some of the feature tests are failing.

In this experiment we seek to validate three hypotheses concerning our feature-centric environment . If the result of the experiment reveals that the hypotheses hold, we then have successfully obtained clear evidence that our feature-centric environment supports a developer to perform maintenance and that the feature affinity metric we applied is of value in practice.

### 4.2 Hypotheses

We propose the three null hypotheses listed in Table 1. The goal of the experiment is to refute these null hypotheses in favor of alternative hypotheses which would indicate that a feature-centric environment helps the software engineer to discover and correct a defect and hence improves program comprehension.

| | |
|---|---|
| $H0_1$ | The time to discover the location of the defect is equal when using the standard browser and our feature-centric environment . (formally: $\mu_{D,FB} = \mu_{D,OB}$, where $\mu_{D,FB}$ is the average discovery time using the feature-centric environment and $\mu_{D,OB}$ the average discovery time using OmniBrowser) |
| $H0_2$ | The time to correct the defect is equal when using the standard browser and our feature-centric environment . (formally: $\mu_{C,FB} = \mu_{C,OB}$) |
| $H0_3$ | The feature-centric environment has no effect on the software engineer's program comprehension. (formally: average effect $\mu_{E,FB} = 0$) |

**Table 1.** Formulation of the null hypotheses

### 4.3 Study design

**Study setup.**

During the experiment, subjects were asked to correct two bugs in a complex web-based content management framework written in Smalltalk. Our software system, Pier [11], consists of 219 classes of 2341 methods with a total of 19261 lines of code. The two defects were approximatively equally complex to discover and correct. For both bugs we slightly changed one method in the Pier system. As a result of our change, some (feature) tests failed. We presented the subjects with these failing tests as a starting point for their search for the defect. In Pier a unit test class is dedicated to a certain feature (*e.g.,* copying a Wiki page) and the different methods of a test class are different instantiations of that feature (*e.g.,* different parameters with which the feature is exercised). This is in line with the argumentation presented in the work of Licata [13] saying that features are often encoded in unit test cases.

In our experiment, we introduced two different defects in the *copy page* feature. This feature is tested by a dedicated test class with five test methods. The two defects produce failures in different test methods of the *copy page* test class. For the experiment we select all the five test methods exercising the *copy page feature* and show them in row in the Compact Feature View our feature-centric environment . As these five exercised test methods are variants of the same feature they are clearly related to each other, which means that if one test method reveals a failure but the others don't, it is most likely to spot that failure by just looking at the different methods the failing test is executing.

We conducted the experiment with twelve graduate computer science students as subjects with varying degrees of experience with the Smalltalk programming language and the Squeak development environment. All subjects had between one and five years of experience with the language, but only between zero and four years of experience with the Squeak development environment. None of the subjects was familiar with the design and implementation of the Pier application in detail.

Before starting the experiment, we organized a workshop to introduce the concepts and paradigms of our feature-centric environment . Every subject could experiment with our feature browser for half an hour before commencing our experiment consisting of the task of defect location and correction. Furthermore, we briefly introduced the subjects to the design and the basic concepts of Pier by presenting an UML diagram of the important entities of the application. The experiment was conducted in a laboratory environment, as opposed to the subjects' normal working environment. While performing the experiment, we observed the subjects. Afterwards we asked them to respond to a questionnaire to gather qualitative information about the feature-centric environment . The questionnaire contains several questions about the usefulness of the feature-centric environment to understand the program and to perform the requested maintenance task. For every question, the subjects could choose a rating from -3 to 3, where -3 represents a hinderance to program comprehension, 0 no effect and 3 very useful. In addition, the subjects could provide qualitative feedback, *e.g.,* what shortcomings of the environment suggested improvements. The results of these open suggestions, as well as the observations of the experimenters form the qualitative part of our study.

Every subject had to fix both defects, one using our feature-centric environment and the other one using the standard class browser, *i.e.,* OmniBrowser. Both the debugger as well as the unit-test runner were available for use to complete the task. We prohibited use of every other tool during the experiment. From subject to subject we varied the order in which they fixed the defects as well as the order in which they used the different browsers. Hence there are four possible combinations to conduct an experiment with a subject and each of these four combinations were exercised three times. A concrete combination was randomly chosen by the experimenters for any subject.

**Dependent variables.**

We recorded two dependent variables: (i) the time to discover the location (*i.e.,* the method) where the defect was introduced, and (ii) the time to actually correct the defect completely. We considered the goal as being achieved when all 872 unit tests of Pier ran successfully. The bugs had to be fixed in the right method, thus, they were carefully chosen so that they could only be corrected in this method.

## 4.4 Study Result

Initially we report on the quantitative data we obtained by recording the time the subjects spent to find and correct the defects using the different browsers. Then we evaluate the results from the questionnaire and finally we present qualitative feedback reported by the subjects.

**Time Evaluation.**

Figure 4 compares the average time spent to correct the bugs, the average times were aggregated independent on the used browser or on the order in which the different defect were addressed by the subject. The figure clearly shows that the two bugs were approximately equally complex, thus allowing us to compare the time the different subjects spent in different environments to correct the two defects. We initially selected these two defects after having assessed their complexity in a pre-test with two subjects working in the standard Squeak browser. These two subjects needed approximately the same time to correct both defects and subjectively considered the two bugs as equally complex and difficult to correct.

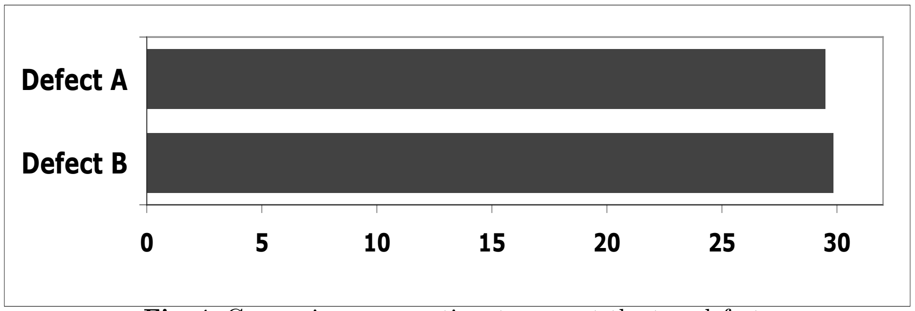


**Fig. 4.** Comparing average time to correct the two defects

Figure 5 compares the total time the subjects spent to discover the location of the defect once using the feature-centric environment and once using the OmniBrowser . The average benefit of using the feature-centric environment is 56 percent which are in total 56 minutes less than using OmniBrowser . During the experiment we considered that the correct location of the defect has been discovered when the subject announces that the defect has to be in the specific faulty method. The subjects were asked to name the faulty method as soon as they believed to have found it. The situation is similar when considering the time spent to fully correct the defects (see also Figure 5), which is the time to discover the defect plus the amount of time to edit and correct the faulty method. Here we get a relative improvement of 33 percent and an absolute of 100 minutes saved when using the feature-centric environment instead of OmniBrowser . Figure 6 presents boxplots showing the distribution of the discovery and correction time the different subjects spent in different browsers. The different defects are not identifiable in these boxplots, only the different browsers. To measure the whole correcting time we considered the bug as being corrected as soon as all the tests of Pier run successfully.

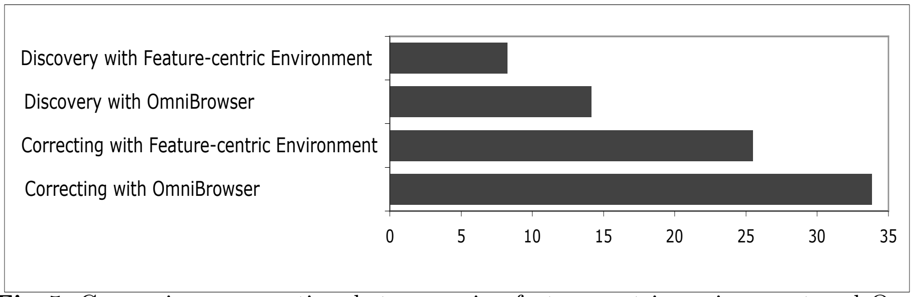


**Fig. 5.** Comparing average time between using feature-centric environment and OmniBrowser to discover resp. correct a defect

**Evaluation of the Questionnaire.**

In our questionnaire we mainly asked the subjects how they rate the effect of the different aspects of the feature-centric environment on program comprehension. We asked about the overall effect on program comprehension, the effect of the feature overview, of the feature tree, feature class browser and of the feature affinity metric. Furthermore, we asked how well certain parts of the feature-centric environment were understood by the subjects and how well they could interact with the different parts. In Table 7 we present the details of our questionnaire and the average results we got from the subjects. They could choose between a rating from -3 to 3, so an average rating of 1.16 for *e.g.,* “General effect on Program Comprehension” reveals a positive, although not a very strong effect. As an example, we depict in Figure 8 the results for the question about

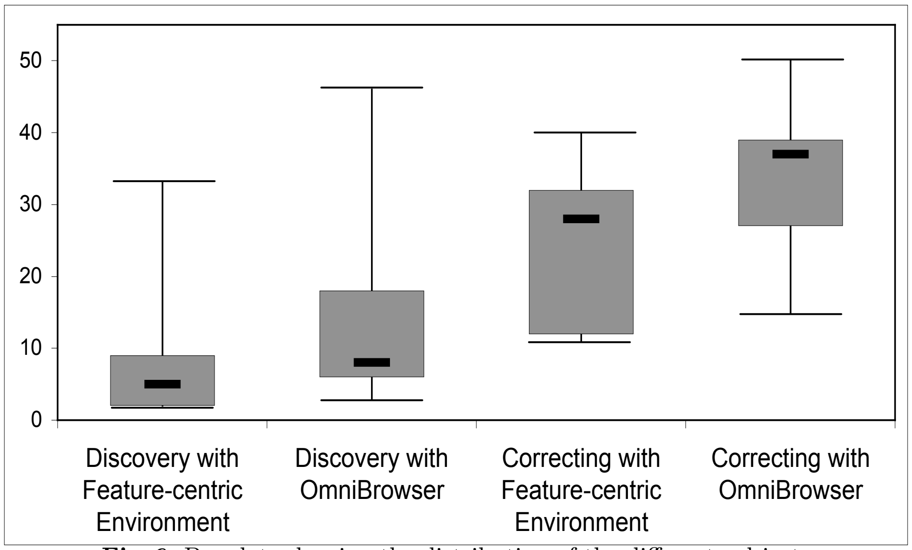


**Fig. 6.** Boxplots showing the distribution of the different subjects

the effect of the compact feature overview on program comprehension. The ratings were in average 1.58 which denotes that the subjects considered the effect in average as "good".

|  | Question | Result |
|---|---|---|
| 1. | General effect on Program Comprehension | 1.16 |
| 2. | Effect of Compact Feature Overview | 1.58 |
| 3. | Effect of Feature Tree Browser | 1.33 |
| 4. | Effect of Feature Class Browser | 1.50 |
| 5. | Effect of Feature Affinity Metric | 1.42 |
| 6. | Understanding the subexpression compression in Feature Tree | 1.58 |
| 7. | Understanding the sequence compression in Feature Tree | 1.75 |
| 8. | Understanding the navigation in Feature Tree | 2.00 |
| 9. | Interaction with Compact Feature View | 1.50 |
| 10. | Interaction between Feature Tree and Feature Class Browser | 1.58 |

**Fig. 7.** Questionnaire.

**Statistical Conclusion.**

To test the first two hypotheses formulated in Table 1 we apply the one-sided independent t-test [17] with an $\alpha$ value of 10% and 22 degrees of freedom. One requirement for applying the t-test is equality of the variance of the two samples. For the two discovery time samples we determine a variance of 92 and 112, respectively, for the correcting time samples the variances are more closes to each other, 102 and 110, respectively. For the correcting time the variance requirement is fulfilled, for the discovery time we are careful and assume that this requirement is not fulfilled. Another requirement for applying the t-test is a normal distribution of the underlying data which we justify with the Kolmogorow-Smirnow-Test. With an $\alpha$ value of 5% the result of this test allows us to assume normal distribution for the correcting time samples, with an $\alpha$ value of 10% we can also assume normal distribution for the discovery time samples.

These preliminary tests allows us to use the t-test at least for the correcting time. We also use it for the discovery time, but we are skeptical about the result in that case. For the discovery time we calculated a t value of 1.32. The t distribution tells us the probability that t > 1.32 is exactly 10% which means that we can just barely reject the null hypothesis $H0_1$. Because the requirements for the t-test are not properly fulfilled and because an $\alpha$ value of 10% is probably to low to justify a rejection of the null hypothesis we cannot prove the positive effect of the feature-centric environment on the discovery time of a defect.

For the time to correct a defect we obtain a t value of 1.86 which is even greater than the t value of the 95% confidence interval (t = 1.717). This means that we can reject $H0_2$ even with an $\alpha$ value of 5% and accept the alternative hypothesis $H1_2$ saying that the feature-centric environment speeds up the time to correct a defect ($\mu_{C,FB} < \mu_{C,OB}$).

To test the third hypothesis we use the results of the questionnaire and apply the one-sided Wilcoxon signed-rank test [18]. We cannot assume normal distribution for the underlying data in this case since the ratings were almost all positive. We only apply the test to the answers for the general effect of the feature-centric environment . We calculate a $W$ value of 26 which is exactly

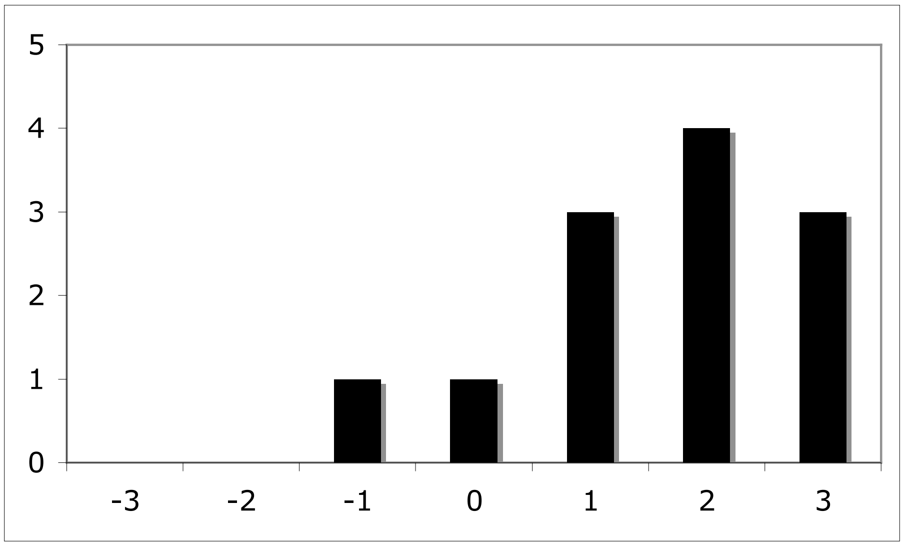


**Fig. 8.** Comparing the average results for the effect of compact feature overview on program comprehension

equal to the S value we find in the tabular denoting the 95% confidence interval. This means that we can reject $H0_3$ with an $\alpha$ value of 5% and hence accept the alternative hypothesis which says that the feature-centric environment has a positive effect on program comprehension.

## 5 Discussion

In this section we report on the main threats to validity of our experiment and on the conclusions we can draw from this study. Furthermore, the subjects provided us with a wide range of constructive criticism and suggestions for improvements to the feature-centric environment , which we also outline in this section.

### 5.1 Threats to Validity

We distinguish between external, internal and construct validity [19].

– *External validity* depends on the subjects and the software system in which subjects are asked to correct defects during the experiment. In our case the software system, Pier, is a complex real-world application comparable to other industrial object-oriented systems. The subjects, however, are students and research assistants who may not be directly comparable to programmers in industry. Furthermore, the experiment was conducted in a laboratory environment which biases the performance of the subjects. Hence the results are not directly applicable to settings in practice, although we consider the aforementioned influences as small.

- *Internal validity* is jeopardized by the fact that not all subjects had the same amount of experience with the programming language and especially with the Squeak development environment. While everybody was quite familiar programming in Small-talk (at least one year of experience), the particular environment was new to some of the subjects. But because these problems were present for both defects and browsers, we believe that they do not bias the results tremendously. Furthermore, we changed the order of the bugs and browsers from subject to subject, hence the results were only slightly biased by the fact that subjects had more insight into the development environment and also into the software system when fixing the second bug than they had for the first bug. However, the different amount of experience of the subjects is nonetheless a shortcoming of this study.
  Another important issue is that subjects could also use other tools than just the two different browsers, *e.g.,* the debugger. As it is usually necessary to use a debugger in a dynamic language to find a bug, we did not prohibit its usage. Some subjects performed the task predominantly by using the debugger to find the bug whereas they made the necessary corrections often in the provided browser. These other available tools clearly bias the result of our experiment to a degree which is hard to estimate.
  Yet another important issue is that the subjects did not know or use the feature-centric environment previously. It is a complete new environment with a very different approach to look at a software system and its features. The other environment (*i.e.,* the OmniBrowser) was well-known by all subjects, since the paradigm applied in this browser is the standard way of browsing source code in most Smalltalk dialects (and indeed for other object-oriented languages).

- *Construct validity.* Our measure, the time to find and correct a defect, is adequate to assess the contribution of the feature-centric environment to maintenance performance. However, this time is certainly biased by many other factors than the browsers in use, such as the experience of the developer, her motivation, the use of other tools, etc. To assess the effect of the feature-centric environment on program comprehension we used our questionnaire to obtain feedback on how the subjects personally judge the effects on program comprehension. These answers are certainly subjective and may hence not be representative. Thus the applied measures are not a perfect assessment of the effects of the feature-centric environment on maintenance performance and on program comprehension, although at least the former is still relatively well assessed with the applied measures.

### 5.2 Study Conclusion

Two main issues of our study are: (1) the subjects participating the experiment do not have the same experience with the programming language and the development environment, and (2) they were unfamiliar with our feature-centric environment as they had never used it before. On the other hand they are familiar with the standard environment. Another important issue is that due to the limited number of subjects participating in the experiment, it is difficult to draw statistically firm conclusions. This study nonetheless gives us worthwhile insights into how software engineers use and judge our feature-centric environment and defines a framework for further studies of this kind. The results we obtained motivate us to proceed with our work on this environment and the subject feedback provides us also several new ideas and approaches that we intend to pursue. We conclude that performing this study was crucial to validate and improve our work.

We outline in the following three interesting ideas and suggestions we obtained as feedback from the subjects who participated in the experiment:

– *Bidirectional interactions.* Providing the capability to navigate the tree by clicking and selecting the textual representations of the methods in the feature artifact browser and being able to click on nodes in the tree to select the same methods in the compact feature view is a useful improvement to the feature-centric environment . This helps one to navigate and understand more quickly the structure and implementation of a feature.
– *Bind tree to debugger.* Using a debugger is not an easy task since we only see a slice of a program in the debugger but not the overall structure. If we could use the feature tree to step through a running program to debug it, we would get a better overview and understanding of the overall structure of a feature. Hence a promising extension of the feature-centric environment would be to add debugging facilities to the feature tree, such as stepping through a program, inspecting variables and changing methods on the fly.
– *Delta debugging.* Using the feature-centric environment to discover and correct a defect in a feature is a frequent task which we can ease by analyzing test cases representing features. Careful analysis of test cases with delta debugging approaches [20] allows us to rate methods used in a feature according to their probability being responsible for a defect. If we present the methods in the compact feature view sorted by their probability to contain erroneous code a developer can very quickly focus on the right methods to correct a defect.

## 6 Related Work

The key focus of our work is on understanding object-oriented software, with a particular emphasis on the role of features to support program comprehension. Many researchers have identified the problem of understanding object-oriented software [3, 4, 21, 5]. Furthermore, the potential of feature-centric approaches in software engineering to understand complex software systems has recently generated much interest [22, 10].

Wilde and Huitt [5] described several problem areas related to object-oriented code and they suggested tracing dependencies was vital for effective software maintenance. To fully understand a method, its context of use must be found by tracing back the chain of messages that reach it, together with the various chains of methods invoked by its body [5].

Nielson and Richards [23] investigated how difficult it was for experienced software engineers to learn and use the Smalltalk programming language. They found the distributed nature of the code caused problems when attempting to understand a system.

Chen et. al. also highlighted class hierarchies and messages as posing difficulties when trying to maintain object-oriented code [24].

Some researchers have performed experiments obtaining evidence to support their claim that understanding object-oriented software to perform maintenance tasks is difficult.

Brade et. al. [25] presented a tool to provide explicit support for delocalized plans (conceptually related code that is not localized contiguously in a program). They developed a tool called Whorf which provided hypertext links between views of the software to highlight interactions between physically disparate components. They performed an experiment with software engineers and measured how quickly it took them to identify relevant code to perform an enhancement to a software system, once with paper documentation and once with their Whorf tool. Their results show that the the Whorf tool improved efficiency to perform the maintenance task. The focus of this work was to support documentation strategies for object-oriented programs.

Dunsmore et. al. [4] highlighted the shortcomings of traditional code inspection techniques when faced with the problem of delocalization in object-oriented software. They present the results of an empirical investigation to measure how long it took software engineers to locate defects in a software system.

In contrast to the work of Dunsmore et. al. and Brade et. al. [25, 4], our feature-centric environment provides the software engineer integrated development environment support to locate the parts of the system that are relevant to a feature.

Substantial research has been conducted on runtime information visualization to understand object-oriented software [26, 12, 27, 28]. Various tools and approaches make use of dynamic (trace-based) information such as Program Explorer [29], Jinsight and its ancestors [30], and Graphtrace [27]. De Pauw et. al. present two visualization techniques. In their tool Jinsight, they focused on interaction diagrams [30]. Thus all interactions between objects are visualized.

Reiss [28] developed *Jive* to visualize the runtime activity of Java programs. The focus of this tool was to visually represent runtime activity in real time. The goal of this work is to support software development activities such as debugging and performance optimizations.

The focus of our feature-centric environment was to incorporate interactive and navigable visualizations of the dynamic behavior of features into the development environment.

## 7 Conclusion

In this paper we presented our feature-centric environment which allows us (1) to visually compare several features of a software system, (2) to visually analyze the dynamic structure of a single feature in detail and, (3) to navigate, browse and modify the source artifacts of a single feature in a feature artifact browser focusing on the entities actually used in that feature. All these visualizations are enriched with the feature affinity metric to highlight parts of a feature relevant to a specific maintenance task.

The views on features are fully interactive and interconnected to ease and enhance their usage in maintenance activities. We validated our feature-centric environment by carrying out an empirical study with twelve graduate computer science students. The results of our experiment are promising because they clearly reveal that the feature-centric environment has a positive effect on program comprehension and in particular on the efficiency in discovering the exact locations of software defects and in correcting them efficiently. We recognize that as our experiment had only a low number of participating subjects, it is difficult to generalize the results. However, feedback of the users in addition to the quantitative results of our analysis are encouraging. This motivates us to continue investigation and development of a feature-centric environment that represents features as first-class entities for the software engineer. In particular, we aim to focus on ideas mentioned in Section 5.2, for example providing bidirectional interaction to the feature-centric environment or to add debugging facilities to the feature tree.

## References

1. Basili, V.: Evolving and packaging reading technologies. Journal Systems and Software **38**(1) (1997) 3–12
2. Corbi, T.A.: Program understanding: Challenge for the 1990's. IBM Systems Journal **28**(2) (1989) 294–306
3. Demeyer, S., Ducasse, S., Mens, K., Trifu, A., Vasa, R.: Report of the ECOOP'03 workshop on object-oriented reengineering (2003)
4. Dunsmore, A., Roper, M., Wood, M.: Object-oriented inspection in the face of delocalisation. In: Proceedings of ICSE '00 (22nd International Conference on Software Engineering), ACM Press (2000) 467–476
5. Wilde, N., Huitt, R.: Maintenance support for object-oriented programs. IEEE Transactions on Software Engineering **SE-18**(12) (December 1992) 1038–1044
6. Nielsen, J., Richards, J.T.: The Experience of Learning and Using Smalltalk. IEEE Software **6**(3) (1989) 73–77
7. Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., Kay, A.: Back to the future: The story of Squeak, A practical Smalltalk written in itself. In: Proceedings OOPSLA '97, ACM SIGPLAN Notices, ACM Press (November 1997) 318–326
8. : Cincom Smalltalk (September 2003) http://www.cincom.com/scripts/smalltalk.dll/.
9. Mehta, A., Heineman, G.: Evolving legacy systems features using regression test cases and components. In: Proceedings ACM International Workshop on Principles of Software Evolution, New York NY, ACM Press (2002) 190–193

10. Eisenbarth, T., Koschke, R., Simon, D.: Locating features in source code. IEEE Computer **29**(3) (March 2003) 210–224
11. Renggli, L.: Magritte – meta-described web application development. Master's thesis, University of Bern (June 2006)
12. Jerding, D., Stasko, J., Ball, T.: Visualizing message patterns in object-oriented program executions. Technical Report GIT-GVU-96-15, Georgia Institute of Technology (May 1996)
13. Licata, D., Harris, C., Krishnamurthi, S.: The feature signatures of evolving programs. In: Proceedings IEEE International Conference on Automated Software Engineering, Los Alamitos CA, IEEE Computer Society Press (October 2003) 281–285
14. Greevy, O.: Enriching Reverse Engineering with Feature Analysis. PhD thesis, University of Berne (May 2007)
15. Philippe Flajolet, Paolo Sipala, J.M.S.: Analytic variations on the common subexpression problem. In: Automata, Languages, and Programming. Volume 443 of LNCS., Springer Verlag (1990) 220–234
16. Hamou-Lhadj, A., Lethbridge, T.: An efficient algorithm for detecting patterns in traces of procedure calls. In: Proceedings of 1st International Workshop on Dynamic Analysis (WODA). (May 2003)
17. Kanji, G.K.: 100 Statistical Tests. SAGE Publications (1999)
18. Wilcoxon, F.: Individual Comparisons by Ranking Methods. International Biometric Society (1945)
19. O'Brien, M., Buckley, J., Exton, C.: Empirically studying software practitioners - bridging the gap between theory and practice. In: Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005), IEEE Computer Society Press (2005)
20. Zeller, A.: Isolating cause-effect chains from computer programs. In: SIGSOFT '02/FSE-10: Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering, New York, NY, USA, ACM Press (2002) 1–10
21. Hamou-Lhadj, A., Braun, E., Amyot, D., Lethbridge, T.: Recovering behavioral design models from execution traces. In: Proceedings IEEE European Conference on Software Maintenance and Reengineering (CSMR 2005), Los Alamitos CA, IEEE Computer Society Press (2005) 112–121
22. Antoniol, G., Guéhéneuc, Y.G.: Feature identification: a novel approach and a case study. In: Proceedings IEEE International Conference on Software Maintenance (ICSM 2005), Los Alamitos CA, IEEE Computer Society Press (September 2005) 357–366
23. Nielson, F.: The typed lambda-calculus with first-class processes. In Odijk, E., Syre, J.C., eds.: Proceedings PARLE '89, Vol II. LNCS 366, Eindhoven, Springer-Verlag (June 1989) 357–373
24. Chen, J.B., Lee, S.C.: Generation and Reorganization of Subtype hierarchies. Journal of Object Oriented Programming **8**(8) (January 1996) 26–35
25. Brade, K., Guzdial, M., Steckel, M., Soloway, E.: Worf: A visualization tool for software maintenance. In: Proceedings of IEEE Workshop on Visual Languages, IEEE Society Press (1992) 148–154
26. Greevy, O., Lanza, M., Wysseier, C.: Visualizing live software systems in 3D. In: Proceedings of SoftVis 2006 (ACM Symposium on Software Visualization). (September 2006)
27. Kleyn, M.F., Gingrich, P.C.: GraphTrace — understanding object-oriented systems using concurrently animated views. In: Proceedings OOPSLA '88 (International

Conference on Object-Oriented Programming Systems, Languages, and Applications. Volume 23., ACM Press (November 1988) 191–205
28. Reiss, S.P.: Visualizing Java in action. In: Proceedings of SoftVis 2003 (ACM Symposium on Software Visualization). (2003) 57–66
29. Lange, D., Nakamura, Y.: Interactive visualization of design patterns can help in framework understanding. In: Proceedings ACM International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 1995), New York NY, ACM Press (1995) 342–357
30. De Pauw, W., Helm, R., Kimelman, D., Vlissides, J.: Visualizing the behavior of object-oriented systems. In: Proceedings OOPSLA '93. (October 1993) 326–337

# Bridging the Gap Between Morphic Visual Programming and Smalltalk Code

Noury Bouraqadi[1] and Serge Stinckwich[2]

[1] École des Mines de Douai, France
bouraqadi@ensm-douai.fr
[2] GREYC - Université de Caen, France
Serge.Stinckwich@info.unicaen.fr

**Abstract.** In this paper, we claim that both prototype-based visual programming and traditional Smalltalk class-based programming are required for developing applications with a GUI. We introduce Easy Morphic GUI (EMG), a framework that connects Morphic and EToys visual manipulation and scripting facilities with the usual Smalltalk development environment tools. The Squeak platform is used here as a playfield for our experiments. A step-by-step tutorial is used to illustrate the main aspects of the EMG framework. We also introduce two reuse operators: EMBED and CLONE in order to build new GUIs out of existing ones. EMBED inserts a GUI into another one, while CLONE makes the destination look the same as the original. Static and dynamic version of these operators are also investigated.
**Keywords:** Visual Programming, GUI Building, Reuse, Prototype.

## 1 Introduction

### 1.1 Context: Squeak and Morphic

Squeak [1] is a feature-rich, platform-independent and open source implementation of the Smalltalk programming environment, whose virtual machine is entirely written in Smalltalk. Squeak includes network and multimedia (sound, graphics, video, ...) support, an integrated development environment similar to others Smalltalk flavors and a constructivist learning environment for children called EToys, based on a GUI model named Morphic.

Morphic[2] was invented for the Self [3] prototype-based programming language. Self used to be the successor language to Smalltalk. The Morphic user interface was developped by Randall Smith and John Maloney. And when John left the Self project, he ported Morphic to Squeak as a replacement for MVC. Every display object (windows, menus, ...) in Morphic is a Morph, i.e an instance of a subclass of the class Morph.

Following the Smalltalk philosophy, Morphic objects are uniform: they have the same basic structure (e.g. color, position, extent) and can be manipulated in the same way. Because morphs are concrete objects, they can be manipulated visually as real objects. Each morph can be selected in order to bring up a set of icons, called *halos*, which allow visual manipulation (e.g. moving, rotating, resizing, cloning, changing colors or layout). Being robust, morphs still work after being manipulated, even if their geometric properties (e.g. rotation or a resizing) are impacted.

On top of Morphic we find EToys, a powerful scripting visual programming language. EToys provide an interface where the developer program by dragging and dropping together small snipets of code which directly manipulate visual objects. There is no separation between the GUI and the model. Hence, EToys open up the opportunity to explore visual programming but in a rather constrained environment (some programs are difficult if not impossible to code using EToys mainly because many objects and messages are unavailable for visual programming). EToys programming share also similarities with event-based programming or context-based programming [4], where form and function are usually merged [5].

EToys can be classified as a *direct manipulation interface*, a term coined in 1983 by Ben Schneiderman [6]. That is a human-computer interaction style that allow users to directly manipulate objects presented to them with actions that resemble to physical one. The advantages are numerous. The user has at his disposal at a given moment only the behaviors associated with the object which he handles.

Though interesting, we believe that the power of Morphic and EToys is not fully unleashed.

### 1.2 Motivation

When programming in Squeak, developers are provided with two programming interfaces. On the one hand Morphs can be manipulated and composed visually through halos and different pop-up menus. On the other hand, Browsers allow writing Smalltalk code. However, when it cames to building software with GUI, developers often code the full GUI as plain Smalltalk code. This task is cumbersome. Therefore, they lose the benefit of Smalltalk dynamicity and incremental programming. They have to fully code the GUI description, then create an instance. For example, if the look or the layout is not appropriated, they have to delete the GUI instance, change the code and then recreate the instance to check whether it fits the desired visual properties.

When using Morphic visual capabilities, one can instantiate some morphs and even compose them. But, the link with code has to be done in an ad hoc manner. No guidelines are provided. More importantly, the created morph description can not be stored together with code through for instance change sets or Monticello repositories. Morphic does allow exporting morphs in files, but the code is not included in those files. Squeak also allows exporting a whole project through the "image segment" concept. However, the obtained file contain all objects only accessible from the roots of the image segment. As a consequence, added classes are often ignored on serialization into image segment.

### 1.3 Requirements

In this paper, we introduce Easy Morphic GUI[3] (EMG) a framework to bridge the gap between, on the one hand Morphic and EToys visual manipulation and scripting facilities, and on the other hand “plain” Smalltalk coding activities. Our goal is to integrate more smoothly the UI design into the incremental style of development of Smalltalk. The key requirements for this integration are:

**Use visual programming tools for GUI development.** Building interfaces is a rather boring and time-consuming operation, because they often look similar. By dragging and dropping existing widgets, it is possible to assemble complex programs in literally seconds rather than by specifying them textually. A lot of commercial development environments like VisualBasic or Delphi already exploit this advantage for fast prototyping applications.

**Use Smalltalk powerful development tools for coding business classes.** "Turtles all the way down" is not necessarily a good choice for UI design: not everything should be implemented visually. Smalltalk already provides very powerful development tools (class browser, instance inspectors, ...) and we like to stick with these tools in order to implement domain specific classes.

**Save GUIs together with the application code.** Smalltalk also provides suitable solutions for code storage and project management, like change sets, Monticello (Squeak). But usually, preserving the user interface is not that simple. It requires either a tool that converts UI into code or one that stores code as objects (not the case for Squeak).

**Support GUI versioning to allow roll backs during the project life-time.** Designing a new UI is an incremental task that need several test and try, before the final build is obtained.

**Provide operators to reuse GUIs.** As we already mentioned, interfaces often look the same. So like domain code that can be reused by subclassing or composing existing classes, GUIs should be reusable in other contexts than the one where they were first defined.

In the remainder of this paper we first provide an overview of the EMG framework (section 2) where we show how EMG does satisfy the four first requirements. EMG features are illustrated by a step-by-step example. Section 3 describes how EMG satisfies the last requirement. We present GUI reuse operators and their functioning through some examples. Next, we present in section 4 work related to EMG. Finally, future works and perspectives are drawn together with the conclusion in section 5.

## 2 An Overview of Easy Morphic GUI

The starting point of Easy Morphic GUI (EMG) is that morphs are naturally manipulated as individual objects while Smalltalk code is mainly class centered.

[3] Freely available at http://csl.ensm-douai.fr/EasyMorphicGUI

EMG bridges the gap between those two worlds by mixing prototype-based programming languages concepts with the class related ones. On the one hand, developers build the GUI as a prototype by visual manipulations but on the other hand, business code is implemented by coding the appropriate classes. EMG, as describe below, provides a framework that allows linking the two parts.

### 2.1 Description

Essential to the EMG framework is the EMGGuiMorph, the root of the GUI classes hierarchy. We call GUIs, instances of EMGGuiMorph and its subclasses. GUIs are morphs that act as containers for other morphs (called sub-morphs) dedicated to human-machine interaction. Besides, GUIs hold instance variables and methods that allow accessing business objects. Hence, they act as glue between visual objects (i.e. morphs) and business ones.

Each GUI class has a special instance called *prototype*[4]. Instances of a GUI class are created as copies of the prototype of that class. In order to build a GUI, developers subclass EMGGuiMorph and then visually setup the prototype. EMGGuiMorph extends default Morphic halos with menus that eases the creation of new morphs and their layout. Note that because we are building the GUI visually, we can use the full power of Morphic and related libraries such as EToys.

EMG relies on the *Mediator* design pattern [7] for connecting visual objects to business ones. GUIs act as a mediator that encapsulates the interaction between morphs and business objects. References from morphs to GUIs are set in an interactive way, through pop-up menus, or if not available using inspectors[5]. For example, buttons implemented by morphic developers have a menu to setup the message to be sent on a click. The menu allow choosing the message receiver among available morphs, and provide the selector and the arguments. When using EMG, the receiver should be the GUI. Of course, this means that the GUI class have to implement methods to be called for mediation.

References from business objects to the GUI, often based on the *Observer* design pattern, are set in the GUI class. This is performed either in the initialize method or through lazy initialization. Of course business objects can refer to each other directly. Symmetrically, morphs belonging to the same GUI can reference each other straightly. However, references between business objects are set within business classes code, while references between morphs are set interactively through pop-up menus or inspectors.

### 2.2 A First Simple Example Step-by-Step

We present here how to build a GUI for a counter using EMG. Our goal is to have a counter that can be handled through the GUI shown on figure 1. The

[4] The prototype can be accessed through a class instance variable (i.e. an instance variable declared in the metaclass).

[5] Morph the superclass of all morphs is extended with a few facility methods that allow retrieving the GUI to which a morph belongs.

counter is incremented by a click on the "+" button and decremented once the "-" button is selected. The text field both displays the counter's value and allows editing it.

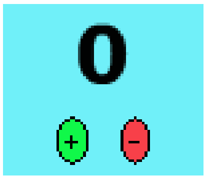


**Fig. 1.** The counter GUI

The business code for our example is a trivial counter class. The corresponding code is provided by figure 2. Nothing special to mention except that the EMGCounter class inherits from Model. It notifies its dependents when the count instance variable changes (see line 12). Our goal is to have the GUI be registered as a dependent and be updated when the counter changes.

Figure 3 provides the definition of EMGCounterGUI, the GUI class for our counter. We can see that EMGCounterGUI inherits from EMGGuiMorph the support for prototype management and related operators (see section 2.3 for more details about EMGGuiMorph). The link to counter is done through an instance variable (line 2) which is set on creation time.

The counter instance variable setter (lines 10–15) registers the GUI as a dependent of the counter. Then, it updates the display through the updateDisplay message. The updateDisplay message is part of the EMG API. It is sent by the update: method in class EMGGuiMorph in order to keep the GUI display coherent state with business objects. In our example, the updateDisplay message ensures that the text field named countText displays the actual value of the counter. The text field is retrieved using the submorphRecursivelyNamed: message (see line 25). This message is implemented by the EMGGuiMorph based on the Null Object Pattern [8]. This pattern allows displaying and testing GUIs in early development stages. Indeed, since the GUIs are build visually, they are empty when prototypes are first created. However, a GUI class may contain some references to submorphs using the submorphRecursivelyNamed: message (cf. line 25 of fig 3). If no submorph holds the provided name, the answer of this message is an object instance of EMGUndefinedMorph. This class redefines the doesNotUnderstand: method in order to notify the developer that a message is not understood. Developers can ignore these messages and proceed with the execution.

Class EMGCounterGUI also provides methods for user interaction. Methods increase (lines 30–31) and decrease (lines 27–28) implement actions to perform when clicking on buttons. Method countFromText: (lines 33–38) aims at updating the counter when the text field content is modified. In order to notify the user that the modification is recorded, we make the counter GUI flashes (line 38).

```
Model subclass: #EMGCounter
    instanceVariableNames: 'count'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'EasyMorphicGUI−Counter Example'

EMGCounter >> count
    ↑ count

EMGCounter >> count: newValue
    count := newValue.
    self changed

EMGCounter >> initialize
    super initialize.
    self count: 0

EMGCounter >> increment
    self count: self count + 1

EMGCounter >> decrement
    self count: self count − 1
```

**Fig. 2.** Definition of the counters class

```
EMGGuiMorph subclass: #EMGCounterGUI
    instanceVariableNames: 'counter'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'EasyMorphicGUI−Counter Example'

EMGCounterGUI >> counter
    ↑ counter

EMGCounterGUI >> counter: newCounter
    counter ifNotNil: [counter removeDependent: self].
    counter := newCounter.
    counter ifNotNil: [
        counter addDependent: self.
        self updateDisplay]

EMGCounterGUI >> createCounter
    ↑ EMGCounter new

EMGCounterGUI >> initialize
    super initialize.
    self counter: self createCounter

EMGCounterGUI >> countTextField
    ↑ self submorphRecursivelyNamed: #countTextField

EMGCounterGUI >> decrease
    self counter decrement

EMGCounterGUI >> increase
    self counter increment

EMGCounterGUI >> countFromText: aText
    | newCount |
    newCount := aText asString asInteger.
    newCount ifNil: [↑ self].
    self counter count: newCount.
    self flash

EMGCounterGUI >> updateDisplay
    super updateDisplay.
    self countTextfield
        contents: self counter count printString
```

**Fig. 3.** Definition of the counter's GUI class

Now we can build visually the GUI. The following expression allows displaying the counter's prototype.

EMGCounterGUI openPrototypeInWorld.

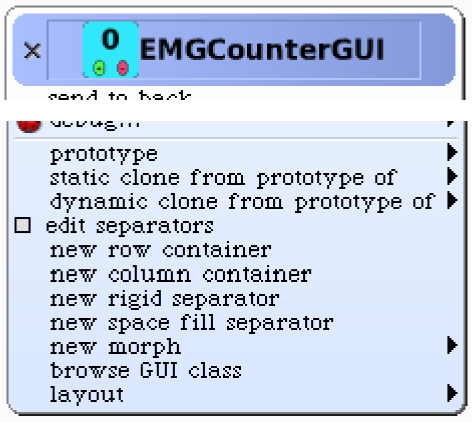


**Fig. 4.** Morphic halos are extended with menus to build and manage the GUI's prototype

On creation, the prototype is but an empty rectangle. The missing buttons and text field can be created from the World menu[6] or the "Parts Bin"[7] and dropped into the prototype. Alternatively the GUI morphic halos can be used (see figure 4). We have extended them with menus helping the construction of GUIs. Besides, we rely on Squeak support for visual operations undo.

Created morphs have to be setup to finish linking them to business code. We rely for this on existing morphic capabilities. For example, the name[8] of the text morph has to be set to "countTextField" (Figure 5-a). The label, action selector and target of buttons for counter incrementing / decrementing can be set through morphic halos menus (Figure 5-b). Other properties such as colors and layout can be setting in a similar way. Once terminated, the prototype has to be saved through our extension of morphic menus (Figure 5-c).

## 2.3 GUI Prototype Management

As said above, GUIs are built based on a prototype. We rely then on the "Prototype" design pattern. So, new instances are created by deep copying the prototype.

[6] Sub-menu "new morph..."

[7] Parts Bin is a visual repository of existing Morphs that can be opened from the World menu, sub-menu "objects".

[8] Actually, it is the morph's "external name".

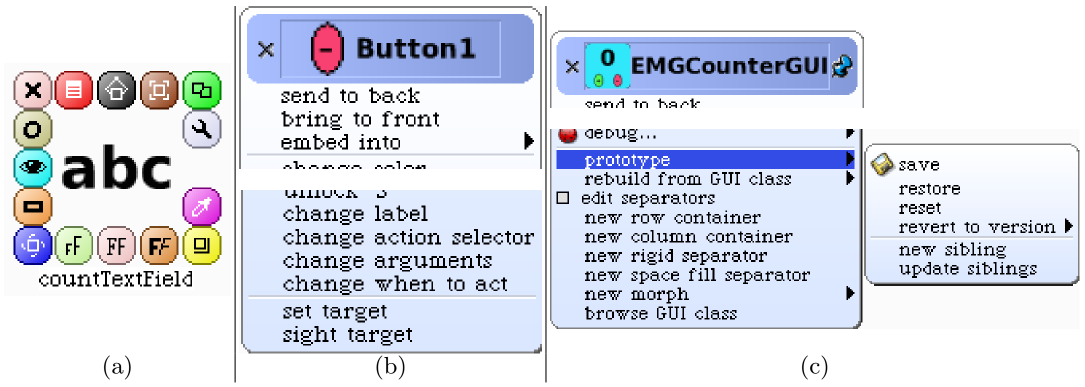


**Fig. 5.** Setting up morphs for the counter's GUI

The "Prototype" design pattern leaves the developers free about where to store prototypes and how to copy them. In our case, we choose to have prototypes stored at the class level. The metaclass of EMGGuiMorph does define an instance variable to reference the prototype. The new method is redefined to use this instance variable and copy the prototype for creating new instances.

However, referencing the prototype in class instance variables is not enough. Indeed, the prototype is lost on file outs or when we commit with project management system like Monticello. To avoid this situation, we introduced a save–restore mechanism accessible through the prototype's morphic menu (see Figure 5-c). It does rely on storing a serialized form of the prototype in a class method. To avoid the literals count limits fixed by the Smalltalk compiler, the byte array produced by the serialization is compressed and stored as a string. Therefore, when filing in a GUI class, the prototype can be restored by retrieving the prototype bytes and deserializing them.

An interesting side effect of this storage solution is GUI automatic version management. We rely here on Squeak automatic method versioning system to provide GUI developers with the ability to revert back to old versions. This feature together with Morphic undo support contributes to make GUI development even more comfortable.

## 3 GUI Reuse

When building interactive software, developers have to deal with at least three concerns: the business, the GUI and their interactions. The business concern is implemented through objects such as the counter in our previous example. The GUI concern is implemented through morphs. Interactions between morphs and business objects reified as mediators instances of EMGGuiMorph and its subclasses.

Through the business – mediator – GUI decomposition, EMG enables separation of concerns when building some software. Development can be separated in time and distributed between different developers. The most appropriate tools can be used for each part such as visual operations for GUI and browsers for the rest. EMG does also enable separation of concerns when it come to reuse. Business or mediator classes can be reused through usual reuse operations, and particularly inheritance. In this section, we focus on GUI reuse.

## 3.1 GUI Reuse Operators

Reuse operators allow building new GUIs out of existing ones. Our proposal is based on two binary operators: EMBED and CLONE.

Each one of these operators requires two operands. The first one, called *source*, is the GUI to reuse. It remains unchanged once the construction is over. The second operand, called *destination*, is the GUI that is being built. Its appearance including the set of submorphs it contains is changed by the reuse operator. It is important to stress that the two operands may be instances of unrelated classes. The only constraint is that they should be GUIs, i.e. their classes should inherit from EMGGuiMorph.

The EMBED operator inserts the source GUI into the destination. As opposite to CLONE which totally changes the destination appearance, the EMBED operator extends the destination. The only modification of the destination is the addition of the source GUI as a submorph. Other destination submorphs, its color and its size to name a few, are not changed. Therefore, if EMBED is applied several times on the same destination with different sources, the destination will include all sources.

The CLONE operator makes an already existing destination GUI have precisely the same appearance as the source one (e.g. same color, dimension, layout and submorphs[9]). If CLONE is applied several times on the same destination with different sources, the destination GUI will end up with same appearance as the last used source GUI.

We distinguish two variants for the CLONE operator: Static-CLONE and Dynamic-CLONE. The Static-CLONE variant does only perform the cloning. The Dynamic-CLONE operator goes beyond. In addition to cloning, it also sets up a dependency link between source and destination GUIs. Whenever source appearance evolves, the destination is updated.

We have chosen not to propose a dynamic variant for the EMBED operator. This decision is based on the observation that an embedded GUI is often visually adapted to fit with its container and other morphs. An unsupervised update may break the appearance of the container. However, developers still can do updates manually through our extension of morphic halo menus.

For the same reason, we chose to keep by default the prototype–sibling relationship "static". New instances of a GUI class are created by cloning the class's

[9] Submorphs are copied.

prototype using the Static-CLONE operator. However, prototypes provide a morphic menu to update all their siblings. Symmetrically, developers can update a single GUI (through a menu) from its prototype. They also can also make the GUI become a "dynamic clone" of the prototype. Actually, these operations are not restricted to the prototype of the same class. A GUI can be changed to resemble the prototype of any other class.

Which operator should the developer use ? The first one (EMBED) is more suitable if you need to build a composite GUI based on several existing ones (see the alarm example below). The second one (CLONE) will be preferred when a new model is build (see for example the circular counter below).

## 3.2 Examples

**Reusing the counter GUI with the CLONE Operator** In this example, we show how we construct a circular counter GUI by cloning it. Figure 6 provide the implementation of the EMGCircularCounter class. Our goal is to build a simple GUI for circular counters that has exactly the same appearance and interactions as the GUI for plain counters. Therefore we simply create a subclass of EMGCounterGUI which uses EMGCircularCounter as shown in figure 7. Last, we make the prototype of EMGSimpleCircularCounterGUI be a dynamic clone of the prototype of EMGCounterGUI by mean of a menu. Figure 8-a shows the menu. The resulting circular counter GUI is presented on figure 8-b. It is worth noting that the inheritance link between EMGSimpleCircularCounterGUI and EMGCircularCounter, and the clone link between their prototypes are totally decoupled. A prototype of another class can be used as a source for the CLONE operator.

**Reusing the counter GUI with the EMBED Operator** In this example, we build a simple alarm. Its GUI is constructed by embedding two simple circular counters GUI described in section 3.2. Again we start by implementing business object which correspond here to the EMGAlarm as shown in figure 9.

Next we implement the GUI class EMGAlarmGUI as shown on figure 10. The run method is performed by a button in the GUI (see figure 12). Besides the business part which is setting up and starting the alarm, this method also makes some actions on the GUI. The run button is locked and a colon label blinks until the alarm rings (i.e. when the run message to the alarm returns).

Blinking is not part of label morph's behavior. We introduced it using EToys to demonstrate the use of visual programming in EMG and how to link it to the GUI code. Scripts we implemented for this example are shown on figure 11.

Finally, the alarm GUI prototype is built by embedding instances of EMGSimpleCircularCounterGUI. This operation is performed simply through a drag and drop thanks to Morphic visual operations (see figure 12-a). The resulting GUI is shown on figure 12-b.

```
EMGCounter subclass: #EMGCircularCounter
    instanceVariableNames: 'min max'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'EasyMorphicGUI−Counter Example'

EMGCircularCounter>>initialize
    self min: 0.
    self max: 9.
    super initialize

EMGCircularCounter>>valueInRange: integer
    integer > self max
        ifTrue: [↑ self min].
    integer < self min
        ifTrue: [↑ self max].
    ↑ integer

EMGCircularCounter>>count: newValue
    | actualNewValue |
    actualNewValue := self valueInRange: newValue.
    super count: actualNewValue

EMGCircularCounter>>max
    ↑ max

EMGCircularCounter>>max: newMax
    max := newMax

EMGCircularCounter>>min
    ↑ min

EMGCircularCounter>>min: newMin
    min := newMin

EMGCircularCounter class>>min: min max: max
    ↑ self new
        min: min;
        max: max;
        yourself
```

**Fig. 6.** Implementation of circular counters

```
EMGCounterGUI subclass: #EMGSimpleCircularCounterGUI
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'EasyMorphicGUI−Counter Example'

EMGSimpleCircularCounterGUI>>createCounter
    ↑ EMGCircularCounter min: 0 max: 9
```

**Fig. 7.** Implementation of simple GUIs for circular counters

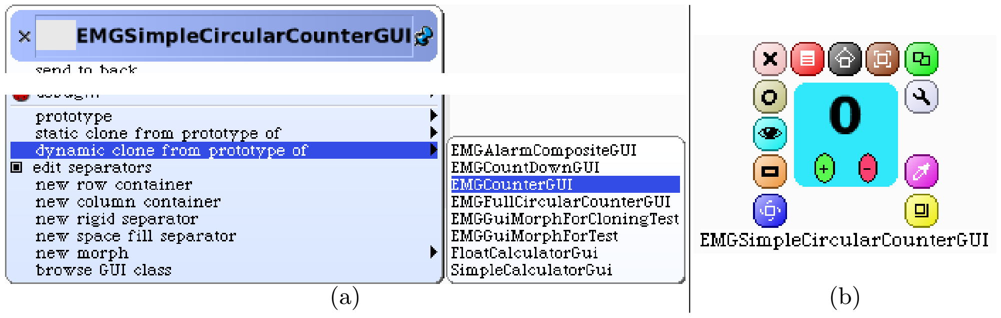


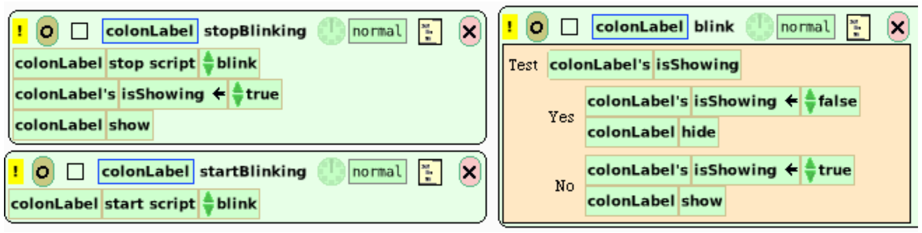
**Fig. 8.** Building circular counter's GUI by dynamic cloning

```
Model subclass: #EMGAlarm
    instanceVariableNames: 'hour minute'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'EasyMorphicGUI−Alarm Example'

EMGAlarm>>initialize
    super initialize.
    self hour: 0 minute: 0

EMGAlarm>>hour: newHour minute: newMinute
    hour := newHour.
    minute := newMinute.
    self changed

EMGAlarm>>hour
    ↑ hour

EMGAlarm>>minute
    ↑ minute

EMGAlarm>>isTimeToRing
    | now |
    now := Time now.
    ↑ now hour = self hour and: [now minute >= self minute]

EMGAlarm>>ring
    AbstractSound stereoBachFugue play

EMGAlarm>>run
    | delay |
    delay := Delay forSeconds: 1.
    [self isTimeToRing] whileFalse: [delay wait].
    self ring
```

**Fig. 9.** Implementation of alarm

```
EMGGuiMorph subclass: #EMGAlarmGUI
    instanceVariableNames: 'alarm'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'EasyMorphicGUI−Alarm Example'

EMGAlarmGUI>>initialize
    super initialize.
    self alarm: EMGAlarm new

EMGAlarmGUI>>alarm
    ↑ alarm

EMGAlarmGUI>>alarm: newAlarm
    alarm ifNotNil: [alarm removeDependent: self].
    alarm := newAlarm.
    alarm ifNotNil: [alarm addDependent: self].

EMGAlarmGUI>>hourCounter
    | hourCounterGUI |
    hourCounterGUI := self submorphRecursivelyNamed: #hourCounter.
    ↑ hourCounterGUI counter

EMGAlarmGUI>>minuteCounter
    | minuteCounterGUI |
    minuteCounterGUI := self submorphRecursivelyNamed: #minuteCounter.
    ↑ minuteCounterGUI counter

EMGAlarmGUI>>updateDisplay
    super updateDisplay.
    self hourCounter count: self alarm hour.
    self minuteCounter count: self alarm minute

EMGAlarmGUI>>run
    | runButton colonLabelPlayer |
    self alarm
        hour: self hourCounter count
        minute: self minuteCounter count.
    runButton := (self submorphRecursivelyNamed: #runButton).
    runButton lock.
    colonLabelPlayer := (self submorphRecursivelyNamed: #colonLabel) player.
    colonLabelPlayer startBlinking.
    [self alarm run.
    runButton unlock.
    colonLabelPlayer stopBlinking.
    ] fork
```

**Fig. 10.** Implementation of alarm's GUI class



colonLabel stopBlinking normal
colonLabel stop script blink
colonLabel's isShowing ← true
colonLabel show

colonLabel startBlinking normal
colonLabel start script blink

colonLabel blink normal
Test colonLabel's isShowing
Yes colonLabel's isShowing ← false
colonLabel hide
No colonLabel's isShowing ← true
colonLabel show

**Fig. 11.** EToys scripts used for alarm GUI

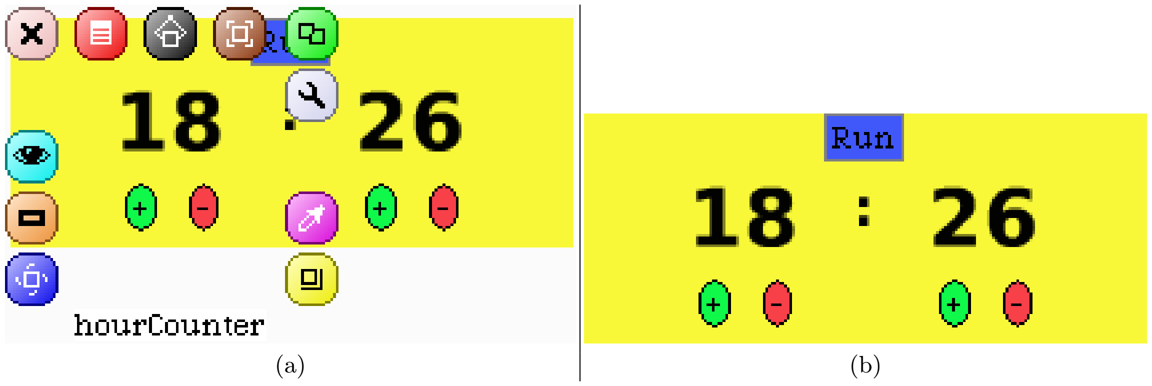


**Fig. 12.** Building circular counter's GUI by dynamic cloning

## 4 Related Work

For building GUIs, many generic and abstract models, were already proposed in the literature. The main aim of these models is to obtain a better comprehension of the existing interactive systems, and to define suitable software architectures for the development of new systems. All models define an interactive system with two components: an interface component and an applicative component. UI models could be classified in four main classes: linguistics models, interactors models, direct manipulation interfaces and hybrid approaches.

**Linguistics models** [9] [10] are based on a linguistic approach of the interaction which identifies three key aspects: 1) lexical aspects indicate all things that can be assimilated to an input (click, drag and drop) or output (icons) vocabulary. 2) syntaxic aspects indicate grammars of input representing valid sequences of actions, or the space and temporal aspects of the display. 3) semantic aspects correspond to the functional part of the application, which lastly determines the meaning of an action and generates errors.

**Interactors models** that carry out the separation of UI concerns in several objects.

**MVC (Model View Controller)** is the most well known and one of the oldest patterns in UI development. The most influential aspect of this pattern was the clear separation between domain objects that model the real world and presentation objects, seen on the screen. MVC users are encouraged, first of all, to create the model classes that represent the domain layer in their application. However, though very powerful, we find that MVC main drawback is that views are static entities. Developers manipulate view descriptions (i.e. code) instead of live objects. Therefore interactive development is difficult if not impossible.

**VisualWorks' Application Model** VW provides a variant of MVC. It introduces an intermediate "application model" between views and models. An application model consists of a behavior that is required to support the user's interaction. For example, the information in the model may represent a selection in a menu, or the contents of a paste buffer. Widgets do no more observe domain objects directly, instead they observe the application model.

**MVP (Model-View-Presenter)** Dolphin Smalltalk use the MVP pattern, a derivative of the MVC pattern, first introduced in C++ by Taligent [11]. Its aim is to provide a cleaner implementation of the Observer connection between Application Model and view. In MVP, the Presenter gets some extra power compared to Controller. Its purpose is to interpret events and perform any sort of logic necessary to map them to the proper commands to manipulate the model in the intended fashion. Most of the code dealing with how the user interface works is coded into the Presenter, making it much like the "Application Model" in the MVC approach.

**Direct manipulation interface** is a user interface style that was defined by Ben Shneiderman [6] whose intention is to allow a user to directly manipulate objects presented to them, using actions that correspond to the physical world.

- **Widgets based UI** Visual Basic (VB) and other related business frameworks use a Widget-based architecture. Developers write application specific forms that use generic controls provided by the framework. The form describes the layout of controls. By means of very simple observer mechanism, the form observes the controls and handle methods that react to interesting events raised by controls. The form is usually build with the help of a visual editor. Programming in VB consist of visually arrange controls components on a form, specify attributes and actions of those components, and write additional lines of code for adding more functionality. This is a very similar approach to the Morphic one, but the domain code is inextricably linked with the interface logic.
- **Naked Object Pattern** With naked objects [12] the domain objects are rendered visible to the user by means of a completely generic presentation layer. This user interface is automaticaly generated from an underlying business model definition. But contrary to Morphic, the emphasis is not on making objects more tangible to the programmer but rather on making them more tangible to the end-user of the system.
- **Fabrik** Dan Ingalls' and Scott Wallace's Fabrik[13] was one of the first direct manipulation of objects system in Smalltalk. Fabrik propose a kit of computational and user-interface components that can be "wired" together to build new components and useful applications.

**Hybrid approaches** mix several approaches in an uniform framework.

- **PAC (Presentation-Abstraction-Control)** is another derivative of MVC that divide an UI object into three components [14]: a lexical (presentation), syntactic (control) and semantic (presentation) components. Interactive components of the PAC model are based on a linguistic approach of interaction. The control component maintains a link between the presentation and abstraction components, but is also responsible to communicate with sub-UI components as the PAC model is recursive.
- **Tweak** On the one hand, Morphic is a suitable architecture as far as direct manipulation is involved but support reusability very badly. On the other hand, MVC doesn't support direct manipulation. Tweak[10] try to combine the best of both worlds. In Tweak, each graphical object exists in a "dual" representation - as a model like object (called a "Player") and as view like object (called a "Costume"). Unfortunately, there is no available academic description of Tweak, so it's a bit difficult to understand how this integration is done.

None of these approaches support all the keys requirements that we had stated in section 1.3.

[10] http://tweak.impara.de/

## 5 Conclusion and Future Work

While coding classes does fit well developing applications business code, the development of GUIs is often cumbersome. The most natural approach is developing GUIs using visual tools. In this paper we presented Easy Morphic GUI (EMG) a framework that eases building applications with Morphic GUIs. Thanks to a few design patterns and particularly mediator and prototype, EMG does bridge the gap between Smalltalk code and Morphic–EToys visual manipulation and scripting capabilities. Developers have total freedom for building GUIs in a WYSIWYG way, including using some higher level tools such as EToys. EMG does also introduce a couple of operators that encourage GUI reuse.

Regarding future work, a first one is about the generalization of the EMG framework to the whole Morphic hierarchy. Our proposal is to refactor Morphic in order to construct every morph visually. Separation between business – mediator – appearance as introduced by EMG, combined with GUI reuse operators is likely to improve reuse and modularity.

Better visual tools are also needed for morphs manipulations. So far, not all morph properties and relationships can be set through clicks and menus. Inspectors and debuggers are used as an alternative for missing feature. But this solution is not totally satisfactory.

Last, automatic code generation and update can make developers work even more easier. For example, methods for retrieving submorphs of a GUI can be automatically produced when embedding a morph into a GUI. Other connections between business objects, the mediator and GUI morphs can also be generated, making GUI construction with EMG even more easy.

## References

1. Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., Kay, A.: Back to the future. the story of squeak, a practical smalltalk written in itself. In: Proceedings of OOPSLA'97, Atlanta, Georgia, ACM (October 1997) 318–326
2. Maloney, J.: Introduction to Morphic: The Squeak User Interface Framework. In: Squeak: Open Personal Computing and Multimedia. Prentice Hall (2002) 39–67
3. Ungar, D., Smith, R.B.: Self: The power of simplicity. In: Proceeding OOPSLA'87. Volume 22 of ACM SIGPLAN Notices. (1987) 227–242
4. Gaelli, M., Nierstrasz, O., Stinckwich, S.: Idioms for composing games with etoys. In: The Fourth International Conference on Creating, Connecting and Collaborating through Computing (C5 2006), IEEE Computer Society (2006)
5. Westerlund, B.: Form is function. In: DIS 2002, Serious reflection on designing interactive systems. (2002) 117–124
6. Shneiderman, B.: Direct manipulation. a step beyond programming languages. IEEE Transactions on Computers **16**(8) (August 1983) 57–69
7. Gamma, E., Helem, R., Johnson, R., Vlissides, J.: Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
8. Woolf, B.: The null object pattern. PLOP'96 Writers Workshop (1996)
9. Pfaff, G., ed.: User Interface Management Systems, Springer-Verlag (1983)

10. UIMS: A metamodel for the runtime architecture of an interactive systems. In ACM, ed.: ACM SIGCHI Bulletin. Volume 24. (1992) 32–37
11. Potel, M.: Mvp : Model-view-presenter : the taligent programming model for c++ and java. http://www.wildcrest.com/Potel/Portfolio/mvp.pdf (1996)
12. Pawson, R.: Naked objects. PhD thesis, Department of Computer Science, Trinity College, Dublin (June 2004)
13. Ingalls, D.: Fabrik: A visual programming environment. In ACM, ed.: OOPSLA'88. (1988)
14. Coutaz, J.: Architectural models for interactive software. In Press, C.U., ed.: European Conference on Object-oriented Programming. (1989) 382–399

# Object Flow Analysis — Taking an Object-Centric View on Dynamic Analysis

Adrian Lienhard[1], Stéphane Ducasse[2], Tudor Gîrba[1]

[1] Software Composition Group, University of Bern, Switzerland
[2] LISTIC, University of Savoie, France

**Abstract.** To extract abstract views of the behavior of an object-oriented system for reverse engineering, a body of research exists that analyzes a system's runtime execution. Those approaches primarily analyze the control flow by tracing method execution events. However, they do not capture information flows. We address this problem by proposing a novel dynamic analysis technique named Object Flow Analysis, which complements method execution tracing with an accurate analysis of the runtime flow of objects. To exemplify the usefulness of our analysis we present a visual approach that allows a system engineer to study classes and components in terms of how they exchange objects at runtime. We illustrate and validate our approach on two case studies.

## 1 Introduction

A large body of research exists for supporting the reverse engineering process of legacy systems. However, especially in the case of dynamic object-oriented programming languages, statically analyzing the source code can be difficult and inaccurate. Dynamic binding, polymorphism, and especially behavioral and structural reflective capabilities pose limitations to static analysis.

Many approaches tackle these problems by investigating the dynamic information collected from system runs [1–4]. Most proposed approaches are based on execution traces, which typically are viewed as UML sequence diagrams or as a tree structure representing the sequence and nesting of method executions [4–7]. Such views analyse message passing to reveal the control flow in a system or the communication between objects or between classes [8, 9].

Various approaches extend method execution tracing with object information to improve object-oriented program understanding. For example, they trace instantiation events to analyze where objects are created [4, 9–11]. Other approaches analyze object reference graphs to make object encapsulation explicit [12] or to find memory leaks [13].

While data flows have been widely studied in static analysis [14], none of the above mentioned dynamic analysis approaches captures the runtime *transfer* of object references. In this paper we present *Object Flow Analysis*, a novel dynamic analysis approach, which captures the complete and continuous path of objects. We propose a meta-model that explicitly captures the transfer of object references.

To exemplify the usefulness of our analysis, we propose the following application for facilitating program comprehension of object-oriented legacy systems. A difficulty with studying the control flow of a program, *e.g.,* using an UML sequence diagram, is that the propagation of objects is hard to understand. For example, it is often not obvious how objects created in one class propagate to others. Also, inspecting how objects refer to each other, *e.g.,* using object inspectors, does not reveal how the object reference graph evolved. Our goal is to analyze how the objects are passed at runtime to facilitate understanding the architecture of a system. We want to address the following explicit questions that arose from our experience maintaining large industrial applications written in dynamic languages:

– Which classes exchange objects?
– Which classes act as object hubs?
– Given a class, which objects are passed to or from it?
– Which objects get stored in a class, and which objects just pass through it?
– Which objects are passed through multiple classes?

To address the questions, we present a prototype tool that is based on Object Flow Analysis. It provides visualizations to facilitate exploring the results of our analysis. We implemented the tracing infrastructure in Squeak[3], an open source Smalltalk dialect, and we implemented the meta-model and the visualizations in VisualWorks Smalltalk using Moose [15] and Mondrian [16].

*Outline.* In the next section we emphasize the need for complementing execution traces with object flow information to provide a more exhaustive fundament for object-oriented program analysis. Section 3 discusses Object Flow Analysis, our novel dynamic analysis technique that tracks how objects are passed through the system. Section 4 presents our approach to exploit object flows for studying the behavior of object-oriented programs at the architectural level. Section 5 evaluates our visual approach based on two case studies, and Section 6 provides a discussion. Section 7 reports on the state-of-the-art and Section 8 presents the conclusions.

## 2 Challenges Understanding Object Propagation

Unlike pure functional languages where the entire flow of data is explicit, in object-oriented systems the flow of objects is not apparent from the source code. However, also with today's dynamic analysis approaches, understanding object flows is hard.

The predominantly captured data of dynamic object-oriented program behavior are execution traces. An execution trace can be represented as a method call-tree. Figure 1 illustrates a small excerpt of such a tree from one of our case studies (a Smalltalk bytecode compiler). It shows as nodes the class name of the receiver and the method that was executed.

[3] See http://www.squeak.org/

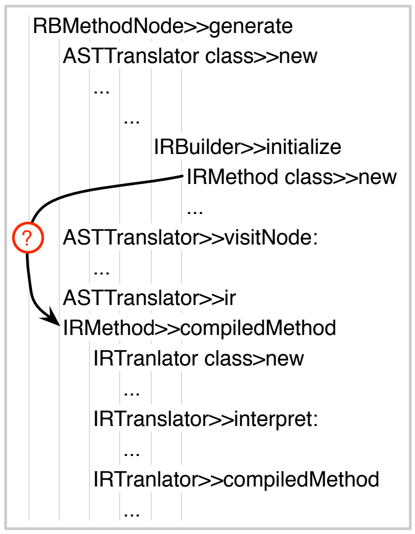


**Fig. 1.** Excerpt of an execution tree.

A limitation of execution traces is that they typically provide little information about the actual objects involved. In Figure 1 we see that an IRMethod instance is created in IRBuilder. Later an IRMethod instance is sent the message compiledMethod in RBMethodNode. Some of the execution trace approaches record the identity of the receiver object of a method execution. With this information we can reveal that the IRMethod instance created is the identical one in both places of the trace.

However, the execution trace cannot answer how this instance was passed there. The instance could be passed from IRBuilder to RBMethodNode via a sequence of method return values through other classes, but it could as well be stored in a field and then be accessed directly later on.

Speculating about the answer is hampered even more because execution traces are usually very large. Figure 1 only shows the first five levels and ten method executions. The actual execution tree from which the example is taken is 46 levels deep and comprises 4793 method executions.

Gschwind et. al. propose a dynamic analysis that captures also the objects passed as arguments [17]. They argue that this information was essential to gain a deeper insight into the program execution. Walker et. al. visualize the operation of a system at the architectural level and note in their discussion that “it may be useful to capture the migration of objects” [10].

Object passing and sharing complicates program comprehension because it can introduce complex object interrelationships. Nevertheless, object passing, *i.e.,* the transfer of object references, is an essential feature of object-orientation. A large body of research has been invested into controlling object aliasing at the type system level to provide a strong notion of object encapsulation [18, 19]. However, such advanced type systems are still in the realm of advanced research.

Our goal is to analyze how the objects are passed at runtime to support program comprehension of object-oriented legacy systems. This analysis is especially interesting for the objects that are not encapsulated, *i.e.,* objects that are aliased and that are passed around. At the heart of our analysis are the following two questions:

– In which method executions was an object made visible through a reference?
– Where did a specific object reference come from?

The first question reveals all locations, *i.e.,* arguments, return values, instance and global variables, the object is passed through (including those in which it does not receive messages). The second question reveals the path of the object, that is, how it is passed from one to another location, starting from where it is instantiated.

In the next section we introduce the concept of Object Flow Analysis and present its underlying meta-model. Based on the captured information we can then precisely answer how the IRMethod instance in Figure 1 is passed through the system.

## 3 Object Flow Analysis

In this section we present Object Flow Analysis, our approach to track how objects are passed through the system at runtime. This technique is based on an explicit model of object reference and method execution.

### 3.1 Representing Object References

The key idea we propose to extract such runtime information is to record each situation in which an object is made visible in a method execution through an object reference. We represent in our meta-model each such situation by a so-called *Alias* (see Figure 2).

In our program execution representation, an object alias is created when an object is (1) instantiated, (2) stored in a field (*i.e.,* instance variable) or global, (3) read from a field or global, (4) stored in a local variable, (5) passed as argument, or (6) returned from a method execution.

The rationale is that each object alias is bound to exactly one method execution (referred to as *Activation* in our meta-model), namely the method execution in which the alias makes the object visible. By definition, arguments, return values, and local variables are only visible in one method activation. In contrast, objects that are stored in fields (*i.e.,* instance variables) or globals, can be accessed in other activations as well. Therefore, we distinguish between *read* and *write* access of fields and globals.

With the record of all aliases of an object and their relationships to method activations, we can now determine where the object was visible during program

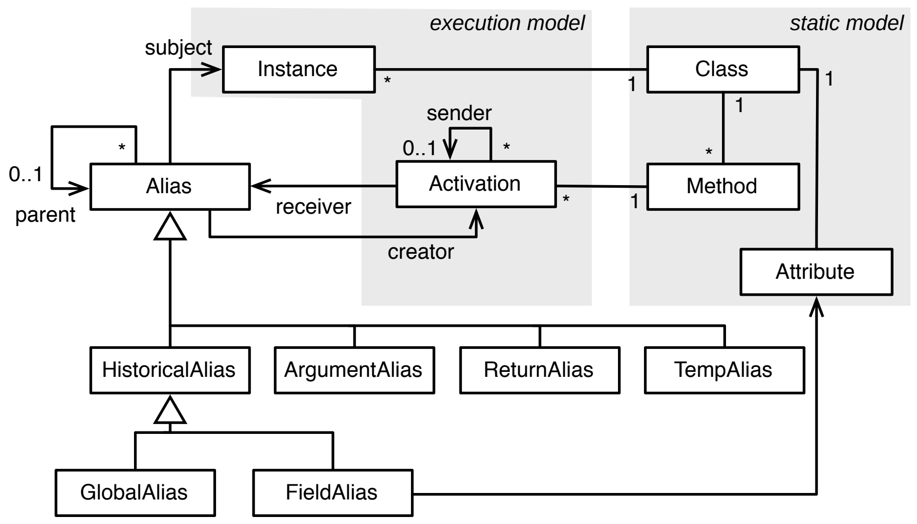


**Fig. 2.** The Object Flow meta-model extends the static and execution meta-models with the notion of Alias.

execution. The other key information from which we can extract the object flow resides in the relationships among the aliases.

Apart from the very first alias, which stems from the object instantiation primitive, all aliases are created from a previously existing one. This gives rise to a *parent-child* relationship between aliases originating from the same object. With this relationship we can organize the aliases of an object as a tree where the root is the alias created by the instantiation primitive. This tree represents the object flow, *i.e.,* it tells us how the object is passed through the system.

## 3.2 Control Flow vs. Object Flow Perspective

To illustrate and discuss details of the object flow construction we introduce a concrete example taken from one of our case study applications, namely the Smalltalk bytecode compiler (see Section 5 for more details). The example used in this section shows the interplay of important classes of the last two compiling phases (translating the abstract syntax tree (AST) to the intermediate representation (IR), and translating the IR to bytecode).

The center of interest is the instance of the class IRMethod. It acts as a container of IRSequence instances, which group instructions and form a control graph. We now take two complementary perspectives to study the program behavior in which the IRMethod instance is used. The first perspective emphasizes the control flow, that is, the sequence and nesting of the executed methods. The second perspective, illustrates the one we gain from studying object flows.

*Control flow perspective.* The execution trace in Figure 1 illustrates the order and nesting of the executed methods through which the IRMethod instance is passed. RBMethodNode≫generate is the first executed method. On the left side of Figure 3 we see its source code. It creates an instance of ASTTranslator, which in turn instantiates and stores an instance of IRBuilder in a field. IRBuilder in its constructor method initialize instantiates an IRMethod and stores it in the field named ir.

After this, the control is returned to RBMethodNode which then sends to ASTTranslator the message visitNode:. ASTTranslator is a visitor which traverses the AST and delegates the building of the IR to the IRBuilder instance. In the process IRBuilder creates several new sequences.

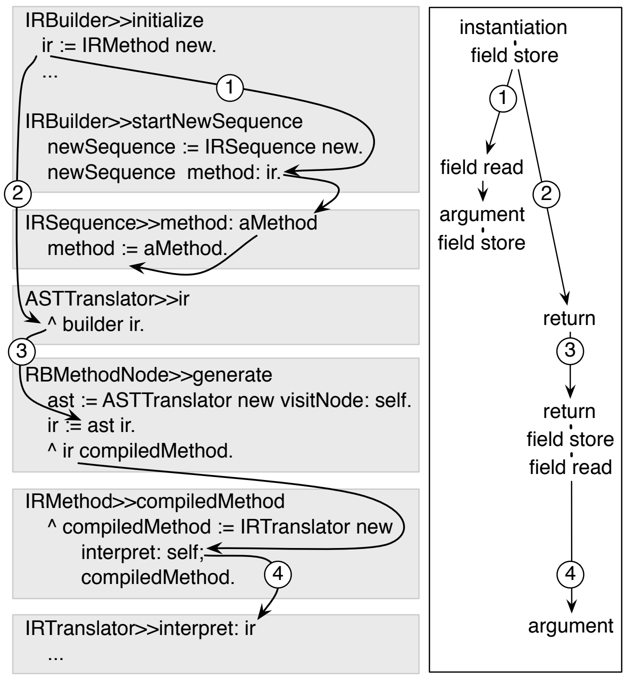

```
IRBuilder>>initialize
  ir := IRMethod new.
  ...

IRBuilder>>startNewSequence
  newSequence := IRSequence new.
  newSequence  method: ir.

IRSequence>>method: aMethod
  method := aMethod.

ASTTranslator>>ir
  ^ builder ir.

RBMethodNode>>generate
  ast := ASTTranslator new visitNode: self.
  ir := ast ir.
  ^ ir compiledMethod.

IRMethod>>compiledMethod
  ^ compiledMethod := IRTranslator new
      interpret: self;
      compiledMethod.

IRTranslator>>interpret: ir
  ...
```

**Fig. 3.** Object flow of an IRMethod.

When the IR is built, the IRMethod object is obtained by sending ir to the ASTTranslator which indirectly gets it from the IRBuilder instance. The execution now continues by sending compiledMethod to the IRMethod instance which eventually generates bytecode.

*Object flow perspective.* In this perspective we take the point of view of how the IRMethod instance flows through the system. The methods on the left side of Figure 3 are ordered by the time at which the instance is passed through them (rather than by the order of the control flow). The right side of Figure 3 illustrates the corresponding flow as a tree. Nodes represent aliases and edges are created depending on the parent-child relationship of the aliases.

This tree represents the object flow of the IRMethod instance. The flow starts with the root alias *instantiation* in the method IRBuilder≫initialize where the IRMethod instance is created. The object is then directly assigned to a field named ir (represented as a field store alias).

During the lifetime of IRBuilder the object is read from the field (1) and then passed as argument to IRSequence objects where it is stored in a field called method. Notice that in the actual execution the branch starting with (1) happens multiple times, but Figure 3 only shows one for conciseness.

When RBMethodNode≫generate requests the IRMethod instance, the object is first returned from the IRBuilder to the ASTTranslator (2) (this happens through a getter not shown here). Only then it is returned to RBMethodNode (3). This last return alias directly gets stored into the field ir of RBMethodNode.

A special case of aliasing is when an object passes itself as argument. In our code example this happens when the method compiledMethod of IRMethod is executed through the field read alias in RBMethodNode. The object instantiates an IRTranslator and passes itself to it as argument (see bottom of Figure 3). The argument alias which is created in IRTranslator has as parent alias the field read alias, that is, the alias which was used to activate the object that passed itself. This property of our model assures that the object flows are continuous.

In the next section, to illustrate the usefulness of Object Flow Analysis, we present a visual approach to answer the reverse engineering questions formulated in Section 1.

## 4 Visualizing Object Flows between Classes

Based on our meta-model we can analyze the transfer of object references between classes to answer questions such as:

- Which classes exchange objects?
- Which classes act as object hubs?
- Given a class, which objects are passed to or from it?
- Which objects get stored in a class, and which objects just pass through it?
- Which objects are passed through multiple classes?

We propose two explorative and complementary views to address the above questions:

- The *Inter-unit Flow View* depicts units connected by directed arcs subsuming all objects transferred between two units (Section 4.1). By unit we understand a class, or a group of classes that a software engineer knows they conceptually belong together (*e.g.,* all classes in a package, in a component, or in an application layer like the domain model or the user interface).
- The *Transit Flow View* allows a user to drill down into a unit to identify details of the actual objects and of the sequence of their passage (Section 4.2).

## 4.1 Inter-unit Flow View

Figure 4 shows an Inter-unit Flow View produced on our compiler case study. The nodes represent units (*i.e.,* either individual classes or groups of classes), and the directed arcs represent the flows between them. The thickness of an arc is proportional to the number of unique objects passed along it.

A force based layout algorithm is applied (nevertheless, the user can drag nodes as she wishes). This layout results in a spatial proximity of classes and units that exchange objects. This supports a software engineer in building a mental model and systematically exploring the software architecture.

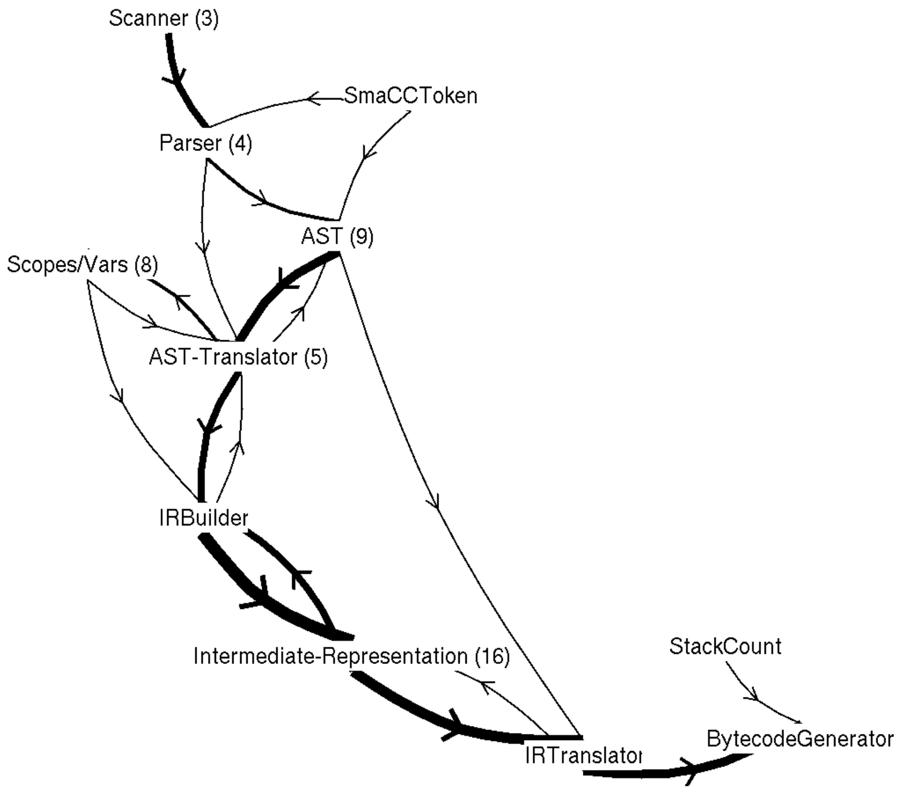


**Fig. 4.** Inter-unit Flow View of the bytecode compiler.

**Constructing the visualization.** The Object Flow model shows how objects are passed between other objects. As the goal of our visualization is to show how objects are passed through *classes*, we aggregate the flow at the level of classes and groups of classes (units). In our prototype units are created by the system engineer using a declarative mapping language (similar to the approach of Walker et. al. [10]). Rules are provided to map classes to units based on different properties such as the package they are contained in, their inheritance relationship, or a pattern matching rule based on their names. For instance, the first rule below maps all classes in the AST-Nodes package to the unit AST. The second rule maps IRInstruction and all classes inheriting from it to the unit IR.

```
classes containedInPackage: 'AST-Nodes' mapTo: 'AST'
classes hierarchyRootedIn: 'IRInstruction' mapTo: 'IR'
```

For the proposed visualization we do not take into account (1) through which instances of a class objects are passed, and (2) the flow of objects that are only used within one class. Another important property is that we treat the flows through collections transparently. This means that when an object is passed from one class to a collection, and later from the collection to another class, the intermediate step through the collection is omitted in the visualization. The flow directly goes from one class to the other and no node gets created for the collection class. This abstraction makes the visualization much more concise and emphasises the conceptual flows between application classes.

*Example.* Let's consider again Figure 4, which shows the Inter-unit Flow View of the Smalltalk bytecode compiler case study. Various classes are aggregated to units, displayed with the number of contained classes in brackets. For instance, the group AST (9) contains the nine classes representing the abstract syntax tree.

The visualization shows which classes exchange objects. For example, there are many objects passed from the Scanner to the Parser or from Intermediate-Representation to IRTranslator. On the other hand, we also see which classes are distant in that objects can only flow between them via several other classes.

Considering the thick arcs, we can detect a propagation of objects from Scanner (top) to BytecodeGenerator (bottom-right) traversing the Parser (top). This corresponds to the conceptual steps of a compiler.

An interesting exceptional flow is the one from AST to IRTranslator. It contains exactly one object, the IRMethod instance we encountered in the previous examples. As we can find out with the features introduced below, the object flow starts at IRBuilder and passes via AST-Translator. This corresponds to the object flow illustrated in Figure 3.

**The chronological propagation of objects.** The Inter-unit Flow View shows an overview of the entire execution. However, as not all objects are passed around at the same time, we are also interested in the chronological order to identify the phases of a system's execution. For example, in a program with an UI the phases may be related directly to the exercised features.

With our prototype implementation the user can scope the visualized object flow information to a specific time period by using a slider representing the time-line. The position of the slider defines up until which point in time object flows

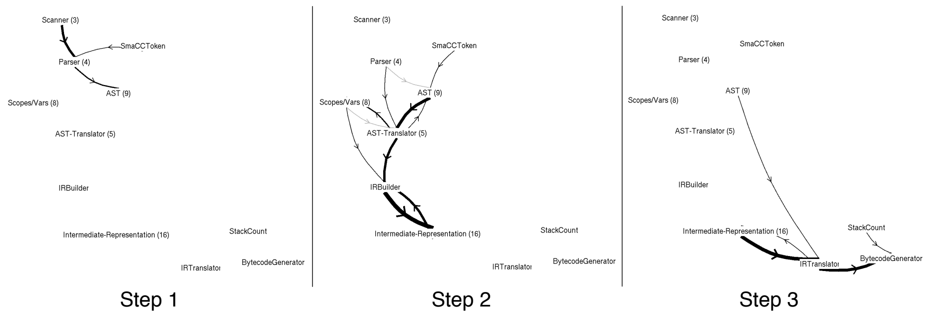


**Fig. 5.** Chronological propagation of flows in the compiler.

are taken into account. A recently active arc is displayed in dark gray which then fades and eventually becomes invisible. The goal of this feature is to facilitate investigation of how objects are propagated during a program execution.

Figure 5 illustrates three snapshots in the evolution of the compiler execution (compare with Figure 4). In the first step we see that objects are passed from Scanner to Parser and from Parser to AST. In the second step, many objects are passed between AST and AST-Translator, IRBuilder and Intermediate-Representation. In the third step, many objects pass from Intermediate-Representation to Bytecode-Generator.

**Highlighting spanning flows.** With the aforementioned features we can see which units directly exchange objects and when. However, we cannot see if there exist objects that are passed from one unit to another *indirectly*, *i.e.,* spanning intermediate units.

This information is useful to understand which units act as steps in object flows leading to a unit. The same holds for the objects passed outside a unit where it is interesting to know to which other units the objects are forwarded and which paths are taken.

In our prototype the user can select a unit. Thereafter, all arcs that contain objects being passed to the selected unit are highlighted in orange and all arcs with objects passed from the selected entity are highlighted in blue.

Figure 6 shows twice the same visualization (compare with Figure 4) but with different classes selected. In Figure 6.A Parser is selected. We see that objects are passed to it directly from Scanner (orange arc). On the other hand, the objects it passes outside reach many different units, the longest path reaches the Intermediate-Representation unit (blue arcs). In Figure 6.B IRBuilder is selected. We see that it obtains objects from most above units and forwards objects to almost all units below.

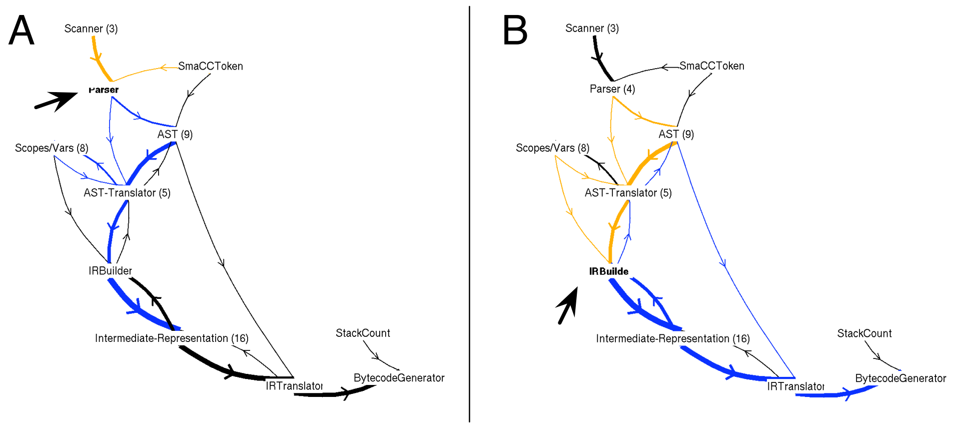


**Fig. 6.** Orange and blue arcs indicate flows leading to and coming from selected unit Parser (A), resp. unit IRBuilder (B).

This view highlights from where objects are passed to a unit and which routes are taken. This tells us, for example, how dependent a unit is on other units, *e.g.,* IRBuilder depends on objects created by or passed through all upper classes except for Scanner and SmaCCToken. The highlighted outgoing flows, on the other hand, tell us how influential a class is.

## 4.2 Transit Flow View

The aforementioned visualization lacks information about the actual objects being passed through a unit. To facilitate investigating this information our prototype allows the user to drill down to access detailed information about the objects transiting a unit.

Figure 7 illustrates the Transit Flow View for the class IRBuilder. It lists from top to bottom all instances that transit IRBuilder grouped by their class. The objects inside a class are grouped by their arrival time. For each instance the point in time when it was passed into or out of the class is indicated with a rectangle. An orange rectangle shows that the object is passed *in*; a blue rectangle that it is passed *out*. A line is displayed during the time when the object is stored in a field (or contained in a collection that is stored in a field).

The Transit Flow View shows when flows take place and how many instances of which class are involved. Further exploration reveals: (1) objects passed through directly (orange/blue pairs without line), (2) objects stored in fields or collections (line), (3) objects created (the first rectangle is not orange, therefore, the object is created in the class), and (4) objects passed in or out multiple times (several rectangles for the same object).

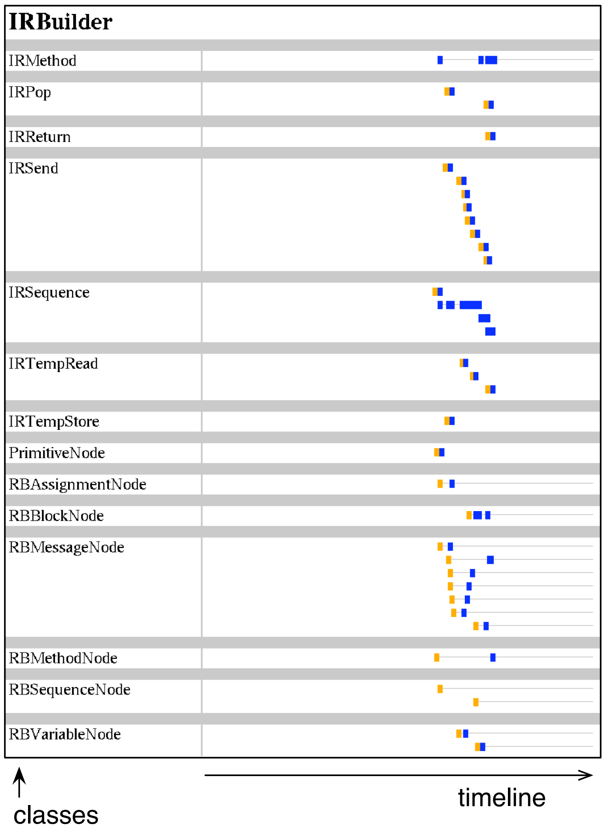


**Fig. 7.** IRBuilder Transit Flow View.

For example, in Figure 7, the IRMethod instance is created in IRBuilder, it is stored in a field, and it is passed out multiple times. Intermediate representation instances are created in IRBuilder but are not stored in it, whereas the instances at the bottom (AST nodes) are passed from outside and are stored.

## 5 Case Studies

In this section we provide an overview of the results we obtained from applying the visualizations on two case studies: a Smalltalk bytecode compiler and a health insurance web application. The objective of these investigations is to evaluate the usefulness of the two visualizations and to learn about a practical exploration process using our tool. Both case studies are implemented in Squeak, an open-source Smalltalk dialect. Our choice of those two case studies was motivated by the following reasons: (1) they are both non-trivial and model a very different domain, (2) we have access to the source code, and (3) we have direct access to developer knowledge to verify our findings.

The table below shows the static dimensions of the code:

| | Compiler | Insurance App. |
|---|---|---|
| Classes | 127 | 308 |
| Methods | 1'912 | 4'432 |
| Lines of code | 11'208 | 40'917 |

## 5.1 Bytecode Compiler

We chose the Smalltalk bytecode compiler as a case study because we wanted to understand its underlying mechanism to use it as basis for the future implementation of our Object Flow Analysis infrastructure. The compiler is a complex program, yet its domain is well known.

To generate experimental data we run the compiler on a typical method source code which includes class instantiations, local variable usage, a conditional and a return statement.

The Inter-unit Flow View illustrated by Figure 4 shows the final state of the view after several iterations of exploring and refining the mappings of units. We describe the exploration process of our tool which crystallized from the two case studies in the next section.

Using the Inter-unit Flow View (see Figure 5) we could correctly, *i.e.,* in line with the documentation, extract the key phases of the compiler: (1) scanning and parsing, (2) translating AST to the intermediate representation, and (3) translating the intermediate representation to bytecode.

With the help of the highlighting feature we obtained more detailed knowledge about the system. For example, IRBuilder plays a key role as it is a hub through which objects from the upper units in the view are passed to the lower ones. Using the Transit Flow View (see Figure 7) we studied detailed interrelationships between the units. For example, we could see where exactly the transitions between the three different representations happen. For the transition from AST to IR (phase 2) we see that the unit AST-Translator passes all AST nodes to IRBuilder, which in turn creates the intermediate representation objects.

In the the remaining part of this section we want to shed light on an interesting aspect of our approach we noticed in this case study.

*Inversion of execution flow.* The object flows do not necessarily evolve in the same direction as the execution flow. For instance, the Parser creates the Scanner and then regularly accesses it to get the next token. An analysis of the execution trace shows the call relation Parser $\rightarrow$ Scanner. The object flow view, on the other hand, shows the conceptually more meaningful order Scanner $\rightarrow$ Parser.

The reason is that with Object Flow Analysis we can provide object-centric views, which abstract implementation details, *e.g.,* the distinction of sender and receiver of a message. This trait also distinguishes our approach from the ones that are based call graphs in which edges point from the sender to the receiver class of a method execution [8, 9].

There are two ways how objects are passed to an instance: (1) objects are pushed to an object, *i.e.,* passed to the instance as method arguments, or (2) objects are pulled by the instance, *i.e.,* received as return value in response to a message send. In the latter case (2) the objects flow in the opposite direction compared to the message sends.

To further illustrate this point, let's consider again the introductory example of the IRMethod instance. In the execution trace excerpt, shown in Figure 1, the first executed method in which the IRMethod instance occurs is RBMethodNode≫generate. Studying this method first, however, leaves us with the question of how the object is set up and how it is passed there – something which is only visible later (or deeper) in the execution tree. In contrast, Object Flow Analysis is capable of providing a more meaningful viewpoint for studying the life cycle of the instance as illustrated by Figure 3.

### 5.2 Insurance Web Application

This industrial application was put into production six years ago and since that time has undergone various adaptations and extensions. The analysed scenario comprises the oldest and most valuable part for the customer, the process of creating a new offer. It is composed of ten features, including adding persons, specifying entry dates, selecting and configuring products, computing prices, and generating PDFs.

We first report on the exploration process, which we refined in this second case study and then discuss our investigations.

*Step 1: Creating coarse-grained units.* We started by investigating the Inter-unit Flow View with units corresponding to packages. Although we pictured first coherences among the packages, the view was hard to work with because there were many web-related packages that seemed less interesting for now. Therefore, we put all classes of the web packages into one unit, representing the UI layer.

*Step 2: Re-grouping to appropriate units.* The resulting view was already more concise. Now focusing on the domain model, we saw many packages corresponding to individual products, each package containing the product classes and associated calculation model classes. We re-grouped the classes into a Products unit and a Calculation Models unit because we wanted to learn about the higher-level concepts rather than how products differ.

This change dramatically improved the view. We obtained only nine units and we could identify interesting flows between them (see Figure 8). For instance, with the help of the Transit Flow View, we could understand how versioning works and how products and calculation models relate to each other. Products pass dates to the package responsible for versioning and in turn calculation models are passed to the products (we used the Transit Flow View to access this information).

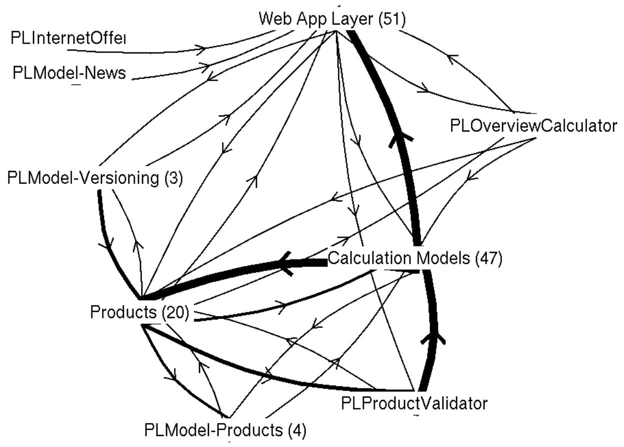


**Fig. 8.** Inter-unit Flow View of the insurance application case study.

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

To obtain more details about those classes we used the highlighting feature to show which other units are involved in passing objects with respect to the selected class. In the case of OverviewCalculator we see that it passes a person and a date to products which forwards date and eventually returns price objects to the calculator.

As soon as we had gained an overview of the domain model, we further explored the UI layer by extracting classes responsible for generating web views.

*Discussion.* The exploration process we took, which proved useful, was to first gain a coarse-grained view (step 1), find appropriate units (step 2), and only then get into more detail (step 3). Our internal declarative mapping language was helpful to create conceptual groups of classes with varying level of detail. It was essential to be able to specify units that crosscut the package structure.

From the Inter-unit Flow View, conceptual relationships between units were intuitively understandable. The presented information is high-level and thus appropriate for studying an unfamiliar system. Yet, means are provided to drill down to gain more detailed knowledge where appropriate.

In contrast to the compiler case study, the feature for investigating the chronological propagation of objects was not particularly useful. A plausible explanation is that the compiler has a much stronger notion of sequentially transforming one representation to another. The exercised features of the health insurance application, on the other hand, do not exhibit this characteristic.

## 6 Discussion

The application of Object Flow Analysis proposed in this paper focuses on studying classes or intentional groups of classes referred to as units. Hence, in our information space, classes are the basic parts representing fixed points on which the object flows are mapped.

As a point of variation, the basic entities could be instances. This would allow one to study the object flows at a much more detailed level ("which objects pass through a particular instance?"). Whether this information is valuable depends on the task at hand. The objective of the approach presented in this paper is to study a system at the architectural level, therefore, we focused on classes.

*Other applications of Object Flow Analysis.* We strongly believe that Object Flow Analysis can be successfully exploited in many more ways, which yet have to be discovered. Our approach maps object flows to structural entities. For instance, a very different way of looking at object flows is to consider dynamic boundaries. In previous work we described an approach to analyse the flow of objects between *features* to detect feature runtime dependencies [20]. Another promising application of our object flow analysis technique could be to analyze object flows between *threads*, which would reveal how objects are shared between them and how they are transferred.

*Scalability.* Naturally, the additional information about object flows do not come for free. Namely, a larger amount of data has to be dealt with. We adopt an offline approach, that is, at runtime the tracer gathers aliasing and method execution events. After execution, the data is then fed into the analysis framework on top of which our prototype is implemented.

In a typical program, the number of alias events is higher compared to method execution events. However, as the figures in the table below show, the relative increase is moderate.

| | Compiler | Insurance App. |
|---|---:|---:|
| Method executions | 11'910 | 120'569 |
| Aliases | 16'033 | 197'499 |
| Ratio | 1.3 | 1.6 |

Summarizing, Object Flow Analysis, which gathers both object aliasing and method executions, consumes about 2.5 times the space of conventional execution trace approaches. Our approach deals with the potential large number of events by providing an abstract view at the architectural level. Detailed information about the objects is only shown on demand.

While a factor of 2.5 is not negligible, we believe that the complementary information about the flow of objects justifies the overhead.

*Limitations.* Considering recall, a noteworthy limitation of our approach is the well-known fact that dynamic analysis is not exhaustive, as not all possible paths of execution are exercised [21]. Therefore, a dynamic analysis always has to be understood in the context of the actual execution.

Like with most other dynamic analysis approaches, scalability may be a limiting factor. The Transit Flow View is most vulnerable because it shows single objects. While in our case studies this view scaled well (the largest one displaying about 100 instances that were passed between two units), it may be cumbersome to study when containing thousands of instances. A solution to this problem may be to even further compact the representation, to apply filters to sort out less interesting objects, or to make selected application classes transparent like collections.

*Language independence.* To perform object-flow analysis, we need to capture details in the trace that reveal the path of objects through the system. We chose Smalltalk to implement our Object Flow Tracer because of its openness and reflective capabilities, which allowed us to evaluate different alias tracking techniques. We are currently implementing an Object Flow Tracer for Java. Particular difficulties are the instrumentation of system classes and the capturing of object reference transfers. The meta-model (both the static and dynamic part) and the visualizations we describe are language independent.

## 7 Related Work

Dynamic analysis covers a number of techniques for analyzing a program's runtime behavior [6, 21, 22]. Many techniques focus on analyzing execution traces [1, 5, 8, 23].

Many different approaches exist to make execution traces accessible. For instance, Lange and Yuichi built the Program Explorer to identify design patterns [2]. Pauw et. al. propose a tool to visually present execution traces to the user [1]. They automatically identify reoccurring execution patterns to detect domain concepts that appear at different locations in the method trace. Scenariographer is a tool which computes groups of similar sequences of method executions to reveal class usage scenarios [23]. While those approaches target understanding a system through the analysis of method executions our approach provides a complementary view based on the the flow of objects. As discussed, an advantage of Object Flow Analysis is that it can reveal more meaningful architectural views because object flows do not depend on implementation details like caller-callee relationships.

Most approaches, including the ones mentioned above, primarily analyze the program's execution *behavior*. Other approaches analyze the *structure* of object relationships. Super-Jinsight visualizes object reference patterns to detect memory leaks [13], and the visualizations of ownership-trees proposed by Hill et. al. show the encapsulation structure of objects [12]. Those two approaches are based on heap snapshots whereas our approach has an explicit notion of the *evolution* of object references in the form of object flows. With our meta-model we can accurately track continuous flows of the objects which is key to understanding flows spanning multiple classes.

Dynamic data flow analysis is a method of analyzing the sequence of actions (define, reference, and undefine) on data at runtime. It has mainly been used for testing procedural programs, but has been extended to object-oriented programming languages as well [24, 25]. Since the goal of those approaches is to detect improper sequences on data access, they do not capture how objects are passed through the system, nor how read and write accesses relate to method executions. To the best of our knowledge, Object Flow Analysis is the only dynamic analysis approach that explicitly models object reference transfers.

A large body of research has also been invested into facilitating program comprehension through static analysis. The static analysis of points-to analysis is an active research area. Challenges are precision and cost of the algorithms. The visualizations we present in this paper could potentially also be based on a static data flow analysis instead of a runtime analysis of object flows. The drawback of a static analysis is that it provides a conservative view (which in some cases may even include infeasible execution paths of the program). Our analysis, on the other hand, produces a precise under-approximation. An advantage is that it allows the user to constrict analysis to the features of interest and thus directly relate them to the obtained knowledge. Our approach trades off precision for completeness, and therefore we have to anticipate that the results do not apply to all possible program executions.

## 8 Conclusions

The hallmark of object-oriented applications is the deep collaboration of objects to accomplish a complex task. Understanding such applications is then difficult since reading the classes only reveals the static aspects of the computation. While dynamic analysis approaches offer solutions, they often focus only on the execution of a program from a message passing point of view.

In this paper we identified a missing aspect of dynamic object-oriented program analysis, namely the tracking of how objects are passed through the system. We introduce our approach, named Object Flow Analysis, in which we treat object references as first class entities to track the flows of objects. This approach complements the view on method executions with the view on objects.

To show the usefulness of our approach, we exemplified it with an application in the form of two visualizations: Inter-unit Flow View and Transit Flow View. We used these visualizations to explore the object flows between classes and we applied them on two case studies. These initial experiments showed promising benefits of this new perspective.

We strongly believe that our approach opens a new perspective on dynamic analysis and we intend to further pursue different applications based on it.

*Acknowledgments.* We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Analyzing, capturing and taming software change" (SNF Project No. 200020-113342, Oct. 2006 - Sept. 2008) and the Cook ANR project "COOK (JC05 42872): Réarchitecturisation des applications industrielles objets".

## References

1. De Pauw, W., Lorenz, D., Vlissides, J., Wegman, M.: Execution patterns in object-oriented visualization. In: Proceedings Conference on Object-Oriented Technologies and Systems (COOTS '98), USENIX (1998) 219–234
2. Lange, D., Nakamura, Y.: Interactive visualization of design patterns can help in framework understanding. In: Proceedings ACM International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 1995), New York NY, ACM Press (1995) 342–357
3. Richner, T., Ducasse, S.: Recovering high-level views of object-oriented applications from static and dynamic information. In Yang, H., White, L., eds.: Proceedings of 15th IEEE International Conference on Software Maintenance (ICSM'99), Los Alamitos CA, IEEE Computer Society Press (September 1999) 13–22
4. Antoniol, G., Guéhéneuc, Y.G.: Feature identification: a novel approach and a case study. In: Proceedings IEEE International Conference on Software Maintenance (ICSM 2005), Los Alamitos CA, IEEE Computer Society Press (September 2005) 357–366
5. Kleyn, M.F., Gingrich, P.C.: GraphTrace — understanding object-oriented systems using concurrently animated views. In: Proceedings OOPSLA '88 (International Conference on Object-Oriented Programming Systems, Languages, and Applications. Volume 23., ACM Press (November 1988) 191–205
6. Greevy, O., Ducasse, S.: Correlating features and code using a compact two-sided trace analysis approach. In: Proceedings of 9th European Conference on Software Maintenance and Reengineering (CSMR'05), Los Alamitos CA, IEEE Computer Society (2005) 314–323
7. El-Ramly, M., Stroulia, E., Sorenson, P.: Recovering software requirements from system-user interaction traces. In: Proceedings ACM International Conference on Software Engineering and Knowledge Engineering, New York NY, ACM Press (2002) 447–454
8. Zaidman, A., Calders, T., Demeyer, S., Paredaens, J.: Applying webmining techniques to execution traces to support the program comprehension process. In: Proceedings IEEE European Conference on Software Maintenance and Reengineering (CSMR 2005), Los Alamitos CA, IEEE Computer Society Press (2005) 134–142
9. De Pauw, W., Kimelman, D., Vlissides, J.: Modeling object-oriented program execution. In Tokoro, M., Pareschi, R., eds.: Proceedings ECOOP '94. LNCS 821, Bologna, Italy, Springer-Verlag (July 1994) 163–182

10. Walker, R.J., Murphy, G.C., Freeman-Benson, B., Wright, D., Swanson, D., Isaak, J.: Visualizing dynamic software system information through high-level models. In: Proceedings OOPSLA '98, ACM (October 1998) 271–283
11. Ducasse, S., Lanza, M., Bertuli, R.: High-level polymetric views of condensed run-time information. In: Proceedings of 8th European Conference on Software Maintenance and Reengineering (CSMR'04), Los Alamitos CA, IEEE Computer Society Press (2004) 309–318
12. Hill, T., Noble, J., Potter, J.: Scalable visualisations with ownership trees. In: Proceedings of TOOLS '00. (June 2000)
13. De Pauw, W., Sevitsky, G.: Visualizing reference patterns for solving memory leaks in Java. In Guerraoui, R., ed.: Proceedings ECOOP '99. Volume 1628 of LNCS., Lisbon, Portugal, Springer-Verlag (June 1999) 116–134
14. Hind, M.: Pointer analysis: Haven't we solved this problem yet? In: 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01), Snowbird, UT (2001)
15. Nierstrasz, O., Ducasse, S., Gîrba, T.: The story of Moose: an agile reengineering environment. In: Proceedings of the European Software Engineering Conference (ESEC/FSE 2005), New York NY, ACM Press (2005) 1–10 Invited paper.
16. Meyer, M., Gîrba, T., Lungu, M.: Mondrian: An agile visualization framework. In: ACM Symposium on Software Visualization (SoftVis 2006), New York, NY, USA, ACM Press (2006) 135–144
17. Gschwind, T., Oberleitner, J.: Improving dynamic data analysis with aspect-oriented programming. In: Proceedings of CSMR 2003, IEEE Press (2003)
18. Hogg, J., Lea, D., Wills, A., deChampeaux, D., Holt, R.: The Geneva convention on the treatment of object aliasing. SIGPLAN OOPS Mess. **3**(2) (1992) 11–16
19. Noble, J., Potter, J., Vitek, J.: Flexible alias protection. In Jul, E., ed.: Proceedings ECOOP '98. Volume 1445 of LNCS., Brussels, Belgium, Springer-Verlag (July 1998)
20. Lienhard, A., Greevy, O., Nierstrasz, O.: Tracking objects to detect feature dependencies. In: Proceedings International Conference on Program Comprehension (ICPC 2007), Washington, DC, USA, IEEE Computer Society (June 2007) 59–68
21. Ball, T.: The concept of dynamic analysis. In: Proceedings European Software Engineering Conference and ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSC 1999). Number 1687 in LNCS, Heidelberg, Springer Verlag (sep 1999) 216–234
22. Systä, Koskimies, Müller: Shimba — an environment for reverse engineering Java software systems. Software — Practice and Experience **1**(1) (January 2001)
23. Salah, M., Denton, T., Mancoridis, S., Shokoufandeh, A., Vokolos, F.I.: Scenariographer: A tool for reverse engineering class usage scenarios from method invocation sequences. In: Proceedings of ICSM 2005 (21th International Conference on Software Maintenance), IEEE Computer Society Press (September 2005)
24. Chen, T.Y., Low, C.K.: Dynamic data flow analysis for C++. In: APSEC '95: Proceedings of the Second Asia Pacific Software Engineering Conference, Washington, DC, USA, IEEE Computer Society (1995) 22
25. Boujarwah, A.S., Saleh, K., Al-Dallal, J.: Dynamic data flow analysis for Java programs. Information & Software Technology **42**(11) (2000) 765–775

# Part III

# Language Features

# Context-Oriented Programming: Beyond Layers

Martin von Löwis[1], Marcus Denker[2], and Oscar Nierstrasz[2]

[1] Operating Systems and Middleware Group
Hasso-Plattner-Institute at
University of Potsdam, Germany
http://www.dcl.hpi.uni-potsdam.de
[2] Software Composition Group
University of Bern, Switzerland
http://scg.iam.unibe.ch

**Abstract.** While many software systems today have to be aware of the context in which they are executing, there is still little support for structuring a program with respect to context. A first step towards better context-orientation was the introduction of method layers. This paper proposes two additional language concepts, namely the implicit activation of method layers, and the introduction of dynamic variables.

## 1 Introduction

Context-oriented programming provides mechanisms in the programming language to make the dependency of the program on its context explicit. Costanza and Hirschfeld [1] have previously proposed *method layers* to explicitly represent context dependency in a program. A new module concept, the layer, helps programmers to refactor a program to separate context-dependent behavior from the main program logic. There are several implementations of method layers, including the original ContextL implementation [1].

While method layers enable the modularization of the program with respect to context-awareness, the mechanism by which method layers are activated still poses a challenge. In ContextL, method layers need to be explicitly activated, for example, after evaluating some condition on sensor readings from the system's environment. While such explicit context activation can work in many cases, there are also cases where no single point in the progam exists that could detect the change in context, and consequently activate a method layer. In particular, if the context may change at any time during program execution, implicit activation of method layers becomes necessary.

In addition, method layers focus on context-dependent algorithms. In many cases, algorithms might not change as the context changes; what will change is the data on which the algorithms operate. For example, in client-server systems, each new request will might establish a new context for the algorithm, including information such as the user performing the request, correlation to earlier actions of the user (*i.e.,* sessions), transactional context of the operation, and so on. A method that needs to consider such context then must find out what the current context is. As methods call each other in a recursive manner, the context information must be passed from the point where the context is observed to the point where the context is consumed; many contemporary programming languages still don't address this need explicitly.

Two examples of contextual behavior are the dependency on the current user of a system, and the dependency on the current device on which information is displayed.

For context-dependent output, it is common that different output algorithms are used, *e.g.,* output to a PostScript file typically requires algorithms different from output to an HTML page. In an object-oriented program, output logic is commonly spread over multiple methods. To denote the various output modes, different method layers can be defined. When rendering the output, the proper layer can be activated, and all output methods will adapt themselves according to the output context.

Dependency on the current user has various facets: it may be that users have different roles, and the program behavior should again change significantly depending on the role in which a user acts. This dependency can be modeled using method layers again, defining different layers for the different roles. However, dependency on the current user also often involves operating on different data: in an email system, different users operate on different mailboxes. The program logic for accessing a mailbox will be the same independent of the user; the only difference is in which mailbox is presented to the user. A common approach to implement this dependency is to use the user's identification as a key into some associative data structure. In order to implement that lookup, the current user needs to be known in all places of the program.

In some cases, a combination of both contextual state and contextual behavior is necessary. For example, a web client application may need to provide a User-Agent header when requesting certain pages from a web server. The User-Agent header is a text string indicating the web browser which sent the request. Web servers sometimes respond with different pages, depending on what browser made the request. If the access is made in an automated manner, the web client library may either send no User-Agent header at all, or send one that indicates an automated agent. With context-oriented programming, the web client application can set the context for all HTTP requests to include a specific User-Agent header. This involves both a behavior change, and relies on contextual data: the header should be sent only if it was configured, and if so, the string that is sent is determined by contextual data.

In this paper, we discuss extensions to context-oriented programming that we developed as part of PyContext, a framework for context-oriented programming in Python. We studied a number of existing Python application to analyze what kind of context-dependency is present in these systems, and then defined a set of Python constructs that can help to better structure programs with respect to context-dependency.

Our main contributions are twofold: we present an extension to the concept of method layers to support implicit activation of layers, and we propose dynamic variables as a means to access context-dependent state.

In section 2, we present our analysis of existing software systems. Then we describe the notion of method layers (section 3), our extensions to that (section 4), and the PyContext implementation (section 5). Finally, we present some related work (section 6) before concluding (section 7).

## 2 Context-dependent behavior

We have analyzed a number of non-trivial Python applications with respect to dependency on context. This study both validates our assertion that context-dependency is common in current software systems, and helps us to identify concepts that can be added to programming languages to better support context-orientation.

We selected three applications that we considered to be naturally context-aware. The applications we studied were Django [2], roundup [3], and SCons [4]. Whereas the first two are both used in the domain of web applications, the third is from the area of software engineering and maintenance.

Django is a framework for web applications. It includes

– an object-relational mapper, to support persistent storage of application objects
– an HTML templating engine, to allow separation of the web page design from the application design, and
– a framework for web applications, including an HTTP server and a component system for web applications

Roundup is a bug-tracking application, allowing users to report issues found with a software system over the web or via email, and developers to collect information about these issues in order to eventually resolve them.

SCons is a software construction tool that can be used in the build process of software systems.

We identified context-awareness in these applications first through systematic inspection. As a starting point, we looked for known cases of context-awareness, such as dependency on system configuration and dependency on the current user of the system. Using source code review, we identified a number of places where context-dependency occurs. In the next step, we tried to generalize these findings and derive more systematic ways of identifying context-awareness. We present these findings starting with the specific examples, then going on to the generalizations.

### 2.1 Case studies

**Django.** Django supports a component system for web applications, where an individual web application can be deployed into an existing Django installation, and transparently respond to HTTP requests targeted at that application. For this to work, Django uses two pieces of contextual information:

1. The URL of the HTTP request consists of contextual information provided by the client-side web browser; a part of the URL is meant to identify the application that should serve the request.
2. As neither the Django code nor the application code should incorporate a static mapping between URLs and applications, Django supports a URL configuration file (urls.py) that defines a mapping from URLs to application. During run-time of the web server, the server should respond to changes of the configuration file, adding new web applications as they are deployed.

To deal with the dependency on configuration data, Django internally maintains a global variable (conf.settings) that contains all configuration information. To react to dynamic changes of the configuration files (such as the url mappings), a separate thread reads the modification times of the configuration files every second, and restarts the web server from scratch if a change in the configuration files has been found.

To gain access to the URL path of the HTTP request, both an HTTP request object and a path object are passed through several layers of software until a URLResolver object maps the URL path to a callback function that is the entry point to the web application.

The HTTP request object is further passed as an explicit parameter to various routines, and provides additional contextual information. In particular, the request is passed to several layers of software called middleware. One such middleware module provides session context, based on information in the request.

**Roundup.** In roundup, several instances of the bug tracker may be running on a single machine. Each tracker instance acts as a context of execution for the tracking software, providing its own relational schema for trackers, its own database of recorded issues, its own set of access control lists, and so on.

Instead of recording the “current" instance in a global variable, roundup creates, per HTTP request to the tracker, a Client object which encapsulates:

– the current tracker,
– the current HTTP request,
– the id of the end-user invoking the current request (which may be obtained from the request, or through other authentication mechanisms), and
– the issue or issue detail on which an operation is to be performed.

Actions are then processed using the command pattern [5], where the command object is created with a reference to the Client object, and then the command is run.

**SCons.** SCons maintains an “environment" object, which contains information about the context of the current build activity. This environment is filled with information from various sources, such as:

– the host system and system type on which the build is run,
– locations of tools needed in various build steps,
– configuration information from the build scripts about parameters to be passed to the build steps,
– files that act as input and output to the build steps, and
– actions that still need to be run, or have already been completed.

This environment is passed as an explicit last parameter to all methods.

## 2.2 Detecting context-dependency

In studying context-dependency in these three applications, we noticed two kinds of code structure which hint towards context-dependent behavior:

1. In many cases, things are explicitly called “context" — authors of the software were clearly aware of the contextual nature of these aspects of the system.
2. In some cases, parameters occurred primarily as “pass-through" parameters, *i.e.,* they propagate recursively through a chain of method calls, until they are eventually consumed by some leaf function. As these parameters sometimes make it into interface definitions, the caller needs to pass the parameter independent of whether the callee actually needs it. This, in turn, may cause the caller to require the parameter as an input argument as well, even though it has no need for the parameter.

These two structures now allow for a more systematic search for contextual code. Unfortunately, detecting pass-through parameters in an automated manner requires tools that analyze the source code statically, inspecting each function's parameters, and usage of the parameters within the function. As no such tools are available today, we were only able to detect a few more cases of contextual information with manual inspection, by looking for functions that don't use parameters they receive.

Searching for things called “context" is much easier in a systematic manner than searching for pass-through parameters; we found these additional cases in the systems studied:

1. In Django, there is a module called `context_processors`. These are functions that return dictionaries of context information, such as the current user, the permissions of the current user, whether or not debugging information should be displayed, what (natural) language a web page should be displayed in, and so on.
2. The Django templating engine maintains a set of variables which can be accessed in rendering the page template, for place holders, conditional HTML inclusion, and repeated blocks of HTML. The collection of these variables is called “context".
3. The same holds for the roundup templating engine, which also calls all variables used for templating collectively “context".

## 3 Context-Oriented Programming

In [1], the authors propose the following language constructs to support context-oriented programming, for an implementation called ContextL:

– layers, which identify groups of classes and methods that will be used together in the dynamic scope of a program,
– layered classes, which are classes that have different partial class definitions for different layers,
– layered methods, which are defined through partial definitions for different layers,
– layered slots, which are instance attributes whose values depend on the active layer, and
– explicit layer activation, which selects a certain set of partial definitions in the dynamic scope of control flow.

With layers, it becomes possible to factor out those parts of method and class definitions that have been written for a specific context. For the example of different output methods given in the example, a method layer can be defined for each different mode of data output. A method that performs the actual output might then get multiple partial definitions, one for each layer. Some parts of the algorithm might be independent from the output algorithm, *e.g.,* the algorithm that iterates over all pieces of information and call the output method on each piece. If a certain kind of output is requested, the program needs to activate the corresponding layer, and call the output algorithm in that context.

As layers can be defined for independent contexts, multiple independent layers can be activated simultaneously, causing the partial definitions for all of these layers to become active. In the output example, a separate layer might be developed for a context in which special support for handicapped people is necessary (*e.g.,* by enlarging all text to improve readability). In such a case, a single layered method might need to take multiple partial definitions into account. ContextL defines a mechanism for combining partial methods, where each layer activation can modify the previous definition of a method with another fragment. In the definition of the partial method, the developer needs to specify whether the fragment is executed before, after, or instead of the original method. In the latter case, the developer can also choose to call the original method inside the fragment, making the fragment run around the original method.

## 4 Context beyond layers

We propose two further mechanisms to support context-oriented programming as defined above: implicit layer activation, and dynamic variables.

A shortcoming of the existing implementations of method layers is that layers must be activated explicitly rather than implicitly as the context of the application changes. While it is possible and desirable in many cases to explicitly control activation of layers (typically after evaluating some condition on the program context), having to explicitly activate layers may sometimes violate modularity of contextual behavior definitions: If the condition that should trigger the activation can become true at any time in the program (and if it is necessary that the program react on the context change quickly), the check for the change of the condition needs to be added in many places of the code, potentially to the degree that these replicated changes become larger than the actual contextual behavior.

In our case studies, we observed that much context-dependency in applications is not in contextual behavior, but in contextual state. It was often the case that the program computed some variables from contextual conditions, and then indirectly called methods that needed to work with these variables.

In the case of the web applications (see section 2), the applications typically combined all contextual data associated with the current request object, which then is passed as an explicit parameter to all methods. If some interface does not provide for passing the request, the methods implementing it have no way of determining the context.

For configuration data, global variables were commonly used. While global variables work fine in single-threaded programs or programs where threads only read the values, they become a maintenance challenge in multi-threaded programs. For example, debugging and tracing support might be active only during a part of the program (the part which the developer currently studies), in these cases, the applications typically added additional parameters to the methods in order to make the context available.

## 5 PyContext

PyContext is a framework for context-oriented programming similar to ContextL [1], extending it with concepts that we determined to be desirable. PyContext also builds on the support for context-orientation that is already present in Python 2.5 [6].

We present PyContext in three stages. First, we review a recent addition to the Python programming language (the **with** statement) that allows one to scope contextual behavior to the dynamic extent of a block of code. We found that mechanism to be a useful basis for supporting cases of context-orientation where the program detects the context at some point, and then requires all subsequent actions to take that context into account.

Then, we discuss how we make the concepts found in ContextL available to Python programs. Specifically, we added support for the definition of method layers, for the definition of layered classes, and for the (nested) activation of layers.

Finally, we introduce the extensions we made in PyContext which aren't available in ContextL, namely the implicit activation of layers and the definition of dynamic variables.

## 5.1 Context managers in Python 2.5

In the Python Enhancement Proposal 343 [7], a new keyword **with** is introduced into the language; this is related to a kind of object protocol called "context managers". While this statement was introduced independently of this work, we found it to be extremely useful to represent context-dependency in a Python program, avoiding the need to come up with our own extension to the Python syntax.

This statement can be either used in the form:

```
with expression:
    statements
```

or as:

```
with expression as variable:
    statements
```

In either form, the expression is evaluated, and should result in a object that implements the "context manager" interface [7]. According to that interface, a method __enter__ is called on the context manager, and the statements are executed. Finally, __exit__ is called on the context manager. The language guarantees that __exit__ is called regardless of how the execution of the statements terminates (*i.e.,* whether they run to the end, are terminated through a return, continue, or break statement, or whether an exception is raised). The method __exit__ receives as a parameter information indicating whether the block was left normally, or by means of an exception.

In the second form, the result of the __enter__ method is assigned to the variable.

The PEP mentions the following use cases for this kind of statement:

- Synchronization and locking of blocks of code: the context manager should be an object supporting mutual exclusion; __enter__ should acquire a lock; __exit__ should release it. Python 2.5 provides such a context manager in its threading library, allowing users to write

  ```
  some_lock = threading.RLock() # create recursive lock
  # ...
  with locking(some_lock):
      some_code
  ```

- Symmetric acquisition and release of resources, such as operating file system handles. In Python 2.5, the file object was extended to become a context manager, making it possible to write statements like

  ```
  with open(pathname, "r") as f:
      for line in f:
          process(line)
  ```

  In this fragment, the file f will be closed automatically at the end of the for loop, even if an exception is raised during processing.

– Similarly, in a transactional system, the context manager might control the transaction context, automatically committing the transactions if the statements complete successfully, and rolling back the transaction if an exception is raised.
– In Python's module for decimal floating point arithmetic, the semantics of operations depends on a decimal context; this controls rounding and error mode of all operations. Applications typically want to apply the same rounding and error mode for the duration of an entire computation, and can use the **with** statement to scope the dynamic extent of the decimal context.

### 5.2 Layers

Similarly to ContextL, PyContext supports the notion of method layers. A layer is a collection of partial method definitions, spanning possibly multiple classes. The partial method definitions can augment or replace an existing method; the same method may have partial definitions in different layers. Dynamically, layers can get activated in the control flow of program; the meaning of a method of a class is then obtained by combining the partial definitions of the methods for all active layers.

As an example of a definition of layered methods, consider the following use case, which originates from the domain of internet client applications. Python offers several libraries to access HTTP URLs (universal resource locators): httplib, which offers direct access to the all wire details of the HTTP protocol, and urllib, which allows a more abstract API. As an example of using urllib, a download of a remote document can be written as follows:

```
import urllib
f = urllib.urlopen("http://www.esug.org")
print f.read()
```

In this case, the urllib library handles all details of the HTTP protocol. In some applications, it is necessary to tailor the protocol interaction to include a User-Agent header in each HTTP request, so that the web server is tricked into believing that the page was requested through a specific web browser (such as Microsoft's Internet Explorer), instead of detecting that it was programmatically accessed through Python's urllib module.

To achieve this functionality, the existing httplib library needs to be augmented to include the User-Agent header. Modifying the library to always set User-Agent to indicate a specific web browser is not acceptable; passing that configuration as a parameter is not possible, since urllib will use httplib in a hard-coded way. Instead, the information of whether a User-Agent header should be sent and, if so, which one, should be defined by the application context.

Using PyContext, it becomes possible to formulate the context using a method layer, and activate that method layer using the **with** statement:

```
import urllib
from useragent import HTTPUserAgent, MSIE6AgentString
```

```
with HTTPUserAgent(MSIE6AgentString):
    f = urllib.urlopen("http://www.esug.org")
    print f.read()
```

In this fragment, a method layer HTTPUserAgent is activated for a block of code, causing all HTTP requests in this block of code to include a User-Agent header; invocations of urllib outside this block (*e.g.,* in a different thread) are not affected. The API of urllib had not to be changed to introduce this context-awareness.

To define the method layer itself, a subclass of context.layers.Layer must be defined

```
from context import layers

class HTTPUserAgent(layers.layer):
    def __init__(self, layer):
        self.layer = layer
```

To define the methods of the layer, the syntax must indicate both what class a partial method belongs to, as well as what layer it belongs to. In PyContext, this is denoted through multiple inheritance (indicating an extension to both the class and the layer). To implement the HTTPUserAgent layer, the method end headers of httplib.HTTPConnection must be extended to explicitly add the User-Agent header. In addition, any attempt to send an additional User-Agent header must be blocked, *e.g.,* preventing urllib from adding its own User-Agent setting. The partial HTTPConnection class then reads as

```
class HTTPConnection(HTTPUserAgent, httplib.HTTPConnection):

    # Always add a User-Agent header
    @layers.before
    def endheaders(self, context):
        with layers.Disabled(HTTPUserAgent):
            self.putheader("User-Agent", context.layer.agent)

    # suppress other User-Agent headers added
    @layers.instead
    def putheader(self, context, header, value):
        if header.lower() == 'user-agent':
            return
        return context.proceed(header, value)
```

In order to produce the complete semantics of a method from its partial definition, the function decorators before, after, and instead can be used. An activation of a layer combines all methods with their newly-activated fragments, making the partial definitions run either before, after, or instead of the original method definition.

Within the context of an activated layer, it is sometimes necessary to disable the layer for further additional recursive calls. In PyContext, this can achieved with the Disabled context manager, which is also demonstrated in the example above.

**Implicit layer activation.** To support applications that need to factor out context activation from the main program logic, PyContext offers a mechanism to implicitly activate layers. Each layer may define a method active, which determines whether the layer is active. Then, when a layered method is called, the framework first determines which layers are active, and then produces a composition of all method definitions for all active layers. For that to work, the framework needs to know which layer object needs to be checked. Therefore, a function layers.register_implicit needs to be called to subscribe layers for the activation check.

We can imagine a number of design alternatives for defining the semantics of implicit activation. As the layer should be activated depending on context, one question is how often the context should be checked (*i.e.,* how often the active method should be called). One option would be to do this regularly, or whenever an external stimulus arrives (such as an interrupt). While there are cases where either of these approaches work well, there are also cases where they fail, *e.g.,* because a context change does not lead to a hardware or software interrupt. To support the most general case, PyContext evaluates the activation condition on each method invocation of a method potentially affected by layer activation. That may produce a lot of overhead; to reduce that overhead, the layer definition may apply caching techniques in case recomputation of the condition is not necessary every time.

## 5.3 Context variables

To ease the programming of applications that rely on contextual state, PyContext offers contextual variables, which maintain their value during the dynamic extent of a **with** statement. In that sense, they are similar to dynamic scoping of values in languages like Lisp or SNOBOL4.

A dynamic variable is represented in a globally-accessible Python object. A set method returns a context manager which sets the variable to its new value during execution of __enter__, and restores the previous value when __exit__ is called. Reading the variable is done through a get method, which returns the value of the variable in the current context.

As an example, consider a web application where the web programming framework provides the notion of a session context. With PyContext, the web framework may expose the session object to the application using a context variable. In this example, the access to the variable is still wrapped with a convenience function:

```
from context import Variable

_session = Variable()
def current_session():
    return _session.get()

def process_request(request):
    session = lookup_session(request)
```

```
with _session.set(request):
    dispatch_request(request)
```

In this code, the scoping of the variable session is still static, and still follows the semantics of the Python language. What is dynamically determined is the value associated with the variable. Rather than having a fixed binding of the variable to a value at any point in time, it depends on the execution context, and the dynamic extent of the **with** statement to determine the value the variable possesses.

More precisely, executing the set method on a variable under control of a **with** statement will bind a new value to the variable, hiding the previous value. Invoking the get method on the dynamic variable object fetches the value bound most recently to the variable in the current thread's context. Leaving the **with** statement restores the binding to the prior value of the variable.

In the example, we only show that the variable is read in the same module where it is written. However, the read access might also happen in any other module, as long as that module imports the module where the variable is defined.

### 5.4 Implementation strategy

The current implementation of PyContext does not perform any modifications to the Python Virtual Machine. Instead, the implementation was completely achieved as a library of regular classes using mechanisms already provided by the language.

Layers are implemented using the meta-object protocol in Python. Layer definitions are based on a custom meta-class provided by PyContext, allowing one to collect all partial method definitions at the point of definition of the layered classes. The original method is replaced by a proxy method which then dispatches to the various partial methods that need to be called, in the (reverse) order of layer activation.

Dynamic variables are implemented using thread-local storage provided by the Python threading library.

Layer activation and binding of dynamic variables uses the concept of context handlers introduced in Python 2.5, as discussed above.

## 6 Related work

PIE [8–11] extends the Smalltalk object model with of *views*. PIE provides code with multiple views, *i.e.,* representing design decisions from the perspectives of different developers. PIE views need to be explicitly activated similar to ContextL layers, PIE does not support any abstractions for implicit activation or contextual state.

Us [12] supports subjective programming where message lookup depends not only on the receiver of a message, but also on a second object, called the *perspective*. The perspective allows for layer activation similar to ContextL. Us does not support implicit activation of layers.

ContextL [1, 13, 14] is a language to support Context-Oriented Programming (COP) in LISP; we have discussed the relationship to PyContext above. Context-oriented Programming is discussed in more detail in [15].

Hanson and Proebsting discuss in [16] the notion of dynamic variables. They identify two constructs needed for dynamic variables; a **set** operation that binds the variable to a value, and a **use** operation, that makes it available in the local static scope of a block of code. Our approach deviates slightly from this pattern, as each read operation is denoted as a method call, rather than bringing the variable into local scope for a block of text. Our approach also differs in that no type declarations are necessary for the variables, consistent with the rest of the Python language, which does not require type declarations either. Hanson and Proebsting list a number of other languages that also provide dynamic variables, and discuss the impact of the introduction of dynamic variables to a language; their conclusions apply to this work also.

It's interesting to observe that a number of implementation strategies have been devised for dynamic variables in the past. Hanson and Proebsting mention that various implementations maintain a linked list of all variables that is traversed to determine the location where the value was last bound. They then propose an alternative algorithm that uses a stack walk, similar to the one implemented for exception handling. PyContext uses yet another approach for determining the value of a dynamic variable, namely by using thread-local storage [17].

## 7 Conclusion and future work

Existing software systems, in particular web applications and other server applications exhibit a great degree of context dependency which currently cannot be expressed adequately and explicitly in the program. With context-oriented programming, these dependencies become explicit, allowing readers of the program to understand more easily how context affects the program's behavior. In addition, properly-designed constructs in the language can help developers to express the context-dependency more concisely.

While we have studied existing software and proposed constructs that assist in addressing context-awareness. Further research is needed to determine the applicability of these constructs to other application domains and other programming languages, *e.g.,* by adding these constructs to existing implementations of context-oriented programming such as ContextL [1].

In this paper, we have mostly neglected issues of performance of the implementation. We believe that efficient implementations of the new programming concepts are possible, but have not made efforts yet to study the performance impact in a realistic scenario, or to improve the performance where that might be necessary.

**Acknowledgements**

We thank Robert Hirschfeld for the productive discussions, and for bringing together the authors of the paper in the first place. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project “Analyzing, capturing and taming software change" (SNF Project No. 200020-113342, Oct. 2006 - Sept. 2008).

## References

1. Costanza, P., Hirschfeld, R.: Language constructs for context-oriented programming: An overview of ContextL. In: Proceedings of the Dynamic Languages Symposium (DLS) ’05, co-organized with OOPSLA’05, New York, NY, USA, ACM Press (October 2005)
2. Holovaty, A., Kaplan-Moss, J.: The Django Book. Apress (2007)
3. Jones, R.: Roundup: an Issue-Tracking System for Knowledge Workers. (2007)
4. Knight, S.: SCons User Guide 0.97. (2007)
5. Gamma, E., Helm, R., Vlissides, J., Johnson, R.E.: Design patterns: Abstraction and reuse of object-oriented design. In Nierstrasz, O., ed.: Proceedings ECOOP ’93. Volume 707 of LNCS., Kaiserslautern, Germany, Springer-Verlag (July 1993) 406–431
6. Kuchling, A.M.: What’s new in Python 2.5. Technical report, Python Software Foundation (2007)
7. van Rossum, G., Coghlan, N.: The “with” statement (Python enhancement proposal 343). Technical report, Python Software Foundation (2006)
8. Bobrow, D.G., Goldstein, I.P.: Representing design alternatives. In: Proceedings of the Conference on Artificial Intelligence and the Simulation of Behavior. (July 1980)
9. Goldstein, I.P., Bobrow, D.G.: Descriptions for a programming environment. In: Proceedings of the First Annual Conference of the National Association for Artificial Intelligence. (August 1980)
10. Goldstein, I.P., Bobrow, D.G.: Extending object-oriented programming in Smalltalk. In: Proceedings of the Lisp Conference. (August 1980) 75–81
11. Goldstein, I.P., Bobrow, D.G.: A layered approach to software design. Technical Report CSL-80-5, Xerox PARC (December 1980)
12. Smith, R.B., Ungar, D.: A simple and unifying approach to subjective objects. TAPOS special issue on Subjectivity in Object-Oriented Systems **2**(3) (1996) 161–178
13. Costanza, P., Hirschfeld, R., Meuter, W.D.: Efficient layer activation for switching context-dependent behavior. In: Joint Modular Languages Conference 2006 (JMLC2006). LNCS, Oxford, England, Springer (September 2006)
14. Costanza, P., Hirschfeld, R.: Reflective layer activation in contextl. In: SAC ’07: Proceedings of the 2007 ACM symposium on Applied computing, New York, NY, USA, ACM Press (2007) 1280–1285
15. Robert Hirschfeld, Pascal Costanza, O.N.: Context-orientied programming. Journal of Object Technology (March-April 2008) (to appear).
16. Hanson, D.R., Proebsting, T.A.: Dynamic variables. Technical Report MSR-TR-2000-109, Microsoft Research (November 2000)
17. IEEE: POSIX P1003.4a — Threads Extension for Portable Operating Systems. (1992)

# Forward Chaining in HALO

## An Implementation Strategy for History-based Logic Pointcuts

Charlotte Herzeel, Kris Gybels, Pascal Costanza, Coen De Roover, and Theo D'Hondt

Vrije Universiteit Brussel
{kris.gybels|charlotte.herzeel|pascal.costanza|cderoove|tjdhondt}@vub.ac.be

**Abstract.** In aspect-oriented programming, pointcuts are formulated as conditions over the context of dynamic events in the execution of a program. Hybrid pointcut languages also allow this context to come from interaction between the pointcut language and the base program. While some pointcut languages only allow conditions on the current execution event, more recent proposals have demonstrated the need for expressing conditions over a history of joinpoints. Such pointcut languages require means to balance the expressiveness of the language with the additional memory and runtime overhead caused by keeping a history of joinpoint context data. In this paper, we introduce a logic-based pointcut language that allows interaction with the base program as well as pointcuts over a history of joinpoints. We introduce forward chaining as an implementation model for this language, and discuss possible optimization strategies for the additional overhead.

## 1 Introduction

A good modular design decomposes program concerns into separate modules each implementing a different concern. Some concerns are however inherently crosscutting, which means that their implementation is scattered over different modules. Aspect-oriented Programming (AOP) focuses on the modularisation of such crosscutting concerns [1]. An AOP language provides a notion of joinpoints which are events in the execution of a program, a pointcut language to concisely describe multiple joinpoints and advices which affect the program behavior at the joinpoints captured by a pointcut.

Research into pointcut languages has shown that these can be made more expressive in several ways [2]. Two of these ways are: allowing pointcuts to be expressed over a history of joinpoints [3, 4], and allowing pointcuts to interact with the base language through a mechanism like hybrid pointcuts [5], which allows messages to be sent to objects. Several pointcut languages are also based on logic programming. In this approach, joinpoints are represented as logic facts and pointcuts as logic queries over these facts. In this paper, we focus on how logic-based pointcut languages [6, 4, 7, 8], can be combined with history-based

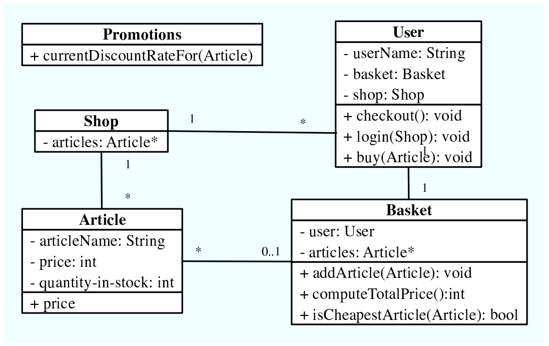


**Fig. 1.** A small e-shop application.

and hybrid pointcuts. Most logic-based pointcut languages are based on Prolog, which uses backward chaining to evaluate logic queries. In this paper, we demonstrate why this chaining strategy fails for a combination of history-based and hybrid pointcuts, and introduce forward chaining as an alternative. Specifically, the contributions of this paper are:

- We introduce a novel logic-based pointcut language, HALO, which allows both pointcuts over a history of joinpoints as well as a mechanism to interact with the base language;
- We introduce forward chaining, in particular the Rete algorithm, as an implementation strategy for such a language; including our extension to Rete to support expressing temporal relations between joinpoints in the history;
- We discuss how the predicates for expressing temporal relations in HALO enable space optimization of the joinpoint history.

In the next section we introduce a small running example demonstrating the need for a pointcut that is both history-based and interacts with the base program. In Section 3 we introduce a logic-based pointcut language that supports both features. In Section 4 we contrast backward and forward chaining, and more extensively discuss the Rete-based implementation of HALO. The final sections of the paper contrast HALO with related work, and discuss future work and our conclusions.

## 2 Running example

We use a simple e-commerce application as a running example to illustrate HALO. A UML diagram for this example is given in Figure 1. Users have an account and have to log in before they can add articles to their shopping basket of which the total price is calculated when they check out.

To attract customers, the shop occasionally engages in promotional marketing campaigns. The application therefore has a singleton class `Promotions` which gives the current rate of discount for each article. Several variations of this class are possible. It can be implemented as a simple table that stores the current discount rate for each article. The computation can however be more complex, for example based on the current amount of stock for the article.

In this example, one possible effect of the promotions is that banners pop-up to advertise promotions. Another possible effect of the promotions is that customers get a discount on articles when they check out. Giving discounts is based on the *past* discount rate for an article. The idea is that if a promotion for an article was advertised, the user should still get the discount when she checks out, even if the promotion is no longer active. The latter can be implemented as a "discount" aspect. Depending on the shop's strategy, the aspect can give the rate from when the user logged in or from when she added the article to her basket.

This is a small example, but it is sufficient to motivate a pointcut language with two features: (1) the ability to interact with the base language to invoke the `Promotions` object's `current-discount-rate-for` method, and (2) the ability to refer to past joinpoints. While the aspect can be expressed in a pointcut language lacking either feature, it requires some effort to do so. Notably, the programmer has to provide the necessary mechanism for manually accessing and recording the past discount rate by writing two pieces of advice: The first one defines a pointcut to capture when the user logs in, and the advice body invokes the `currentDiscountRateFor` to record the discount rate of the articles. The second advice defines a pointcut to capture when the checkout happens, and determines the right discount rate for an article from the recorded rates. This approach becomes more complicated the more pointcuts involve additional past joinpoint or past data. In the next section, we introduce a logic-based pointcut language that supports both features, which allows the pointcut to be expressed more concisely. A detailed comparison of the implementation of the e-shop application solelely relying on CLOS and a version implemented using HALO was reported on in previous work [9]. The focus of this paper is the definition and implementation details of the HALO language.

## 3 The HALO Language

HALO ("History-based Aspects using LOgic") is a novel logic-based pointcut language for Common Lisp [10] that allows pointcuts to be expressed over a history of joinpoints as well as allowing interactions with the base language. In contrast with earlier work on logic-based pointcut languages that offer a history of joinpoints [4], HALO imposes a fixed set of temporal predicates for expressing pointcuts over the history, drawn from temporal logic. In the next section, we discuss how imposing a fixed set of built-in predicates allows a run-time space optimization of the joinpoint history based on the semantics of the predicates. The related work section contrasts this approach with the static analysis of the open set of predicates of earlier work. In this section, we first discuss the background on logic-based pointcut languages, and briefly give an overview of temporal logic programming. We then explain the HALO language in more detail.

## 3.1 Logic Pointcuts over Joinpoint Histories

*Background on logic-based pointcut languages* Previous work on CARMA [6] and other logic-based pointcut languages [4, 7, 8] has demonstrated the suitability of logic programming [11] as the basis for a pointcut language. The core idea behind these languages is to represent joinpoints as logic facts and write pointcuts as logic queries over these facts. In particular, variations of the logic programming language Prolog [12] are often used as query languages.

*Temporal logic* To deal with temporal relations between joinpoints in the joinpoint history, HALO is in particular based on temporal logic programming. Between any two joinpoints generated during the execution of a program there exists a temporal relation. This relation can henceforth be used to concisely describe sequences of joinpoints in a pointcut. HALO offers a set of higher-order predicates to describe temporal relations between pointcuts. Higher-order temporal predicates are important for expressing hybrid pointcuts as they indicate at what time the interaction with the base language should occur. In addition, as explained in Section 4.4, a predefined set of temporal predicates makes it possible to optimize the memory usage of a history-based pointcut language.

There are different variants of temporal logic programming [13]. These variants primarily differ in the way time is modeled (i.e. discrete or continuous, finite or infinite, etc.). A discrete time model is most suited for a pointcut language as joinpoints are discrete events occurring during the execution of a program. The predicates available for expressing temporal relations constitute another important difference. Most of these predicates express an ordering relation and can either be past- or future-oriented, while in some logics both are provided. For example, J-LO [14] uses linear temporal logic which provides future-oriented predicates. HALO, on the other hand, uses a past-oriented subset of metric temporal logic programming (MTL) [15].

## 3.2 HALO Advice Language

As the focus of research involving HALO is the pointcut language itself, HALO adopts the advice mechanism used in most other general-purpose aspect languages. The body of the advice is written in the base program language, in this case Common Lisp. Unlike other advice languages, there is no construct to distinguish between “before” and “after” advices. Instead, the joinpoint model includes distinct “entry” and “return” joinpoints on which the application of an advice has the same effect as a “before” or “after” advice in other languages.

An example to illustrate the form in which advices are written:

```
(at ((gf-call 'buy ?arguments)
     (escape ?time (get-universal-time)))
   (print "Buy was invoked with arguments: " ?arguments)
   (print " at time " ?time))
```

| | | |
|---|---|---|
| pointcut | :: | (< primitive_pointcut >< escape > ∗ < tpointcut >) |
| pointcut | :: | (< primitive_pointcut >< escape > ∗(since < tpointcut >< tpointcut >)) |
| tpointcut | :: | ({< temporal >\| not} < pointcut >) |
| primitive_pointcut | :: | < gf_call >\|< gf_return >\|< get >\|< set >\|< create > \| < m_return >\|< m_call > |
| escape | :: | (escape ?variable < lisp-form >) |
| gf_call | :: | (gf-call ?gfName ?arguments) |
| | | Generic function call joinpoint |
| m_call | :: | (method-call ?methodName ?arguments ?specializers) |
| | | Method call joinpoint |
| gf_return | :: | (gf-return ?gfName ?arguments ?rvalue) |
| | | Generic function return joinpoint |
| m_return | :: | (method-return ?methodName ?arguments ?specializers ?rvalue) |
| | | Method return joinpoint |
| get | :: | (slot-get ?obj ?slotName ?value) |
| | | Slot get joinpoint |
| set | :: | (slot-set ?obj ?slotName ?oldValue ?newValue) |
| | | Slot set joinpoint |
| create | :: | (create ?className ?instance) |
| | | Instance creation joinpoint |
| temporal | :: | most-recent \| all-past \| cflow |
| | | Temporal relations |

**Fig. 2.** Grammar for HALO pointcut language. For conciseness we have depicted the arguments of the different predicates as logic variables preceded by a "?".

This is an example of a straightforward logging advice. Its pointcut comprises two conditions using the predicates `gf-call` and `escape`. The advice body consists of two calls to `print`. As is also illustrated in the example, it is possible to pass logic variables from the pointcut to the advice body. Logic queries, and thus pointcuts, can have multiple solutions with different values for the variables. The advice is executed for every solution of the query.

## 3.3 HALO Pointcut Language

In a logic pointcut language, pointcuts are expressed as queries using logic predicates. The built-in predicates of HALO fall into two classes: the primitive predicates that distinguish between types of joinpoints and higher-order temporal predicates for dealing with temporal relationships between joinpoints. The predicates are summarized in Figure 2.

Note that as HALO is a pointcut language for Common Lisp, Lisp-style list syntax is used for logic pointcut queries. Variables are written with a question mark, as in `?var`. For example, the expression `(gf-call 'buy ?args)` would be written in Prolog as `gf-call(buy, Args)`.

**Joinpoint Type Predicates** HALO's joinpoint model, as with most other pointcut languages, consists of the key events in the execution of an object-oriented program. In the case of Common Lisp, there are seven types of joinpoints: the instantiation of a class, the invocation of and return from a generic function[1], the execution of and return from a method, and the accessing or changing of a slot (instance variable).

Figure 2 lists HALO's joinpoint predicates. They each have a number of arguments exposing data of the joinpoint. The `gf-call` and `method-call` predicate respectively capture invocations of a generic function and executions of specific methods of a generic function. They each expose the arguments the function is invoked with: i.e. the actual run-time objects. The name of the function is exposed as a symbol. The `method-call` predicate has an additional parameter that exposes the specializers of the method, i.e. the argument types specified in the method signature, which are used to select a specific method of a generic function. The corresponding `gf-return` and `method-return` predicates select return joinpoints for generic functions and methods respectively. They have a similar parameter list as the `gf-call` and `method-call` predicates, but additionally expose the return value. The `slot-get` and `slot-set` predicates respectively capture slot access and change joinpoints. They expose the object whose slot is referenced, the name of the slot, its value, and its new value in the case of `slot-set`. The `create` predicate captures class instantiation joinpoints and exposes the class's name and the actual instance.

### Temporal Predicates

*Temporal predicates overview* The temporal predicates in HALO allow for pointcuts that express a temporal relation between past joinpoints. This is not limited to joinpoints which are in a control flow relationship. Rather, a history of past joinpoints is kept which can be referred to using the temporal predicates. The temporal predicates are higher-order predicates that take pointcuts as arguments. To establish some terminology, consider the following pointcut:

```
((gf-call 'checkout ?argsC)
 (most-recent (gf-call 'buy ?argsB)))
```

The first condition is referred to as the outer pointcut, the single condition used as argument to the temporal predicate `most-recent` is referred to as the inner pointcut.

The temporal higher-order predicates share the same basic semantics. An inner pointcut is evaluated against a subset of joinpoints relative to the joinpoints

[1] Methods in the Common Lisp Object System are not associated with a single class and do not have a hidden "receiver" argument as in other languages. Method dispatch occurs on the dynamic type of all arguments. Methods are grouped into a generic function, which performs the dynamic dispatch. The concept of sending a message with name $n$ to an object comes down to invoking the generic function named $n$ with that object as one of the arguments.

matching the outer pointcut. The actual subset, of course, depends on the particular temporal predicate. In the above example, the inner pointcut `(gf-call 'buy ?argsB)` is thus evaluated against joinpoints in the past of the joinpoints matching the outer pointcut `(gf-call 'checkout ?argsC)`.

In total, HALO hase four temporal predicates built-in: `most-recent`, `all-past`, `since` and `cflow`. The `all-past` and `most-recent` predicates match the inner pointcut against *all* past joinpoints relative to the joinpoint matched by the outer pointcut. The predicates differ in that the `all-past` has solutions for all past joinpoints that match, while the `most-recent` predicate only has a solution for the most recent joinpoint that matches. The `cflow` predicate is a variation of the `most-recent` predicate which additionally checks that no corresponding `return` joinpoint has occurred for the joinpoint captured by the inner pointcut (it is therefore similar to the `cflow` construct in AspectJ).

The `since` temporal predicate is the more difficult one of the four predicates as it has two inner pointcuts. The first inner pointcut is evaluated against the past joinpoints relative to the joinpoints captured by the outer pointcut. The second inner pointcut is evaluated against the joinpoints in-between the two other joinpoints.Examples are given in the following section, after a further explanation of variable sharing and the `escape` mechanism.

*Variable sharing* Variables can be shared between the inner and outer pointcuts. As the semantics of the temporal predicates is that the inner pointcut is evaluated against the past of the joinpoint captured by the outer pointcut, variables are bound by the outer pointcut. For example, the following pointcut captures invocations of the `buy` function for a user buying an article, and gives all users that previously also bought the same article:

```
((gf-call 'buy (?user1 ?article))
 (all-past (gf-call 'buy (?user2 ?article))))
```

In this example, the outer pointcut captures a `buy` call and exposes the arguments of the call in the `?user1` and `?article` variables, the inner pointcut then matches on all previous calls to `buy` with the same article object as argument.

**Hybrid Pointcuts** The `escape` predicate can be used to include Lisp code referring to logic variables in a pointcut definition. The example below shows a pointcut capturing invocations of a generic function named `buy`, where the `escape` predicate is used to ask the price of an article (second argument) and to bind the result to a logic variable `?price` (first argument). Whenever such a pointcut is evaluated, the piece of Lisp code is executed using the bindings available for the logic variables, resulting in a new variable binding which in return is used in the evaluation of the rest of the pointcut. But if the return value of the Lisp code is `nil`, the condition has no solution[2]. In the example, the constraint `(greater-than ?price 10)` is checked for a value `?price` computed at the Lisp level – or in other words, the binding for `?price` is not logically derived.

[2] `nil` also denotes false in Lisp

1 `(gf-call 'login <kris> <shop>)`
2 `(gf-call 'buy <kris> <dvd>)` where the current discount rate for `<dvd>` is 0.05
3 `(gf-call 'checkout <kris>)`
4 `(gf-call 'login <kris> <shop>)`
5 `(gf-call 'buy <kris> <game>)` where the current discount rate for `<game>` is 0.05
6 `(gf-call 'buy <kris> <book>)` where the current discount rate for `<book>` is 0.10
7 `(gf-call 'buy <kris> <cd>)` where the current discount rate for `<cd>` is nil
8 `(gf-call 'checkout <kris>)`

**Fig. 3.** A sample history of joinpoints (to simplify the example, only generic function calls are considered).

```
((gf-call 'buy (?user ?article))
 (escape ?price (price ?article))
 (greater-than ?price 10))
```

The `escape` predicate can be in a temporal predicate, but a restriction on the variables that can be used in its condition applies: Only variables that are used in non-`escape` conditions in the same inner pointcut can be used. This is because the Lisp code of the `escape` conditions inside the `most-recent` is evaluated when `user2` buys an article. The following piece of advice is triggered to print the price of an article bought by a user, and its price when previously bought by another user:

```
(at
    ((gf-call 'buy (?user1 ?article))
     (most-recent
        (gf-call 'buy (?user2 ?article))
        (escape ?price2 (price ?article))
        (escape ?name2 (user-name ?user2))))
 (print "Article previously bought by "
        ?name2 " for " ?price2 " EUR"))
```

So the variable `?price2` will refer to the past price of the article, which is possibly different from the price of the article when the second buyer purchases the article.

## 3.4 Further Examples

To clarify the way the temporal predicates are matched, we give a few further examples based on the sample execution trace shown in Figure 3. Note that we use the notation `<`*`name`*`>` to denote object identifiers (e.g. `<cd>` represents an object, obviously intented to be an instance of `Article`). We explain whether and why the joinpoint matches on joinpoint 8 (some examples also match on other joinpoints).

Given the sample execution history depicted in the figure, the following pointcut matches on joinpoint 8 with one solution:

```
((gf-call 'checkout ?user)
 (\emph{most-recent} (gf-call 'buy (?user ?article))
              (escape ?rate (current-discount-rate-for (singleton-instance 'promotions)
              ?article))))
```

For the match with joinpoint 8, the solution gives the article the user represented by the object `<kris>` last bought, and the discount rate at the time the article was bought. Given the execution history in Figure 3, this means the exposed discount rate is 0.10 for the article `<book>`. This pointcut captures join point nr. 8 because it matches the outer pointcut `(gf-call 'buy ?user ?article)` and because of the presence of join point nr. 6 that matches the inner pointut. Join point nr. 7 does not match the inner pointcut as the Lisp form in the `escape` condition evaluates to `nil`.

The following pointcut has multiple solutions for joinpoint 8, one for each article the user checking out ever bought and for which there was a promotion (the articles `<book>` and `<game>` and `<dvd>` bought by user `<kris>`):

```
((gf-call 'checkout ?user)
 (\emph{all-past} (gf-call 'buy (?user ?article))
          (escape ?rate (current-discount-rate-for (singleton-instance 'promotions)
          ?article))))
```

Our last example is a variation of the above one. It again has multiple solutions for the match with joinpoint 8, but only those articles bought since the last `login` (only the articles `<book>` and `<game>` purchased by `<kris>`):

```
((gf-call 'checkout ?user)
 (\emph{since}
   ((most-recent (gf-call 'login (?user ?shop))) )
    (all-past (gf-call 'buy (?user ?article))
              (escape ?rate
                      (current-discount-rate-for (singleton-instance 'promotions)
                      ?article))))))
```

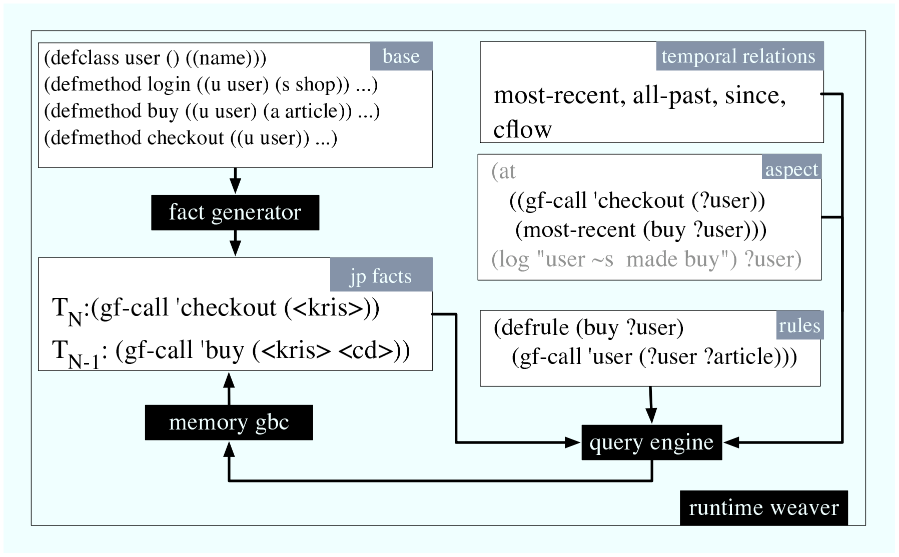
In more detail, this pointcut is matched at join point nr. 8, because it matches the outer pointcut `(gf-call 'login (?user ?shop))`, and exposes the discount rate of all `buy` join points (namely nr. 5 and 6) that match the second argument of the since predicate, since the last `login` join point that matched the first argument of the since predicate (join point nr. 4). Note that the `buy` join points and the `login` join point are again matched in the past of the `checkout` join point.

## 3.5 Defining Rules

Programmers can define rules for new predicates using the `defrule` construct. As in other logic-based pointcut languages [6, 4], this mechanism can be used to define new joinpoint predicates. This is simply a matter of using an existing joinpoint predicate in the definition of the rule. For example, the rule definition below extends HALO with a new pointcut predicate that captures invocations of a generic function called `checkout`:

```
(defrule (checkout-gf-call ?args)
  (gf-call 'checkout ?args))
```

Note that rules do not have to define predicates about joinpoints. Only rules based on other joinpoint predicates define a new joinpoint predicate. This is unlike the named pointcut mechanism in AspectJ for example, in which the conditions of a named pointcut always have to include a primitive or user-defined pointcut.



base
(defclass user () ((name)))
(defmethod login ((u user) (s shop)) ...)
(defmethod buy ((u user) (a article)) ...)
(defmethod checkout ((u user)) ...)
fact generator
jp facts
$T_N$:(gf-call 'checkout (<kris>))
$T_{N-1}$: (gf-call 'buy (<kris> <cd>))
memory gbc
temporal relations
most-recent, all-past, since, cflow
aspect
(at ((gf-call 'checkout (?user)) (most-recent (buy ?user))) (log "user ~s made buy") ?user)
rules
(defrule (buy ?user) (gf-call 'user (?user ?article)))
query engine
runtime weaver

**Fig. 4.** HALO weaver schema.

## 4 HALO Implementation

In this section we present the implementation strategy for evaluating HALO pointcuts. We present the overall architecture of the weaving process which involves a runtime weaver for intercepting joinpoints and a query engine for checking pointcuts for matches. We contrast the use of backward and forward chaining query engines for supporting HALO to compare with related work on logic-based pointcut languages and to demonstrate why forward chaining is necessary. This is followed by an extensive discussion of the Rete forward chaining algorithm and the necessary extensions for supporting HALO's temporal predicates and `escape` mechanism. Finally, we discuss how the semantics of the predicates is exploited to optimize the joinpoint history, so that joinpoints are removed from the history when they are no longer relevant for matching pointcuts.

### 4.1 HALO Weaver Architecture

A schema of the dynamic weaving process, responsible for combining HALO code and base code, is depicted in Figure 4. The weaver is responsible for mapping the key events in the execution of a Common Lisp program to logic facts and storing them in a fact base [3]. In our concrete implementation this is achieved by

[3] Note that we assume a sequential execution of the base program. The current implementation of the HALO weaver also operates on a "per thread" basis. A version of HALO for concurrent programming where pointcuts are shared between multiple threads is left for future work.

wrapping the generic function call, instance creation and slot access protocols in Common Lisp through the CLOS Metaobject Protocol [16] to attach code for generating the facts. Secondly the weaver is responsible for weaving in the proper advice code at each event; The proper advice code is computed by trying to resolve the pointcuts given the fact base. The latter is done by a *query engine*, which is basically an interpreter for our logic language HALO. In the following sections we outline the decisions made for implementing the HALO query engine. Another problem we examine in the follow-up text involves optimization strategies for memory management of the fact base, a common problem in history-based logic pointcut languages.

## 4.2 Implementing the Query Engine

We provide a more detailed discussion of the differences between forward and backward chaining [17] to compare with related work on logic-based pointcut languages (cfr. Section 5). Most current logic-based pointcut languages are based on Prolog, which is in turn based on backward chaining. We contrast these two approaches for evaluating logic queries and discuss why forward chaining is necessary to support a combination of hybrid pointcuts and reasoning over a history of pointcuts as in HALO.

The two approaches to chaining can be best contrasted by representing a logic query graphically, as in Figure 5 depicting the following piece of advice. When a user checks out, he gets a discount on the total amount purchased, and this discount is based on a rate that was promoted when the user logged into the shop:

```
(at
    ((gf-call 'checkout (?user))
     (most-recent (gf-call 'login (?user ?shop))
                  (escape ?rate (current-rate ?shop))))
    (discount ?user ?rate))
```

The left upper corner of Figure 5 depicts a sample program run. As explained in the previous section, the weaver records facts for each join point: the resulting fact base is also depicted in Figure 5.

When using a weaver with a backward-chained query engine, a problem arises with evaluating `escape` conditions at the right time. When using a weaver with a backward chained query engine, at every joinpoint, the weaver produces logic facts for describing that joinpoint and then invokes the query engine to check if any pointcuts match. In the example, when using a backward chainer, the weaver would launch the pointcut as a query at every joinpoint. In backward chaining, a logic query or pointcut is evaluated by finding rules to evaluate the conditions in the pointcut, and recursively finding rules for the conditions in those rules. In other words, using the graphical representation, the query is evaluated from the bottom to the top. The process stops when it can find logic facts for all of the conditions, meaning the query or pointcut follows logically from the facts. Resolving this query using backward chaining results in a bottom-up traversal of

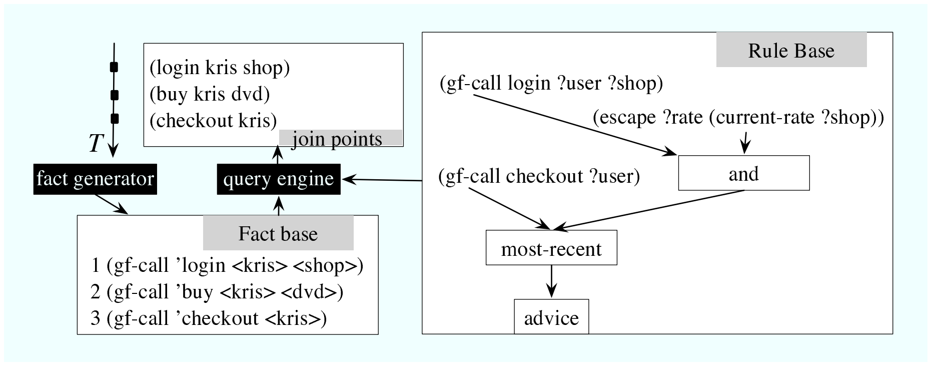

(login kris shop)
(buy kris dvd)
(checkout kris)
join points
T
fact generator
query engine
Fact base
1 (gf-call 'login <kris> <shop>)
2 (gf-call 'buy <kris> <dvd>)
3 (gf-call 'checkout <kris>)
Rule Base
(gf-call login ?user ?shop)
(escape ?rate (current-rate ?shop))
and
(gf-call checkout ?user)
most-recent
advice

**Fig. 5.** Execution of a program represented as facts. HALO pointcut represented as a tree.

the tree depicted in rule base of Figure 5. In order to resolve the query, a `most-recent` relation must hold between the result of resolving the left-input and the right-input of the node labelled `most-recent` . This requires to search the fact base for a fact that matches the pattern `(gf-call 'checkout (?user))`, another fact that matches the pattern `(gf-call 'login (?user ?shop))` and given those bindings to resolve the `escape` condition by executing its piece of Lisp code. Escape conditions make it possible to expose context from the base program (see Section 3.3); When combined with temporal operators, escape conditions expose *past* program context. Coming back to our example, this means the `escape` condition should be evaluated in relation to the program state when the `login` join point occurred. In other words, the promotional rate exposed via `?rate` through the `escape` predicate, needs to be bound to the discount rate active at login time. However, backward chaining does not support this semantics. At any time between the `login` join point and resolving the pointcut, the state of the object `<shop>` might have changed.

Supporting the evaluation of `escape` conditions at the right time fits better in the model of forward chaining, particularly the Rete forward chaining algorithm. When using a forward-chained query engine, the relationship between weaver and query engine is reversed. Rather than the weaver invoking the query engine to check if a pointcut matches, the query engine responds to changes in the fact base and informs the weaver if there are any new matches for pointcuts. In the Rete forward chaining algorithm, a representation for pointcuts similar to the one in Figure 5 is used. This representation is extended with memory. The memory serves to remember partial matches for pointcuts. Overall, the algorithm works as follows: When a fact is inserted in the fact data base, find all rules for which the fact matches a condition and try to resolve the rule given the fact base at that time; In addition, it records all conclusions found in-between in the fact base. So in the example depicted in Figure 5, when the fact `(gf-call 'login <kris>)` is inserted in the fact base, the `escape` condition of the rule depicted in the same figure is evaluated and asserted in the fact base. At a later time, when the fact `(gf-call 'checkout <kris>)` is asserted, it is combined with the solution memorized for the match to the partial pointcut `((gf-call 'login (?user ?shop)) (escape ?rate (current-rate ?shop)))`. Note that the `escape` condition is thus evaluated at the time the joinpoint happens that matches this part of the pointcut, thus implementing the HALO semantics. In the next section we discuss how the Rete algorithm is further extended to support HALO's temporal operators and discuss how its apparent drawback on memory usage can be optimized in HALO.

## 4.3 Temporal Extensions to Rete

In this section we discuss how the Rete algorithm can be extended to support HALO's temporal predicates and its `escape` mechanism. We begin by briefly discussing the standard Rete algorithm [4] .

*Basic Rete* Rete represents rules – or pointcuts in HALO– as a network consisting of nodes containing a memory table. For each condition in a rule, the network contains a "filter" node. For each logical "and" between conditions in rules, it contains a "conjunctive" join node. When new facts are added, they are inserted in the filter nodes. A filter node checks whether the fact unifies with its condition, and if so, memorizes it in its memory table and notifies the join node that it is linked to. For example, the filter node for the condition `(test 3 ?x)` will memorize a fact `(test 3 4)` but not a fact `(test 5 4)`. Conjunctive join nodes have an incoming link from one filter node and another join node or filter node. When a conjunctive join node is notified that a new fact was memorized, it checks whether this fact matches with the facts memorized by the other incoming node. Specifically, it checks whether they have the same values for common variables. If this is the case, this combination is memorized and the next join node is notified.

As an example, consider the query given below. The Rete network for the query is shown in Figure 6. The figure shows the state of the memory tables after adding the fact shown on the left of the figure. Note that the notation `<name>` is used for an object identifier (a pointer in the actual implementation).

```
((gf-call ?operation (?arg1 ?arg2))
 (gf-call 'foo (?arg1 ?arg2))
 (gf-call ?operation (1 ?arg2)))
```

As the rule has three conditions, the Rete network contains three filter nodes (the circles). These filter nodes are connected to one another by means of conjunctive join nodes (the squares). The bottom node (the triangle) is a query conclusion node. When it is notified of new facts, it means a solution for the query was derived. This is the case in this example, where the bottom node is triggered for

[4] We discuss the original Rete algorithm. Improved versions, like Rete II and Rete III exist, but unlike the original Rete, are proprietary algorithms that have not been published.

the match where `?operation` has the value `foo`, `?arg1` the value 1 and `?arg2` the value `<lotte>`.

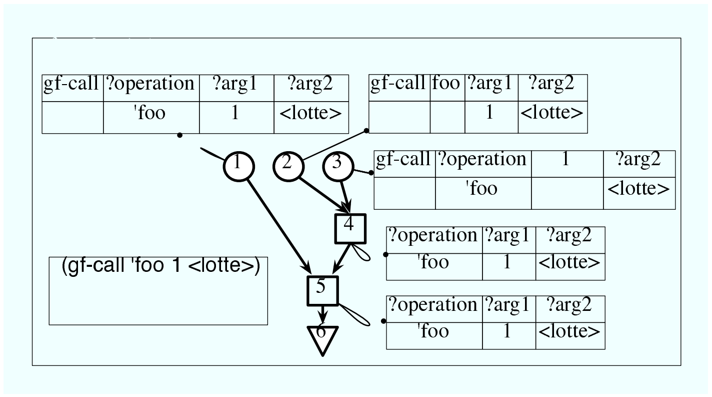


**Fig. 6.** A standard Rete network.

*Temporal frames in memory tables* In standard logic, any fact is unambiguously "true." In temporal logic however, facts are true in a temporal frame, a certain moment in time.[5] To support this, memory tables are extended to record in which temporal frame their entries are considered true. In Figure 7, this is the gray column in the left of memory tables.

*Temporal join nodes* Supporting the temporal operators in Rete is done by introducing new types of join nodes. One new type of join node is added for each of the temporal operators `most-recent`, `all-past`, `since` and `cflow`. When temporal join nodes are notified of new incoming facts, they combine the new facts with those in the memory table of its other incoming node similar to conjunctive join nodes in regular Rete. The join nodes in the extended Rete are restricted to combining entries that meet constraints on the temporal frames. Figure 8 displays the different temporal constraints for the different types of join nodes: $T_{left}$ and $T_{right}$ refer to the temporal frames associated with the outer and inner pointcut respectively (which are always depicted as respectively the left and right inputs of the temporal join node). In the case of the `since` operator, $T_{left}$ refers to the outer pointcut, $T_{right}$ points to the second argument of `since` and $T_{middle}$ points to the first argument. The behavior of the `most-recent` and `all-past` join nodes further differs in that an `all-past` passes all matches to its output node, while a `most-recent` join node only passes one match. Specifically, when a new entry is made in its left input node, it tries to match it with the entries

[5] The term "temporal context" is used in literature, but we use "temporal frame" to avoid confusion with "context" in the sense of joinpoint context data.

in its right memory table, starting from the most recent entry and only passes the first successful match.

For example, Figure 7 shows the network for the following pointcut:

```
((gf-call 'checkout (?user))
 (most-recent (gf-call 'buy (?user ?article))))
```

The network consists of one temporal join node for the `most-recent` condition, its left input is a filter node for the one condition in the outer pointcut `(gf-call 'checkout (?user))`, and its right input is a filter node for the one condition in the inner pointcut `(gf-call 'buy (?user ?article))`.

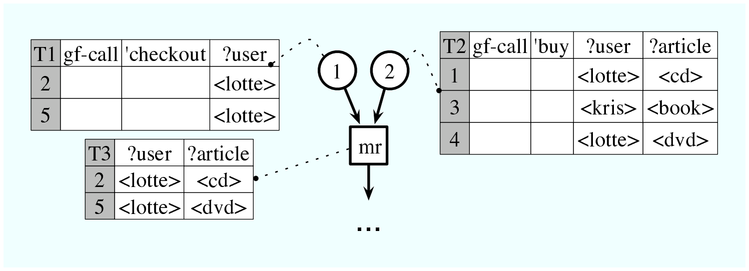


**Fig. 7.** Rete containing a square-shaped temporal join node.

Figure 7 also shows the state of the memory tables after processing the following series of fact insertions. At time 1, a fact `(gf-call 'buy <lotte> <cd>)` is inserted which matches only the right filter node and is memorized. The join node is notified, but as the memory table of its left input is empty, it does not do anything. At time 2, a fact `(gf-call 'checkout <lotte>)` is inserted which matches only the left filter node and is memorized. As the join node is notified, it combines the new entry of its left input with the most recent matching entry in the memory table of the right input node. The entries match if they have the same values for the common variables and if the constraint $T_{right} < T_{left}$ between the temporal frames of the entries is met. The entry that was made at time 1 in the right filter node matches, as it also has the object `<lotte>` as value for the variable `?user`. Thus, this combination is memorized in the join node's memory table. At time 3, the fact `(gf-call 'buy <kris> <book>)` is inserted and at time 4 the fact `(gf-call 'buy <lotte> <dvd>)`. At time 5 the fact `(gf-call 'checkout <lotte>)` is inserted. It matches the left filter node, so the join node is notified. The join node combines the new entry with the right filter node's memory table. Two combinations are possible: one with temporal frame 4 and one with temporal frame 1, because both have the object `<lotte>` as value for `?user`. However, due to the recent matching semantics of the `most-recent` operator, only the combination with the entry of temporal frame 4 is made. A new entry is thus made in the join node's memory table which is true at temporal frame 5, with the object `<dvd>` as value for the variable `?article`. Conversely, were the operator `most-recent` replaced by the operator `all-past`, all matching combinations would have been memorized.

– • `and`: $T_{left} = T_{right}$
  • `most-recent` $T_{right} < T_{left}$
  • `all-past`: $T_{right} < T_{left}$
  • `since`: $T_{right} > T_{middle}$ and $T_{right} < T_{left}$ and $T_{middle} < T_{left}$

**Fig. 8.** Constraints in temporal join nodes.

*Control flow join nodes* Control flow join nodes operate slightly differently from `most-recent` join nodes. Figure 9 shows the Rete network for the following pointcut:

```
((gf-call 'update-line ?args1)
 (cflow (gf-call 'update-figure ?args2)))
```

The memory table for a `cflow` join node's right input node is extended to record the time at which the return of the captured joinpoint occurs. When a return joinpoint is encountered, the weaver notifies control flow join nodes of the time of the corresponding invocation joinpoint. The nodes that have an entry for that time in their right input node's memory table add the time at which the return occurred. The entry will no longer be used to make combinations with entries coming from the left node.

Figure 9 reflects the Rete after the insertion of the following conclusions. At time 1, a call to the generic function `update-figure` occurs and a fact `(gf-call 'update-figure <fig1>)` is inserted in the network, making an entry in the memory table of the first filter node. Immediately thereafter, the weaver detects the return of that same generic function call and notifies the control flow join node: The return time, namely 2, is added to the entry for the generic function call in the memory table of the temporal join node's right input node. If subsequently at time 3, a fact `(gf-call 'update-line <line1>)` is added, this is memorized in the first filter node and the control flow join node is notified. However, as the control flow join node cannot find an unfinished generic function call entry in its right input node for which the constraint $T2 < T1$ succeeds, it cannot memorize a conclusion. Assume that next at time 4, the fact `(gf-call 'update-figure <fig2>)` and at time 5, the fact `(gf-call 'update-line <line2>)` are inserted in the network. This time, when the temporal join node is notified, it is able to derive a conclusion as it can combine the entry memorized at time 4 in its left input node with the entry memorized at time 5 in its right input node. As such, the Rete network concludes that the invocation of the generic function `update-line` with argument `<line2>` is in the control flow of the call to the generic function `update-figure` with an argument `<fig2>`.

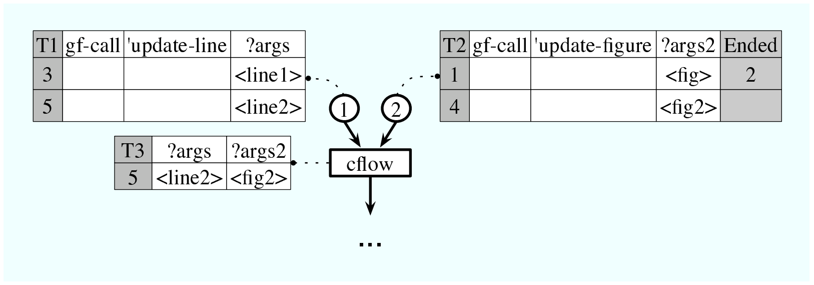


**Fig. 9.** Rete containing a control flow join node.

*Escape nodes extension* Another extension to the Rete algorithm is used to handle the `escape` conditions. Such conditions are represented as nodes in the Rete network similar to join nodes, though they only have one input. Figure 10 shows the Rete network for the following pointcut:

```
((gf-call 'checkout (?user))
 (most-recent
    (gf-call 'buy (?user ?article))
    (escape ?rate (current-discount-rate (singleton-instance 'promotions) ?article))))
```

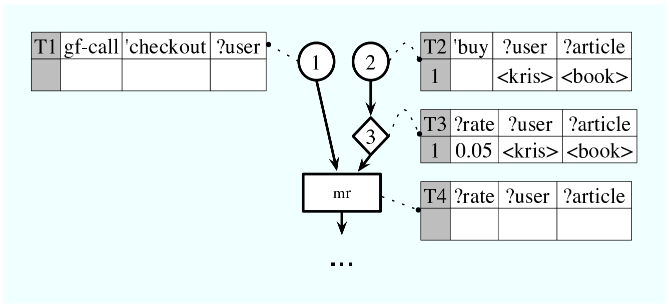


**Fig. 10.** Rete containing an diamond-shaped escape filter node.

When an escape filter node is notified of a memorization in its input node, the Lisp form is executed after all logic variables are replaced by the values from the received notification. Subsequently, if the result of this evaluation is different from `nil`, it is memorized in the escape filter node and the escape filter node's output node is notified. For example, if a fact `(gf-call 'buy <kris> <book>)` is inserted in the network from Figure 10, the escape filter node is notified and evaluates the Lisp form `(current-discount-rate (singleton-instance 'promotions) <book>)`.

## 4.4 Memory Table Garbage Collection

It is not very economical to keep all the entries in the memory tables in the Rete network during the entire run of the program. Due to the semantics of the temporal predicates, certain entries in the memory tables can become irrelevant as they will never produce new combinations. We show how this is exploited to provide automatic garbage collection of such irrelevant entries, and discuss which nodes do not actually need to keep a memory table at all. [6]

*Entries no longer most recent* Entries in the right input memory table of `most-recent` temporal join nodes, can be removed when new entries with the same values for the variables are added. In fact, only the values for the variables that are in common with the left input memory table need to be the same. This is because when an entry is added to the left input's memory table, the join node will combine it with the most recent matching entry in the right input node. The match requires that the values for the variables in common between the two input nodes are the same. Thus, if there is an older entry in the right memory table that also matches with the new entry in the left, it will still not produce a combination. Thus, such entries can be removed.

Consider the example of Figure 7 discussed previously. At time 4, an entry is made in the memory table of the join node's right input node. The entry of time 1 can then be removed because it has the same value for the variable `?user`. Note that the values for the variable `?article` are different, but this variable is not used in the join node's left input node.

Figure 11 gives an example with nested `most-recent` predicates for the following piece of advice:

```
(at ((gf-call 'checkout ?user1)
      (most-recent (gf-call 'checkout ?user2)
           (most-recent (gf-call 'buy ?user2 ?article2))))
      (format t "~s just bought ~s" ?user2 ?article2))
```

A sample program run is depicted in the same figure. In addition, the figure displays tables labelled `LT` (life time): The intervals stored by these tables indicate the begin and end point for the interval during which entries in the memory tables are kept. Note that though the entries in the third filter node are removed as new entries are made, the derived conclusions are not also removed at the same time: at time 7 for example, when the entry made for `(gf-call 'buy <lotte> <dvd>)` is removed, the derived conclusion for time 5 in the first `most-recent` join node is kept. This ensures that at time 8 it can be used to match the pointcut. But this does not mean the derived conclusion is kept forever. The first `most-recent` join node is itself the input of another `most-recent` join node. The input nodes of this second join node share no variables. So the

[6] Dynamically adding pointcuts would raise the issue of whether these can match against facts produced before the addition. If so, this creates a conflict with the ability to optimize the history. HALO does not support dynamically adding pointcuts at this time.

entry for time 5 in the output memory table of the first join node is removed when any other entry is made, which in this example will happen the next time a user checks out if he bought something (e.g. if the user `lotte` does another checkout).

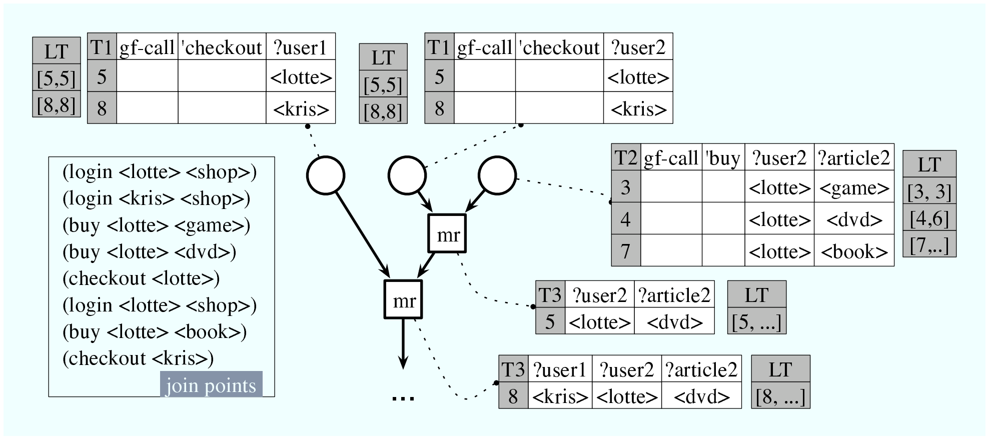


**Fig. 11.** Garbage collection of nested temporal join nodes.

*Combinations of* `since` *and* `most-recent` When the first argument pointcut of a `since` condition is a `most-recent` condition, the memory tables for the second argument pointcut's network can be garbage collected whenever entries are removed from the `most-recent` node's memory table. For example, consider the Rete network shown in Figure 12 for the following piece of advice that makes sure a discount is given for each article bought during a single shopping session:

```
(at
    ((gf-call 'checkout (?user))
      (since (most-recent (gf-call 'login (?user ?shop)))
             (all-past (gf-call 'buy (?user ?article)))))
     (discount ?article (current-discount-rate
                          (singleton-instance 'promotions)
                          ?article)))
```

Intuitively, in this pointcut, the joinpoints in the history for the buy calls of a user can be removed once she logs in again. This is illustrated for the sample program depicted in the figure, and the Rete network is shown after the execution of the entire program. When the second call `(login <lotte> <shop>)` at time 5 happens, the entry labelled 1 is removed in the right input node of the `most-recent` join node (table T2). This also implies we can safely remove all entries memorized in the right input network of the `since` join node, which have the same binding for the variable `?user`, namely `<lotte>`, which were made before time 5. This is because the temporal constraint of the `since` temporal join node, which is $T_3 > T_2$ in Figure 12, will never be fulfilled for those entries anymore.

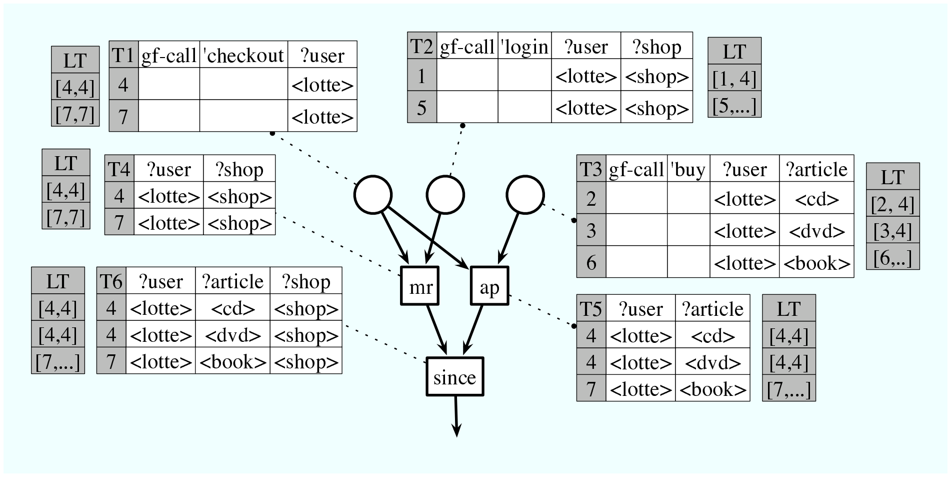
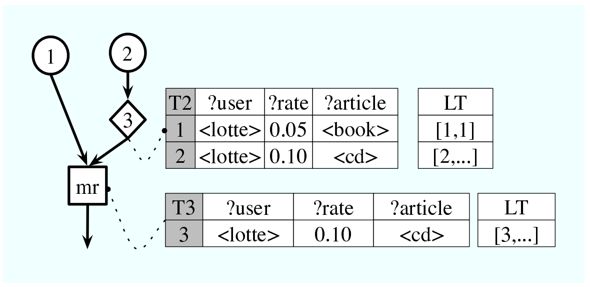


**Fig. 12.** Rete network illustrating automatic garbage collection for `since`.

*Nodes without memory* Not all nodes need to keep a memory table, the exceptions are: nodes that are the left input of a temporal join node (thus not conjunctive join nodes), the last node that triggers the advice code, and nodes that are the input of an escape node. Keeping a memory table for the left input of a temporal join node is not necessary: new entries coming in from the right conceptually need to be matched with entries from the left, but they can only match if the left entries are in a temporal frame which is in the future of the right entry. Due to the order in which facts are added by the weaver, such entries can not yet exist. An escape node never consults the memory of its input node, rather whenever new facts come in through its input, it executes Lisp code and records the result in its own memory table. Thus the input node of an escape does not actually need to keep a memory table. The last node that triggers the advice code does not need to keep a memory table: these are never joined to other nodes for deriving conclusions.

As an example, the Rete network of Figure 10 is depicted again in Figure 13, annotated with the life time of facts. The two filter nodes no longer have a memory. The second figure shows the network after a different series of insertions: the facts `(gf-call 'buy <lotte> <book>)` at time 1 and `(gf-call 'buy <lotte> <cd>)` at time 2. When the first fact is inserted, assuming the discount rate of the `<book>` is 0.05, this is memorized in the escape node . Similarly, as the second conclusion is inserted and if the discount rate of the `<cd>` object is 0.10, this is also memorized. In addition, the previous conclusion of time 1 memorized in the escape node is deleted, as this conclusion will not be used anymore to derive conclusions by the most-recent join node.



| T2 | ?user | ?rate | ?article | LT |
|---|---|---|---|---|
| 1 | <lotte> | 0.05 | <book> | [1,1] |
| 2 | <lotte> | 0.10 | <cd> | [2,...] |

| T3 | ?user | ?rate | ?article | LT |
|---|---|---|---|---|
| 3 | <lotte> | 0.10 | <cd> | [3,...] |

**Fig. 13.** Rete network from Figure 10 annoted with life time of facts for garbage collection

*Entries marked as no longer used* Obviously, in the case of a `cflow` join node, the entries that are marked as "no longer used" can in fact simply be removed. Note that this means that older entries in the memory table can become the most recent match again. This is why `cflow` nodes are an exception to the first case for garbage collection described above.

## 4.5 Further Optimization Strategies

The memory table garbage collection optimization strategy outlined in the previous sections is orthogonal to the shadow weaving optimization strategy performed in other logic-based pointcut languages [6, 4]. In that optimization, which is based on abstract interpretation and partial evaluation, the pointcuts are statically analyzed to determine which joinpoints never affect the matching of a pointcut so that the weaver does not need to intercept these joinpoints. That optimization can be adapted to HALO as well: for each shadow point one can record which filter nodes the join points from that shadow *can* match against. In many cases, this will actually be zero filter nodes, and thus no joinpoint fact needs to be generated at all. In other cases, only a small subset of the filter nodes will have to be checked. In any case, we note that the shadow weaving optimization strategy is orthogonal to optimizing the joinpoint history dynamically. Shadow weaving optimizes the joinpoint history so that joinpoints that are *never* relevant are not intercepted. The technique discussed in this paper optimizes the joinpoint history so that joinpoints that *are* relevant are removed when they are *no longer* relevant. This is further explained in the related work section in comparison with Alpha.

## 5 Related Work

*OReA & Hybrid Pointcuts* We have simplified the "hybrid pointcuts" mechanism [5] in this paper to an explicit `escape`. In the work of D'Hondt on OReA [5], the goal of "hybrid advices" is to be transparent: a condition in a logic pointcut can be re-defined as a method, and vice-versa. The pointcut language and base language are changed so that when no rule is defined for a logic condition, the condition will be evaluated by sending a message instead. This can be easily achieved in HALO as well: if no rule exists for a logic condition, it is translated to an `escape` condition. But we have not demonstrated this in this paper in order to focus on the issue of supporting "hybrid pointcuts" in a language like HALO that supports pointcuts over a history of joinpoints. OReA also supports interaction from the base and advices languages with the rule language, which we have not considered in HALO so far. OReA is actually a family of logic pointcut languages, which includes a forward-chaining-based variant. But this is not based on the Rete network and lacks the necessary support for memorizing past evaluations of hybrid pointcuts. While OReA supports hybrid pointcuts in both directions in a transparent manner, it does not support pointcuts over a history of joinpoints.

*Alpha* A closely related approach to our work is Alpha [4], a logic-based pointcut language for expressing pointcuts over a history of joinpoints. Alpha includes information about the state of objects and the static structure of the program in the fact base. Full Prolog can be used to write pointcuts as logic queries over the historic fact base. A pre-defined set of logic rules for expressing temporal relations is provided, but this can be extended by the programmer. While Alpha also has a mechanism for letting the pointcut language interact with the base program, as discussed in Section 4.2, the use of standard Prolog only allows interaction with the base program at the current joinpoint. So (as discussed in Section 2), this means the "past rate discounting" aspect must be expressed as two pointcuts and advices. Thus, while Alpha is more expressive than HALO in terms of providing a richer joinpoint model and the use of full Prolog to reason about the past history of joinpoints, it is also less expressive in allowing hybrid pointcuts to interact with the base program. Because of the open set of temporal predicates, partial evaluation of pointcuts is used to optimize the memory required to keep the historic fact base. The analysis is done statically and determines which joinpoints may possibly affect pointcuts. For these, the shadow weaving technique well-known in aspect weaver construction is applied so that only those joinpoints are actually intercepted. A similar technique can be used in HALO, though this is an area of future work. In Alpha, the joinpoints that are intercepted are kept in the fact base indefinitely, except if the static analysis can determine that they are only used in matching pointcuts as the current joinpoint and not as past joinpoints. Thus, if a pointcut expresses the equivalent of HALO's `most-recent` predicate, information about all joinpoints matching the `most-recent` condition is kept indefinetely. In contrast, in HALO, the set of temporal predicates is fixed, which means the implementation knows about the semantics of the predicates, which is used to perform a dynamic analysis of the fact base so that matches are removed from memory tables if they are no longer relevant.

*Context-Aware Aspects in Reflex* Reflex is a kernel for multi-language aspect-oriented programming, implemented as an object-oriented framework. In earlier work, the framework was extended with the necessary support for context-aware aspects, which also combines interacting from the pointcut language with referring to past joinpoints [18]. Context definitions can be implemented in the framework as objects with a method that indicates whether the context is active. The proposed pointcut language does not allow pointcuts over past joinpoints. Rather, the framework provides support for defining "context restrictors" that can be used in a pointcut to restrict it not just based on the current joinpoint but also on past activation of a context: for example, depending on whether a context was active during the creation of the object in which the current joinpoint occurs. Internally, these restrictors add additional pointcuts and advices to the program to capture the state of the context objects for later reference. In HALO, this splitting of pointcuts into parts that are evaluated at different times, and keeping the past state exposed by contexts automatically, arises from the Rete network.

*EAOP, J-Lo & Tracematches* In several other approaches that allow pointcuts over a history of joinpoints, including EAOP [19], Tracematches [3] and J-LO [14], implementation strategies based on state machines are investigated. The state machines are used to evaluate temporal relations between pointcuts, which in the Tracematches and J-LO approaches, are expressed in AspectJ. The state machines formalism inherently does not support a memory, thus when variable sharing is allowed between the non-temporal AspectJ pointcuts, this requires an additional form of memory. The logic chaining formalism we have started from in this paper on the other hand inherently uses such a memory. As for interaction with the base language, current versions of Tracematches extend AspectJ with a `let` pointcut similar to the `escape` discussed in this paper [20], but this mechanism is not covered in [3] and [21], and only examples for accessing the current joinpoint reflection object are discussed in [22]. Furthermore, a `let` pointcut condition is limited to using variables from its enclosing symbol, meaning only variables defined at the current joinpoint, in contrast with HALO's `escape` predicate which also allows use of variables defined at past joinpoints.

*Rete* Work by Teodosiu and Pollak [23], and more recent work by Berstel [24] propose extensions of the Rete algorithm for temporal event management. No foundation based on temporal logic is considered, meaning temporal constraints are expressed over explicit timestamps, and no higher-order predicates for expressing temporal relations are provided. Furthermore, the temporal constraints always involve a fixed interval of past event, which is motivated by the need to garbage collect memory table entries. In contrast, we have shown how an appropriate most recent joinpoint matching semantics for the temporal predicates still allows for garbage collection.

## 6 Conclusions and Future Work

As stated in the introduction, the contributions of this paper are three-fold. Firstly, we introduced a novel logic-based pointcut language which has features for expressing pointcuts over a history of joinpoints and allowing interaction with the base language. We introduced a novel such language, HALO, which is based on temporal logic and supports `escape` conditions.

Secondly, we introduced forward chaining as an implementation mechanism for such a language. Earlier work on similar logic-based languages such as Alpha and OReA supported either feature but not a combination of both. As we have shown, the backward chaining evaluation strategies used in these approaches are insufficient for supporting the combination. We demonstrated the Rete forward chaining algorithm as a particular forward chaining algorithm that can support a language that combines both features; we showed how Rete can be extended with support for verifying temporal relations between facts and interacting with the base language.

Thirdly, we demonstrated how the Rete network can be further optimized so keeping a full history of joinpoint facts is not necessary. Only nodes that are the right input of a temporal node actually need a memory table. Furthermore, because the set of temporal predicates is built-in in the language (rather than an open set), the known semantics of these predicates can be exploited to perform a dynamic analysis of the memory tables. Certain nodes can perform a garbage collect of their previous entries in the memory table when new entries are made.

As future work, we consider extending HALO with predicates that offer a static model of the base application, as in other logic-based pointcut languages [6, 4]. Predicates for such a model could be easily added, and would exist in the temporal Rete network as facts that are eternally true. HALO does not currently support recursive rules, which have so far not been proven useful in our examples, and the restriction allows rules to simply be inlined. In previous work, recursive rules have been proven useful for writing pattern-based pointcuts that detect recursive patterns in the static model of the base application [6]. Rete can support recursive rules, but the impact of our additions, especially the `escape` predicate, needs to be further investigated. Furthermore, while the temporal relations expressed by the current set built-in higher-order temporal operators are similar to the pre-defined time stamp comparison predicates used in Alpha [4], the use of full Prolog in the latter potentially allows additional temporal relations to be expressed. We are currently investigating additional temporal predicates for expressing more interesting temporal relations.

## References

1. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: Proceedings of the European confer-

ence on Object-Oriented Programming, Springer-Verlag (jun 1997)

2. Kiczales, G., Mezini, M.: Separation of concerns with procedures, annotations, advice and pointcuts. In: European Conference on Object-Oriented Programming, ECOOP 2005. (2005)
3. Allan, C., Avgustinov, P., Christensen, A.S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G., Tibble, J.: Adding trace matching with free variables to aspectj. In: OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications, New York, NY, USA, ACM Press (2005) 345–364
4. Ostermann, K., Mezini, M., Bockisch, C.: Expressive pointcuts for increased modularity. In: European Conference on Object-Oriented Programming. (2005)
5. D'Hondt, M., Jonckers, V.: Hybrid aspects for weaving object-oriented functionality and rule-based knowledge. In: Proceedings of the Fourth International Conference on Aspect-Oriented Software Development. (2004)
6. Gybels, K., Brichau, J.: Arranging language features for more robust pattern-based crosscuts. In: Proceedings of the Second International Conference on Aspect-Oriented Software Development. (2003)
7. Havinga, W., Nagy, I., Bergmans, L., Aksit, M.: Detecting and resolving ambiguities caused by inter-dependent introductions. In: Proceedings of 5th International Conference on Aspect-Oriented Software Development, AOSD2006. (2006)
8. Windeln, T.: Logicaj - eine erweiterung von aspectj um logische meta-programmierung. Diploma thesis, CS Dept. III, University of Bonn, Germany (Aug 2003)
9. Herzeel, C., Gybels, K., Costanza, P.: Modularizing crosscuts in an e-commerce application in Lisp using HALO. In: Proceedings of the International Lisp Conference 2007. (2007)
10. Bobrow, D., DeMichiel, L., Gabriel, R., Keene, S., Kiczales, G., Moon, D.: Common lisp object system specification. Lisp and Symbolic Computation **1**(3-4) (January 1989) 245–394
11. Kowalski, R.: Predicate logic as programming language. In: IFIP Congress. (1974) 569–574 Reprinted in Computers for Artificial Intelligence Applications, (eds. Wah, B. and Li, G.-J.), IEEE Computer Society Press, Los Angeles, 1986, pp. 68–73.
12. Cohen, J.: Describing prolog by its interpretation and compilation. Commun. ACM **28**(12) (1985) 1311–1324
13. Gergatsoulis, M.: Temporal and modal logic programming languages. In Kent, A., Williams, J.G., eds.: Encyclopedia of Microcomputers. Volume 27., New York, Marcel Dekker, Inc (2001) 393–408
14. Bodden, E.: J-LO - A tool for runtime-checking temporal assertions. Master's thesis, RWTH Aachen university (2005)
15. Brzoska, C.: Temporal logic programming with bounded universal modality goals. In: Proceedings of the Workshop on Executable Modal and Temporal Logics, London, UK, Springer Verlag (1993) 21–39
16. Kiczales, G., des Rivières, J., Bobrow, D.G.: The Art of the Metaobject Protocol. MIT Press (1991)
17. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach. 2nd edition edn. Prentice-Hall, Englewood Cliffs, NJ (2003)
18. Tanter, É., Gybels, K., Denker, M., Bergel, A.: Context-aware aspects. Lecture Notes in Computer Science, Proceedings of the 5th International Symposium on Software Composition (SC 2006) **4089** (2006) 227–242

19. Douence, R., Motelet, O., Südholt, M.: A formal definition of crosscuts. In: REFLECTION '01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, London, UK, Springer-Verlag (2001) 170–186
20. Sittampalam, G.: Abc version 1.1.1 release announcement http://abc.comlab.ox.ac.uk/archives/announce/2006-Mar/0000.html.
21. Avgustinov, P., Bodden, E., Hajiyev, E., Hendren, L., Lhotak, O., de Moor, O., Ongkingco, N., Sereni, D., Sittampalam, G., Tibble, J., Verbaere, M.: Aspects for trace monitoring. In: Invited paper at FATES/RV 2006. (2006)
22. Avgustinov, P., Tibble, J., Bodden, E., Lhotak, O., Hendren, L., de Moor, O., Ongkingco, N., Sittampalam, G.: Efficient trace monitoring. Technical Report abc-2006-1, ABC Group (2006)
23. Teodosiu, D., Pollak, G.: Discarding unused temporal information in a production system. In: Int. Conf. on Information and Knowledge Management, Baltimore. (1992)
24. Berstel, B.: Extending the rete algorithm for event management. In: Proceedings of the Ninth International Symposium on Temporal Representation and Reasoning. (2002)

# Practical, Pluggable Types

Niklaus Haldiman, Marcus Denker and Oscar Nierstrasz

Software Composition Group
University of Bern – Switzerland
http://scg.iam.unibe.ch

**Abstract.** Most languages fall into one of two camps: either they adopt a unique, static type system, or they abandon static type-checks for run-time checks. Pluggable types blur this division by (i) making static type systems optional, and (ii) supporting a choice of type systems for reasoning about different kinds of static properties. Dynamic languages can then benefit from static-checking without sacrificing dynamic features or committing to a unique, static type system. But the overhead of adopting pluggable types can be very high, especially if all existing code must be decorated with type annotations before any type-checking can be performed. We propose a practical and pragmatic approach to introduce pluggable type systems to dynamic languages. First of all, only annotated code is type-checked. Second, limited type inference is performed on unannotated code to reduce the number of reported errors. Finally, external annotations can be used to type third-party code. We present TypePlug, a Smalltalk implementation of our framework, and report on experience applying the framework to three different pluggable type systems.

## 1 Introduction

Any static type system attempts to reduce the number of errors that may occur at run-time by restricting the programs that one is allowed to write. Statically-typed programming languages implicitly take the position that this is a good deal. In return for the minor inconvenience of not being able to write certain kinds of relatively uncommon programs, we obtain guarantees that no catastrophic run-time errors will occur.

Dynamically-typed languages assume the opposing standpoint. Precisely those interesting programs — for example, those exploiting behavioural reflection — would be rejected. Furthermore, although there are many different approaches to static typing, a statically-typed programming language must commit to one. Dynamic languages do not need to make this kind of premature commitment. Finally, there is the argument that static type systems are responsible for generating a false sense of security. Certain catastrophic errors can certainly be detected, but others cannot.

Gilad Bracha has argued that it is possible to have one's cake and eat it too [1]. Instead of static type systems being mandatory, they should be *optional.* This means that a type system should neither require syntactic annotations in the program source, nor should it affect the run-time semantics of the language. An optional type system would report possible errors, but would not prevent the program from being run (and possibly failing at run-time). In addition, type systems should be *pluggable*, that is, one should be able to choose the kind of static checks one would like to perform. This is especially interesting given the relatively recent emergence of more exotic type systems for reasoning about confinement [2], aliasing [3], scopes [4] and so on.

A number of pluggable type systems have been proposed over the last years [5–7]. The key difficulty with pluggable types, however, is a practical one: *In the presence of a large, existing software base, how can one benefit from a pluggable type system without annotating the legacy code?*

We present a practical and pragmatic approach to pluggable types that significantly reduces the overhead of adopting a new type system. TypePlug is a framework supporting pluggable types implemented in Squeak Smalltalk. Type systems are defined by specializing the framework. Types are declared using an annotation framework. TypePlug reduces the cost of adopting a type system by means of the following techniques:

1. Only annotated code is type-checked.
2. Type inference is applied to unannotated code to reduce propagation of errors.
3. Explicit type casts shortcut type-checking.
4. External annotations are supported to declare types for third-party code.

Section 2 provides a brief introduction to TypePlug using an example of a non-nil type system. In Section 3 the TypePlug framework is described in some detail. Section 4 discusses experience using TypePlug to define a class-based type system and a confinement type system. Section 5 discusses the results obtained in the context of related work. We conclude in Section 6 with some remarks on future work.

## 2 TypePlug in a Nutshell

TypePlug provides a framework to optionally annotate code with type declarations and to type-check annotated code. We will demonstrate how optional types and type checking can be used in a typical coding session.

To keep the presentation as simple as possible, we use the example of a non-nil type system. This type system has exactly one type, the nonNil type. If a variable has the nonNil type it cannot hold the value nil. Anything not typed nonNil is considered to potentially evaluate to nil (there is no explicit nil type).

To declare a variable to be of type nonNil we add the annotation **<:nonNil :>** to it. In the following class definition of a two-dimensional line we declare the instance variable endPoint to be nonNil:

```
Object subclass: #Line
    uses: TReflectiveMethods
    typedInstanceVariables: 'startPoint endPoint <:nonNil :>'
    typedClassVariables: ''
```

Notice the trait[1] TReflectiveMethods that we use in this class. It pulls in the annotation framework that we need to annotate its methods. We assume that we have not added any other annotations yet to the methods of this Line class. If we now browse the source of one of the methods we can experience type checking in action. Here is a method of a line that moves it horizontally by a number of units:

```
moveHorizontally: anInteger
    startPoint := self movePoint: startPoint horizontally: anInteger.
    endPoint := self movePoint: endPoint horizontally: anInteger <– type 'TopType' of
         expression is not compatible with type 'nonNil' of variable 'endPoint'.
```

When the source of a method is displayed it is type-checked and type errors are shown inline. The type error found in the above method is highlighted. What it says is that the expression self movePoint: endPoint horizontally: anInteger was found to have the top type which means that its type is *unknown*. Since the instance variable endPoint requires the type nonNil, this results in a type error.

To fix this error we have to look into the method movePoint:horizontally: and declare its return type to be nonNil. This is what it looks like:

```
movePoint: aPoint <:nonNil :> horizontally: anInteger
    ↑ (aPoint addX: anInteger y: 0) <:nonNil :>
```

A return type annotation must be added to the return statement expression. Notice how parentheses are needed here to apply the annotation to the whole expression. Adding a return type to a method has two repercussions: in call sites of the method the message send expression will be typed accordingly and within the method its return statements will be type-checked. Here this means that the expression aPoint addX: anInteger y: 0 must be of type nonNil (assume we annotated the addX:y: method with a nonNil return type already, so no type error occurs here).

To show an example of an argument type we annotated the first argument of movePoint:horizontally: with a nonNil type as well. Adding an argument type declares the argument to be of that type within the method, and it is also a requirement to arguments passed when the method is used. Because of this requirement we now get a different type error when we look at moveHorizontally: again:

```
moveHorizontally: anInteger
    startPoint := self movePoint: startPoint horizontally: anInteger <– in message
         'movePoint:horizontally:' of class 'Line' argument 'TopType' is not compatible
         with expected type 'nonNil'.
    endPoint := self movePoint: endPoint horizontally: anInteger.
```

[1] A trait is a set of methods that can be included in the definition of a class [8].

The startPoint instance variable does not have an annotation so its type is the default, the top type, which is not what is required for the first argument to movePoint:horizontally:. The obvious fix to this type error is to declare startPoint to be nonNil as well. A different strategy that TypePlug offers is to *cast* an expression to an arbitrary type, in this example by adding the annotation **<:castNonNil :>**.

```
moveHorizontally: anInteger
    startPoint := self movePoint: startPoint <:castNonNil :> horizontally: anInteger.
    endPoint := self movePoint: endPoint horizontally: anInteger.
```

With a cast a programmer asserts to the type checker that the type of a given expression is known. Casts are useful for one-off resolutions of type errors where more type annotations would be overkill or impossible, for example because the cast expression uses untyped third-party code.

Let's annotate one more method of Line with a return type:

```
hasStartPoint
    ↑ (startPoint notNil) <:nonNil :>
```

We are still operating under the assumption that no other type annotations have been made than those previously discussed. We surely have not annotated any notNil method with types, so how come we don't get a type error here because the expression startPoint notNil can't be proven to be nonNil? The reason is that the type checker tries to infer return types of methods if they are not explicitly annotated. The notNil method has two implementations, one in the base class Object — simply returning false — and one in UndefinedObject — simply returning true. So obviously invocations of notNil will always return a nonNil boolean value, which is what the type checker figured out in this case, sparing us from explicitly annotating the notNil method with a return type.

While this preceding coding session serves as a lightweight introduction to TypePlug, it also illustrates an important issue when programming with optional types. Once one adds just one type to previously untyped code the need for more type annotations usually arises to type-check other parts of the code. Most likely, these additional types prompt for even more type annotations. This means that quite a bit of work is usually required from a programmer if he introduces types in a section of his code. A framework for optional typing should be aware of this problem and minimize the work required by the programmer. As demonstrated in the coding session, two of our strategies in this area are type inference and casts, reducing the need for explicit type annotations.

## 3 The TypePlug Framework

A type system in TypePlug is specified by first defining the types of the system and then the properties of the types. This mainly consists of defining mappings from Smalltalk elements to types as well as operations on types. Since a key goal of TypePlug is to enable the creation of new type systems without much effort, the number of properties that have to be defined is kept to a minimum while retaining considerable flexibility.

Concretely, a new type system is created by subclassing the TPTypeSystem class and overriding some of its methods. As a quick reference and overview, Table 1 lists all of the methods of TPTypeSystem that can be overridden when implementing a new type system. In the following discussion we will touch on the details of the most important of these methods.

| | |
|---|---|
| systemKey | unique key for this system. *Defined as class method.* |
| annotationValueToType:inContext: | converts annotations to types. *Abstract, must be overridden.* |
| is:subtypeOf: | defines the subtyping relation. *Abstract, must be overridden.* |
| unifyType:with: | defines the type unification operation. *Abstract, must be overridden.* |
| typeForArray: | maps arrays to types. |
| typeForBlock: | maps blocks to types. |
| typeForGlobal:value: | maps global variables to types. |
| typeForLiteral: | maps literals to types. |
| typeForPrimitive: | maps primitives to types. |
| typeForPrimitiveNamed:module: | maps named primitives to types. |
| typeForSelfInClass: | maps self pseudo-variables to types. |
| typeForSuperInClass: | maps super pseudo-variables to types. |
| methodsForMessage:... | customizes the set of methods to be considered when type checking message sends. |
| transformMethodType:... | transforms methods type before they are used in the type checker. |
| transformType:... | transforms types before they are used in the type checker. |
| assignmentTo:... | customizes type checking for assignments |
| displayClass: | creates a custom string version of classes. |
| displayMethod:in: | creates a custom string version of methods. |

**Table 1.** Methods of TPTypeSystem to override in subclasses

## 3.1 Defining a Type System

**Persephone.** Persephone [9, 10] brings sub-method reflection to Squeak. The standard model of Squeak and generally any Smalltalk system does not provide a model for sub-method structure. Methods are just represented as text and bytecode.

Persephone enhances Squeak with a model for methods based on an abstract syntax tree (AST). Any entity of the AST can be annotated with metadata. Annotations can be visible in the source code, they can be added in various places in a method, namely to instance and class variables, method arguments, return statements, local variables and block variables. For example, the expression anExpression **<:aSelector: anArgument:>** attaches an annotation with one argument to the expression anExpression.

The argument of an annotation can be any Smalltalk expression. In the simplest case, when the argument is just a literal object, the value of the annotation is set to this literal object. When the argument is an expression, the value of the annotation is the AST of this expression. We can specify when this AST is evaluated, either at compile time or later at run-time. In addition we provide a reflective interface to query and set annotations at runtime.

**Annotations, Types and Their Representation.** TypePlug gathers type information from annotations made in Squeak source using the Persephone framework. Such an annotation places the requirement that the annotated subexpression should have the given type.

In addition to type annotations TypePlug supports *cast annotations*. Casts are unchecked — they always succeed, regardless of whether they can statically be proven type safe. Cast annotations are formed by preceding a type annotation with cast and capitalizing its first character, *e.g.,* **<:castNonNil :>** is a cast to type nonNil.

Since TypePlug supports an arbitrary number of coexisting pluggable type systems, any expression can have several annotations, one per type system. In the source code, annotations are simply added one after the other. For this to work we need a way to distinguish annotations for different type systems. Every type system defines a unique *system key* that is used to identify its annotations. The system key is a symbol returned from the class method systemKey.

For the non-nil type system, the key is nonNil, thus an annotation for the unique type in the system is **<:nonNil :>**. This is a special case — usually a type system will have more than one single type, and keys will usually have a colon at the end to signal that. For example, the confinement type system discussed in Section 4.2 has the key confine:. In an annotation for that system, the actual type appears as the *annotation value* after the key, *e.g.,* **<:confine: toClass :>**.

The type system must define its valid annotation values and thus its valid types. It does so by defining a method annotationValueToType:inContext: which returns instances of its types (types in TypePlug are instances of subclasses of the class TPType). The annotationValueToType:inContext: method takes as the first argument an abstract syntax tree (utilizing the Refactoring Browser AST nodes) of such an expression. This is a great help for type systems which might have complex annotation values, so they do not need to do their own parsing. The second argument is an instance of TPContext which describes the context of the annotation, *i.e.,* in which method the annotation at hand is located. In a type system, the interpretation of an annotation might depend on where it is found.

**The Top Type.** One type is predefined and shared by all type systems: the top type. It is assumed to be a supertype of every other type in a type system.

Within the TypePlug framework, the top type can appear in many places where a type is expected, and if it does it means one of two things:

1. This can be any type.
2. Nothing is known about this type.

An example of the first meaning is the default argument type for methods: by default, if a method argument is not explicitly annotated with a type, it is assumed to have the top type. The second meaning can be observed in the typing methods discussed in the next section: all of these by default return the top type to state that, *e.g.,* the type of literals (defined by typeForLiteral:) is unknown — unless, of course, defined differently by the type system.

Generally speaking, every typed thing has the top type if it is unannotated or has an unknown type. This may seem like an insignificant implementation detail but the top type is an important device to usefully type check only partially annotated source. With its property of being a supertype of every other type it guarantees that, *e.g.,* unannotated source code type-checks safely.

**Typing Elements of Smalltalk Syntax.** Type systems can define the types of a range of basic constructs of Smalltalk syntax: types for literals, global variables, primitives, arrays, blocks and pseudo-variables such as self and super. This is achieved by implementing any of the typeFor* family of methods defined on type systems, as listed in Table 1. By default all of these methods return the top type.

In the non-nil type system, the implementation of the typing methods is very simple. Here are two of the more interesting ones:

```
typeForLiteral: aValue
    ↑ aValue ifNotNil: [self singleType] ifNil: [self topType]

typeForSelfInClass: aClass
    ↑ (UndefinedObject includesBehavior: aClass)
            ifTrue: [self topType]
            ifFalse: [self singleType]
```

Obviously, the type for a literal is always nonNil except when the value is nil. To assess the type of self we need to be slightly cautious since within nil's class, UndefinedObject, we cannot say that self is nonNil. But the same is true for all superclasses of UndefinedObject since their methods could be called from the nil instance (the includesBehavior: method returns true if the argument is the receiver or a superclass of the receiver).

**Subtyping and Unification.** The two most important operations a type system must define in our model are a subtyping relation and a type unification operation. Each of these is heavily used at the core of the type-checking algorithm as described in Section 3.2. Responsibilities for subtyping and unification are assigned to the type system rather than to types, though the implementation of a particular type system may delegate these responsibilities to the types themselves.

The subtyping relation is defined by implementing the is:subtypeOf: method which takes two types as arguments. It should return true if the first type is a subtype of the second in the context of this type system, false otherwise. While type systems are free to define whatever subtyping relation they please, it should usually be reflexive and transitive to be of practical use. The second argument may be the top type, representing an unknown type. The top type is considered to be a supertype of every other type of any type system by definition, so is:subtypeOf: is never called with the first argument being the top type.

The unification operation creates a type that represents the union of two types. Again, type systems are completely free to define this in any way that is appropriate. Generally, the union of two types should be the most specific common supertype, but a type system can also work with an explicit union type that it defines. In code, the unification operation is defined by implementing the unifyType:with: method which should return the result of unifying the two types passed as arguments.

Since the non-nil type system only knows a single type, subtyping and unification are absolutely trivial. The nonNil type is a subtype of itself and the nonNil type unified with itself gives the nonNil type again:

```
is: aType subtypeOf: anotherType
    ↑ (self isNilType: aType) and: [self isNilType: anotherType]

unifyType: aType with: anotherType
    ↑ ((self isNilType: aType) and: [self isNilType: anotherType])
        ifTrue: [self singleType]
        ifFalse: [self topType]
```

**The Built-in Static Type System.** Although Smalltalk is dynamically typed, its source code nevertheless contains some inherent static type information. For example, the class of an object is statically known if it appears as a literal in the source code. Static type information such as this can be very valuable to the type-checking algorithm and to any given pluggable type system. It is therefore important to capture this information and make it available in a convenient form. TypePlug achieves this with a built-in static type system which is itself implemented as an ordinary pluggable type system.

Every expression is assigned one of the following static types:

- The self type, the type of the pseudo-variable self.
- The super type, the type for the pseudo-variable super.
- Class types, the types for objects whose class is known.
- Object types, the types for objects whose exact value is known, *e.g.,* literals and globals.
- Block types, the type for literal blocks.

– The top type, meaning that nothing is known about the static type of that expression.

Object types of a given class are all subtypes of the corresponding class type, *e.g.,* the object type for the integer literal 42 is a subtype of the class type for SmallInteger.

The block type is a class type, since the class of a literal block is obviously known. The same goes for the super type. The self type, however, is not a class type — in class hierarchies the pseudo-variable self can refer to any of the subclasses of the class it appears in.

In contrast to user-defined pluggable type systems, the static type system is always present underneath every other type system. The built-in static types are available to implementors of pluggable type systems, and can be exploited when defining subtyping and unification for those systems. As a general principle, whenever an implementation gets hold of a type of some expression (*e.g.,* the two types passed as arguments into the unification operation) it can also access the static type of that same expression.

## 3.2 Type Checking

The heart of TypePlug is its approach to type-checking. The type-checking algorithm has been designed to make it easy to plug in a new type system. Furthermore, since type annotations in pluggable type systems are optional, the algorithm must deliver useful results in the face of an only partly annotated codebase. As a consequence of these considerations we established two guiding principles for the approach to type checking:

1. Only code that contains type annotations or uses annotated methods will be type-checked.
2. Where code to be type-checked refers to unannotated code, static types and type inference will be used to infer as much type information as is practical.

The combination of these two principles makes it possible to deal well with a mix of typed and untyped code.

**Ensuring Type Safety.** Type checking in TypePlug is applied per method and per type system. The type checker takes a type system and a method as input, statically checks the method for type safety and returns detailed results about a type error if there is one.

Source code is analyzed by traversing its abstract syntax tree (AST) representation. A type is assigned to every expression node in the AST. The type of an expression is determined by the typing methods of the type system and — where possible — by type inference, taking into account the type and cast annotations in the source.

While the type checker traverses the AST, at certain points type safety is ensured by doing subtyping checks. The three points are assignments, return statements and message sends. Type safety checks for assignments and return statements are trivial, but message sends deserve an extended discussion.

**Checking Message Sends.** Ensuring the type safety of message sends is the most difficult problem in type checking Smalltalk. In general, the class of a receiver of a message is not statically known, so there is usually no way to statically determine which method will be invoked at runtime or if a matching method even exists. Clearly, any static type-checking algorithm relying on partial type annotations must be pragmatic here and make some compromises.

One possible solution to improve the situation is to extensively use type inference to determine the class of message receivers. We do utilize some simple forms of type inference but in the context of a specific pluggable type system, not to determine the class of expressions.

We use a simple scheme that nevertheless yields useful results. Our approach tries to look up the *set of methods* that could be invoked for a specific message send and involve the whole set during type checking. The static type system is used to make this set as small as possible. Thus our approach is similar to standard type inference techniques [11], but with a far simpler reduction strategy.

Three cases are distinguished based on the static type of the receiver:

1. If the receiver class is statically known, *i.e.,* the receiver has a static class type, the method invoked at runtime can be looked up precisely and the set consists of that one method. Note that object types, block types and super types all fall under this case since they are class types.
2. If the receiver has a self type we need to consider the class the method being type-checked belongs to. The set consists of all implementors of the message in this class and its subclasses. It is necessary to include subclasses because the self pseudo-variable can refer to an instance of a subclass if the method is called from a subclass.
3. If the receiver does not have a static type, *i.e.,* its static type is the top type, we have to resort to a very broad strategy: the set consists of all implementors of a message, *i.e.,* all methods of the message's name implemented in the whole system. This case is the most common, unfortunately.
   A pluggable type system might carry information that can be used to further reduce the set of methods in this case. That is why type systems get the chance to implement their own strategy for this third case (by overriding the methodsForMessage method).

Once the set of methods to be considered has been fixed, the actual type check of the message send consists of asserting that the types of the arguments are subtypes of the respective types of the methods parameters. If a method parameter does not have a type it is assumed to have the top type which means that untyped parameters effectively are not type-checked. If the set of methods is empty a type error is raised.

This approach as a whole has the desirable property of catching most type errors. But it has the undesirable property of possible raising too many type errors. When unrelated classes have methods with the same name but with different parameter types, the "all implementors" strategy might label a message send as a type error even though at runtime the error would never occur. This drawback becomes less problematic when one considers what actually happens when type checking arguments in a message sends. If a method does not have any type annotations argument type checking is always successful. Only methods that declare types on their arguments can provoke type errors. Since we do not expect to operate in a fully annotated code base these questionable type errors should not occur often.

One problem remains: This approach cannot guarantee that a receiver actually responds to a message at runtime, *i.e.,* that a receiver actually implements a method of that name. We only guarantee type safety for the case that a receiver actually responds to a message.

## 3.3 Type Inference

We treat type inference as a crucial tool to enhance the user experience of TypePlug. It spares a programmer from exhaustively annotating source code with types.

Our type checker has two specific limited forms of type inference built in, for local variable types and for method return types. Both of those are optional, insofar as the type checker just *tries* to infer types but does not depend on a conclusive result. What is more, the pluggable type systems themselves do not have to care about type inference — it emerges from our model of type systems and does not restrict expressiveness within that model.

Type inference for local variables is not particularly novel or interesting, so we will only discuss our approach to return type inference here.

**Return Type Inference.** When the returned result of a message send is used, the type checker needs to make a useful assumption about the return type of the message. Again, Smalltalk's dynamic properties make it impossible to know which method will actually be invoked at runtime. So we use the same rules as with argument type checking described in Section 3.2 to get a set of methods to be considered. The return type of the message is then the union of all return types of those methods.

If a method was not annotated with a return type, its return type is by default considered to be the top type. But to improve the quality of type checking and comply with the requirement to infer as many types as possible, the type checker will try to infer the return type of an unannotated method. Inference basically works the same way as type checking (implementation-wise it is in fact identical): we walk the AST of the method and keep track of the types of AST nodes. The inferred return type simply becomes the union of the types of all return statements in the method. If the method's last statement is not a return statement, the type defined by the type system for `self` is also part of this union since by default Smalltalk methods return `self`.

When inferring the return type of a method the result can obviously be improved by inferring the return types of message sends within that method as well. In fact the inferencer could drill even deeper and walk the whole graph of method calls to make the result as precise as possible. This is not realistic for performance reasons, so we limit the inferencer to an *inference depth*. It defines how deep the inferencer looks into the call graph.

Increasing the inference depth is very costly in terms of performance, since the total number of methods considered grows very fast. While using TypePlug with real world code we discovered that 3 is about the highest tolerable inference depth. We use various caching strategies to improve performance, but in general inferred return types can't be kept in a cache very long because the conditions that lead to a cache invalidation are very costly to detect, especially with a high inference depth (*i.e.,* when the the return type annotation of a method changes, the inferred return types of all its callers might change).

**The Impact of Return Type Inference.** To assess the impact of return type inference, we carried out the following experiment. In a stock Squeak image without any explicit type annotations we tried to infer the return type of all methods of classes in the category Files−Directories. This category contains Squeak's abstraction of file system directories, with a total of 8 classes with 210 methods. This serves as an example of a typical small package of user code that one would want to type check.

In the second stage of the experiment, we added as many return type annotations to methods of the Object class as possible. Object is the base class for almost all other classes in Smalltalk; it implements some heavily used methods such as = (equality), class and copy, so annotating those should improve return type inference (because overridden methods inherit types from superclasses). With this small set of type annotations we again tried to infer all return types in the Files−Directories class category We used an inference depth of 1, meaning that the type inferencer is allowed to dive one level deep into messages used in an examined method, but not further. We ran this experiment for both the non-nil and the class-based type system (discussed in Section 4.1). The results are summarized in Table 2.

| | *Non-nil* | *Class-based* |
|---|---|---|
| Total methods | 210 | 210 |
| Inferred, without annotations | 67 | 66 |
| in percent | 31.9% | 31.4% |
| Inferred, with annotations | 73 | 77 |
| in percent | 34.8% | 36.7% |

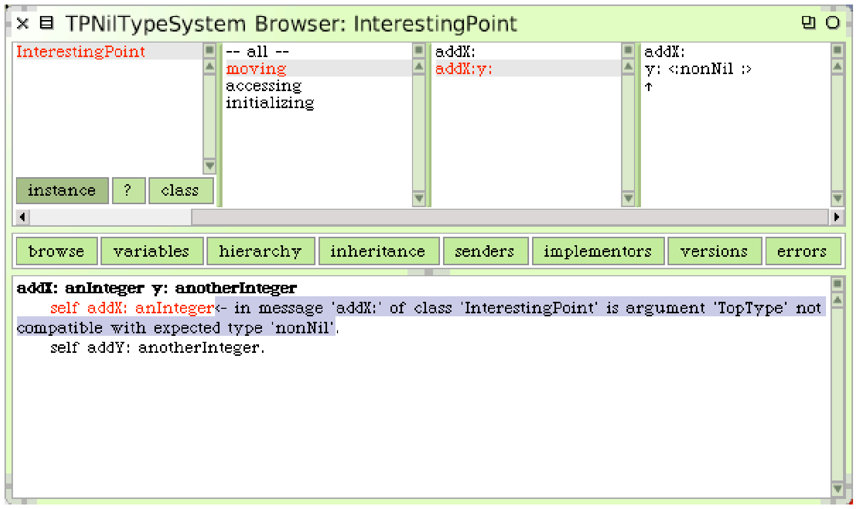
**Table 2.** Return type inference for class category Files−Directories in a stock Squeak image, version 3.9 final

In both type systems, for about 31% of all methods the return type could be inferred even without any type annotations present in the image. This shows that return type inference does add significant value to the system as a whole, since in this package for more than 30% of methods explicit return type annotations are not even needed. 39 methods (19% of all methods) do not contain a return statement, so their return type is trivially inferred to be the type of `self`.

With type annotations for `Object`, inferred return types are about 3% more for the non-nil type system and 5% more for the class-based type system. This improvement is quite impressive considering that only a few annotations were added to `Object`. These results suggest that the cumulative effect of additional annotations to other commonly used classes such as numbers, strings and collections should be good.

## 3.4 Programming Environment Integration

In Smalltalk IDEs, browsers are traditionally used to navigate and modify the source code in an image. Part of TypePlug is a type browser for Squeak, a browser that enhances a standard browser with some type-specific behavior. The type browser has three main features: it integrates type checking, it provides an alternative way of introducing type annotations without changing source code (external type annotations) and it exports types to a distributable form.



```
addX: anInteger y: anotherInteger
    self addX: anInteger<- in message 'addX:' of class 'InterestingPoint' is argument 'TopType' not compatible with expected type 'nonNil'.
    self addY: anotherInteger.
```

**Fig. 1.** The Type Browser

Figure 1 shows the type browser in action. Compared to a standard browser, the type browser has an extra panel to the right of the list of methods. This panel shows the method type of the currently selected method, *i.e.*, its argument and an up arrow (↑) to represent the return statement, including types if there are type annotations. Selecting one of these types brings up the type in the bottom (source code) area of the browser where it can be edited.

**External Type Annotations.** For a type system to support useful type checking it is often necessary to annotate at least a minimal set of methods of the standard Smalltalk classes with types. For example, you may want to annotate the return type of the hash method of Object with a nonNil type. But normally it is not a good idea to add such an annotation by modifying the source of a method. Redefining methods in standard classes such as Object works well locally in an image, but problems arise when the code or the types should be packaged up for distribution, reside in a version control system, or undergo any other form of migration between images.

To address these issues, the type browser offers a way to add *external type annotations* which are separate from the source code of a method. External types can simply be added and modified through the extra types panel of the browser, which registers types with an external cache but does not change the source code of a method. Additionally, the type browser offers the option to export the types of all methods of a class, including both external and in-source types. Exported types of several classes can easily be bundled, distributed and imported into any image.

The external type annotation support of the type browser is an important tool for the development and use of pluggable type systems. It allows developers of a type system to distribute a set of types for a Squeak base image, but it also allows other parties to create and distribute type packages for arbitrary type systems.

## 4 Case Studies

In addition to the non-nil type system presented in Section 2, we used TypePlug to implement a class-based type system and a confinement type system. We present in some detail the first of these, and briefly summarize our experience with the second.

### 4.1 A Class-Based Type System

We use TypePlug to implement an expressive class-based type system, sporting many features of modern statically typed languages. It supports generic types, polymorphic methods, type unions and typed blocks. The syntax for type annotations in this system is summarized in Table 3. This is expressive enough to meaningfully annotate most Smalltalk code with class-based types.

The type of instances of a class is simply the name of that class, *e.g.,* booleans and integers have the types Boolean and Integer respectively. Classes themselves (as opposed to their instances) have a type, too. Such a class type is formed by appending class to the name of the class, *e.g.,* Boolean class for the type of the class Boolean.

We will not discuss all other kinds of types in detail. Instead we will look closely at the important subtyping relation and the polymorphism features of the type system.

| | | |
|---|---|---|
| *Class* ::= | | |
| | *ClassName* | |
| | Self | |
| | Instance | |
| *Type* ::= | | |
| | *Class* | (Simple type) |
| | *Class* class | (Class-side type) |
| | *Type* \| *Type* | (Union type) |
| | Block args: {*Type**} return: *Type* | (Block type) |
| | *Class* (*ParamName*: *Type*)+ | (Generic type) |
| | Param *ParamName* | (Type parameter) |
| | MethodParam *ParamName* | (Type parameter) |

**Table 3.** Grammar for class-based types

**Subtyping.** The usefulness of a class-based type system such as this one depends largely on the definition of subtyping. Smalltalk code in general is not well suited to be typed using classes. For example, classes might override methods while breaking assumptions about types made in the superclass, and other classes might "delete" methods defined in superclasses. All of these things function well considering Smalltalk's dynamic properties, but are not readily compatible with the notion of subtyping.

On the basis of these considerations we define a subtyping relation that is based on the *type interface* of a class. We define this to be the set of class method types that have some type annotation (a method type is a method's name with its argument and return types). Table 4 demonstrates this with an example. The type interface of the type Fruit is the method type of mixWith: (with argument Fruit and return type Array E: Fruit) plus of course any method types inherited from superclasses. isVegetable is not part of the type interface since it does not have any type annotations.

The definition of our subtyping relation consists of two clauses. The first clause forms the basis for the relation with a standard contravariant definition for structural subtyping:

1. Type A is a subtype of type B if the type interface of B is a subset of the type interface of A and for all method types mB of the interface B and the corresponding method type mA of the interface A it is true that: the argument types of mB are subtypes of the respective argument types of mA and the return type of mA is a subtype of the return type of mB.

Taken in isolation, this is a standard subtyping relation from the literature (*e.g.,* [12, page 182]) based on structural interfaces, using the usual contravariant subtyping rule for arguments.

According to this first clause, in the fruit example from Table 4 it is clear that both types Apple and Orange are subtypes of the type Fruit; their type interface contains the (inherited) mixWith: method with types. Also, more surprisingly, the

Apple and Orange types are both subtypes of each other — their type interfaces are identical since they both implement an annotated color method.

There is one obvious problem with subtyping based only on the above first clause: if this were the complete subtyping rule then every simple type would be a subtype of every other simple type in a codebase without any type annotations. This would not be useful, which motivates the second clause of our definition:

2. We consider the type B of the class B which is a subclass of A. If the type interface of B is identical with the type interface of A (*i.e.,* the methods of class B do not add or change any type annotations compared to A), then the type B has no subtypes except itself.

This means that for types based on classes that do not define any typed methods, subtyping is based on type identity only.

This rather unusual subtyping definition has some interesting implications. First, thanks to the second clause, it works well with untyped classes. Second, it puts a lot of responsibility into the hands of the programmer who makes type annotations. For example, a newly created class is not a subtype of its superclass. The programmer must add at least one method with a type annotation to the new class (while not violating subtyping rules) to make the new class a subtype of its superclass. In general, to get subtyping relations in a class hierarchy, type annotations must be added to all the classes in the hierarchy that should participate in the subtyping relation.

**Genericity.** *Generic types* are types parameterized by at least one named type parameter. A generic type parameter consists of a name followed by a colon and the type it should be bound to. A typical example of generic types are collections, *e.g.,* the type of an OrderedCollection. This type has a single parameter E referring

```
class Fruit subclassing: Object
    mixWith: aFruit <:type: Fruit :>
        ↑ (Array with: self with: aFruit) <:type: Array E: Fruit :>
    isVegetable
        ↑ false


class Apple subclassing: Fruit
    color
        ↑ (Color red) <:type: Color :>


class Orange subclassing: Fruit
    color
        ↑ (Color orange) <:type: Color :>
```

**Table 4.** Fruit bowl code

to the type of the elements of the collection. An OrderedCollection of integers then has the type OrderedCollection E: Integer.

In the context of a class with a generic type, type parameters can be used just like any other type: the type Param E refers to the type that the parameter E takes in a generic type. By using a type parameter in any method a type implicitly becomes generic, *i.e.,* there is no explicit declaration of type parameters. For example, the type of an OrderedCollection is generic because the parameter Param E appears in the methods of the class OrderedCollection, *e.g.,* as the argument type of the add: method.

Apart from type parameters with class scope as used in generic types, our class-based type system also supports type parameters with method scope. Methods making use of such type parameters are *polymorphic methods.* The quintessential and most simple example of a polymorphic method is the id: method which takes an argument and does nothing but return that argument again. To type id: we can give its argument the type MethodParam A, a type parameter named A with method scope. As the return type of id: we use the same type parameter again, MethodParam A. This expresses that we expect the return type of id: to be the type of its argument.

During type checking of code invoking a polymorphic message, the type checker infers the value of all type parameters from the type of the concrete arguments passed (in the type system implementation, this is achieved via the hook method transformMethodType:...).

An interesting example of a polymorphic method is ifTrue:ifFalse: in the class Boolean:

```
ifTrue: aBlock <:type: Block args: {} return: MethodParam R :>
ifFalse: anotherBlock <:type: Block args: {} return: MethodParam R :>
    ...
    ↑ (...) <:type: MethodParam R :>
```

This example demonstrates that polymorphic methods are needed to type some crucial innards of Smalltalk and also illustrates block types.

## 4.2 A Confinement Type System

As an example very different from the non-nil and class-based type systems we defined a confinement type system. This type system implements a very specific kind of confinement: confined instance variables. References to mutable objects such as a collection are often considered to be private to a class when they are stored in an instance variable; these references should not be shared with other classes. The confinement type system can guarantee that such confined instance variables do not leak from their class and thus are not modified outside their class. In Smalltalk, all instance variables are always private, encapsulated by the instance. In a way, this type system expresses an extended form of privateness for instance variables.

The confinement type system demonstrates how TypePlug enables type systems with fairly complex semantics to be implemented succinctly. The implementation consist of only 4 classes with a total of about 110 lines of code.

This type system exploits TypePlug's type inference in an unusual way: whether a method possibly returns a confined reference is determined only by return type inference, not by explicit return type annotation. Confined type annotations are added only to instance variables. During type checking, type inference automatically takes care of analyzing the flow of these confined references through methods.

A further interesting aspect concerns the *unconfined types* supported by our confinement system. Unconfined annotations are a pragmatic tool for a programmer to signal to the type checker that he knows the annotated expression does not evaluate to a confined value. The confinement type system is conservative in that it considers the result of any message sent to a confined reference to be confined as well. But some methods (such as `copy` to copy an object) will always return new and thus unconfined references. The return types of such methods can be annotated with an unconfined type. Unconfined types effectively "override" the effect of confined types—this use of overriding could be a pattern useful in other type systems.

For further details, consult the full report on TypePlug [13].

## 5 Discussion and Related Work

### 5.1 Reflection and Exceptions

In a dynamically-typed language like Smalltalk there are programming idioms that our type checker simply is not capable of handling. We discuss two problems of our approach: typing code that uses the reflective features of Smalltalk and exception handling.

*Reflection.* Static type checking is about deducing static properties of a system before it is run, whereas reflection (not just introspection) means changing the structure and behavior of a system at runtime. Typing reflective systems thus poses many problems [14].

The `perform:` method is used to send a message to an object with the message name determined at runtime. It takes a message name as its argument and sends that message to the receiver. The return type depends on the value of the argument which in general is not known statically, so there is no way to reasonably type check a `perform:` message, its arguments and its return value. However, in many cases we can add an explicit type cast to a perform expression, thus providing the type checker with the information needed to check a perform expression.

The `doesNotUnderstand:` idiom poses a more severe problem for our type checking algorithm. The `doesNotUnderstand:` method allows classes to intercept runtime errors occurring when a message was sent that is not understood by the receiver.

This means any object possibly accepts a message sent to it even though its class does not explicitly implement a method of the same name. This is a problem because type checking by default assumes that it can find all implementors of a given message name. To avoid false positives when type-checking, the only recommendation we can make is to avoid this idiom in areas of code that should be type safe.

*Exceptions.* TypePlug currently does not model exceptions. In general, every message send in Smalltalk could result in an exception being signaled. The concept of exceptions is fortunately orthogonal to our notion of type safety. For our purposes, if an exception is signaled it simply means that the execution of a method stops at that point; it does not in any way affect the types of variables or other expressions. TypePlug simply does not deal with exceptions at the moment, not diminishing the usefulness of type checking.

### 5.2 Related Work

In the following, we give an overview of related work. After a short presentation on relevant general work on type systems and type inference in the context of Smalltalk, we focus on related work on pluggable type systems.

**Type Systems and Type Inference for Smalltalk.** Smalltalk as a practical dynamically typed system has seen many proposals for adding static type systems [15–18]. The most recent effort at conceiving and implementing a practical type system for Smalltalk was *Strongtalk* [19]. Some of these type systems support optional typing, but they provide just a single type system, not addressing pluggability.

*Type Inference* was originally researched in the context of functional languages [20]. Type inference then was applied to Smalltalk [21]. Palsberg and Schwartzbach presented the first algorithm that can type check completely untyped Smalltalk code [11]. The Cartesian product algorithm (CPA) by Agesen [22] provides a substantial improvement in both precision and efficiency. Even with this advanced algorithm, type inference does not scale to larger programs. Efficiency and scalability thus is a focus of current research. Demand-driven type inference (DDP) [23] provides scalability by analyzing type information on demand and selectively reducing precision. RoelTyper [24] uses heuristics for providing type information of instance variables. RoelTyper, like DDP, provides high performance at the cost of reduced precision.

**Pluggable Type Systems.** Pluggable type systems were originally proposed by Gilad Bracha [1]. A number of implementations of pluggable type systems have been published.

**JavaCOP** [7] is a program constraint system for implementing practical pluggable type systems for Java. The authors present a framework that allows additional types to be added to Java source code, based on Java's annotation facilities. They then define a declarative, rule-based language that can express rules as constraints on AST nodes, making use of the information from annotations as well as from Java's static type system. These rules form the semantics of pluggable type systems and can be enforced by a type checker that hooks into the compile process.

Annotations in JavaCOP are similar to TypePlug's, but are restricted to class and variable declarations by the Java language, so something like our casts on arbitrary expressions cannot be supported. The way type systems are implemented with rules in a domain-specific language is very different from TypePlug's type system model. The rule language enables very fine grained control over type checking. The authors prove the validity and versatility of this approach by implementing an impressive number of pluggable type systems, ranging from confined types to reference immutability and checks of the kind usually done by coding style analyzers. On the other hand, our approach with a rather fixed type checking algorithm and customization is arguably a bit simpler and results in simpler implementations at least for some applications. For example, our non-nil type system is extremely simple with about 30 (mostly very trivial) lines of code while JavaCOP's equivalent presumably needs several non-trivial rules. JavaCOP does not provide any special support for working with untyped legacy code.

**Fleece** [6] also explores the notion of pluggable type systems, but explicitly for dynamically typed languages. As an example of a dynamically typed language the report introduces and defines $\mathcal{R}_{sub}$, a subset of the Ruby programming language, which is then used throughout the report. It develops the notion of annotators that automatically add and propagate annotations (types) on nodes of the AST of a program. A special case of an annotator is the programmer who can manually add annotations to a program. Fleece's handling of annotations correspond in many aspects to TypePlug's. Automatic annotators play the role of the inference part of TypePlug's type checking.

Compared to TypePlug, Fleece is both more general and more restricted. It is more general, because it is independent of the language $\mathcal{R}_{sub}$; it copes with any other language that can be expressed in the grammar form that it expects. However, it is more restricted because it has not been shown to handle a real dynamically typed language. $\mathcal{R}_{sub}$ is a very restricted form of Ruby with, *e.g.,* no inheritance or support of the standard library. The feature that distinguishes TypePlug is the ability to pragmatically work with an actual full implementation of Smalltalk, thus supporting type-checking of legacy code.

**JastAdd**, an extensible Java Compiler, has been extended with a pluggable type system by Ekman and Hedin [5]. The paper presents only one type system, a non-nil type system similar to that of TypePlug. Of all three discussed other frameworks, it is the only one which provides support for reducing the number of annotations needed when dealing with untyped legacy code. It supports inference of non-null types, but it does not allow for type annotations without changing the source code.

## 6 Conclusion and Future Work

TypePlug is intended to offer a very pragmatic framework for implementing pluggable type systems. New type systems are defined by specializing the TP-TypeSystem class and overriding the methods for subtyping and type unification. TypePlug makes it relatively easy to obtain the benefits of a pluggable type system even if the underlying legacy code base remains largely free of type annotations. This is achieved by (i) only type-checking annotated code, (ii) using static type information and limited type inference to reason about unannotated code, and (iii) externally annotating third party code. Explicit type casts can also be used to avoid annotating code that is known to be safe.

We have made experience with three very different kinds of type systems, including a classical class-based type system, a non-nil type systems and a type system for confined types.

There are numerous further directions to explore. Presently TypePlug does not take exceptions into account. Type inferencing is quite slow, and inferencing beyond a depth of 3 levels is unpractical. We plan to integrate modern type inference algorithms and a heuristical type analysis system in the spirit of Roel-Typer.

An interesting field for future research is to explore type checking in the presence of reflection, especially how to type check a system that can be changed reflectively at runtime. It would also be interesting to explore ways of adding optional run-time type checks by making further use of the Persephone framework for sub-method reflection, thus complementing TypePlug's static type checking. Pluggable type systems might be able to benefit from information provided by any other pluggable type system, rather than just the built-in static type system. This sharing of type information is another direction to consider.

### Acknowledgements

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

## References

1. Bracha, G.: Pluggable type systems (October 2004) OOPSLA Workshop on Revival of Dynamic Languages.
2. Grothoff, C., Palsberg, J., Vitek, J.: Encapsulating objects with confined types. In: OOPSLA '01: Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications, New York, NY, USA, ACM Press (2001) 241–255
3. Smith, F., Walker, D., Morrisett, J.G.: Alias types. In: ESOP '00: Proceedings of the 9th European Symposium on Programming Languages and Systems, London, UK, Springer-Verlag (2000) 366–381

4. Zhao, T., Noble, J., Vitek, J.: Scoped types for real-time Java. In: RTSS '04: Proceedings of the 25th IEEE International Real-Time Systems Symposium (RTSS'04), Washington, DC, USA, IEEE Computer Society (2004) 241–251
5. Ekman, T., Hedin, G.: Pluggable non-null types for Java. In: Proceedings of TOOLS Europe 2007. (2007) To appear.
6. Allwood, T.: Fleece, pluggable type checking for dynamic programming languages. Master's thesis, Imperial College of Science, Technology and Medicine, University of London (June 2006)
7. Andreae, C., Noble, J., Markstrum, S., Millstein, T.: A framework for implementing pluggable type systems. In: OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, New York, NY, USA, ACM Press (2006) 57–74
8. Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., Black, A.: Traits: A mechanism for fine-grained reuse. ACM Transactions on Programming Languages and Systems (TOPLAS) **28**(2) (March 2006) 331–388
9. Denker, M., Ducasse, S., Lienhard, A., Marschall, P.: Sub-method reflection. Journal of Object Technology **6**(9) (September 2007) To appear.
10. Marschall, P.: Persephone: Taking Smalltalk reflection to the sub-method level. Master's thesis, University of Bern (December 2006)
11. Palsberg, J., Schwartzbach, M.I.: Object-oriented type inference. In: Proceedings OOPSLA '91, ACM SIGPLAN Notices. Volume 26. (November 1991) 146–161
12. Pierce, B.: Types and Programming Languages. The MIT Press (2002)
13. Haldimann, N.: TypePlug — pluggable type systems for Smalltalk. Master's thesis, University of Bern (April 2007)
14. Alanko, L.E.: Types and Reflection. Ph.D. thesis, University of Helsinki (November 2004)
15. Borning, A.H., Ingalls, D.H.: A type declaration and inference system for Smalltalk. In: Proceedings POPL '82, Albuquerque, NM (1982)
16. Johnson, R.E.: Type-checking Smalltalk. In: Proceedings OOPSLA '86, ACM SIGPLAN Notices. Volume 21. (November 1986) 315–321
17. Graver, J.: Type-Checking and Type-Inference for Object-Oriented Programming Languages. Ph.D. thesis, University of Illinois at Urbana-Champaign (August 1989)
18. Palsberg, J., Schwartzbach, M.I.: Object-Oriented Type Systems. Wiley (1993)
19. Bracha, G., Griswold, D.: Strongtalk: Typechecking Smalltalk in a production environment. In: Proceedings OOPSLA '93, ACM SIGPLAN Notices. Volume 28. (October 1993) 215–230
20. Milner, R.: A theory of type polymorphism in programming. Journal of Computer and System Sciences **17** (1978) 348–375
21. Suzuki, N.: Inferring types in smalltalk. In: POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM Press (1981) 187–199
22. Agesen, O.: Concrete Type Inference: Delivering Object-Oriented Applications. Ph.D. thesis, Stanford University (December 1996)
23. Spoon, S.A., Shivers, O.: Demand-driven type inference with subgoal pruning: Trading precision for scalability. In: Proceedings of ECOOP'04. (2004) 51–74
24. Wuyts, R.: RoelTyper, a fast type reconstructor for Smalltalk (2005) http://decomp.ulb.ac.be/roelwuyts/smalltalk/roeltyper/.

# Part IV

# Concurrency

# Transactional Memory for Smalltalk

Lukas Renggli and Oscar Nierstrasz

Software Composition Group, University of Bern
`{renggli,oscar}@iam.unibe.ch`
scg.unibe.ch

**Abstract.** Concurrency control in Smalltalk is based on locks and is therefore notoriously difficult to use. Even though some implementations provide high-level constructs, these add complexity and potentially hard-to-detect bugs to the application. Transactional memory is an attractive mechanism that does not have the drawbacks of locks, however the underlying implementation is often difficult to integrate into an existing language. In this paper we show how we have introduced transactional semantics in Smalltalk by using the reflective facilities of the language. Our approach is based on method annotations, incremental parse tree transformations and an optimistic commit protocol. We report on a practical case study, benchmarks and further and on-going work.

## 1 The need for transactions

Smalltalk has inherently weak support for concurrent programming and synchronization. Smalltalk-80 [1] only proposes semaphores as a means for synchronizing processes and achieving mutual exclusion. The ANSI standard of Smalltalk [2] does not refer to synchronization at all.

Only a few current Smalltalk implementations provide more sophisticated synchronization support. VisualWorks Smalltalk provides a reentrant lock implementation that enables a process to enter the same critical section multiple times. Other processes get blocked until the owning process leaves the critical section. Unfortunately this kind of synchronization has its drawbacks and is notoriously difficult to use [3]:

**Deadlocks.** If there are cyclic dependencies between resources and processes, applications may deadlock. This problem can be avoided by acquiring resources in a fixed order, however in practice this is often difficult to achieve.

**Starvation.** A process that never leaves a critical section, due to a bug in the software or an unforeseen error, will continue to hold the lock forever. Other processes that would like to enter the critical section starve.

**Priority Inversion.** Usually schedulers guarantee that processes receive CPU time according to their priority. However, if a low priority thread is within a critical section when a high priority process would like to enter, the high priority thread must wait.

Squeak Smalltalk [4] includes an implementation of monitors [5], a common approach to synchronize the use of shared data among different processes. In contrast to mutual exclusion with the reentrant lock, with monitors, a process can wait inside its critical section for other resources while temporarily releasing the monitor. Although this avoids deadlock situations, the use of monitors is difficult and often requires additional code to specify guard conditions. Moreover, if the process is preempted while holding the monitor, everybody else is blocked. Beginners are often overwhelmed by the complexity of using monitors as Squeak does not offer method synchronization as found in Java.

Transactional memory [6, 7] provides a convenient way to access shared memory by concurrent processes, without the pitfalls of locks and the complexity of monitors. Transactional memory allows developers to declare that certain parts of the code should run atomically: this means the code is either executed as a whole or has no effect. Moreover transactions run in *isolation*, which means they do not affect and are not affected by changes going on in the system at the same time. Upon *commit* the changes of a transaction are applied atomically and become visible to other processes. Optimistic transactions do not lock anything, but rather conflicts are detected upon commit and either lead to an *abort* or *retry* of the transaction.

Most relational and object databases available in Smalltalk provide database transactions following the ACID properties: Atomicity, consistency, isolation, and durability. However, they all provide this functionality for persistent objects only, not as a general construct for concurrent programming. These implementations often rely on external implementations of transactional semantics. GemStone Smalltalk [8] is a commercially available object database, that directly runs Smalltalk code. As such, GemStone provides transactional semantics on the VM level. Guerraoui et. al. [9] developed GARF, a Smalltalk framework for distributed shared object environments. Their focus is not on local concurrency control, but on distributed object synchronization and message passing. They state that *"A transactional mechanism should however be integrated within group communication to support multi-server request atomicity."* [10]. Jean-Pier Briot proposed Actalk [11], an environment where Actors communicate concurrently with asynchronous message passing. The use of an Actor model is intrusive. It implies a shift of the programming paradigm, where there is no global state and therefor no concurrency issues.

In this paper we present an implementation of transactions in Squeak based on source-code transformation. In this way most code is free of concurrency annotations, and transactional code is automatically generated only in the contexts where it is actually needed.

The specific contributions of this paper are:

- The implementation of transactional semantics in Smalltalk, using the reflective capabilities of the language.
- A mechanism to specify context-dependent code using method annotations, for example to intercept the evaluation of primitive methods.

– Incremental, on-the-fly parse tree transformation for different execution contexts.
– Efficient, context-dependent code execution using the default execution mechanisms of the VM.

Section 2 presents some basic usage patterns of our implementation. Section 3 shows the implementation of transactions in Squeak without modifying the underlying VM. Section 4 validates our approach by running a collection of benchmarks and by applying the concept to a real world application. Section 5 concludes this article with some remarks about ongoing and future work.

## 2 Programming with transactions

Transactions offer an intuitively simple mechanism for synchronization concurrent actions. They do not require users to declare specific locks or guard conditions that have to be fulfilled. Moreover transactions can be used without prior knowledge of the specific objects that might be modified. Transactions are global, yet multiple transactions can run in parallel. The commit protocol checks for conflicts and makes the changes visible to other processes atomically.

```
tree := BTree new.
lock := Semaphore forMutualExclusion.

" writing "
lock critical: [ tree at: #a put: 1 ].

" reading "
lock critical: [ tree at: #a ].
```

```
tree := BTree new.


" writing "
[ tree at: #a put: 1 ] atomic.

" reading "
tree at: #a.
```

**Fig. 1.** Lock-based vs. Transactional accesses of a shared data structure.

On the left side of Figure 1 we see the traditional way of using a semaphore to ensure mutual exclusion on a tree data structure. The key problem is that *all* read and write accesses to the tree must be guarded using the same lock to guarantee safety. A thread-safe tree must be fully protected in all of its public methods. Furthermore, we cannot easily have a second, unprotected interface to the same tree for use in a single-threaded context.

On the right side of Figure 1 we present the code that is needed to safely access the collection using a transaction: the write access is put into a block that tells the Smalltalk environment to execute its body within a transaction. The read access can happen without further concurrency control. As long as all write accesses occur within the context of a transaction, read accesses are guaranteed to be safe. The optimistic commit protocol of the transaction guarantees safety by (i) ensuring that no write conflicts have occurred with respect to the previous saved state, and (ii) atomically updating the global object state.

To make the code using transactions as simple as possible we provide two methods for running code as part of a transaction. These methods are extensions to the standard Smalltalk library, and do not affect the language syntax or runtime.

– #atomic causes the receiving block closure to run as a new transaction. Upon termination of the block, any changes are committed atomically. If a conflict is detected, all modifications are cancelled and a commit conflict exception is raised.
– #atomicIfConflict: causes the receiving block to run as a new transaction. Instead of raising an exception if a conflict occurs, the block argument is evaluated. This enables developers to take a specific action, such as retrying the transaction or exploring the conflicting changes.

Further convenience methods can easily be built out of these two methods, for example a method to retry a transaction up to fixed number of times, or only to enter a transaction if a certain condition holds.

## 3 Inside transactions

We introduce transactions to Smalltalk without modifying the underlying Virtual Machine (VM). Our approach is based on earlier proposals in which source code is automatically and transparently transformed to access optimistic transactional object memory, rather than directly accessing objects [12, 13]. The key advantage of this approach is that most source code can be written without embedding any explicit concurrency control statements. Transactional code is automatically generated where it is needed. Furthermore, in contrast to the earlier approaches, we generate the needed transactional code dynamically where and when it is needed, and caching the generating code for future invocations.

In a nutshell, our approach works as follows:

– Every method in the system may be compiled to two versions: one to be executed in the normal execution context, and the other within a transactional context. Contrary to the other approaches we do this incrementally and on the fly using a compiler plugin.
– State access in transactional methods is automatically transformed to use an indirection through the transaction context.
– We use method annotations to control the automatic code transformation or to provide different code. Unlike earlier approaches, we take into account the use of primitives and exception of file-system access by providing alternative code to be used in a transactional context.
– When entering a transactional context we record the transaction (an object) in the current process (also an object).
– All objects touched during a transaction are atomically taken a snapshot of. Each snapshot consists of two copies of the original object: one that reflects the initial state and one that is altered during the transaction. For efficiency reasons immutable objects are excluded from snapshots.

– Upon commit we check for conflicts by atomically comparing the state of the object at the beginning of the transaction to the current version in memory. If no conflict is detected, the changes are committed. In case of a conflict the system is left in the state as it was before the transaction and an exception is raised that provides information for further reflection, namely all the changes, the conflicting changes and the transaction itself.

The key novelties of our approach lie in the use of annotations and reflection to lazily generate the transactional versions of methods, and the ability to provide alternative code to use in place of primitives during transactions.

In the following two sections we describe (1) the compilation to transactional code, and (2) the implementation of the transactional object model.

## 3.1 Compiling to transactional code

We transform methods by changing read and write accesses to make use of transactional object memory. Methods are transformed using a new version of the behavioral reflection framework Geppetto [14] which is based on sub-method reflection [15], allowing us to declaratively reify and transform an abstract syntax tree (AST) before compiling to byte-code.

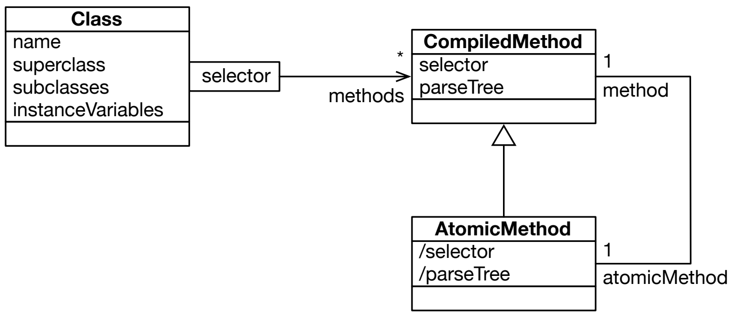


**Fig. 2.** Static Compilation Model

Whenever the source code of a method is accepted, as seen in Figure 2, our compiler plugin creates an additional compiled method that implements the behaviour to be used within the context of a transaction. The following basic transformations are performed:

1. Reading from instance variables and global variables is transformed to send the messages #atomicInstVarAt: or #atomicValue respectively. This allows us to implement these two messages to read the current value from within a transactional context instead of directly accessing the variables within the receiving object.

2. Writing to instance and global variables is transformed to send the messages #atomicInstVarAt:put: or #atomicValue: respectively. Again this allows us to intercept state access and handle it from within the current transaction.
3. Sending a message from inside a transactional method will actually send a different message name, namely we prepend #__atomic__ to the original selector name.

```
BTree≫at: aKey put: anObject
    | leaf |
    leaf := root leafForKey: aKey.

    leaf insertKey: aKey value: anObject.
    root := leaf root.
    ^ anObject
```

```
BTree≫__atomic__at: aKey put: anObject
    | leaf |
    leaf := (self atomicInstVarAt: 1)
        __atomic__leafForKey: aKey.
    leaf __atomic__insertKey: aKey value: anObject.
    self atomicInstVarAt: 1 put: leaf __atomic__root.
    ^ anObject
```

**Fig. 3.** Original vs. transformed code.

Having applied these three transformations to the code, the two compiled methods are stored in the method dictionary of their owning class. To tell the two methods apart, the atomic version of the method has #__atomic__ prepended to its name. These methods are hidden, and are only called from generate code within an atomic context. Transactional methods are filtered from the code editors, so they are not visible to the developer and development tools but only to the VM. On the left side of Figure 3 we present the code of a method as the developer implemented it, whereas on the right side we show the same method as it is compiled for the atomic context. All message sends are prepended with #__atomic__ and instance variable access is dispatched trough special methods using the slot index as argument.

A transaction is created by sending the message #atomic to a block containing normal (non-transactional) Smalltalk code. The code within such a block is statically transformed to evaluate within a transactional context. We have seen an example for such a call in the introductionary example in Figure 1. Methods that send #atomic are special, because the code outside this block is compiled normally, whereas we apply the transformations as described above to the inside of the block closure.

Squeak includes a few primitive methods that access and modify state. The most prominent of these are #at: and #at:put: to access the elements of variable-sized objects. Moreover there are also some central collection and stream methods that are implemented within the VM for efficiency. As primitive operations are written in C and statically compiled into the VM, we cannot use Geppetto to modify state-access. The only possibility to reify these methods is to replace them with non-primitive methods.

We make use of annotations to further control the way in which transactional code may be generated. Table 1 summarizes the effect of the following annotations:

| **Annotation** | **Source Code** | **Transform** |
|---|---|---|
| (no annotation, default) | method body | yes |
| <atomic:> | argument | yes |
| <atomicUseUntransformed> | method body | no |
| <atomicUseUntransformed:> | argument | no |

**Table 1.** Method annotations are used to control how the compiler transforms source code for the transactional context.

- <atomicUseUntransformed> avoids doing any code transformation. This means the normal and the transactional method will be the same, so no transformation is needed. In the current implementation this is mostly used for exception handing, as this code should continue to work through the boundaries of transactions.
- <atomic:> uses the method identified as its argument as the source for the code transformation. We use this for primitives that are implemented for efficiency reasons only. For example the method #replaceFrom:to:with:-startingAt: in the class Array calls the primitive 105 and is used to copy elements from one collection to another one. With the method annotation we tell the compiler that it should instead transform and install the method #atomicReplaceFrom:to:with:startingAt: that has the same behavior but is implemented in Smalltalk.

```
Array≫replaceFrom: start to: stop with: replacement startingAt: repStart
    "Primitive. This destructively replaces elements from start to stop
    in the receiver starting at index, repStart, in the collection,
    replacement. Optional."

    <primitive: 105>
    <atomic: #atomicReplaceFrom:to:with:startingAt:>
    super replaceFrom: start to: stop with: replacement startingAt: repStart

Array≫atomicReplaceFrom: start to: stop with: replacement startingAt: repStart
    | index repOff |
    repOff := repStart - start.
    index := start - 1.
    [ (index := index + 1) <= stop ]
        whileTrue: [ self at: index put: (replacement at: repOff + index) ]
```

- <atomicUseUntransformed:> uses the method identified as its argument as untransformed atomic code. We use this mainly in infrastructural code to dispatch primitive requests that access state to the working copy of the receiver. For example indexed slot access is handled through primitives in Squeak. The method #at: in the class Object calls the primitive 60 to fetch the contents of an indexed element. The method annotation tells the compiler to use #atomicAt: instead. This method delegates the request to the current working copy of the object.

```
Object≫at: index
    "Primitive. Assumes receiver has indexed slots. Answer the value of an
    indexable element in the receiver. Fail if the argument index is not an
    Integer or is out of bounds."

    <primitive: 60>
    <atomicUseUntransformed: #atomicAt:>
    self primitiveFail

Object≫atomicAt: index
    ^ self workingCopy at: index'
```

Compiling all the methods of the system is costly both in time and memory. Most methods available in the system are never called from within a transactional context and therefore do not need to be translated. The dynamic nature of Smalltalk makes it difficult to determine statically the required set of transactional methods, however it allows us to compile methods lazily when they are about to be executed. This produces a slowdown the first time a method is executed within a transactional context, but subsequent invocations are dispatched using the normal mechanisms of the VM and therefore run at full speed.

Most transactional systems prohibit system calls and filesystem access during transactions [16, 17]. Our approach allows replacement code to be specified for use within a transactional context. For example, when deleting a file the action is recorded with a custom change object and atomically applied together with the other changes upon successful commit of the transaction:

```
FileDirectory≫deleteFile: aString
    <primitive: 'primitiveFileDelete' module: 'FilePlugin'>
    <atomicUseUntransformed: #atomicDeleteFile:>

FileDirectory≫atomicDeleteFile: aString
    Processor activeProcess currentTransaction
        addChange: (CustomChange onApply: [ self deleteFile: aString ])
```

Our model also allows exceptions to be thrown and handled inside the transaction boundaries. An exception that leaves the boundaries of a transaction causes that transaction to abort and the exception to be re-raised in the non-transactional context.

## 3.2 Transactions at runtime

When entering a transaction we create a new transaction object and store it in an instance variable of the current process, as depicted in Figure 4. When leaving a transaction we set the current transaction reference back to nil. In this way we can efficiently determine the current transaction from anywhere in our application. Moreover we capture an escape continuation upon entry, to be able to abort the current transaction by doing a non-local jump to the calling context.

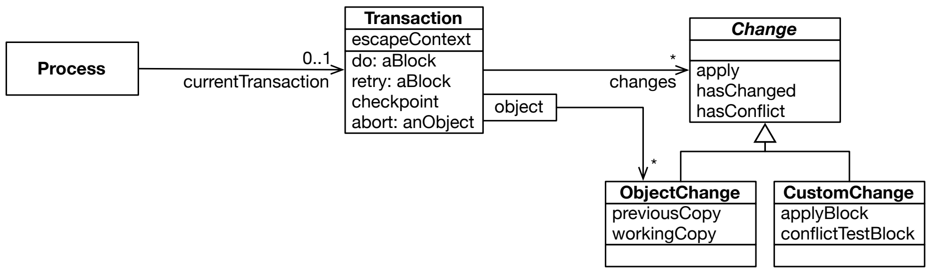


**Fig. 4.** Dynamic model of transactions at runtime

After having entered a transactional context, all the executed code is in its transformed form. This means that state access goes through special accessor methods and all message sends are redirected to their transactional counterparts. Like that transactional code execution works the same as normal code execution: it shares the same object memory but it uses a different access strategy to access state.

We adopt a conventional optimistic transaction protocol [18]. Whenever an object is touched within the context of a transaction for the first time (read or written), the transaction instantiates a new change object ObjectChange. This change object contains references to two copies of the object. The previousCopy contains an immutable copy for detecting conflicts. The workingCopy is a mutable copy of the object being used during the transaction. The change object knows if it has a conflict (the original object is not the same as the previous copy) and if it has changed (original object is not the same as the working copy).

We also provide a custom change object CustomChange that is used to record irreversible actions that should only be applied during the atomic commit phase if there are conflicts. We have seen the use of such a custom change in Section 3.1, where we presented a possible solution for file-deletion within a transactional context.

At the end of the transaction we have to acquire a form of "global lock" on the object memory to be able to check for conflicts and commit the changes. We use #valueUnpreemptively implemented on block closures to ensure that no other process is running at the same time. As a first step we check if any of the changes we gathered during the transaction has a conflict and raise an exception if this is the case. Otherwise we copy the changes from the working copies to the original objects. The time required to hold the lock and to validate and apply the changes linearly depends on the number of objects involved in the transaction.

These are some important properties of our transactional model [19]:

**Repeatable read.** Reads are repeatable. Since data that is read within a transaction is copied, repeated reads from within a transaction are consistent. Changes outside the transaction are not visible after a first read.

**Optimistic write.** Our transactional memory writes optimistically [20]. The transaction boundaries are controlled by working on copies of the objects.

**Lazy version management.** We create copies of objects that are read and written within transactions. This requires an extra redirection for accessing the state and a considerable amount of memory and processing time for copying the involved objects. Aborting a transaction is cheap as no state has to be restored.

**Lazy conflict detection.** Assuming that conflicts are rare, conflicts are checked before committing data. This check happens atomically together with the commit.

**Lazy conflict resolution.** Conflicts are resolved by dropping (or retrying) the transaction that produces the conflict when committing. The changes are eventually collected by the garbage collector.

## 4 Validation

First we assess the cost transactions by means of benchmarks that compare the running time of actions performed with and without transactions. Then we compare the cost of thread-safety realized with semaphores to that of our implementation with transactional memory.

### 4.1 Micro benchmarks

We performed several micro benchmarks to establish the runtime cost of using our implementation of transactional memory for Smalltalk. Table 2 shows the times and ratios of performing basic actions, such as invoking a method or accessing state. $t_1$ is the time required to perform the action $10^7$ times outside a transactional context, and $t_2$ is the time required to perform the same action within a transactional context. The benchmarks were performed on an Apple MacBook Pro, 2.16 GHz Intel Core Duo in Squeak 3.9. The required transactional methods were compiled in advance.

The activation time is the time required to enter a transaction as compared to the time required to evaluate a block closure. The *ratio* indicates that entering a transaction is 31 times slower than entering a block closure. This results from the fact that entering a transactions requires several objects to be instantiated to track the changes of the transaction. Moreover the transaction is recorded in the current process and an escape context must be captured to be able to abort a running transaction.

Normal method invocation does not show any speed penalty. In all the benchmarks we assume that the transactional methods are already compiled. For some common selectors, such as #+, #*, #=, #size, #at:put:, #new, #class, *etc.*, Squeak uses special byte codes to make the invocation about twice as fast as a common send. In a transactional context these byte codes cannot be used anymore and have to be replaced by normal message sends, resulting in a penalty for special method invocations.

| Operation | $t_1$ | $t_2$ | ratio |
|---|---|---|---|
| Activation | 2.75 | 85.27 | 31.03 |
| Method invocation | 1.98 | 1.98 | 1.00 |
| Special method invocation | 1.14 | 2.00 | 1.75 |
| Instance variable read | 1.03 | 20.72 | 20.08 |
| Instance variable write | 1.13 | 21.04 | 18.60 |
| Indexed variable read | 1.11 | 19.92 | 17.93 |
| Indexed variable write | 1.21 | 20.22 | 16.75 |
| Global variable read | 1.03 | 20.89 | 20.25 |
| Global variable write | 1.15 | 21.72 | 18.92 |

**Table 2.** $t_1$: time in seconds for $10^7$ runs in a non-transactional context, $t_2$: time in seconds for $10^7$ runs in a transactional context, ratio: $t_2/t_1$, the penalty when used in a transactional context.

State access within a transactional context is fairly expensive. For instance, indexed and global variable reads and writes produce very similar results: in the current implementation these are about 20 times slower than their non-transactional counterparts. As we have seen in Section 3.2, accessing state of an object requires to lookup the current transaction, the change object and to dispatch the state access to its working copy. This whole procedure involves several message sends that cannot be easily optimized in Smalltalk. Further improvements are possible by writing primitives (or introducing new byte-codes) that can more efficiently dispatch that kind of request.

Here we have been comparing the cost of thread-safe actions to unsafe actions. A fairer comparison would be that between thread-safe actions implemented with semaphores and thread-safe transactions. We discuss this in the following section.

## 4.2 Real world example using transactional memory

We applied our transactional model to Pier, a web-based content management system [21, Chapter 3]. Pier uses a tightly-connected graph of objects to represent pages and their content. Edit operations on pages use the Command design pattern and are executed while holding a global lock. Some operations, such as adding or removing a page, require the system to walk through the whole object graph to invalidate links. This results in a significant amount of time that all other commands are blocked, even though edit operations rarely conflict with each other.

Before executing a command Pier checks for conflicts on the current page, to avoid that changes of other users accidently get overridden. It does not check for conflicts that could be caused by the need to update links in other parts of the object model. Pier normally does not lock read operations, such as browsing the web site, as they are very common and would introduce a major bottleneck. In rare cases users could therefore encounter an inconsistent state of a particular page.

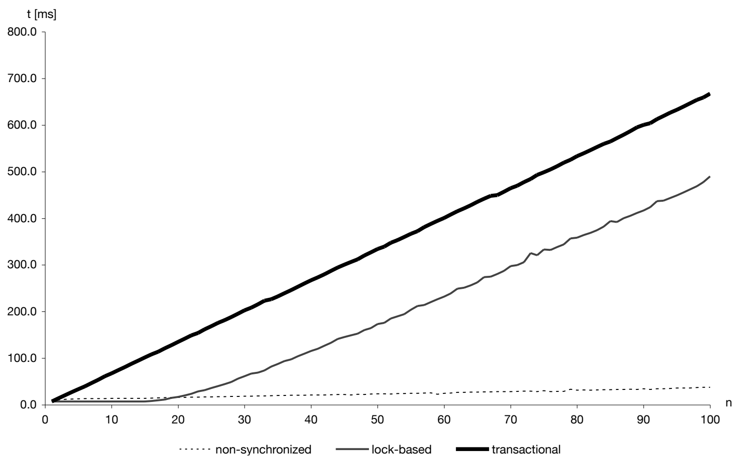


**Fig. 5.** Average execution time for non-synchronized, lock-based and transactional execution time $t$ to complete $n = 1..100$ concurrent edit operations in Pier.

To assess the effectiveness of transactional memory for Smalltalk, we remove the global lock in Pier and wrap the execution of the command within a transactional context. This means that edit commands can now be evaluated concurrently while still ensuring consistency. Moreover we could remove the manual checks for conflicts as these are now detected and handled by the transaction in a complete manner. Page views are now guaranteed to see a consistent state of the web site, as all the changes are applied atomically through a transaction.

Figure 5 shows the average execution time of an edit command that changes the contents of a single page. Using a script we simulated $n = 1..100$ concurrent edit operations on different pages, so no conflicts could occur. Interestingly the overhead is just over 100 ms for transactions over locks. The transactions are short, involve only few objects and little state access. Memory requirements are moderate: the edit operation touches 39 objects, whereof the transaction requires 2 copies of each object to track changes. In this particular use case, a single transaction consumes 2'556 bytes of additional memory.

We believe that the transactional approach would be considerably faster than the lock-based one, if the Squeak VM would exploit multiple CPUs to process concurrent requests.

## 5 Conclusion and future work

Smalltalk VMs traditionally offer poor support for concurrency control. Existing Smalltalk dialects provide only lock-based concurrency control, with the exception of GemStone Smalltalk, which provides transactions only for database code. In this paper we have presented an implementation of optimistic transactions for Squeak Smalltalk without modifying the underlying VM.

Our prototype implementation demonstrates that any Smalltalk can profit from having a transactional model. The implementation can be potentially ported to any of today's available Smalltalk platforms, as it is purely based on parse tree transformation of source code. The fact that the whole implementation is written in Smalltalk makes it an ideal platform to experiment with different transaction policies and implementation strategies. Changes to the transactional runtime system and transactional code can be applied and compiled on the fly, so there is no need to restart or rebuild the system.

Our approach works well with external libraries. New code that is loaded into the Smalltalk environment is transformed lazily within the context of a transaction. Primitive methods, filesystem I/O and exceptions work well together with transactions, as special transformation rules can be specified using method annotations. Contrary to other approaches our implementation integrates well with garbage collection, as the transactions are fully implemented in the object system of Smalltalk.

State access within a transaction is about 20 times slower than usual, which is a big penalty to pay. The integration of transactions with the object model at the VM level would certainly lead to much better performance, however we would also lose the flexibility to be able to quickly change the semantics of the transactional mechanisms. Code not using transactions continues to work exactly as before. The traditional mechanisms used for concurrency control can be even mixed with transactions.

As future work we would like to investigate how to further improve the speed of our model. We would like to investigate other areas of applicability, such as atomic loading of source code. In Smalltalk this is traditionally done in an incremental manner and poses certain problems, for example when the application is supposed to continue running while loading.

Furthermore we would like to see how to apply our approach to other dynamic programming languages, such as Python or Ruby. We expect the implementation in those scripting languages to be much more difficult than in Smalltalk, as both languages have major parts of their library implemented in C. Also they both lack direct support to transform source code using high-level AST representations.

Our approach to implementing optimistic transactions in Smalltalk can be seen as a special case of *context-oriented programming* [22], a programming paradigm that supports context-dependent behaviour. Transactional behaviour is automatically dispatched whenever we enter a transactional context. We believe that this approach can be extended more generally to support other forms of context-dependent concurrency control: instead of littering code with explicit calls to specific concurrency mechanisms, one should be able to simply annotate code with the concurrency properties one would like to ensure, and depending on the runtime context the appropriate behaviour will be automatically selected. We also intend to explore more efficient approaches to implementing contextual behaviour, in particular the use of *scoped reflection* [23] to control the temporal and spatial context in which reflective behaviour is active.

**Acknowledgments**

We thank Tudor Gîrba for his careful review of a draft of this paper. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project “Analyzing, capturing and taming software change" (SNF Project No. 200020-113342, Oct. 2006 - Sept. 2008).

## References

1. Goldberg, A., Robson, D.: Smalltalk 80: the Language and its Implementation. Addison Wesley, Reading, Mass. (May 1983)
2. American National Standards Institute, Inc.: Draft American National Standard for Information Systems — Programming Languages — Smalltalk. American National Standards Institute (1997)
3. Harris, T., Fraser, K.: Language support for lightweight transactions. In: Object-Oriented Programming, Systems, Languages, and Applications. ACM Press, New York, NY, USA (October 2003) 388–402
4. Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., Kay, A.: Back to the future: The story of Squeak, A practical Smalltalk written in itself. In: Proceedings OOPSLA ’97, ACM SIGPLAN Notices, ACM Press (November 1997) 318–326
5. Hansen, P.B.: Monitors and Concurrent Pascal: a personal history. ACM Press, New York, NY, USA (1996)
6. Herlihy, M.P.: Wait-free synchronization. ACM Transactions on Programming Languages and Systems **13**(1) (January 1991) 124–149
7. Herlihy, M.P., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: Proceedings of the 20. Annual International Symposium on Computer Architecture. (1993)
8. Butterworth, P., Otis, A., Stein, J.: The GemStone object database management system. Commun. ACM **34**(10) (1991) 64–77
9. Guerraoui, R., Garbinato, B., Mazouni, K.R.: The garf library of dsm consistency models. In: EW 6: Proceedings of the 6th workshop on ACM SIGOPS European workshop, New York, NY, USA, ACM Press (1994) 51–56
10. Guerraoui, R., Felber, P., Garbinato, B., Mazouni, K.: System support for object groups. In: OOPSLA ’98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, New York, NY, USA, ACM Press (1998) 244–258
11. Briot, J.P.: Actalk: A testbed for classifying and designing actor languages in the Smalltalk-80 environment. In Cook, S., ed.: Proceedings ECOOP ’89, Nottingham, Cambridge University Press (July 1989) 109–129
12. Hindman, B., Grossman, D.: Atomicity via source-to-source translation. In: MSPC ’06: Proceedings of the 2006 workshop on Memory system performance and correctness, New York, NY, USA, ACM Press (2006) 82–91
13. Adl-Tabatabai, A.R., Lewis, B.T., Menon, V., Murphy, B.R., Saha, B., Shpeisman, T.: Compiler and runtime support for efficient software transactional memory. In: PLDI ’06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation, New York, NY, USA, ACM Press (2006) 26–37
14. Röthlisberger, D., Denker, M., Tanter, É.: Unanticipated partial behavioral reflection: Adapting applications at runtime. Journal of Computer Languages, Systems and Structures (2007) To appear.

15. Denker, M., Ducasse, S., Lienhard, A., Marschall, P.: Sub-method reflection. Journal of Object Technology **6**(9) (September 2007) To appear.
16. Lie, S.: Hardware support for unbounded transactional memory. Master's thesis, Massachusetts Institute of Technology (May 2004)
17. Moore, K.E., Bobba, J., Moravan, M.J., Hill, M.D., Wood, D.A.: LogTM: Log-based transactional memory. In: Proceedings of the 12th International Symposium on High-Performance Computer Architecture. IEEE Computer Society (February 2006) 254–265
18. Lea, D.: Concurrent Programming in Java, Second Edition: Design principles and Patterns. 2nd edn. The Java Series. Addison Wesley (1999)
19. Bobba, J., Moore, K.E., Yen, L., Volos, H., Hill, M.D., Swift, M.M., Wood, D.A.: Performance pathologies in hardware transactional memory. In: Proceedings of the 34rd Annual International Symposium on Computer Architecture. International Symposium on Computer Architecture (June 2007)
20. Kung, H.T., Robinson, J.T.: On optimistic methods for concurrency control. ACM TODS **6**(2) (June 1981)
21. Renggli, L.: Magritte – meta-described web application development. Master's thesis, University of Bern (June 2006)
22. Costanza, P., Hirschfeld, R.: Language constructs for context-oriented programming: An overview of ContextL. In: Proceedings of the Dynamic Languages Symposium (DLS) '05, co-organized with OOPSLA'05, New York, NY, USA, ACM Press (October 2005)
23. Nierstrasz, O., Denker, M., Gîrba, T., Lienhard, A.: Analyzing, capturing and taming software change. In: Proceedings of the Workshop on Revival of Dynamic Languages (co-located with ECOOP'06). (July 2006)

# Linguistic Symbiosis between Actors and Threads

Tom Van Cutsem[⋆], Stijn Mostinckx[⋆⋆], and Wolfgang De Meuter

{tvcutsem|smostinc|wdmeuter}@vub.ac.be
Programming Technology Lab
Vrije Universiteit Brussel
Brussels – Belgium

**Abstract.** We describe a linguistic symbiosis between AmbientTalk, a flexible, domain-specific language for writing distributed programs and Java, a conventional object-oriented language. This symbiosis allows concerns related to distribution (service discovery, asynchronous communication, failure handling) to be handled in the domain-specific language, while still enabling the reuse of existing software components written in a conventional language. The symbiosis is novel in the sense that a mapping is defined between the concurrency models of both languages. AmbientTalk employs an inherently event-driven model based on actors, while conventional object-oriented languages employ a concurrency model based on threads. The contribution of this paper is a linguistic symbiosis which ensures that the invariants of the event-driven concurrency model are not violated by engaging in symbiosis with multi-threaded programs.

**Keywords**: actors, threads, events, linguistic symbiosis, AmbientTalk

## 1 Introduction

The hardware advances in networking technology of the past decade have resulted in novel kinds of distributed systems, commonly referred to as *mobile ad hoc networks*. Such networks are no longer populated by large, immobile servers and desktop machines interconnected by reliable network cables, but rather are populated by small, mobile handheld computers or cellular phones interconnected by highly volatile wireless communication links. These hardware changes profoundly affect the design of software that has to run reliably in such an environment.

Our approach to tackle the novel distributed computing issues in mobile networks is based on dedicated programming language support. Language support can aid the programmer to build software that has been designed to run in such inherently volatile networks. Novel language constructs help the software

[⋆] Research Assistant of the Fund for Scientific Research Flanders, Belgium (F.W.O.)
[⋆⋆] Funded by a doctoral scholarship of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen), Belgium.

developer both during the design phase (different programming paradigms foster different solutions to similar problems) as well as during actual development (dedicated language constructs can abstract away many low-level issues).

We have developed a domain-specific language called *AmbientTalk* which provides built-in language constructs specifically geared towards distributed development for highly asynchronous, volatile, infrastructure-shy mobile ad hoc networks [1]. Naturally, the advantage of this approach is that programs related to AmbientTalk's problem domain can be more easily expressed. The downside of having a domain-specific language is that it is either not always appropriate to model regular application logic or simply that a lot of existing software components developed in other languages cannot be reused.

In this paper, we describe a *linguistic symbiosis* between the domain-specific, actor-based AmbientTalk language and the general-purpose object-oriented and multithreaded Java language. The symbiosis is profitable for both symbionts: AmbientTalk profits from the large amount of available Java software components, while Java profits from AmbientTalk's high-level support for distributed programming.

The main difficulty to be tackled by the AmbientTalk/Java symbiosis is the mapping of both languages' fundamentally different concurrency models. While AmbientTalk is entirely built on an event-driven actor-based architecture, Java employs a traditional multithreaded model. A linguistic symbiosis between two such models is not without danger: the event-driven model enforces certain concurrency constraints which could be violated by Java's multithreaded concurrency if Java objects access AmbientTalk objects without proper provisions. The contribution of this paper is a mapping between actors and threads which does not violate the properties of the event-driven actor model of AmbientTalk.

## 1.1 Motivation

Before describing the AmbientTalk/Java symbiosis and the problems that have to be tackled, we briefly highlight why AmbientTalk enforces an actor-based, event-driven model rather than a traditional multithreaded model. The first motivation for employing such a model has to do with the beneficial properties of actor-based distributed communication in the context of mobile ad hoc networks. In such networks, network partitions occur much more frequently because e.g. two mobile devices may move out of one another's wireless communication range. Such network partitions are often temporary: the partition is healed when the mobile devices move back into one another's communication range. The asynchronous nature of the actor model enables distributed communication to be decoupled in time by means of an actor's built-in message queues. For example, an object can send an asynchronous message to a disconnected object without being blocked because the message can be stored in a message queue while the recipient is disconnected. A more extensive discussion of the beneficial properties of actors in mobile networks can be found elsewhere [2].

The second motivation behind employing an event-driven concurrency model stems from the inherently event-driven nature of distributed systems. Devices may join or leave the network and messages can be received from remote devices at any point in time. In an event-driven concurrency model, event handlers implicitly denote an atomic block, restricting the non-determinism to the order in which events are processed. This is in contrast with pre-emptive multithreading where the programmer should manually denote such atomic blocks to ensure that the code is consistent with every possible interleaving of all threads in the system.

## 1.2 Problem Statement

We now describe the main concurrency problem that has to be tackled by the AmbientTalk/Java symbiosis. As will be explained in more detail in section 2.3, each AmbientTalk object is associated with *a single* actor and only this actor may directly invoke the object's methods. This ensures that race conditions on regular objects cannot occur. However, when an AmbientTalk object is passed as a parameter to a Java method invocation (via symbiosis), the object can become accessible to Java threads. When these threads in turn invoke a method on the AmbientTalk object (via symbiosis), the AmbientTalk object and any other accessible objects may be manipulated concurrently both by the object's actor and by other Java threads, violating the guarantee that race conditions on AmbientTalk objects cannot occur. As an example of how race conditions on objects could be caused, consider this following code which registers an AmbientTalk object as a Java `ActionListener` on an AWT `Button` instance.

```
actor: {
  // code executed by AmbientTalk actor
  def obj := object: { ... }
  button.addActionListener(object: {
    def actionPerformed(actionEvent) {
      // code executed by Java thread
    };
  });
}
```

The `actionPerformed` method is invoked by the AWT framework's thread when the button is clicked. Hence, objects visible to both the actor and the anonymous listener object, such as the `obj` object, could be manipulated by multiple concurrent threads, requiring the introduction of locks and other synchronisation constructs in AmbientTalk. In other words, AmbientTalk's actor model would have to be abandoned when engaging in symbiosis with Java. In section 4 a symbiotic protocol mapping is described that enables an event-driven actor language to safely engage in symbiosis with a multithreaded programming language, such that problems like the one illustrated above are avoided.

Although most prevalent object-oriented languages are multithreaded, that does not prevent programs written in those languages to foster an event-driven programming *style* (which is upheld by convention, not enforced by the language). Therefore the symbiosis described later pays specific attention to map the event-driven style of the thread-based language directly onto the appropriate event constructs of the event-driven programming language.

## 2 AmbientTalk

Before describing the AmbientTalk/Java symbiosis, we briefly introduce the AmbientTalk language, particularly emphasising its concurrency model. Although AmbientTalk is domain-specific in terms of its abstractions for distributed programming, it is a full-fledged object-oriented programming language in its own right. AmbientTalk inherits most of its standard language features from Self, Scheme and Smalltalk. From Scheme, it inherits the notion of true lexically scoped closures. From Self and Smalltalk, it inherits an expressive block closure syntax, the representation of closures as objects and the use of block closures for the definition of control structures. AmbientTalk's object model is derived from Self: classless, slot-based objects using delegation [3] as a reuse mechanism.

The concurrent and distributed features of the language have a different lineage. Rather than employing a multithreaded concurrency model, AmbientTalk's model is founded on the actor model of computation [4] and its many incarnations in languages such as Act1 [5], ABCL [6] and Actalk [7]. However, AmbientTalk's closest relative is the E programming language [8] (further described in section 7.1). E combines actors and objects into a unified model called *communicating event loops*, which is based on event loop concurrency, described in section 2.2.

### 2.1 Object-oriented Programming

The following code illustrates standard object-oriented programming in AmbientTalk.

```
def Account := object: {
  def balance := 0;
  def init(amount) { balance := amount };
  def deposit(amnt) { balance := balance + amnt };
  def withdraw(amnt) { balance := balance - amnt };
};
def LimitAccount := object: {
  super := Account;
  def limit := 0;
  def init(lowest, amount) {
    super :=  Account^new(amount);
    limit := lowest;
  };
  def withdraw(amnt) {
    (self.balance - amnt < limit).ifTrue: {
      raise: TransactionException.new(self, amnt);
    } ifFalse: {
      super^withdraw(amnt);
    }
  };
};
def account := LimitAccount.new(-500, 1000);
account.deposit(20);
account.balance; // returns 1020
```

Two prototypes are defined, one for `Account` objects and one for `LimitAccount` child objects, which set a limit to the amount of money that can be withdrawn from the account. Objects can be created ex-nihilo, by cloning or by instantiating objects. Instantiating an object is done by sending it the message `new`, which creates a shallow copy of that object and initialises the copy using its `init` method, which plays the role of "constructor". AmbientTalk's object instantiation is similar to class instantiation, except that the new object is a clone of an existing object, rather than an empty object allocated from a class.

Every object has a field slot named `super` denoting the "parent object" to which it delegates messages it cannot handle. The parent of an `Account` object is `nil`, the parent of a `LimitAccount` object is an `Account` object. Next to this implicit delegation, which occurs when an object receives a message it does not understand, AmbientTalk also allows objects to explicitly delegate a request to another object. The expression `obj^m()` delegates the message `m` to `obj`, leaving `self` bound to the sender. A traditional "super send" in AmbientTalk is then a message that is delegated to the object stored in an object's `super` slot.

Block closures are constructed by means of the syntax `{ |args| body }`, where the arguments can be omitted if the block takes no arguments. Note that AmbientTalk supports both traditional canonical syntax as well as keyworded syntax for method definitions and invocations.

### 2.2 Event Loop Concurrency

AmbientTalk's concurrency model is based on communicating event loops [8]. In this model, an actor is represented as an event loop, rather than as a traditional "active object". An *event loop* is a thread of execution that perpetually processes *events* from its *event queue* by invoking a corresponding *event handler*. Communicating event loops enforce three essential concurrency control properties:

**Property 1 (Serial Execution)** *An event loop processes incoming events from its event queue one by one, i.e. in a strictly serial order.*

Property 1 ensures that the handling of a single event is atomic: race conditions on the event handler's state while handling the event cannot occur.

**Property 2 (Non-blocking Communication)** *An event loop* never *blocks waiting for another event loop to finish a computation. Rather, all communication between event loops occurs strictly by means of asynchronous event notifications.*

Property 2 ensures that event loops can never deadlock one another. However, in order to guarantee progress, event handlers should not execute e.g. infinite `while` loops. Long-running actions should be performed piecemeal by means of scheduling recursive events, such that an event loop always gets the chance to respond to other incoming events. An event loop can only be suspended when its event queue is empty.

**Property 3 (Exclusive State Access)** *An event loop has* exclusive *access to its mutable state. In other words, two or more event loops may* never *have direct access to shared mutable state.*

Property 3 ensures that event handlers never have to lock mutable state. Mutating another event loop's state has to be performed indirectly, by asking the event loop to mutate its own state via an event notification.

Event loop concurrency avoids deadlocks and race conditions *by design*. The non-determinism of the system is confined to the order in which events are processed. In standard pre-emptive thread-based systems, the non-determinism is much greater because threads may interleave upon each single instruction. In the following section, we describe how the abstract event loop model is incorporated into the AmbientTalk language.

## 2.3 AmbientTalk actors

In AmbientTalk, actors are represented as event loops: the event queue is represented by an actor's message queue, events are represented as asynchronous message sends, and event handlers are represented as the methods of regular objects. The actor's event loop thread perpetually takes a message from the message queue and invokes the corresponding method of the object denoted as the receiver of the message.

In AmbientTalk, each object is said to be *owned* by exactly one actor. Only an object's owning actor may directly execute one of its methods. Objects owned by the same actor communicate using standard, sequential message passing. It is possible for objects owned by an actor to refer to objects owned by other actors. Such references that span different actors are named *far references* (the terminology stems from E [8]) and only allow asynchronous access to the referenced object. Any messages sent via a far reference to an object are enqueued in the message queue of the object's owning actor and processed by that actor itself. AmbientTalk borrows from the E language the syntactic distinction between sequential message sends (expressed as `o.m()`) and asynchronous message sends (expressed as `o<-m()`). In the remainder of this paper, we assume that asynchronous sends do not return a value. Another semantics for return values is discussed in future work (see section 8).

Figure 1 illustrates AmbientTalk actors as communicating event loops. The dotted lines represent the control flow of event loops which perpetually take messages from their message queue and synchronously execute the corresponding methods on the actor's owned objects. The control flow of an event loop never "escapes" its actor boundary. When communication with an object in another actor is required, a message is sent asynchronously via a far reference to the object. The message is enqueued and eventually processed by the receiver object's own actor.

Because objects residing on different devices are necessarily owned by different actors, the only kinds of object references that can span across different devices are far references. This ensures by design that all distributed communication is asynchronous. AmbientTalk's far references are by default resilient to network disconnections: asynchronous messages may be buffered within the reference while the remote receiver object is disconnected. It is this design that

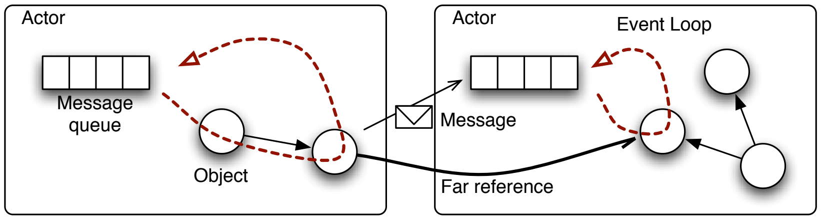


**Fig. 1.** AmbientTalk actors as communicating event loops

makes AmbientTalk's distribution model suitable for mobile ad hoc networks because it allows one to abstract over temporary network partitions.

## 3 The AmbientTalk/Java Symbiosis

Our model for explaining the AmbientTalk/Java linguistic symbiosis is based on that of Inter-language Reflection [9]. In this model, a linguistic symbiosis consists of:

– a **data mapping** which ensures that data in one language looks like data in the other language, such that the symbiosis becomes as syntactically transparent as possible. For example, it is desirable that Java objects are equally represented as objects in AmbientTalk, such that messages can be sent to objects regardless of their native language.
– a **protocol mapping** between the meta-level representation of both languages' data. For example, both AmbientTalk and Java objects communicate by sending messages, but AmbientTalk is dynamically typed while Java is statically typed and exploits type overloading during method lookup. A proper symbiosis needs to map these message sending protocols onto one another.

AmbientTalk has been implemented in Java. Because of this, Java plays two roles: it is both a symbiont language and the implementation language of AmbientTalk (and hence of the linguistic symbiosis itself). Figure 2 illustrates the different objects that play a part in the AmbientTalk/Java symbiosis, according to the implementation model of Inter-language reflection [9]. AmbientTalk objects are physically implemented as Java objects. This is illustrated by means of the "represents" relationship. To enable symbiosis, additional objects are required which denote the *appearance* of objects from one language in the other language. At the implementation level, such appearances are implemented as *wrapper* objects, which wrap an object from a different language and which perform the protocol mapping which translates between the semantics of the symbiont languages. In what follows, we describe the standard data and protocol mappings for the AmbientTalk/Java symbiosis.

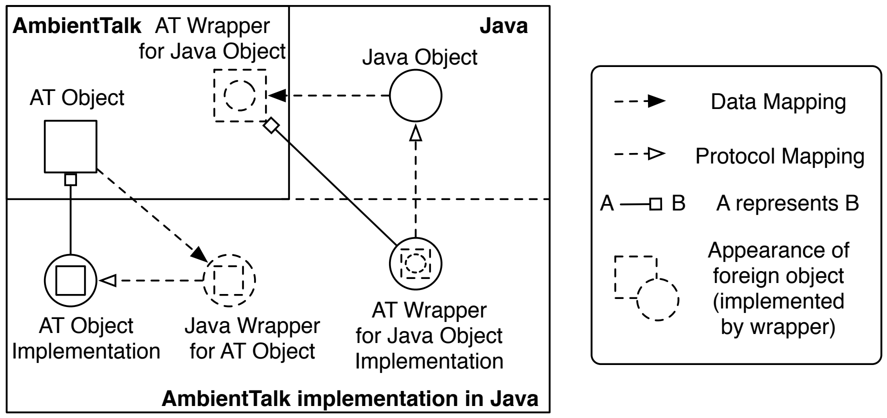

AmbientTalk
AT Wrapper for Java Object
Java
Java Object
AT Object
AT Object Implementation
Java Wrapper for AT Object
AT Wrapper for Java Object Implementation
AmbientTalk implementation in Java
Data Mapping
Protocol Mapping
A B
A represents B
Appearance of foreign object (implemented by wrapper)

**Fig. 2.** Symbiotic representation of AmbientTalk and Java Objects

## 3.1 Data Mapping

AmbientTalk's data mapping is similar to that of other dynamic languages implemented on top of the JVM such as Jython and JRuby (as discussed later in section 7.1). We introduce the linguistic constructs by means of a toy chat program, shown below. This small AmbientTalk program constructs a graphical user interface using the Java Swing framework. The GUI consists of a simple input field and an output text area. The code assumes that all connected chat participants are stored in a `participants` array. When the user enters text in the text field, AmbientTalk code broadcasts the text message to all connected participants. Note the registration of an anonymous AmbientTalk object as an action listener on the text field.

```
def swing := jlobby.javax.swing;
def JFrame := swing.JFrame;
def JTextField := swing.JTextField;
def JTextArea := swing.JTextArea;

// instantiate classes by sending them the "new" message
def frame := JFrame.new("Chat");
def textfield := JTextField.new(20);
def outputArea := JTextArea.new();

// static Java fields appear as fields on class wrapper
frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

// symbiotic method invocations
def pane := frame.getContentPane();
pane.setLayout(jlobby.java.awt.GridLayout.new(2,1));
pane.add(textfield);
pane.add(outputArea);
// the anonymous object is an AmbientTalk object that
// masquerades as a Java ActionListener object
textfield.addActionListener(object: {
  def actionPerformed(actionEvent) {
    // String returned by getText() is converted into AmbientTalk text
    def msg := textfield.getText();
    // participants is an array of all connected participants
    participants.each: { |chat| chat<-tell(msg) }
    // AmbientTalk text is turned into a String
    outputArea.append(msg);
  };
});
frame.setVisible(true);
```

**Appearance of Java objects in AmbientTalk** In order for AmbientTalk objects to talk to Java objects, they first need to get access to Java classes. From classes, objects can then be referenced via static fields or by instantiating the referenced classes. Java classes are organised hierarchically by means of packages. We have chosen to mimick this structural hierarchy by means of simple objects whose public slot names correspond to nested Java package or class names. The root of this hierarchy is named `jlobby`[1]. As can be seen from the code example, package objects can be created by selecting the slot with the appropriate name from `jlobby`.

Given a package object like the `swing` object in the chat example, it suffices to select the slot matching the Java class name to refer to a Java class as if it were an AmbientTalk object. Java classes appear as AmbientTalk objects whose fields and method slots correspond to public static fields and methods in the Java class. Hence, these fields and methods can be accessed or invoked using regular AmbientTalk syntax.

Java classes can be instantiated in AmbientTalk similar to how AmbientTalk objects are instantiated, i.e. by sending `new` to the wrapper for the class, which returns a wrapped instance of the Java class. Arguments to `new` are passed as arguments to the Java constructor. For example, in the chat application above, a new instance of a `JFrame` is created with the title of the frame passed as an AmbientTalk string. Java objects appear as AmbientTalk objects whose field and method slots correspond to public instance-level fields and methods in the Java object. These are accessed or invoked as if they were plain AmbientTalk slots.

The symbiotic architecture has a number of built-in conversions between several native datatypes. For example, AmbientTalk numbers are automatically converted into Java integers (e.g. `JTextField.new(20)`), AmbientTalk text is converted into Java `String`s, AmbientTalk arrays into Java arrays, AmbientTalk's `nil` value into Java's `null` value, etc. If the parameter-passed AmbientTalk object is actually a wrapper for a Java object, the unwrapped object is passed instead. These predefined conversions make the symbiosis highly transparent in most cases.

**Appearance of AmbientTalk objects in Java** There are two ways for Java code to gain access to AmbientTalk objects. The first is by embedding an AmbientTalk interpreter in existing Java code. The second is by means of a conversion

[1] Ordinary AmbientTalk programs use an object called the `lobby` to load external objects, hence the name `jlobby` for loading Java classes.

rule when AmbientTalk code invokes a Java method which expects an argument typed as an *interface*. Any AmbientTalk object can be converted by the symbiosis into a Java object which *implements* that interface. Any messages sent by Java objects to this interface object are transformed into AmbientTalk message sends on the wrapped AmbientTalk object. In the chat example, the symbiotic call to `addActionListener` requires a parameter of type `ActionListener`, which is an interface type. Instead of passing a wrapped Java object implementing this interface, it is allowed to pass any AmbientTalk object; the object is not even required to implement all declared interface methods. The anonymous object passed in the above code properly implements the `actionPerformed` callback, and will be notified by Java code whenever the user has entered new text in the text field. A discussion on how the concomitant threading issues are avoided is postponed until section 4.

## 3.2 Protocol Mapping

For a linguistic symbiosis to work correctly, more is required than simply an appearance for the entities of the foreign language. A protocol mapping must be defined between the meta-level operations defined on the appearance of an object in the one language and the meta-level operations defined on the actual object in the other language [9]. In the case of AmbientTalk/Java, we consider three protocols: the slot access/method invocation protocol, which translates between AmbientTalk message sends and Java method invocations, the delegation/inheritance protocol, which translates between object delegation and class inheritance and the actor/thread protocol, which translates between event-driven and multi-threaded concurrency control. We briefly describe the first two protocols below. The thread/actor protocol is described in detail in section 4.

**The Slot Access/Method Invocation Protocol** AmbientTalk's message sending protocol is entirely based on the concept that a selector (a symbol) uniquely denotes a slot in an object. In Java, on the other hand, static types can be used to *overload* a method name. At call time, the static types of the arguments are used to disambiguate the call. Another notable difference is that AmbientTalk supports explicit delegation, which implies that the object bound to `self` during method execution is not necessarily the object in which the method was found during method lookup. In Java, the value of `this` cannot be explicitly set to a separate delegating object. As a consequence, delegating a message to a Java object from within AmbientTalk does not allow the delegator to intercept self-sends performed by Java code.

*Invoking Java methods in AmbientTalk* The AmbientTalk/Java symbiosis treats message sends from AmbientTalk to Java as follows: if a message is sent to a class wrapper, only static fields or methods of the Java class are considered. If the message is sent to an instance wrapper, only non-static fields or methods of the Java class of the wrapped object are considered. If the AmbientTalk selector uniquely identifies a method (i.e. no overloading on the method name is performed in Java), the matching method is invoked. All AmbientTalk arguments are converted to Java objects by means of the data mapping described in the previous section. The Java return value is mapped back to an AmbientTalk value. If the Java method is overloaded based on arity (i.e. each overloaded method takes a different number of arguments), the number of arguments in the AmbientTalk invocation can be used to identify a unique Java method. If the Java method is overloaded based solely on argument types, the interpreter may derive that the actual arguments can only be converted from AmbientTalk to the appropriate Java types for exactly one of the matching overloaded signatures. In the remaining case in which the actual AmbientTalk arguments satisfy more than one overloaded method signature, the symbiotic invocation fails. It is then the AmbientTalk programmer's responsibility to provide explicit type information in the method invocation.

*Invoking AmbientTalk methods in Java* When an AmbientTalk object is passed as an argument to a Java method expecting an object of an interface type, the AmbientTalk object will appear to Java objects as a regular Java object implementing that interface. Hence, messages sent to this wrapped AmbientTalk object appear as regular Java method invocations on an interface type. For example, the action listener in the chat example can be notified as if it were a normal Java object by performing `listener.actionPerformed(event)`.

If Java invokes a method declared in an interface with an overloaded method signature, all overloaded invocations are transformed into the same method invocation on the AmbientTalk object. In other words, the AmbientTalk object does not take the types into consideration. However, if the Java method is overloaded based on arity, the AmbientTalk programmer can take this into account in the parameter list of the corresponding AmbientTalk method, by means of a variable-argument list or optional parameters. Otherwise, the Java invocation may fail because of an arity mismatch.

**The Delegation/Inheritance Protocol** In AmbientTalk, when objects need shared access to state or behaviour, they can do so by designating an object to hold that state or behaviour as their common parent. This use of delegation was first advocated by Lieberman [3] and is a key programming pattern in the prototype-based language Self. In Self, these shared parent objects are called *traits* and they play the role of shared repositories of behaviour normally played by classes in class-based languages [10].

The AmbientTalk/Java symbiosis represents the instance-class relationship of Java objects by means of the delegation link of the AmbientTalk wrappers: the `super` slot of an AmbientTalk wrapper for a Java object always points to the wrapper of that Java object's class. Because of this design, the concept of a class in Java symbiotically appears as the concept of a trait in AmbientTalk.

The delegation relationship between AmbientTalk objects is not mapped to Java inheritance concepts when passing an AmbientTalk object to Java code. To

Java code, the AmbientTalk object appears as an instance of a class implementing some interface. Java interfaces may extend other interfaces. It may be that the AmbientTalk object implements those extended interfaces by delegating to other AmbientTalk objects, but this is an implementation detail, just like it is an implementation detail how a Java class implements the methods of its declared interfaces.

## 4 The Actor/Thread Protocol

In this section, we describe a symbiotic protocol mapping for representing AmbientTalk actors as Java threads, and more interestingly, for streamlining multithreaded concurrency in Java as asynchronous message sending in AmbientTalk.

### 4.1 Viewing Actors as Threads

As described previously, an AmbientTalk actor consists of a message queue, an event loop and a number of objects owned by the actor. The event loop is essentially a thread which perpetually reads the next message from the message queue and invokes the method on an owned object designated as the receiver of the message. The invoked method is executed by the actor's event loop thread.

**Symbiotic invocations on Java objects** If an AmbientTalk object performs a symbiotic invocation on a Java object, the symbiotically invoked Java code is still executed by the actor's event loop thread. From a symbiosis point of view, an actor in AmbientTalk appears as a thread in Java by representing the actor as its own event loop thread. When an actor appears as a thread at the Java level, it has to abide by the rules of shared-state multithreaded concurrency: the event loop thread may take locks on Java objects and use Java's `wait` and `notify` synchronisation primitives. When the symbiotic invocation finally returns, the event loop thread transparently starts executing AmbientTalk code again and is trivially converted into the event loop of an actor again.

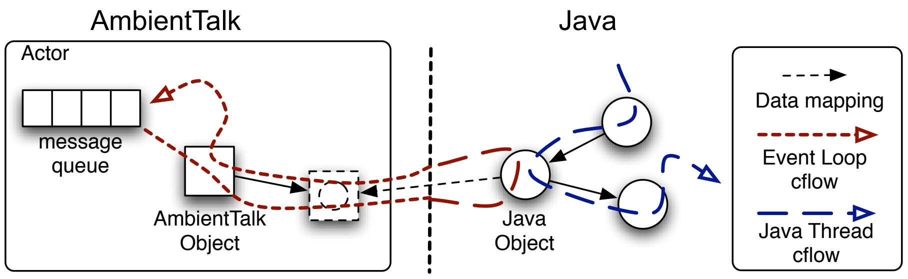


**Fig. 3.** Representing AmbientTalk actors as Java threads

Figure 3 illustrates the control flow of passing from the AmbientTalk level to the Java level. On the left-hand side of the figure, an AmbientTalk actor is executing a symbiotic invocation on a wrapped Java object. Notice how the event loop thread flows from the AmbientTalk into the Java level to execute the invoked Java method. Because other Java threads may be concurrently operating on the same object, synchronisation between the threads may be required.

**Immediate symbiotic invocations** When an AmbientTalk actor appears as a Java thread, it is possible that the thread performs a symbiotic invocation on a wrapped AmbientTalk object. For example, consider an AmbientTalk visitor object that is passed as argument in the `accept` method of a Java object which calls back on the visitor to visit the appropriate type. To enable a Java thread representing an actor to perform invocations on AmbientTalk objects, the symbiosis must define an appropriate equivalent in Java for the "ownership" relationship between objects and actors in AmbientTalk.

When an event loop thread enters the Java level, the symbiosis considers any Java object on which the thread operates as being owned by the actor represented by the thread. Hence, Java objects which are reachable from objects owned by an AmbientTalk actor are transitively considered owned by that actor. When multiple event loop threads access the same Java object, this object becomes owned by multiple actors, which may lead to ill-defined behaviour (see below). When a Java object that is transitively owned by an actor invokes a method on a wrapped AmbientTalk object, the protocol is as follows:

– If the owner of the wrapped AmbientTalk object equals the owner of the Java object, the invocation is performed *immediately*, by the Java thread representing the event loop itself. There is no need to synchronise access with the AmbientTalk actor's event loop because the Java thread *is* that event loop. If access would be synchronised, the event loop would wait for itself, resulting in immediate deadlock.
– If the owner of the wrapped AmbientTalk object does not equal the owner of the Java object, this implies that an AmbientTalk actor has gained direct access to an AmbientTalk object owned by another actor. Hence, the actor has circumvented the exclusive state access property by sharing an object with another actor at the Java level. In this case, the interpreter aborts the symbiotic invocation, rather than allowing race conditions to occur between the actors. The alternative of synchronising access to the AmbientTalk object is not viable, as this would violate the non-blocking communication property: one event loop (the executing Java thread) would be blocked waiting for another event loop (the event loop of the actor owning the wrapped object).

## 4.2 Viewing Threads as Actors

We now describe the protocol mapping that allows the threading model of Java to co-exist with the actor model based on communicating event loops of AmbientTalk. The symbiosis distinguishes between synchronous and asynchronous symbiotic invocations on AmbientTalk objects by Java code.

**Synchronous Symbiotic Invocations** When a Java thread (which does not represent an AmbientTalk actor) performs a method invocation on a wrapped AmbientTalk object, the method invocation *cannot* be executed immediately by the Java thread. Doing this would violate the serial access property of event loop concurrency, which ensures that no race conditions can occur on objects owned by an actor.

The solution employed by the AmbientTalk/Java symbiosis is to regard a Java wrapper for an AmbientTalk object as a *far reference* to the actual AmbientTalk object, and to interpret every Java method invocation on that wrapper as an *asynchronous message send*. The Java thread schedules the symbiotic method invocation for asynchronous execution in the message queue of the actor owning the wrapped AmbientTalk object, rather than synchronously invoking the method on the actual AmbientTalk object itself. Hence, it is the actor's own event loop that will process the symbiotic invocation, ensuring that the serial access property of the event-loop model remains intact.

By turning synchronous Java invocations on wrapped AmbientTalk objects into asynchronous AmbientTalk message sends, the properties of AmbientTalk's actor model remain intact. However, Java's threading model based on synchronous method invocation cannot always deal with this asynchrony: a method invocation normally returns a value or it may throw an exception. To reconcile asynchronous message sends with synchronous method invocations, we employ *futures*, which are a well-known abstraction that represent a handle to the return value of an asynchronous request (see section 7.3).

The asynchronous scheduling of the symbiotic invocation immediately returns a future object to the Java thread that schedules the request. The Java thread can then (explicitly) synchronise on the future, suspending the thread until the future either gets *resolved* with a return value, or until it is *ruined* by an exception (raised in the asynchronously invoked AmbientTalk code). When the event loop of an actor dequeues a symbiotic invocation request, it invokes the AmbientTalk method and uses the return value (respectively a caught exception) to resolve (respectively ruin) the future attached to the symbiotic invocation. This wakes up the waiting Java thread, which can then return from the symbiotic method invocation.

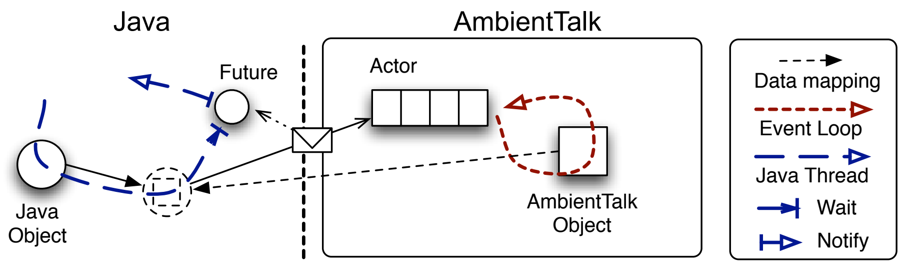


**Fig. 4.** Java method invocations appear as AmbientTalk message sends

Figure 4 illustrates the appearance of Java wrappers for AmbientTalk objects as far references at the AmbientTalk level. On the left-hand side of the figure, a Java thread is performing a symbiotic invocation on a wrapped AmbientTalk object. Rather than making the Java thread enter the AmbientTalk actor, a symbiotic invocation is scheduled in the message queue of the wrapped object's actor. The Java thread synchronises on the associated future object. While the Java thread is blocked, the actor event loop processes its incoming messages. When the symbiotic invocation has been completed, the future object is notified and the symbiotic invocation returns, enabling the Java thread to proceed its execution. At the Java level, the call appeared to be synchronous, at the AmbientTalk level, it appeared to be an asynchronous message send.

**Asynchronous Symbiotic Invocations** Java's native concurrency model is based on shared-state concurrency, i.e. multiple threads communicating by modifying shared objects. For many applications, this concurrency model is inappropriate. Many interactive applications (e.g. games, user interface frameworks) or discrete-event simulations require an event-driven approach. In Java, event-driven programming cannot be enforced. However, an event-driven style can be adopted by using an event loop framework. The Java AWT and Swing toolkits are the quintessence of such an approach.

In event-driven Java frameworks, asynchronous message sends are necessarily represented implicitly by synchronously invoking the methods of so-called *listener* objects. The documentation of most event-driven Java frameworks specifies that such methods must return as soon as possible, and should preferably only schedule tasks for later execution. In effect, this means that asynchronous message passing in Java is implemented as a *second-class* language construct by means of listener invocations. However, our symbiosis can detect such second-class asynchronous message sends and map them onto *actual* asynchronous sends in AmbientTalk, without having to synchronise the Java thread. This is highly desirable because it ensures the responsiveness of the event-driven Java framework.

As an example, consider an AmbientTalk object that is registered as an `ActionListener` on an AWT `Button`. When the AWT event loop invokes the `actionPerformed` method on the AmbientTalk object, this implicitly indicates an event notification, which is conceptually asynchronous. Hence, rather than making the AWT event loop suspend until the AmbientTalk actor has actually processed the `actionPerformed` method, as is the case for a synchronous symbiotic invocation, the method invocation is turned into an actual asynchronous message send, and the event loop thread can return immediately.

The AmbientTalk/Java symbiosis treats the invocation of any method belonging to (an extension of) the `java.util.EventListener` interface as an asynchronous message send, provided the method has a `void` return type and does not declare any thrown exception. It is a convention of Java frameworks that objects representing event listeners be tagged with (a subtype of) this interface. When a method on a wrapped AmbientTalk object representing an `EventListener` is invoked, a symbiotic call is scheduled as described previously, but no future is created on which to synchronise the Java thread. Rather, the Java thread returns immediately. Hence, the Java method invocation `buttonListener.actionPerformed(event)` is effectively interpreted by AmbientTalk as `buttonListener<-actionPerformed(event)`.

The symbiosis described above only covers asynchronous message sends without return values. A discussion on the possibility of allowing symbiotic future-type message sends is postponed until section 8.

### 4.3 Summary

In this section, we have described how actors and threads interact. When AmbientTalk objects invoke methods on Java objects, the invocation is always synchronous. When Java objects invoke methods on AmbientTalk objects, we distinguish three different kinds of symbiotic method invocations:

– Immediate symbiotic invocations occur when a Java thread that represents an AmbientTalk actor calls back on its own objects.
– Synchronous symbiotic invocations occur when a regular Java thread invokes a method on an AmbientTalk object.
– Asynchronous symbiotic invocations occur when a regular Java thread invokes a method that represents an event notification on an AmbientTalk object (i.e. the AmbientTalk object acts as a listener).

## 5 Applications

We now give concrete examples of each of the three symbiotic invocations on AmbientTalk objects described in the previous section, thereby illustrating how the thread/actor protocol mapping behaves in practice.

### 5.1 Immediate Invocation

A traditional advantage of introducing symbiosis between a small language like AmbientTalk and an industry-strength language like Java is that the vast amount of libraries available in the latter language can be reused. However, the use of a Java library is hardly ever restricted to simple one-way calls from AmbientTalk to Java. In many Java frameworks, for example, the framework objects call back on the parameter-passed AmbientTalk objects. We illustrate this interchange of messages by means of the Java Collection Framework. In the example, an AmbientTalk program periodically receives address card objects from PDAs in its environment. It will store these objects in a set to filter out duplicates. Moreover, the address cards should be ordered by name so that the user can be presented with an alphabetical overview of all nearby persons. This can be achieved with the following AmbientTalk code excerpt:

```
def vCardPrototype := object: {
  // implements the java.lang.Comparable interface
  def compareTo(vCard) {
    self.fullName.compareTo(vCard.fullName);
  };
  // define fields for fullName, address, etc.
};
def contacts := jlobby.java.util.TreeSet.new();
whenever: VCard discovered: { |vCard|
  contacts.add(vCard);
};
```

Whenever a vCard object is discovered in the network, the `add` method is invoked on the wrapped `TreeSet`. In order to correctly insert the element, the set repeatedly invokes the `compareTo` method on the Java wrapper for the inserted `vCard` object. As the thread invoking the method is the event loop thread owning the wrapped AmbientTalk object, the method can be executed immediately.

The immediate invocation presented in this section is the most common form of symbiotic invocations, often being the result of passing AmbientTalk objects as parameters to Java libraries. The Visitor design pattern [11] can be seen as the epitome of such an interaction. When traversing a Java data structure with an AmbientTalk visitor, a double dispatch across the language boundary occurs between `accept` and `visitType` messages. Such examples illustrate why it is necessary for the thread/actor protocol mapping to distinguish immediate invocations from synchronous invocations. If the callback operation from Java to AmbientTalk is not executed immediately, but rather by means of a synchronous invocation, the AmbientTalk actor would wait for itself, resulting in immediate deadlock.

## 5.2 Synchronous Invocation

As noted previously, linguistic symbiosis is often useful for a small language like AmbientTalk to reuse the large amount of software components available in a language like Java. However, it is equally viable for the Java programmer to embed AmbientTalk components into an existing Java framework. This allows the Java programmer to profit from e.g. AmbientTalk's language support for distributed programming.

In this section, we illustrate how AmbientTalk unit tests can be incorporated into the JUnit unit testing framework, allowing a Java developer to integrate all unit tests in a consistent testing framework. In AmbientTalk, a unit test is an object whose methods are prefixed with `test`. All unit test objects delegate to the prototypical unit test object, which contains reflective code to invoke all test cases. Note that this object also implicitly implements the `junit.framework.Test` interface.

```
def UnitTestPrototype := object: {
  def testMethods; // array of first-class method objects
  def init() {
    testMethods := retrieveTestMethods(self);
  };
  def countTestCases() { testMethods.length };
  def run(reporter) {
    reporter.startTest(self);
    testMethods.each: { |method| /* perform the test */ };
    reporter.endTest(self);
  };
}
```

Now consider a `TestSuite` that is composed of both unit tests written in Java and unit tests written in AmbientTalk. All unit tests uniformly implement the `Test` interface. In order to incorporate an AmbientTalk unit test into the test suite, it suffices to wrap the AmbientTalk object representing the unit test explicitly in the `Test` interface. The following code excerpt assumes that `exampleATTest` is a reference to an AmbientTalk unit test object.

```
public static void main(String[] args) {
  TestSuite suite = new TestSuite();
  ATObject exampleATTest = /* load AmbientTalk test */;
  Test exampleJavaTest = /* load Java test */;
  suite.addTest((Test) wrap(exampleATTest, Test.class));
  suite.addTest(exampleJavaTest);
  junit.textui.TestRunner.run(suite);
}
// see section 6 for details on wrapping AmbientTalk objects
static Object wrap(ATObject obj, Class interface) {
  return Proxy.newProxyInstance(interface.getClassLoader(),
                              new Class[] { interface },
                              new JavaWrapperForATObject(obj));
}
```

A `TestRunner` executes the test suite by sequentially invoking each unit test's `run` method. This execution is performed by a Java application thread. Because an embedded AmbientTalk unit test should be run inside its owning actor, an invocation of the `run` method on the wrapped AmbientTalk unit test schedules an asynchronous request in the actor owning `exampleATTest`. However, the JUnit framework expects `run()` to be a synchronous invocation, implying that the Java thread should obviously wait for the test to be completed before executing the next test or terminating. Therefore, the Java thread suspends transparently until the AmbientTalk actor has processed the `run` method.

Synchronous invocation is enforced when a Java thread (which does not represent an actor) performs invocations on a wrapped AmbientTalk object. This is typically the case when using AmbientTalk from within a Java application, or when passing AmbientTalk objects to libraries which internally start their own threads.

## 5.3 Asynchronous Invocation

As described in section 4.2, many Java applications are themselves event-driven. We have already shown how AmbientTalk objects can engage in a proper symbiosis with event-driven frameworks without any problem. AmbientTalk objects may implement event listener interfaces whose methods are invoked purely asynchronously. In this section, we illustrate another use of asynchronous symbiotic invocations. The goal is for AmbientTalk code to be able to reuse Java's `java.util.Timer` abstraction. The `Timer` class is typically used to schedule tasks for execution at a later point in time as follows:

```
Timer timer = new Timer();
TimerTask task = new TimerTask() {
  public void run() {
    System.out.println("executing task");
  }
}
// schedule task to be executed in 5 sec
timer.schedule(task, 5000);
```

The `run` method of the `TimerTask` instance is invoked by the `timer` object after 5000 milliseconds have elapsed. In Java, this callback is executed by the thread of `timer`, which is not necessarily the thread that scheduled the task. Hence, explicit synchronisation between both threads is often necessary to prevent race conditions.

In AmbientTalk, we would like to be able to schedule code for execution at a later point in time, without causing race conditions in the actor. Unfortunately, `TimerTask` is not an interface but an abstract Java class, so it cannot be instantiated. Neither can AmbientTalk code create a concrete subclass of `TimerTask`. To make the symbiosis work, a small auxiliary class needs to be written in Java:

```
public class ATTimerTask extends TimerTask {
  public interface AsyncRunnable extends EventListener {
    public void run();
  }
  private AsyncRunnable code;
  public ATTimerTask(AsyncRunnable r) {
    code = r;
  }
  public void run() {
    code.run();
  }
}
```

The above class defines a simple wrapper around an object that understands the message `run`. By means of this auxiliary class, it is easy to wrap AmbientTalk objects in Java `TimerTask` instances:

```
def timer := jlobby.java.util.Timer.new();
def task := jlobby.at.support.ATTimerTask.new(
  object: {
    def run() { system.println("executing task") }
  });
timer.schedule(task, 5000);
```

The above code assumes that the class `at.support.ATTimerTask` is available on the JVM's class path. The thread/actor protocol mapping ensures that the `run` method of the anonymous AmbientTalk object is executed by its owning actor, *not* by the thread of the timer. Moreover, because the `AsyncRunnable` interface has been marked as an `EventListener` interface, the AmbientTalk/Java symbiosis treats the `code.run()` call in the auxiliary Java class as an asynchronous send. This ensures that the Java timer thread is *not* blocked waiting for the AmbientTalk actor to process the invocation, such that it can timely execute other scheduled tasks.

The AmbientTalk/Java symbiosis allows AmbientTalk code to use Java's `Timer` framework in exactly the same way as the framework would be used in Java, except that the multithreaded concurrency of Java is automatically adapted to the event-driven concurrency of AmbientTalk. The symbiosis layer ensures that this mapping is done transparently, without additional programming effort in AmbientTalk itself.

## 6 Implementation

This section describes the detailed implementation of the thread/actor protocol mapping introduced in section 4.2. The thread/actor mapping occurs in the Java wrapper of an AmbientTalk object. This Java wrapper is implemented by means of Java's standard support for dynamic proxies. When an AmbientTalk object is passed as an argument to a Java method requiring a parameter of an interface type, a dynamic proxy implementing that interface is generated by the symbiosis. The proxy requires an object implementing the `InvocationHandler` interface whose `invoke` method is used to intercept messages sent to the proxy. The `JavaWrapperForATObject` class implements Java wrappers for AmbientTalk objects. The essential part of the code is shown below.

```java
class JavaWrapperForATObject implements InvocationHandler {
  final ATObject principal; // the wrapped AmbientTalk object
  final EventLoop owningEventLoop; // the thread of the owner actor
  JavaWrapperForATObject(ATObject atObj) {
    principal = atObj;
    owningEventLoop = EventLoop.fromThread(Thread.currentThread());
  }
  Object invoke(Object rcv, final Method method, Object[] args) throws Throwable {
    final ATObject[] atArgs = new ATObject[args.length];
    for (int i = 0; i < atArgs.length; i++) {
      atArgs[i] = Symbiosis.javaToAmbientTalk(args[i]);
    }
    Event symbioticInvocation = new Event() {
      public Object process() throws Exception {
        ATObject result = Symbiosis.downInvocation(principal, method, atArgs);
        return Symbiosis.ambientTalkToJava(result, method.getReturnType());
      }
    }
    if (owningEventLoop.equals(Thread.currentThread())) { // immediate invocation
      return symbioticInvocation.process(); // immediately perform invocation
    } else { // detecting exclusive access violations
      if (EventLoop.isEventLoop(Thread.currentThread())) {
        throw new RuntimeException("Violated exclusive access property");
      }
      if (Symbiosis.isEvent(method)) { // asynchronous invocation
        owningEventLoop.schedule(symbioticInvocation);
        return null; // void return type
      } else { // synchronous invocation
29          Future f = new Future(); // to make the Java thread wait for the result
30          owningEventLoop.schedule(symbioticInvocation, f);
31          return f.get(); // blocks until invocation has been executed by actor
32        }
33      }
34    }
35  }
```

A `JavaWrapperForATObject` wraps an AmbientTalk object (the `principal`) and the event loop thread of the actor that owns that object (the `owningEventLoop`). The implementation assumes that the thread creating the wrapper is the event loop thread of the owning actor (line 6).

When a method is invoked on the dynamic proxy, the Java arguments to the call are mapped onto AmbientTalk values (lines 9-12). The data mapping is performed by means of the `javaToAmbientTalk` method. Subsequently, an event object is created which can be scheduled in an actor's event queue (lines 13-18). When the event is processed, it transforms the Java method invocation into an AmbientTalk invocation, according to the protocol mapping outlined in section 3.2. The AmbientTalk return value is mapped to a Java value of the appropriate return type (line 16). The rest of the code implements the thread/actor protocol mapping. It is subdivided into four parts:

*Immediate invocation* First, it is checked whether the symbiotic invocation is performed by the owning actor's own event loop thread (line 19). If this is the case, the symbiotic invocation is performed immediately using the current thread. This ensures that e.g. the callback messages described in section 5.1 operate correctly.

*Detecting exclusive access violations* By providing AmbientTalk actors with the power to access Java objects, we have introduced the problem that two or more actors may share their objects via the Java level, bypassing the exclusive access property. However, by treating Java objects as being "owned" by the thread that operates on them, we can detect when the event loop thread of one actor tries to access an object owned by another actor. In the code, it is checked whether the accessing thread is the event loop of another actor (line 22). If this is the case, an AmbientTalk actor has gained access to an object owned by another actor, and the symbiotic invocation is aborted.

*Asynchronous invocation* If the thread performing the symbiotic invocation is a regular Java thread, but the invoked method represents an event notification (i.e. it belongs to an `EventListener` interface, has a `void` return type and does not declare any thrown exceptions), the invocation is scheduled asynchronously and the thread returns immediately (lines 25-27). The `schedule` method places the event in the event queue of the owning actor. This queue is served by the actor's own event loop thread which will eventually invoke the `process` method of the scheduled `Event` object.

*Synchronous invocation* In the final case, the symbiotic invocation is performed by a regular Java thread on a non-event method (lines 29-31). In this case, the Java thread must wait until the owning actor has processed the invocation. This synchronisation is performed by means of a small auxiliary `Future` class. A future is passed to the actor's `schedule` method which will be resolved by the actor with the result of processing the scheduled `Event`. The Java thread subsequently suspends and waits for this result by invoking the future's `get` method.

**Performance Measurements** To determine the performance of symbiotic method invocations from Java to AmbientTalk, we have compared the overhead of synchronous symbiotic invocations (which require the Java thread to schedule an event and suspend on a future) with respect to immediate invocations (which allow the calling thread to immediately execute the symbiotic invocation). The results are shown below[2]. Three kinds of methods are invoked: a method with an empty body and two calculating a linear function requiring 25 resp. 50 iterations. The methods are invoked by Java code ran by the AmbientTalk actor itself (immediate invocation) and by an auxiliary Java thread (synchronous invocation).

| | Runtime (in milliseconds) of | | |
|---|---|---|---|
| | Empty Method | For-loop 25 | For-loop 50 |
| Immediate | 3.24 | 57.35 | 112.51 |
| Synchronous | 4.36 | 58.39 | 113.19 |
| Difference | 1.12 | 1.04 | 0.68 |
| Overhead | 25.69% | 1.78% | 0.60% |

The results show the expected result that synchronous invocations are slower than immediate invocations. Especially for very small methods this overhead is significant. Naturally, the overhead quickly decreases when methods take longer to execute. On average, the overhead introduced by synchronous invocations is .94 milliseconds. One important point is that all synchronous symbiotic invocations were performed on an actor whose message queue was *empty*. As the load of an actor increases, it takes more time before the synchronous invocation is served from the queue which of course drastically influences the runtime of the synchronous invocation.

## 7 Related Work

### 7.1 Linguistic Symbiosis

The reference model that we used for explaining linguistic symbiosis is that of Inter-language Reflection [9] which is itself based on an open design of object-oriented languages [12]. The purpose of these models is to clearly identify which

[2] Average runtime of 300 invocations on an AmbientTalk (version 2.5) object by a Java thread as measured on an Apple Powerbook G4 1Ghz using the HotSpot JVM v1.3.1 on Mac OS X.

objects define the boundaries between two languages (data mapping) and to identify consistent rules for crossing the boundaries (protocol mapping).

Performing a linguistic symbiosis between two languages in order to combine different programming paradigms is not novel. For example, SOUL/Smalltalk is a linguistic symbiosis between a logic programming language and an object-oriented language, where the goal is to use the logic language to *reason about* the object-oriented language [9, 13].

There exist a vast number of dynamic languages that have been implemented on top of the Java Virtual Machine and which provide a symbiotic layer to interface with Java objects. Examples include JRuby, Jython, JScheme, LuaJava and JPiccola. The data and protocol mappings of these languages is often similar in spirit, differing only in the details of e.g. how method overloading is handled. Some implementations are more advanced than others. For example, Jython allows Python classes to subclass Java classes. To the best of our knowledge, none of these linguistic symbioses define a concurrency protocol mapping, often because the two symbionts employ a similar thread-based concurrency model.

Piccola [14] is a *composition language* designed to glue together components from its underlying *host* language. Hence, Piccola requires a linguistic symbiosis with its host language in order to access and manipulate the language's components. For example, JPiccola – which uses Java as the host language – defines a so-called composition style that allows Java AWT components to be easily composed in Piccola. Piccola's concurrency control is based on the $\pi$-calculus, where agent processes communicate via channels. To the best of our knowledge, Piccola does not define a mapping between its host language's concurrency model and the agent and channel abstractions of the $\pi$-calculus, although the Piccola programmer may define such a mapping himself by defining his own composition style.

AmbientTalk's event loop concurrency model is based on that of the E language [8]. E is a language for capability-secure distributed computing in open networks. E also provides symbiotic access to Java, the language in which it is implemented. However, to the best of our knowledge, E does not provide a concurrency protocol mapping as described in this paper. The only documented feature in this regard is E's ability to allow one of its event loops to *morph into* an AWT or Swing event loop thread. In this way, all threads in the system correspond to E event loops, which effectively rules out a coexistence of the thread-based concurrency model of Java with the communicating event loops of E. This solution can be regarded as a specific instance of AmbientTalk/Java's asynchronous symbiotic invocations.

### 7.2 Unifying threads and events

The Scala language offers a concurrency library based on actors that unify threads and events [15, 16]. Scala actors are equipped with two kinds of message reception mechanisms. Thread-based actors have a `receive` operation that suspends the actor's thread until a suitable message specified in the `receive` block arrives in the actor's mailbox. When such a message finally arrives, the thread is resumed and the `receive` operation returns normally. Event-based actors have a `react` operation which, when no suitable message is available in the mailbox, does not suspend the current thread, but rather stores the actor's `react` code as a closure and signals to the executing thread that the actor is "suspended" by throwing a special exception. This exception allows the executing thread of the event-based actor to continue executing other actors. Scala's approach differs from ours in the sense that they provide one unified concurrency model, while we provide a language symbiosis that bridges the distinct concurrency models of two separate languages. Also, the actor model of Scala is more liberal than that of AmbientTalk, in the sense that the three properties of event loop concurrency are not enforced by Scala's model. It is the programmer's own responsibility to guarantee that Scala actors do not perform concurrent updates on shared state but communicate only by message passing.

Li and Zdancewic describe a concurrency model that unifies threads and events, implemented in Haskell [17]. In their approach, the system offers two views on multithreaded programs: a thread view, which allows regular application code to program in a thread-based way, and an event view, which allows programmers to manipulate threads as passive entities by means of an event-driven thread scheduler. Technically, threads are cut into a series of event handlers at points where the code performs blocking I/O operations. The difference with our approach lies again in the fact that a unified model is presented, not a symbiosis between two languages with a separate model. Also, Li and Zdancewic stress the use of their event-driven model to write custom thread scheduling code, to achieve better performance. In our approach, the event-driven model is meant to be used for actual application-level code in order to avoid race conditions and deadlocks by design in highly volatile distributed systems.

### 7.3 Futures

Our synchronisation technique between synchronous Java invocations and asynchronous AmbientTalk message sends is essentially an application of futures [18]. In concurrent object-oriented programming languages, futures have been a recurring language abstraction to reconcile asynchronous message passing with return values (e.g. future-type message passing in ABCL [6], promises in Argus [19] and E [8], wait-by-necessity in Eiffel// [20]). A future is essentially a handle for the result of an asynchronous message send. In our thread/actor protocol mapping, a synchronous Java invocation can be thought of as an asynchronous AmbientTalk message send that returns a future, on which the Java code immediately synchronises. Because of this immediate synchronisation, it has to be noted that the full power of futures is not really required: Java's standard `wait` and `notify` primitives suffice. However, in the next section we discuss how futures could help in achieving true symbiotic future-type message sends. Although our symbiosis hosts a trivial application of futures, to the best of our knowledge it has not before been used to synchronise between method invocations performed in different languages.

## 8 Research Status and Future Work

The AmbientTalk language is implemented as an interpreter on top of the Java Virtual Machine[3]. AmbientTalk is a research artifact, serving as a practical platform for experimentation, but lacking extensive performance optimisations. The interpreter runs on the Java Micro Edition platform and has been successfully deployed on PDAs connected via an ad hoc WiFi network. We are currently planning on applying the symbiosis by distributing a sequential Java application using AmbientTalk as the "distribution middleware".

With respect to the AmbientTalk/Java data mapping, one area of future work is the ability for AmbientTalk objects to "subclass" Java classes. This would allow AmbientTalk objects to function directly as Java anonymous inner classes without the need for manually written adaptor classes such as the `ATTimerTask` introduced in section 5.3, improving AmbientTalk's ability to integrate with existing Java frameworks.

With respect to the AmbientTalk/Java protocol mapping, we are planning on extending it with more elaborate support for future-type message sending. Although we have not discussed it in this paper, AmbientTalk supports asynchronous message sends that return futures. AmbientTalk's futures are based on the E language's non-blocking futures [8]: an event loop is never allowed to block, waiting for a future to become fulfilled. Instead, it is possible to register a listener on the future which is notified asynchronously when the future is fulfilled. Since version 1.5, Java also supports futures (as a library abstraction), but these are traditional futures that allow a thread to block, waiting for the future to become fulfilled. A logical next step in the symbiosis is to define a mapping between AmbientTalk's non-blocking and Java's blocking futures. This protocol mapping would fill a missing gap in the symbiosis: currently a Java invocation is either purely asynchronous or purely synchronous. If the symbiosis would support a future mapping (e.g. by detecting that the invoked Java method's return type is a subtype of `java.util.concurrent.Future`), Java invocations can be turned into future-type asynchronous AmbientTalk message sends.

## 9 Conclusions

We have described the AmbientTalk/Java linguistic symbiosis, which defines a protocol mapping between the event-driven concurrency model of AmbientTalk and the multithreaded concurrency model of Java. The motivation behind the symbiosis is that AmbientTalk can benefit from Java's vast number of existing components, while Java can benefit from AmbientTalk's high-level concurrent and distributed language features. The main problem to be tackled by the symbiosis is that Java's threading model should not violate the properties of the event-driven concurrency model (e.g. the prevention of race conditions on AmbientTalk objects).

[3] The language can be downloaded at http://prog.vub.ac.be/amop/at/download.

The contribution of this paper is a protocol mapping between actors and threads which ensures that the concurrency properties of the actor-based model are preserved. To reconcile Java's synchronous method invocations with AmbientTalk's asynchronous message sends, we have distinguished three different kinds of symbiotic invocations: immediate invocations (performed by the actor's own thread representation), synchronous invocations (which require a Java thread to wait for an event loop) and asynchronous invocations (which treats Java method invocations on listeners as pure asynchronous message sends). We have described an application of each symbiotic invocation by means of concrete examples in our AmbientTalk/Java symbiosis.

**Acknowledgments.** The authors would like to thank Kris Gybels for his suggestions for improvement, in particular the structuring of the section on the actor/thread protocol mapping, and the anonymous referees for their helpful suggestions for improvement.

## References

1. Dedecker, J., Van Cutsem, T., Mostinckx, S., D'Hondt, T., De Meuter, W.: Ambient-oriented Programming in Ambienttalk. In Thomas, D., ed.: Proceedings of the 20th European Conference on Object-oriented Programming (ECOOP). Volume 4067 of Lecture Notes in Computer Science., Springer (2006) 230–254
2. Dedecker, J., Van Belle, W.: Actors for mobile ad-hoc networks. In Yang, L., Guo, M., Gao, J., Jha, N., eds.: Embedded and Ubiquitous Computing. Volume 3207 of Lecture Notes in Computer Science., Springer-Verlag (2004) 482–494
3. Lieberman, H.: Using prototypical objects to implement shared behavior in object-oriented systems. In: Conference proceedings on Object-oriented Programming Systems, Languages and Applications, ACM Press (1986) 214–223
4. Agha, G.: Actors: a Model of Concurrent Computation in Distributed Systems. MIT Press (1986)
5. Lieberman, H.: Concurrent object-oriented programming in ACT 1. In Yonezawa, A., Tokoro, M., eds.: Object-Oriented Concurrent Programming. MIT Press (1987) 9–36
6. Yonezawa, A., Briot, J.P., Shibayama, E.: Object-oriented concurrent programming in ABCL/1. In: Conference proceedings on Object-oriented programming systems, languages and applications, ACM Press (1986) 258–268
7. Briot, J.P.: From objects to actors: study of a limited symbiosis in smalltalk-80. In: Proceedings of the 1988 ACM SIGPLAN workshop on Object-based concurrent programming, New York, NY, USA, ACM Press (1988) 69–72
8. Miller, M., Tribble, E.D., Shapiro, J.: Concurrency among strangers: Programming in E as plan coordination. In Nicola, R.D., Sangiorgi, D., eds.: Symposium on Trustworthy Global Computing. Volume 3705 of LNCS., Springer (2005) 195–229
9. Gybels, K., Wuyts, R., Ducasse, S., D'Hondt, M.: Inter-language reflection: A conceptual model and its implementation. Computer Languages, Systems & Structures **32**(2-3) (2006) 109–124
10. Ungar, D., Chambers, C., Chang, B.W., Hölzle, U.: Organizing programs without classes. Lisp Symb. Comput. **4**(3) (1991) 223–242

11. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
12. Steyaert, P.: Open Design of Object-Oriented Languages, A Foundation for Specialisable Reflective Language Frameworks. PhD thesis, Vrije Universiteit Brussel (1994)
13. Wuyts, R., Ducasse, S.: Symbiotic reflection between an object-oriented and a logic programming language. In: ECOOP 2001 International Workshop on Multi-Paradigm Programming with Object-Oriented Languages. (2001)
14. Achermann, F., Nierstrasz, O.: Applications = Components + Scripts – a tour of Piccola. Software Architectures and Component Technology (2001) 261–292
15. Haller, P., Odersky, M.: Event-based programming without inversion of control. In: Proc. Joint Modular Languages Conference. Springer LNCS (2006)
16. Haller, P., Odersky, M.: Actors that Unify Threads and Events. In: International Conference on Coordination Models and Languages. Lecture Notes in Computer Science (LNCS) (2007)
17. Li, P., Zdancewic, S.: Combining events and threads for scalable network services. In: ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI), ACM Press (2007)
18. Baker Jr., H.G., Hewitt, C.: The incremental garbage collection of processes. In: Proceedings of Symposium on AI and Programming Languages. Volume 8 of ACM Sigplan Notices. (1977) 55–59
19. Liskov, B., Shrira, L.: Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation, ACM Press (1988) 260–267
20. Caromel, D.: Towards a method of object-oriented concurrent programming. Communications of the ACM **36**(9) (1993) 90–102