Reflection Analysis and First Steps in Its Control

Iona Thomas

Université de Lille, Faculté des sciences et technologies

Département de formation doctorale en informatique **École doctorale MADIS Lille**
UFR IEEA

Reflection Analysis and First Steps in Its Control

Analyse de la Réflection et Premiers Pas Vers Son Contrôle

THÈSE

présentée et soutenue publiquement le 7 novembre

pour l'obtention du

Doctorat de l'Université de Lille

(spécialité informatique et applications)

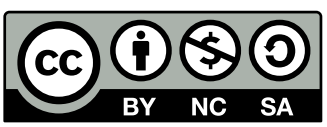
par

Iona Thomas

Composition du jury

Rapporteurs :	Christophe DONY	Professeur des universités Université de Montpellier
	Alain PLANTEC	Professeur des universités Université de Bretagne Occidentale
Examinatrice :	Mireille BLAY-FORNARINO	Professeur des universités Université de Montpellier
Directeurs de thèse :	Stéphane DUCASSE	Directeur de Recherche Univ. Lille, CNRS, Inria, CRIStAL
Co-Encadrant de thèse :	Pablo TESONE	Ingénieur de recherche Univ. Lille, CNRS, Inria, CRIStAL
Invités :	Guillermo POLITO	Chargé de recherche Univ. Lille, CNRS, Inria, CRIStAL

Centre de Recherche en Informatique, Signal et Automatique de Lille — UMR 9189
INRIA Lille - Nord Europe



Acknowledgments

Je remercie la Région Hauts-de-France pour son soutien et le financement de cette thèse.

Abstract

Reflective operations are powerful APIs (Application Programming Interfaces) that enable developers to build advanced tools and architectures. Reflective operations are used to implement tools and development environments (*e.g.* compilers, debuggers, inspectors) and language features (*e.g.* distributed systems, exceptions, proxies).

Programming languages evolve, introducing better concepts, and revising practices and APIs. For example, since 2008, Pharo has evolved from Squeak and its reflective API has evolved accordingly, diverging from the original Smalltalk reflective API. Pharo has a large set of over 500 reflective methods, often built on top of each other. They range from structural reflection to on-demand stack reification.

This thesis describes 3 main works:

- Inventory and classification of reflective operations,
- Assessing dependencies to reflection with mutation analysis,
- Protected visibility modifiers and applicability to reflective infrastructures.

Inventory and classification of reflective operations: Understanding such APIs is tedious. There is a need for a deep analysis of current reflective APIs to understand their underlying use, potential dependencies, and whether some reflective features can be scoped and made optional. Such an analysis is challenged by new metaobjects introduced into the system, such as first-class instance variables, and their mixture with the base-level API.

We propose an inventory and a classification of these operations based on their semantics. We also identify a set of issues of the current implementation and analyze their inter-dependencies. Such an analysis of reflective operations is important to support the rational evolution of the reflective layer and its potential redesign.

Assessing dependencies to reflection with mutation analysis: While reflection is a powerful tool, developers may use it to circumvent data encapsulation and method visibility modifiers. Thus, it is important to assess the extent to which an application relies on reflection. Nonetheless, reflection is mostly incompatible with static analysis as it relies on runtime information. These problems are exacerbated in dynamically-typed languages, where reflective operations are polymorphic with non-reflective ones.

To this end, we present RAPIM, an approach to assess the uses of reflective APIs: it uses mutation analysis with a new mutation operator to handle core reflective methods. We compare the performance of our approach against static analysis on a selection of 5 projects. On four out of five projects, RAPIM disambiguates

more potentially reflective call-sites than static analysis. When code coverage is high, the percentage of disambiguation is three times higher. Finally, we question the relevance of polymorphism between non-reflective and reflective APIs.

Protected visibility modifiers and applicability to reflective infrastructures: In our exploration of means to control reflection, we examined method visibility to separate internal methods from the public API. Protected visibility modifiers provide a way to hide methods from external objects while authorizing internal use and overriding in subclasses. While present in main statically-typed languages, visibility modifiers are not as common or mature in dynamically-typed languages. In Pharo, all methods are public. While only a few reflective methods could be protected due to their uses, many reflective metaobjects could benefit from having a protected modifier for internal methods.

We present PROTDYN, a visibility model for dynamically-typed languages that is computed at compile time. It relies on name-mangling and syntactic differentiation between self vs non-self-sends. Its Pharo implementation is backward compatible with existing programs. We study its performance, memory footprint, and compatibility with existing optimizations to demonstrate its viability.

While this thesis focuses on Pharo, the results can be applied to many dynamically-typed languages.

Keywords: Reflection, Classification, Protected modifier, Mutation analysis, Feature dependency

Résumé

Les opérations réflexives sont des API (interfaces de programmation d'applications) qui permettent aux developpeur·euse·s de construire des outils avancés et de modifier l'architecture d'un langage. Elles permettent de créer des outils pour les environnements de développement (compilateurs, débogueurs, inspecteurs) ou de concevoir de nouvelles fonctionnalités dans un langage (systèmes distribués, exceptions, proxies).

Les langages de programmation évoluent, introduisant de meilleurs concepts et mettant à jour les pratiques et les API. À l'origine proche de Squeak, Pharo a évolué depuis 2008 et ses API réflexives ont divergé de celles d'origine de Smalltalk. Avec plus de 500 méthodes réflexives identifiées, Pharo possède un large panel de fonctionnalités réflexives.

Cette thèse présente trois travaux principaux :

- Un inventaire et une classification des opérations réflexives;
- Analyser les dépendances à la réflexion;
- Les modificateurs de visibilité et leur applicabilité aux infrastructures réflexives;

Inventaire et classification : Il est nécessaire d'analyser les API réflexives actuelles afin de comprendre leurs usages, leurs dépendances et si certaines pourraient être limitées ou optionnelles. Cette analyse est rendue complexe par l'ajout de nouveaux métaobjets dans Pharo, tels que les variables d'instance de première classe et par leur mélange avec les API non-réflexives.

Nous proposons un inventaire et une classification sémantique de ces opérations réflexives. Nous analysons leurs interdépendances et identifions un ensemble de problèmes liés à l'implémentation actuelle. Cette analyse pourra informer les prochaines évolutions de la couche réflexive.

Analyser la dépendance à la réflexion: La réflexion est un outil puissant qui peut contourner l'encapsulation des données et les modificateurs de visibilité. Il est donc important de pouvoir évaluer dans quelle mesure une application repose sur la réflexion. Puisque la réflexion dépend d'informations dynamiques, l'analyse statique est limitée. Ces problèmes sont accrus dans les langages dynamiquement typés en raison du polymorphisme.

Nous proposons donc RAPIM, une approche pour évaluer l'utilisation de la réflexion. Celle-ci utilise l'analyse de mutation avec un nouvel opérateur de mutation pour traiter les méthodes réflexives du noyau. Nous comparons les performances de notre approche à celles de l'analyse statique sur une sélection de cinq projets.

Pour quatre projets, RAPIM désambiguïse plus d'appels potentiellement réflexifs que l'analyse statique. Lorsque la couverture du code est élevée, le pourcentage de désambiguïsassions est trois fois plus élevé. Enfin, nous questionnons la pertinence du polymorphisme entre les API réflexives et non-réflexives.

Modificateurs de visibilité: Dans notre exploration des moyens de contrôler la réflexion, nous avons examiné la visibilité des méthodes pour séparer les méthodes internes de l'API publique. Les modificateurs de visibilité (*protected*) permettent de cacher les méthodes aux objets externes tout en autorisant l'utilisation interne et la surcharge dans les sous-classes. Bien qu'ils soient présents dans les principaux langages statiquement typés, ce type de modificateur n'est pas aussi courant ou mature dans les langages dynamiquement typés. En Pharo, toutes les méthodes sont publiques. Bien que peu de méthodes réflexives puissent être protégées en raison de leur utilisation, de nombreux métaobjets réflexifs bénéficieraient d'un modificateur pour les méthodes internes. Nous présentons PROTDYN, un modèle de visibilité pour les langages dynamiques calculé à la compilation et s'appuyant sur la distinction syntaxique de l'usage (ou non) de *self*. Son implémentation Pharo est rétro-compatible. Nous évaluons ses performances, son utilisation de la mémoire et sa compatibilité avec les optimisations existantes.

Bien que cette thèse se concentre sur Pharo, les résultats peuvent être appliqués à d'autres langages dynamiquement typés.

Mots-clés: Réflexion, Classification, Modificateur de visibilité, Analyse de mutation, Dépendances de fonctionnalités.

Contents

1 Introduction **1**
- 1.1 The Need for Reflective Behavior . . . 3
- 1.2 Issues Raised by Reflective Operations . . . 4
- 1.3 The Need for an Up-to-date Reflective Feature Classification . . . 5
- 1.4 Understanding the Uses of Reflection . . . 6
- 1.5 The Challenges of Reflection Analysis . . . 8
- 1.6 Terminology . . . 9
  - 1.6.1 Message Passing Terminology . . . 9
  - 1.6.2 Visibility Terminology . . . 10
  - 1.6.3 Call-Site Terminology . . . 10
- 1.7 Contributions . . . 11
- 1.8 Structure of the Thesis . . . 12
- 1.9 Publications . . . 13

I Analysing Reflection and Its Uses **15**

2 Classification Of Reflective Operations **17**
- 2.1 Metaobjects, Classes and Their Related APIs . . . 18
  - 2.1.1 Pharo Structural Meta-Model . . . 18
  - 2.1.2 Overview of the Reflective APIs . . . 19
- 2.2 A Classification and Analysis of Runtime Reflective Operations . 20
  - 2.2.1 Methodology . . . 22
  - 2.2.2 Object Inspection Reflective Operations . . . 22
  - 2.2.3 Object Modification Reflective Operations . . . 24
  - 2.2.4 Class Structural Inspection Reflective Operations . . . 24
  - 2.2.5 Class Structural Modification Reflective Operations . . . 26
  - 2.2.6 Method Creation Reflective Operations . . . 27
  - 2.2.7 Structural Queries on Methods Reflective Operations . . . 27
  - 2.2.8 Message Sending and Code Execution Reflective Operations 28
  - 2.2.9 Chasing and Atomic Pointers Swapping Reflective Operations 29
  - 2.2.10 Memory Scanning Reflective Operations . . . 30
  - 2.2.11 Stack Manipulation Reflective Operations . . . 31
- 2.3 Discussions . . . 31
  - 2.3.1 Threats to Validity . . . 32
  - 2.3.2 General Concerns . . . 32
- 2.4 Conclusion . . . 35

3 Analyzing Dependencies Between Reflective APIs 37
3.1 Methodology . . . 38
3.2 Reflective APIs Categories Interdependencies . . . 39
3.2.1 Interdependencies Overview . . . 39
3.2.2 Isolated Reflective Categories . . . 42
3.2.3 Provider Categories . . . 43
3.2.4 Client Categories . . . 43
3.2.5 Iterating and Querying Hierarchy, a Hub Category . . . 44
3.3 Visualising Reflective Operations Intra-category Dependencies and Their Layered Organization . . . 45
3.3.1 Chasing and Swapping Pointers - Find Pointers to . . . 45
3.3.2 Memory Scanning - Instances of a Class . . . 46
3.3.3 Message Sending and Code Execution - Runtime and Evaluation & Message sending and Code Execution - Reflective Message Send . . . 47
3.4 Threats to Validity . . . 48
3.5 Conclusion . . . 49

4 Mutation Analysis for reflection 51
4.1 RAPIM: Reflective API Mutation Analysis . . . 52
4.1.1 Reflective Mutation Analysis Approach . . . 53
4.1.2 Reflective Method Cancelling Operator . . . 53
4.2 RAPIM by Example: a Developer Perspective . . . 54
4.3 Evaluation of Reflective API Identification . . . 59
4.3.1 Chosen Projects . . . 59
4.3.2 Answering RQ1. MUTATION COMPARISON . . . 60
4.3.3 Answering RQ2. EFFECTIVE POLYMORPHISM . . . 62
4.4 Discussion . . . 62
4.4.1 Limitations . . . 62
4.4.2 Threats to Validity . . . 63
4.5 Related Work for Mutation Analysis . . . 63
4.5.1 Overcoming the Limits of Static Analyses . . . 64
4.5.2 Program Analysis and Validation via Mutation Testing . . . 64
4.6 Conclusion . . . 65

II First Steps in Control for Reflection 67

5 State of the Art for Controlling Reflection and Visibility Modifiers 69
5.1 Controlling Reflection . . . 70
5.1.1 Mirror Architecture . . . 70
5.1.2 Ownership-Based Control . . . 71
5.1.3 Secure MOP . . . 71

5.1.4 Optional Reflective Features . . . 72
5.1.5 First-Class References . . . 72
5.1.6 Reflectogram Reification . . . 72
5.1.7 External Reflection . . . 73
5.2 Existing Visibility Modifiers . . . 73
5.2.1 Ruby . . . 74
5.2.2 PHP . . . 75
5.2.3 Python . . . 75
5.2.4 Javascript . . . 75
5.2.5 Statically-Typed Languages: Java, C#, and C++ . . . 76
5.3 Object Encapsulation . . . 76
5.3.1 ConstrainedJava . . . 76
5.3.2 MUST . . . 76
5.3.3 Encapsulation Policies and Visibility Semantics . . . 76
5.4 Conclusion . . . 78

6 Protected Modifier **79**
6.1 PROTDYN: A Protected Modifier Model . . . 80
6.1.1 Properties and Chosen Semantics for a PROTDYN . . . 80
6.1.2 Object-Send and Self-Send Lookup Semantics by Example 81
6.1.3 Changing Visibility in Subclasses . . . 84
6.1.4 PROTECTEDLITE: PROTDYN Semantics . . . 85
6.2 #PHARO Implementation . . . 87
6.2.1 Design Principles . . . 88
6.2.2 Implementation Overview . . . 88
6.2.3 Double Public Registration . . . 89
6.2.4 Selector Mangling for Self-Sends Sites . . . 89
6.2.5 Preventing Selector Mangling Propagation to the Whole System . . . 90
6.3 Discussing Alternative Implementations . . . 91
6.3.1 Lookup Mechanism Modification . . . 91
6.3.2 Run-Time Visibility Checks . . . 92
6.4 Conclusion . . . 92

7 Protected Pharo Evaluation **93**
7.1 Performance Evaluation . . . 94
7.1.1 Experimental Design . . . 94
7.1.2 Methodology and Setup . . . 95
7.1.3 Selected Benchmarks . . . 96
7.1.4 Scenario 1: Lookup Performance . . . 97
7.1.5 Scenario 2: Global Lookup Cache Performance . . . 98
7.1.6 Scenario 3: Lookup Cache Behaviour . . . 99
7.1.7 Scenario 4: Polymorphic Inline Cache PIC Performance . 101

7.2 Memory Use Analysis of #PHARO 102
7.2.1 Methodology and Setup 102
7.2.2 Memory Cost: Results 103
7.3 Applicability to Reflection 104
7.3.1 Applicability to Reflective Methods 104
7.3.2 The Case of doesNotUnderstand: 105
7.3.3 Applicability to Internal Methods of Reflective Classes . . 108
7.4 Conclusion 109

8 Conlusion **111**
8.1 Contributions 111
8.2 Future Work 112

A Exhaustive classification of runtime reflective operations in Pharo **115**
A.1 Object Inspection 115
A.2 Object Modification 116
A.3 Class Structural Inspection 117
A.4 Class Structural Modification 123
A.5 Method Creation 127
A.6 Structural Queries on Methods 127
A.7 Message Sending and Code Execution 128
A.8 Chasing and Atomic Pointer Swapping 131
A.9 Memory Scanning 131
A.10 Stack Manipulation 132

B SMALLTALKLITE Formal Semantics **135**
B.1 SMALLTALKLITE Reduction Semantics 135

C Numerical Results for Benchmarks **141**

List of Figures

1.1 Call-sites terminology. 11

2.1 The structural Pharo metamodel: Class aggregates variables, methods, constant management (SharedPools) and method annotation (Pragma) and exposes related APIs. 19
2.2 Metaobjects controlling the reflective APIs of Pharo. 20

3.1 Reflective category dependency graph. The size of the circle corresponds to the number of selectors in the category. Line thickness depends on the number of dependent selectors. For more details see Figure 3.2. 39
3.2 Matrix of dependencies between categories. The category in row X depends on the category in Column Y if there is a number at the intersection. The number corresponds to the number of different selectors depending on the other category. 40
3.3 Categories according to the number of incoming and outgoing dependencies. The size of each bubble and the number inside of it correspond to the number of reflective categories matching that number of outgoing (x-axis) and incoming (y-axis) dependencies. 41
3.4 Subgraph of Chasing and swapping pointers - Find pointers to. The black selectors belong to the studied category. 46
3.5 Subgraph of Memory Scanning - Instances of a class. Black selectors belong to the studied category. 47
3.6 Subgraph of ***Message sending and code execution - Runtime and Evaluation*** and ***Message sending and code execution - Reflective message send***. 48

4.1 RAPIM: Using mutation analysis to assess reflection usage. For each reflective method removed, all tests are run and results are collected. 52
4.2 Code transformation for *Reflective method canceling*. 54
4.3 Proportion of call-sites types by selector for STON. 55
4.4 Table of the percentage of tests in a given class depending on a reflective method. 57
4.5 Table of the percentage of tests in a given class depending on a reflective category. 57
4.6 Table of Tests depending on a reflective method. A 1 (blue cell) means that the test has failed when this reflective method is removed. 58

6.1 Message sending is modified to distinguish between object-sends and self-sends: only object-sends can invoke protected methods which can also be overridden and taken into account by default method lookup. 82
6.2 Overriding a public method by a protected one leads to buggy situations where self-sends and object-sends can yield two different results. (a) superclass method is used, (b) asymmetrical results. . . 85
6.3 Protected Pharo syntax. 85
6.4 Relations and predicates for PROTECTEDLITE. 86
6.5 Message passing reductions for PROTECTEDLITE. 86
6.6 Self and super-send call site renaming (also called selector mangling). Method names of object-sends are not renamed to protected method name mangling, while self and super sends are. 87
6.7 From the model to the actual implementation: Selector mangling and public method double addition to their class method dictionary. 89
6.8 Limiting the propagation of recompilation with selector mangling for protected to the top of the hierarchy. 90
6.9 Compiling a method with an undefined selector assumes a public message. 91

7.1 Relative run time performances with global cache disabled on Microdown, Deltablue, and Richards benchmarks. Lower is better. A red dot marks the average. 97
7.2 Relative run time performances on the VM without global cache for the Smark Compiler benchmark. Lower is better. The red dot marks the average. 97
7.3 Relative run time performances on the VM with global cache only for Microdown, Smark Deltablue, and Smark Richards benchmarks. Lower is better. A red dot is the average. 98
7.4 Results with lookup cache enabled on the Compiler benchmark. Lower is better. Red dots mark the average. 99
7.5 Cache hits and misses. 100
7.6 Results on the VM with JIT and PICs enabled for Microdown, Deltablue, and Richards. Lower is better. A red dot is the average. 101
7.7 Results on the default VM for the Compiler benchmark. Lower is better. A red dot is the average. 102

B.1 Redex syntax. 136
B.2 SMALLTALKLITE syntax. 136
B.3 Translating expressions to redexes. 137
B.4 Relations and predicates for SMALLTALKLITE. 138
B.5 Reductions for SMALLTALKLITE. 138
B.6 Variable substitution. 139

List of Tables

2.1 Overview of the reflective categories and APIs alphabetically sorted. Leading letters are used for Figure 3.1. 21

4.1 STON's *potentially reflective call-sites* classification with RAPIM. 55
4.2 Links to project repositories used for this analysis. 59
4.3 Projects statistics and their *potentially reflective call-site* classification with RAPIM. 60
4.4 Number of *potentially reflective call-sites* by projects, split by coverage and reflective ambiguity according to static analysis. 61

5.1 Accessibility of methods according to visibility modifiers in different languages. N/A means "Not Applicable" 74

7.1 Microdown memory use with and without #PHARO. Absolute numbers in bytes. Results relative to the baseline are in parentheses. 103
7.2 Table of reflective methods selectors that could be protected according to static analysis of the Pharo standard library/base image. . . 105
7.3 Number of protectable methods in reflective classes' private protocol. 108

C.1 Average run times and standard interval with confidence of 95 % for the different benchmarks: without using the protected modifier library, with the protected modifier library loaded, and with the protected modifier library used. 141

CHAPTER 1

Introduction

Contents

1.1 The Need for Reflective Behavior 3
1.2 Issues Raised by Reflective Operations 4
1.3 The Need for an Up-to-date Reflective Feature Classification . . 5
1.4 Understanding the Uses of Reflection 6
1.5 The Challenges of Reflection Analysis 8
1.6 Terminology 9
1.6.1 Message Passing Terminology 9
1.6.2 Visibility Terminology 10
1.6.3 Call-Site Terminology 10
1.7 Contributions 11
1.8 Structure of the Thesis 12
1.9 Publications 13

> *Reflection is the ability of a program to manipulate as data something representing the state of the program during its execution.*
> [**?**]

Reflective operations are powerful APIs(Application Programming Interfaces) that allow a program to manipulate itself (and its programming language) during its execution. This includes [**?**, **?**]:

- *Introspection*, the ability to examine its own structure and state,
- *Self-modification*, the ability to change itself,
- *Intercession*, the ability to alter the semantics of its programming language.

Reflective operations let developers build advanced tools or architectures that otherwise would have to be implemented in language implementation engines, would require complex infrastructure, or may simply not be possible. These reflective features support the implementation of tools (*e.g.* compilers, debuggers, inspectors),

frameworks and libraries (*e.g.* serialization, persistence, logging), and language infrastructure (*e.g.* exceptions, distributed systems, continuations, green threads). Such a set of tools and frameworks are both used during the development and deployment of applications.

Giving too much power to developers is, however, also a burden. Reflective features defeat static analysis [**?**] and are usable as security exploits. For example, they allow malicious users to violate encapsulation or execute methods that were not intended to be executed [**?**, **?**, **?**, **?**]. They can also break invariants (*e.g.* disabling features of the language or breaking its safety), cause maintenance issues (*e.g.* higher complexity level), performance issues (*e.g.* costly reifications), and negatively impact the stability of the system (*e.g.* manipulating runtime state).

Reflective APIs evolve with their languages. Since 2008 the Pharo programming language continuously evolved: new concepts were added (slots, packages, pragmas...). There is a need for a deep, up-to-date, analysis of available reflective features and their uses. However, reflection is particularly complicated to analyze. Livshits *et al.* claim that

> *Reflection has always been a thorn in the side of Java static analysis tools.* [**?**].

The difficulty for static analysis grows in the case of deeply reflective languages such as Smalltalk descendants[Gold83a]. Certain reflective operations rely on runtime information. Pharo, for example, as a descendant of Smalltalk has advanced reflective operations such as bulk pointer swapping [**?**], on-demand stack reification [**?**], and first-class resumable exceptions. In addition, in Smalltalk-80 and many of its derivatives, reflective facilities are mixed with the base-level API of objects and classes [**?**, **?**, **?**]. They are a key part of the kernel of the language and libraries.

In this thesis, we propose an inventory of Pharo's reflective operations and a classification based on their semantics. This inventory and categorizations are then leveraged to study both dependencies between reflective methods and reflection between categories. This is a first step in understanding how the reflective API could be modularized. We also propose to use mutation analysis to study dependencies from libraries and applications to reflective APIs. Using mutation analysis allows us to get runtime information and we show that with a high enough coverage, we outperform static analysis in classifying potentially reflective call-sites.

Then, we focus on control strategies. We present a simple state of the art about the existing strategies to restrict strategies for reflection, the existing visibility modifiers, and object encapsulation. Note that none of these were adopted into an existing mainstream language. We propose a backward-compatible protected visibility modifier model for dynamically-typed language and its implementation for Pharo. We evaluate its runtime performance, and memory footprint, and discuss its applicability to Pharo's reflective infrastructure.

1.1 The Need for Reflective Behavior

Reflective features in object-oriented languages are central to the development of advanced behavior ranging from enhanced development tools to new paradigm implementation such as Aspect-Oriented Programming [**?**]. In the middle of the 90s, reflection was heavily explored: structural [**?**, **?**], computational [**?**, **?**], message-based [**?**, **?**], compile-time [**?**] and partial reflection [**?**, **?**].

Reflection is an important tool that enables many important features of modern languages [**?**]. For example, message-passing control is one of the cornerstones of a broad range of applications and an important feature of reflective systems. Applications that use message-passing control are roughly sorted into three main categories.

- The first category is *application analysis and introspection* that are based on tools that display interaction diagrams, class affinity graphs, and graphic traces [**?**, **?**, **?**, **?**].

- The second category is *language extension*. In such a case, message passing control allows one to define new features from within the language itself: Garf [**?**], Distributed Smalltalk [**?**], or [**?**] transparently introduce object distribution. Language features such as multiple inheritance [**?**], backtracking facilities [**?**], and instance-based programming [**?**, **?**] have been introduced. Futures [**?**, **?**] or atomic messages [**?**, **?**] are also based on message-passing control capabilities.

- The third category is the *definition of new object models*, introducing concurrent aspects such as active objects (Actalk [**?**]) and synchronization between asynchronous messages (Concurrent Smalltalk[1] [**?**]). Other work proposes new object reflective models such as CodA which is a meta-object protocol that controls all the activities of distributed objects [**?**], meta helix [**?**] or submethod reflection using Abstract Syntax Tree (AST) annotation [**?**, **?**].

More elaborate schemes have been proposed (*e.g. partial behavioral reflection* [**?**, **?**]) that provide a more flexible and fine-grained way to specify both the location being reflected and the metaobject invoked. Context-oriented [**?**] or aspect-oriented programming implementations are often based on reflection [**?**, **?**].

Chari *et al.* [**?**] argue that reflective capabilities increase the maintainability and evolvability of running systems. Often virtual machine implementations impose restrictions on the changes that are possible. They state that execution semantics and memory management at runtime are not supported in mainstream VMs. They support the idea of a VM with its own observability and modifiability at runtime.

[1] Concurrent Smalltalk is based on the extension of the virtual machine and new byte-code definition. However, the synchronization of asynchronous messages uses the doesNotUnderstand: technique.

The importance and need for reflective features are also illustrated by the effort to offer them in more static languages such as C++ [**?**], Ada [**?**], and Java [**?**, **?**, **?**, **?**].

1.2 Issues Raised by Reflective Operations

In the previous section, we have seen that reflective operations allow developers to implement powerful tools and frameworks. This is done by directly accessing and manipulating the state and meta-state of the application.

These direct accesses and manipulations, however, impose issues on the application: security concerns (*e.g.* leaking of information, arbitrary code execution), invariance breaking (*e.g.* disabling features of the language or breaking its safety), maintenance issues (*e.g.* higher complexity level), performance issues (*e.g.* costly reifications), and possible impact on the stability of the system (*e.g.* manipulating runtime state).

Breaking state encapsulation. State encapsulation is a powerful concept highly used in object-oriented programming. Direct access to the internal state of objects not only breaks the original design and intention of the programmer but it might be used to access information that is not intended to be accessed. It produces a leak of information, as having a reference to an object allows attackers to get the state of all objects in the accessible graph. Accessing instance variables directly limits the possibility of building systems with restricted information access.

Changing the behavior or shape of live instances. Reflective operations on meta-objects might modify the behavior of a running application. Modifying the behavior of a running application might affect its stability. For example, by using reflective operations, it is possible to modify a class structure (*e.g.* removing instance variables) or to modify the methods installed in a class. By doing so, it is possible to install invalid methods or affect the structure of live instances producing errors and possible invalid calls that crash the application.

Breaking proxies and membranes. Transparent proxies [**?**] and membranes [**?**] are protection mechanisms allowing one to control access to an object or a group of objects. They require that proxies be used as normal objects. Membranes are used to extend the functionality of the language, for example, to restrict visibility or implement capability models. Moreover, they require that the user of a given object is not able to differentiate if the referenced object is the real object or a proxy to it. However, reflective operations allow one to know if the receiver is a proxy or not, but they also allow the attackers to access the real objects by breaking state encapsulation and traversing the memory.

Arbitrary code execution. Reflective operations allow attackers to execute arbitrary methods on instances. Attackers might execute a primitive or a method that is not allowed in a given instance. These operations bypass the lookup mechanism and there is no way of preventing the execution of a given method. This issue might be used to access instance state, break proxies or membranes, or affect the stability of the system.

Memory scanning. Forging pointers is the operation of creating a reference to an object that one has not and should not have. By using reflective operations, it is possible to obtain references to existing instances not initially known by the attacker. For example, it is possible to obtain references by traversing the stack, accessing all instances of a class, or accessing the internal state of objects.

Impact on maintenance. Reflective operations allow developers to extend the execution of the language and runtime. However, the presence of reflective operations may lead developers to overuse them instead of doing a clean object-oriented design. For example, a developer will be tempted to use dynamic class change instead of using a simple State design pattern. By overusing reflective operations the complexity of the application is higher. This impacts directly their maintenance.

Impact on performance. Extending the language can lead to a slower overall experience. Using reflective operations when base-level operation would be sufficient can also be slower, especially if it requires some reification. Stack reification to access the execution context is an example of a costly operation.

Burdening polymorphism. Two objects of different types sharing a common interface are polymorphic. Polymorphism allows these objects to be used indiscriminately by an application relying only on the common interface. Some reflective operations, like checking for the identity of the object, or its membership to a class or class hierarchy, break the ability to use an instance of another class if needed. This why in Pharo, `respondsTo:` and `species` should be preferred to checks such as `isMemberOf:`.

1.3 The Need for an Up-to-date Reflective Feature Classification

Given the possibilities they offer and the issues they can cause, understanding which reflective operations are available is crucial.

Rivard's analysis. Back in 1996, Rivard [**?**] proposed the first classification of Smalltalk reflective features. Such a classification is, however, old, and includes aspects such as the compiler which are orthogonal to runtime reflective features. In addition, it is based on VisualWorks a proprietary Smalltalk that is not easily accessible nowadays.

More than 25 years of evolution. Between 1996 and 2008, Squeak evolved from the original Smalltalk reflective API with many contributions. In 2008, Pharo was born from Squeak. Pharo on its turn saw many different contributions. To give an idea of the activity in Pharo, since 2019 and the versioning of Pharo on GitHub, Pharo has around 100 yearly contributors (with up to 30 regular ones). As of the writing of this thesis, its commit history counting only since 2019 contains over 20,000 commits.

New reflective features. Finally, Rivard's analysis does not take into account traits [**?**, **?**, **?**, **?**], first-class instance variables, and the introduction of new tools using reflection such as the new inspector framework [**?**], reflectivity [**?**], object-centric debugging [**?**], error handling infrastructure [**?**], and on the fly deprecated message rewritings [**?**] to name a few.

Other limited analysis. Callau *et al.* [**?**] studied the use of dynamic features of programming languages and used Pharo as a case study. Their study is limited and focuses on the use of a limited set of elements. They do not embrace the full reflective APIs. Demers and Malenfant proposed to compare reflective capabilities in logic, functional, and object-oriented programming [**?**], but it is not related to a concrete Pharo implementation.

This shows the need for a deep and up-to-date analysis that embraces the full spectrum of available reflective features.

1.4 Understanding the Uses of Reflection

From a language evolution perspective. We have seen in Section 1.2 that reflective operations and their widespread availability cause issues. Although powerful, reflective features are usable for example in security exploits. For example, they allow malicious users to violate encapsulation and execute methods that were not intended to be executed [**?**, **?**, **?**].

The challenge is now how to improve the modularity and security of the language core, without affecting the features used by tools, frameworks, and libraries. For example, serialization libraries highly use reflective operations to serialize and deserialize objects generically; removing those operations to improve the security of the application impedes the use of such a library.

Packaging away reflective operations or restricting their access requires, however, a complete understanding of their usages and analysis to see if they can be separated from the language kernel and core libraries. Ideally, we would want sensitive reflective operations to be only loaded on demand. The first step is to understand which reflective operations are used, where, and how.

From a developer perspective. From the perspective of deploying an application with less reflective features, developers need to be able to assess how much their application relies on reflection. It is important to understand if a reflective functionality is central to an application or if it is only confined to peripheral parts (*e.g.* tests that are not run in production). However, developers should not need to analyze the code manually.

The need for an automated analysis. An automated analysis should be able to assess how much an application or library relies on reflective operations, and which ones. When performing an analysis, developers and language maintainers should not get drowned in a sea of information. Information should be presented with different levels of granularity. Here are more precise questions to characterize a reflective feature usage in an application or library:

- *General use.* How much does a project rely on reflection in general? The first general objective is to understand if an application uses or not reflective features.

- *Faceted analysis.* How much does it rely on specific reflective features? There are different families of reflective features: simple introspection, encapsulation violation, memory scanning, and more [**?**]. Each of such families has different consequences on security issues. This is why this is important to propose a faceted analysis.

- *Spatial distribution.* Which application/library parts are impacted by a given reflective use? During the assessment developers need to understand the spread of a reflective use.

- *Degraded modes.* How much of the application/library still works without a given reflective use?

- *Coupled uses.* Are there some reflective features that are often used together?

This information can be then used to inform future language or application evolutions. However, getting this information is not as simple as it sounds.

1.5 The Challenges of Reflection Analysis

> *Reflection has always been a thorn in the side of Java static analysis tools. Without a full treatment of reflection, static analysis tools are both incomplete because some parts of the program may not be included in the application call graph, and unsound because the static analysis does not take into account reflective features of Java that allow writes to object fields and method invocations. However, accurately analyzing reflection has always been difficult, leading to most static analysis tools treating reflection in an unsound manner or just ignoring it entirely. This is unsatisfactory as many modern Java applications make significant use of reflection [?].*

While the quote above is about Java, this tension is exacerbated in the case of deeply reflective languages such as Smalltalk descendants. Pharo, for example, as a descendant of Smalltalk is the essence of a reflective language. It offers advanced reflective operations such as bulk pointer swapping [**?**], on-demand stack reification [**?**], and first-class resumable exceptions.

Static analysis is therefore complicated by the following facts:

Reflective operations are mixed with non-reflective ones. In Smalltalk and many of its derivatives, reflective facilities are mixed with the non-reflective API of objects and classes [**?**, **?**, **?**]. There are no separate classes or packages that regroup them. They are a key part of the kernel of the language and standard libraries.

Dynamic decisions are a blind spot of static analysis. Reflective features defeat static analysis because the actual attributes/methods that are being used are decided and even crafted at runtime [**?**, **?**, **?**, **?**, **?**]. It is thus impossible to precisely know using static analyses what methods will be called by a reflective method invocation, or what fields will be read/written using a reflective field access.

Some reflective and non-reflective methods are polymorphic. Moreover, such reflective dependency analysis is even more difficult in dynamically-typed languages where reflective APIs are often designed as normal methods that are polymorphic with non-reflective operations. As already mentioned, in Javascript and Python reflectively accessing an attribute (`object['attribute']`) is syntactically equivalent to array/dictionary accesses (`array[index]`/`dictionary[key]`). In Pharo, 44% of the reflective method's selectors have also non-reflective implementors.

This makes reflective operations look like any other message. Reflective features are handy but they should not be used by accident, and should not be mistaken for regular non-reflective messages. While solutions such as mirrors [**?**] offer a way to separate the base level from the meta-level, it requires the full redesign of some

core functionalities of a language. This does not solve the problems of developers of existing languages.

1.6 Terminology

This section provides definitions for some vocabulary that is used in the rest of the thesis. These definitions are grouped by corresponding chapters. They are ordered so that later definitions can build on the ones defined above.

1.6.1 Message Passing Terminology

Because through this thesis we focus on dynamically-typed languages, we introduce some key vocabulary taken from the Smalltalk terminology regarding message passing.

Definition 1. Message. Messages are the key operations in object-oriented languages, also known in other languages as method invocations. A message is composed of a receiver, a selector, and zero or more arguments.

Definition 2. Message receiver. The message receiver is the object targeted by the message.

Definition 3. Message selector. The message selector is an identifier used to choose what method is executed. In dynamically-typed languages, a selector is used as a method signature. In statically-typed languages, the method signature contains a selector and also the types of its arguments and return value.

Definition 4. Method lookup. The method lookup is the process of searching for the right method to execute from a given message. In single-dispatched dynamically-typed object-oriented languages, the method lookup is a function on the receiver object and the selector. It looks up in the receiver's hierarchy the method whose signature is equal to the selector.

Definition 5. Method activation. A method activation is the execution of a method, triggered by a message-send. To activate a method, first, the method-lookup finds the corresponding method, then the method is executed on the message receiver and arguments.

Definition 6. Current method receiver. The current method receiver is the receiver object that led to the current method activation. It is usually denoted with special keywords or pseudo-variables named self or this.

Definition 7. Message-send site. A message send site is a location in the code where a message is sent, also known in other languages as a call site.

1.6.2 Visibility Terminology

The following definitions focus on visibility and types of message-send sites and are based on the previous fine-grained distinction between message, method activation, and message-send site. These definitions are mostly used in Chapters 5, 6, and 7.

Definition 8. Method visibility. A method is visible from a message-send site if the method lookup finds this method in the receiver's hierarchy. We say a send-site *can see* a method if the method is visible to the send-site.

Definition 9. Visibility semantics. The visibility semantics of a programming language are the rules that decide which methods are visible from which message-send sites.

Definition 10. Visibility mechanism. The visibility mechanism of a programming language is the technique used to guarantee that the language visibility semantics are not violated.

Definition 11. self-send site or self-send. A self-send site (self-send for short from now on) is a message-send site where we can syntactically identify the receiver as the current method receiver (*i.e.* self or this). Unless they are specifically mentioned, we consider super-send sites as self-send sites.

Definition 12. object-send site or object-send. An object-send site (object-send for short from now on) is a message-send site where we cannot syntactically identify the receiver as the current method receiver (*i.e.* it is not self or this).

Definition 13. Protected and public methods. A protected method is a method explicitly annotated by a developer as *protected*. A public method is a non-protected method.

Definition 14. Protectable method A protectable method is a method that could be annotated as *protected* without breaking the current behavior.

1.6.3 Call-Site Terminology

The following definitions focus on call-sites and whether or not they call reflective methods. As defined in the previous section message-send site and call-site are equivalents. These definitions are mostly used in Chapter 4 which focuses on understanding how much an application depends on reflection. Figure 1.1 is supporting those definitions.

Definition 15. Potentially reflective call-site. A call-site is potentially reflective if it could directly call a reflective method (*i.e.* not in a transitive manner). We consider that a call-site is potentially reflective if there is at least one reflective method implementing the call-site signature. For example, at:put: has at least one reflective implementor, therefore all at:put: call-sites are potentially reflective.

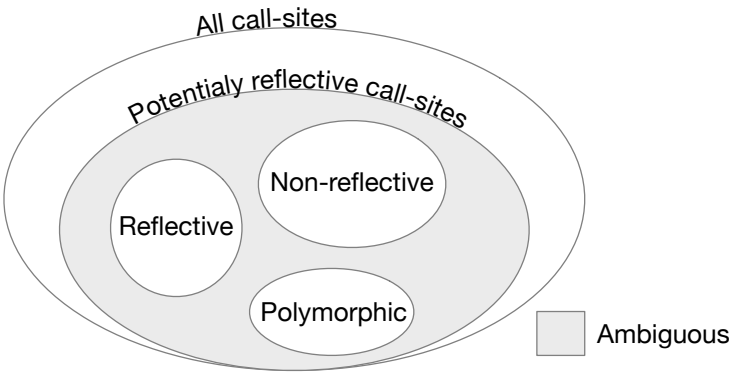


Figure 1.1: Call-sites terminology.

Definition 16. Reflective call-site. A *potentially reflective call-site* that only calls reflective methods.

Definition 17. Non-Reflective call-site. A *potentially reflective call-site* that only calls non-reflective methods.

Definition 18. Polymorphic call-site. A *potentially reflective call-site* that calls both reflective and non-reflective methods.

Definition 19. Ambiguous call-sites. A set of *potentially reflective call-sites* that we cannot identify as reflective, non-reflective or polymorphic.

1.7 Contributions

This section summarises the contributions of this thesis. We first focus on understanding available reflective features and their uses. Then we present the first steps in controlling reflection.

Analysing reflection and its uses.

- an up-to-date catalog of the reflective features in Pharo. We identified over 500 reflective methods.
- a classification and an analysis of such operations based on their semantics.
- a discussion of potential re-designs of such reflective operations.
- a dependency analysis between the reflective methods and between reflective categories.
- an approach based on mutation testing to assess the dependencies of an application on certain reflective APIs.

- RAPIM, an implementation strategy to contextualize the analysis *i.e.* handling the fact that reflective features are often a core part of a language (Python, Pharo...) and cannot be simply removed.
- a way to report results in a structured manner.
- a comparative evaluation of our approach compared to static analysis on a selection of projects.
- An evaluation of the relevance of polymorphism between reflective and non-reflective APIs.

These contributions are of key importance since they set the foundation for a redesign of the reflective capabilities of Pharo. They allow one to understand the current state of reflective APIs and their uses both by other reflective APIs and by libraries and applications. This could lead for example to offer optional reflective operations and more controlled ones in the context of a more secure and modular version of the language.

First steps in control for reflection. Then we focus on control strategies and explore the idea of using visibility modifiers to limit access to the reflective API. The contributions in this area are:

- A state of the art about existing techniques to control reflection, visibility modifiers, and object encapsulation.
- The definition of a protected method model named PROTDYN.
- #PHARO, an implementation for Pharo that supports optional protected modifiers, relies on default method lookup with negligible run-time overhead, and that applies to other dynamic object-oriented languages.
- An evaluation of the performance overhead introduced by our solution.
- An evaluation of the memory footprint impact of our solution
- A discussion of the applicability of such a protected modifier to Pharo's reflective API.

1.8 Structure of the Thesis

Analysing reflection and its uses. The first part of this thesis focuses on understanding reflection and how it is used.

In Chapter 2 we present an inventory of reflective operations in Pharo and a semantic classification. For each category, we analyze the capabilities it provides

and how they are used. We propose other improvement suggestions, design issues, and peculiarities of Pharo that we noticed during our analysis.

Then, in Chapter 3 we focus on dependencies inside the reflective APIs. We discuss interdependencies between previously established reflective categories and between reflective methods. We present the layered architecture of reflective operations.

Finally, in Chapter 4 we propose to use mutation analysis to study the use of reflection inside a library or application. We design reflection-specific mutations to obtain dynamic runtime information which is missing when using static analysis. We then use this information to assess the dependencies on reflective features. We evaluate the results of our approach in comparison to static analysis on a selection of projects and challenge the relevance of polymorphism between reflective and non-reflective APIs.

Through this first part, we provide information and tools to support future rational evolutions of Pharo reflective API.

First steps in control for reflection. The second part focuses on control strategies.

In Chapter 5, we present a simple state of the art of existing techniques to control reflection. We also discuss existing visibility modifiers and object encapsulation.

Then, in Chapter 6 we present PROTDYN, a self-send-based visibility model that introduces protected methods in dynamically-typed languages and computes method visibility at compile time. We present an implementation of PROTDYN in Pharo named #PHARO. This implementation is (1) optionally loadable, (2) does not require any changes to the default method lookup supported by a virtual machine, and (3) is backward compatible with existing code. We also discuss alternative implementation choices.

Finally, in Chapter 7 we evaluate the performance of our solution by measuring the performances of #PHARO. We assess its compatibility with existing runtime optimizations such as global lookup caches and polymorphic inline caches and we also measure its memory footprint. To conclude we discuss its applicability to Pharo's reflective infrastructure.

1.9 Publications

The work presented in this thesis is based on the following publications in international and peer-reviewed journals, conferences, and workshops.

- **International journals:**

Pharo: a reflective language - Analyzing the reflective API and its internal dependencies
I. THOMAS, S. DUCASSE, P. TESONE AND G. POLITO
Journal of Computer Languages, 2024

- **International Conference:**

  A VM-Agnostic and backward compatible protected modifier for dynamically-typed languages
  I. THOMAS, V. ARANEGA, S. DUCASSE, G. POLITO, AND P. TESONE
  The Art, Science, and Engineering of Programming, vol. 8, 2024

  Assessing Reflection Usage with Mutation Testing Augmented Analysis
  I. THOMAS, S. DUCASSE, G. POLITO AND P. TESONE
  21st International Conference on Software and Systems Reuse (ICSR), 2024

- **International Workshop:**

  Pharo: a reflective language - A first systematic analysis of reflective APIs
  I. THOMAS, S. DUCASSE, P. TESONE, AND G. POLITO,
  IWST 23 - international workshop on smalltalk technologies, 2023

The following technical report has also been published:

A classification of runtime reflective operations in Pharo
I. THOMAS, S. DUCASSE, P. TESONE, AND G. POLITO
Technical report - Evref, 2023

Part I

Analysing Reflection and Its Uses

CHAPTER 2

Classification Of Reflective Operations

Contents

2.1 Metaobjects, Classes and Their Related APIs 18
2.1.1 Pharo Structural Meta-Model 18
2.1.2 Overview of the Reflective APIs 19
2.2 A Classification and Analysis of Runtime Reflective Operations . 20
2.2.1 Methodology 22
2.2.2 Object Inspection Reflective Operations 22
2.2.3 Object Modification Reflective Operations 24
2.2.4 Class Structural Inspection Reflective Operations 24
2.2.5 Class Structural Modification Reflective Operations 26
2.2.6 Method Creation Reflective Operations 27
2.2.7 Structural Queries on Methods Reflective Operations 27
2.2.8 Message Sending and Code Execution Reflective Operations 28
2.2.9 Chasing and Atomic Pointers Swapping Reflective Operations 29
2.2.10 Memory Scanning Reflective Operations 30
2.2.11 Stack Manipulation Reflective Operations 31
2.3 Discussions 31
2.3.1 Threats to Validity 32
2.3.2 General Concerns 32
2.4 Conclusion 35

In this chapter, we analyze existing reflective features in Pharo 12. We scope the analysis to runtime reflection to focus on the core reflective features of the language and its associated virtual machine. Pharo inherited the reflective operations and facilities originally present in Squeak and Smalltalk-80 and extended them over the years. Some reflective methods such as Object≫instVarAt: are still present and used, some names have changed and new reflective facilities have appeared.

We chose to use the term *reflective API* to talk about reflective methods, as it highlights the fact that this is a programming interface offered by Pharo to developers.

This chapter acknowledges that the runtime reflection offered by Pharo needed a deep and systematic analysis after the evolution of Squeak, and the subsequent evolution of Pharo since 2008. The analysis of Rivard [**?**] while interesting is dated. Indeed, Pharo metaobjects evolved since this publication [**?**,**?**,**?**,**?**] and got first-class instance variables, and the introduction of new tools using reflection such as the new inspector framework [**?**], reflectivity [**?**], object-centric debugging [**?**], error handling infrastructure [**?**] and on the fly deprecated message rewritings [**?**]. This is not counting the full rewrite of the compiler as we chose to exclude those APIs.

This chapter begins with an overview of the Pharo reflective APIs based on the classes supporting them and their interactions in Section 2.1. In Section 2.2 we present a detailed classification of such operations. For each of the categories, we analyze the capabilities it provides and how they are used in Pharo. The technical report [**?**] lists the methods in each category (See Appendix A for the complete list). Finally, in Section 2.3, we discuss other improvement suggestions, design issues, and peculiarities of Pharo that we noticed during our analysis.

We note that the content of this chapter has been published in the following articles:

- "Pharo: a reflective language - A first systematic analysis of reflective APIs" Thomas *et al.* [**?**] and

- "Pharo: a reflective language – Analyzing the reflective API and its internal dependencies" Thomas *et al.* [**?**] which extends the first one.

2.1 Metaobjects, Classes and Their Related APIs

Before giving an overview of the API, we briefly present the structural metamodel of Pharo. The current version is Pharo 12.

2.1.1 Pharo Structural Meta-Model

A class is a central entity in Pharo's structural meta-model [**?**]. We briefly describe it, since a large part of the API is currently associated with classes.

- A class defines instance variables or slots. Since several versions of Pharo, slots (first-class instance variables) have been introduced and the fusion between instance variables and slots is under development. A class also defines class variables (a.k.a static variables) and uses zero or more shared pools which are collections of constants.

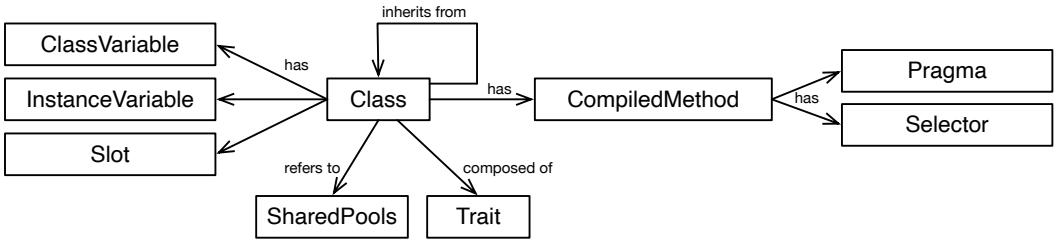


Figure 2.1: The structural Pharo metamodel: Class aggregates variables, methods, constant management (SharedPools) and method annotation (Pragma) and exposes related APIs.

- A class inherits from another class and has zero or more subclasses. Since a couple of versions, a class is composed of traits (class fragments defining methods and state).
- A class contains methods. Methods have a selector and are annotated using zero or more Pragmas [**?**].

2.1.2 Overview of the Reflective APIs

Figure 2.2 shows an overview of the reflective API of Pharo, structured with the classes that expose such APIs in the Pharo 12 release.

MetaObjects. Grey boxes represent first-class objects. Object, Slot, Class, ClassVariable, and CompiledMethod are structural metaobjects. The classes CompiledMethod and CompiledBlock are two entry points to AST nodes and sub-method reflective APIs. We decided not to add such a dimension since submethod reflection is optional and can be seen as compile-time reflection [**?**].

Implementation objects. We put the MethodDictionary, CompiledBlock, and BlockClosure in a white box because it is unclear whether we need or not a metaobject for them. Indeed in Pharo, the method dictionary is rather simple and does not offer a reflective entry point per se. It is more of an implementation object. Similarly, BlockClosure can be introspected as an object, but it is unclear whether it represents a metaobject or not.

Perspective. The dashed package-like packages represent two aspects of the system: on the one hand Memory which allows users to iterate memory with methods such as nextInstance and, on the other hand, Runtime which represents the execution aspect of the system with Context (stack reification), Messages, Thread, and Environment (keeping class and variable binding).

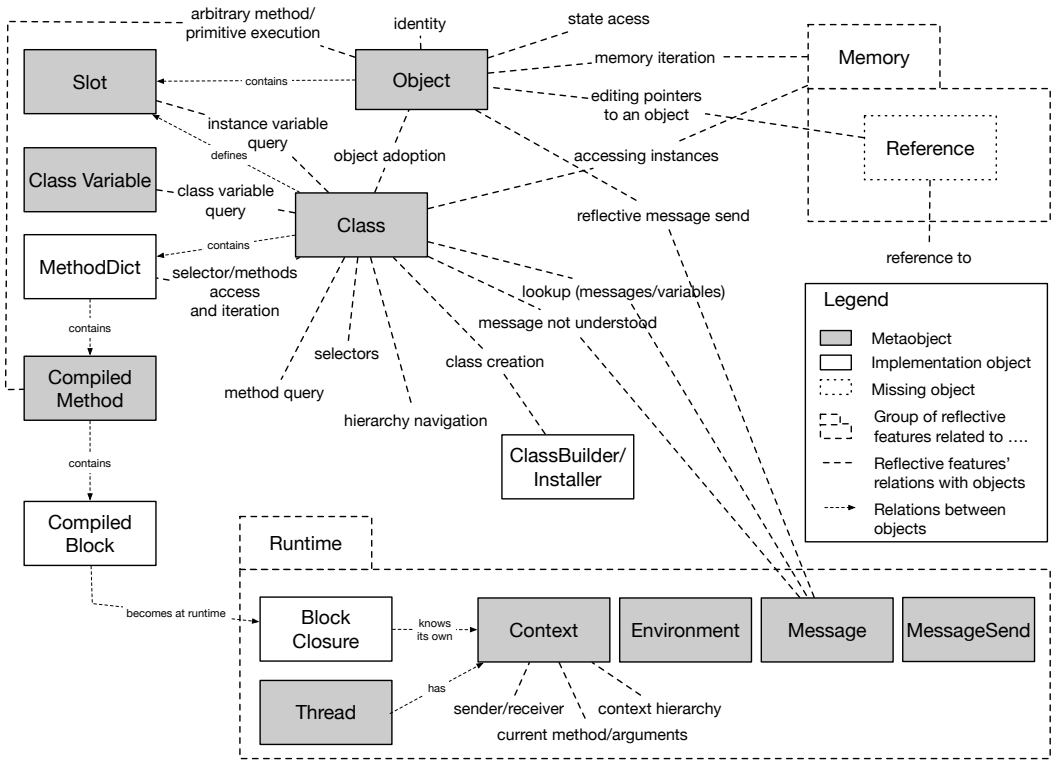


Figure 2.2: Metaobjects controlling the reflective APIs of Pharo.

Note that some APIs are not controlled by metaobjects per se. For example, the Reference API is an API defined on Object as such every object may override it.

2.2 A Classification and Analysis of Runtime Reflective Operations

Rivard [**?**] classified reflective operations in the following categories: *Meta-Operation* (objects), *Structure* (class), *Semantics* (compiler), *Message Sending*, and *Control State* (thread). The *Semantics* part is just a description of the compilation process and involved classes - as such it is not relevant for our analysis since it boils down to adding a new compiled method to a method dictionary. We complement and revisit this classification by adding *References*, and *Memory Scanning* (See Table 2.1).

We propose a detailed and systematic description of the APIs and runtime reflective behavior. Our classification subsumes the one of Rivard. In addition, we distinguish APIs supporting introspection from modification since modification has more impact in terms of state encapsulation. We are aware that systematically presenting lists may be tedious for the reader, this is why the technical report [**?**] describes systematically all the APIs.

Table 2.1 gives an overview of the classification: it groups reflective methods

Categories	APIs
A – Chasing and Swapping pointers	A1 – Bulk pointer swapping A2 – Find pointers to
B – Class structural Inspection	B1 – Class kind testing B2 – Class variable inspection B3 – Class/Metaclass shift B4 – Instance variable inspection B5 – Iterating and querying hierarchy B6 – Pragma B7 – Selectors and methods inspection B8 – Shared pool inspection B9 – Slot inspection B10 – Traits B11 – Variable lookup
C – Class structural Modification	C1 – Anonymous class creation C2 – Class variable modification C3 – Fluid Builder class creation C4 – Hierarchy modification C5 – Instance variable modification C6 – Old class creation C7 – Selector/Method modification C8 – Shared pool modification C9 – Slot modification
D – Memory Scanning	D1 – Memory Scanning D2 – Instances of a class
E – Message sending & code execution	E1 – Arbitrary method/primitive execution E2 – Control message passing E3 – Message send reification E4 – Method lookup E5 – Reflective message send E6 – Runtime and Evaluation
F - Object Inspection	F1 – Accessing object class F2 – Accessing object identity F3 – State inspection
G - Object Modification	G1 – Object class change G2 – State modification
H – Stack Manipulation	H1 – Context H2 – Controlling the stack
I – Structural queries On methods	I1 – Class references I2 – Method element references I3 – Method slot uses

Table 2.1: Overview of the reflective categories and APIs alphabetically sorted. Leading letters are used for Figure 3.1.

into APIs and APIs in high-level categories. The categories are sorted alphabetically. The letters are used to understand Figure 3.1.

For each of the APIs we briefly describe it, list its key methods (we often group similar ones), the offered possibilities, and the areas of improvement when appropriate.

Finally, note that the existence of an API is more important than the fact that we classify it under a given heading. For example, asking an object to reflectively execute a method is listed together with other execution-oriented APIs and not directly in the object-centered API. In addition, the next subsections are organized to follow Rivard's classification order.

Sections 3.2.1 and 3.3 identify dependencies and layers among the categories.

2.2.1 Methodology

To analyze and classify the reflective API we focus on the base image of Pharo 12, build 636[1]. We manually identified reflective methods by reading the code of the base image, specifically code belonging to the explicit list of metaobjects and packages present in Figure 2.1 and Figure 2.2. We identified reflective methods using definitions of reflection from [**?**, **?**] and categorized them based on a categorization that builds on top of Rivard's. We then tagged the identified reflective methods with a pragma[2] parametrized with its reflective category.

Using this methodology we identified and marked 532 methods with 344 unique selectors as reflective. This number shows the extent of the reflective API in Pharo [**?**]. The reflective method category tagging was proposed as a pull request to the Pharo repository and accepted by the community in September 2023[3]. In what follows, we refer as *reflective method* any method that is marked in our list. A *reflective operation* is an operation performing reflection and implemented through one or more methods.

In the remainder of this section, we present and analyze the reflective methods and operations we found, divided into categories. Later, in Section 3.2, we analyze the dependencies between these categories.

2.2.2 Object Inspection Reflective Operations

The first category of reflective operations is centered around object inspection. Rivard [**?**] described these operations in the *Meta-Operations* category, but he grouped inspection and modification. Our category is composed of three subcategories:

- ***State inspection*** to read the values of the variables of an object.

[1] Pharo-12.0.0+build.836.sha.8b241ecb87492515bbdd975557ecf8491a4af88b (64 Bit)

[2] A pragma is a method annotation in Pharo's parlance

[3] https://github.com/pharo-project/pharo/pull/14821

- ***Accessing object identity*** to identify an object.
- ***Accessing object class*** to read the class of an object.

In Pharo, all instance variables are private, meaning they are not readable and writable by any other object. They are only accessible through getter or setter methods. Developers decide which instance variables are accessible by implementing or not methods to access them. Pharo also includes class instance variables and shared variables, these work in the same fashion as instance variables, and the analysis for instance variables is directly extensible to them. Using the ***State inspection*** operations is breaking the encapsulation and bypasses the decisions of the developer.

Several methods exist on the class Object allowing access to the state of internal variables. Key examples of this category are Object≫instVarAt: and Object≫instVarNamed:, which read an instance variable of an object from its index or name respectively. These operations combine well with those in the ***Accessing object class*** subcategory to work on object internal structure *e.g.* Behavior≫allInstVarNames.

Possibilities offered. The ***State inspection*** operations give a uniform API to inspect all the instance variables of any object, including classes. They are particularly useful for designing tools addressing crosscutting needs, like debugging, inspecting an object, serializing it... The ***Accessing object identity*** supports checking the identity of an object. basicIdentityHash is used for implementing identityHash variants, scanning for an object in a method dictionary, and testing.

Examples of uses.

- Serializing objects.
- Inspecting objects.
- Implementing hash methods.

This use raises the question of whether accessing object identity is a reflective operation and not just part of the base-level object API in a language where references are ubiquitous.

Areas of improvement.

- In the analyzed Pharo version, there is currently no provided solution for intercession on state read or write on a class or even on a specific object. This requires using additional libraries or implementing ad hoc solutions [**?**]. Such tools rely heavily on reflection, and loading several tools at the same time might lead to bugs and instabilities due to incompatibilities between them.

2.2.3 Object Modification Reflective Operations

The second category of reflective operations is centered around object modification. It is the counterpart of the first one and it is composed of ***State modification***, ***Manipulating object identity***, and ***Object's class change***.

- ***State modification*** to write the values of variables of an object.
- ***Manipulating object identity*** to manipulate the identity of an object.
- ***Object's class change*** to change the class of an object.

Possibilities offered. The ***State modification*** operations allow one to bypass encapsulation and modifying variables of third-party objects. This could be used to write variables in an unanticipated way when they were originally designed to not be changed via base-level message passing. This is for example useful for deserialization. They allow one to build tools that will modify objects. The ***Object's class change*** operations allow one object to become an instance of another class, which is particularly important in Pharo's live environment when a class has to be rebuilt. The ***Manipulating object identity*** category contains only one operation becomeForward:copyHash:[4]

Examples of uses.

- Copying objects.
- Deserializing objects.
- Modifying object on the fly in the debugger.
- Migrate instances between two class versions.

Areas of improvement. The API on object class change is weak and limited. The object state may be lost in the process and some constraints (the two classes should have the same format) make it difficult to change the actual class. Overriding the methods providing these operations provides a way to limit reflection at the cost of limiting services such as instance migration provided by the environment.

2.2.4 Class Structural Inspection Reflective Operations

This category groups reflective APIs that query the class structure and its constituents: methods, variables (instance/class/slots). It is composed of the following subcategories:

[4]Using becomeForward:copyHash: to swap a reference it is possible to change the object identity.

- ***Class/metaclass shift*** to navigate between a class and its metaclass.
- ***Iterating and querying hierarchy*** to query class hierarchies.
- ***Instance variable inspection***, ***Class variable inspection***, ***Shared pool inspection***, ***Slot inspection*** to query variable definitions. ***Slot inspection*** provides a higher level view compared to ***Instance variable inspection***. Slots are either defined locally in a class or imported, for example from a trait. Thus the existence of the localSlots and slots operations.
- ***Selector and method inspection*** to query the set of methods/selectors implemented by a class.
- ***Variable lookup*** the access the binding of a variable.
- ***Pragmas*** to query pragmas.
- ***Class kind testing*** to query the state of a class (installed, obsolete, anonymous...).

Possibilities offered. The structural class introspection is large. It is mainly used by tools. It supports the interpretation of object inspection. ***Iterating and querying hierarchy*** methods support navigation of the graph with messages such as superclass and allSubclasses. ***Selector and method inspection*** methods allow one to check existing selectors and methods. All variable query operations allow one to list existing variables. Some methods such as isKindOf: or respondsTo: produce suboptimal designs when used at the base level. respondsTo: allows one to query if an object understands a given selector.

Examples of uses.

- Object Serialization and Deserialization.
- Code browsing.
- Object inspection.

Areas of improvement.

- There is spurious redundancy between isClassSide and isMeta. Such double methods should be corrected.
- As a general remark, the question of the systematic application of the Law of Demeter should be discussed because it bloats the API. For example, messages such as selectSuperclasses: / selectSubclasses: do not seem to be necessary. In addition, withAllSuperAndSubclasses and includesBehavior: look superfluous.

- We see the old protocol with cryptic names such as instSize to mean instanceVariableSize.
- The duality of instance variables and slots is an artifact of the current evolution of Pharo. Nevertheless, this is important that in the future, instance variables get fully replaced by slots and that the corresponding reflective APIs get merged.
- The duality of selectors versus methods should be evaluated. Since a method dictionary always has the selector of the method as a key, the API could favor selectors for most of the queries and only favor one access to compiled methods (via methods such as methodNamed:).

2.2.5 Class Structural Modification Reflective Operations

This category is the counterpart of the previous one. It is composed of the following APIs whose objectives are clear: Hierarchy modification, ***Instance variable modification***, ***Shared pool modification***, ***Slot modification***, ***Selector/Method modification***, ***Old class creation***, ***Fluid class creation***, and ***Anonymous class creation***. It focuses on the modification of the structural relation a class has with its constituents.

Possibilities offered. Structural modification operations allow the user to modify the current structure/shape of classes. They include operations to add/remove subclasses, instance variables, and class variables. We distinguish introspection and modification APIs because we want to stress that modification is destructively modifying the executed system and that as such they represent more powerful operations.

In addition to the traditional class creation API (kept for backward compatibility) and the fluid API, Pharo introduced the notion of anonymous classes (message newAnonymousSubclass) [**?**]. It helps to define instance-specific methods.

Examples of uses.

- Reflective code modification.
- Object-Centric Reflection.
- Proxy implementations.

Areas of improvement.

- The unification of Slots and variables should be continued to avoid duplication at the reflective API level.

- About ***Selector/Method modification***. The 'silently' prefix raises the question of the management of the notification of modification. Indeed, some tools need to be notified to react to new elements. Nevertheless, this duality suggests a layered API where low-level API elements are identified.
- The API ***Anonymous class creation*** can be packaged separately from the ***Fluid class creation***.

2.2.6 Method Creation Reflective Operations

The API ***Compiled method creation*** is a low-level API that supports the definition of compiled methods. Such an API is often ignored but it is central because it is responsible for the creation of new compiled methods.

Possibilities offered. ***Compiled method creation*** offers the possibility to create a compiled method and this even without the need for the compiler.

Examples of uses. This API supports the modularization of the system core such as making the compiler optional in the system bootstrap. It is used by Hermes a binary binary code loader. It is an important asset that supports the bootstrap of Pharo [**?**] and ensures that the compiler is loadable into a system without having a compiler to compile and install code.

Areas of improvement. Since the code for creating compiled methods is just the code of the CompiledMethod class. It has not been explicitly designed to be a MOP API. Revisiting this central API in the context of the ***Selector/Method modification*** and the interplay between the two could lead to a stronger MOP.

2.2.7 Structural Queries on Methods Reflective Operations

This category supports the cross-referencing between methods, instance variables, and classes. It is composed of two subcategories:

- ***Method slot uses*** to query usages of variable read/writes.
- ***Method element references*** to query internal implementation of methods.

Possibilities offered. These two subcategories are central for all the cross-referencing and code navigation in Pharo.

Examples of uses. These operations are important for the IDE and tools.

- query method senders and implementors
- query methods reading/writing a variable

Areas of improvement.

- The duality of selectors and methods could be handled better, *e.g.* whichMethodsReferTo: vs whichSelectorsReferTo:. We suggest not exposing the compiled method, since it is always possible to get the method out of a selector. It would lead to a more compact API.
- ***Method slot uses*** looks like an optional API that can be packaged with the tools. A unification between the two APIs would produce a more coherent API.

2.2.8 Message Sending and Code Execution Reflective Operations

This category of operations allows us to explicitly send messages, handle lookup failures, or execute compiled methods. It is composed of different subcategories:

- ***Reflective message send*** to lookup and execute methods.
- ***Arbitrary method/primitive execution*** to execute methods without lookup.
- ***Method lookup*** to simulate the method lookup.
- ***Control message passing*** to control message sends.
- ***Message send reification*** to access message information.

In Pharo, when sending a message to an object, the first step is to search the message selector in the class hierarchy of the message's receiver. This is the lookup. Once a compiled method is found in the receiver's class or one of its superclasses, the method is applied to the receiver. When the lookup does not find any corresponding method, doesNotUnderstand: is sent to the receiver with the message reified as an instance of MessageSend. This allows the receiver to specialize message error. The APIs are central to bringing flexibility to applications. In particular ***Reflective message send*** with its perform: methods is important for pluggable UI logic. While the methods of ***Reflective message send*** do a method lookup, methods of ***Arbitrary method/primitive execution*** allow us to execute directly a compiled method or a primitive operation[5]. While the methods of ***Reflective message send*** do a method lookup, methods of ***Arbitrary method/primitive execution*** allow us to execute directly a compiled method or a primitive[6].

Possibilities offered. These operations allow us to explicitly send a message and handle failure cases. The selector sent is determined dynamically from an input, a string, or a symbol. Run a specific primitive operation or version of a compiled method without requiring installing a method in the class hierarchy.

[5]A primitive operation is a call to a virtual machine internal behavior.
[6]A primitive operation is a call to a virtual machine internal behavior.

Examples of uses.

- Implementing frameworks with naming flexibility such as in SUnit.
- Decouple user interfaces from model objects.
- Proxy implementations.
- Debugging and profiling.

Areas of improvement.

- Pharo offers two ways of representing a message via the class MessageSend and Message. This situation shows that the addition of concepts was not done carefully to avoid duplication. Message represents a message when an error occurs (doesNotUnderstand:). It supports the possibility to perform a lookup via the message sendTo: which is the counterpart of doesNotUnderstand:. MessageSend represents the concept of sending a message and holds in addition a receiver. Such a class is not used by the runtime and offers an API compatible with block closures: The messages value: and its variants allow one to execute a message. We suggest merging MessageSend into Message since this last one is used to reify messages on error.
- Having several ways to express the same behavior can be improved. There are, for example, three different reflective methods implementing similar behavior in ***Arbitrary method/primitive execution***. We suggest that only methods on CompiledMethod should be kept. The definition on ProtoObject would have the pernicious side effect of making domain developers think that it is safe to use such messages.
- The direct execution of a compiled method as offered by the ***Arbitrary method-/primitive execution*** API is dangerous because the system does not check that the executed method can be executed on the receiver. This is usually ensured by the lookup. Therefore while it makes sense to use methods of the ***Reflective message send*** API, we believe domain developers should not be exposed to the ***Arbitrary method/primitive execution*** API. In addition, this API should be packaged separately to expose its nature.

2.2.9 Chasing and Atomic Pointers Swapping Reflective Operations

The APIs in this category are ***Find pointers to*** and ***Bulk pointer swapping*** (supports the atomic swapping to references). The first one is rarely mentioned but Pharo supports the possibility of identifying pointers to a given object (*e.g.* ProtoObject≫pointersTo and ProtoObject≫pointsTo:).

Possibilities offered. ***Find pointers to*** is useful for building tools to identify a memory leak; it is optional and its use is well-scoped.

The second one, ***Bulk pointer swapping***, is one of the hallmarks of Smalltalk reflective APIs. Pharo's implementation implements this operation efficiently by using forwarders at the VM level [**?**]. It should be noted that Pharo offers two semantics for swapping: become: which symmetrically swaps the pointers and becomeForward: which is one way. In addition, one cannot be used to express the other at the language level.

Examples of uses.

- Memory leak analyzers.
- Dynamically updating existing instances to new class shapes.

Areas of improvement. Such API while useful during development sessions should be limited during deployment. A precise analysis of the use of ***Bulk pointer swapping*** should be done to differentiate the places where it is mandatory from the places where it is a convenient optimized solution. It should be noted that ***Find pointers to*** could be implemented on top of a full memory scan API such as the ones presented in the next section. Similarly, a slower version of ***Bulk pointer swapping*** could be possible.

2.2.10 Memory Scanning Reflective Operations

This category contains two subcategories: *Memory scanning* that supports the traversal of the complete heap and Instances of a class that gives access to all the instances of a class.

Possibilities offered. This API is usually not mentioned in the literature but it is at the core of live programming [**?**]. The method nextObject and nextInstance are key to building other functionalities such as allInstances.

Examples of uses. The main use is the migration of instances between two versions of one class. All objects need to be obtained to be migrated to the new class and be potentially rebuilt if there are changes in their shape *e.g.* if a variable is added/removed. Other uses such as collecting all instances of a class are more anecdotal and reflect the lack of an explicit registration mechanism in the domain.

Areas of improvement. Understanding whether such a reflective API can be optional and only be loaded on demand would be a step toward building a more compact, tidier, and secure reflective MetaObject protocol.

2.2.11 Stack Manipulation Reflective Operations

This category groups together all APIs that support traversing and modifying the execution stack. These APIs are accessible from two main entry points: the Process class that provides access to the existing processes and their suspended execution stack, and the thisContext pseudo-variable that provides access to the current method execution. Both these entry points provide instances of Context that represent a method execution and make the execution stack in a linked list.

Possibilities offered. Stack manipulation operations provide on the one hand low-level access to the call stack (*e.g.* sender, programCounter), context meta-data (*e.g.* method), and context operand stack (*e.g.* push:, pop and at:), and on the other hand support for continuations built on top of the previous APIs (*e.g.* return:, resume:).

Examples of uses.

- Implementing exceptions.
- Debuggers.
- Bytecode interpretation.

Areas of improvement. Stack manipulation support for stack modification and intercession is, at the moment of this writing, limited. A single class Context is allowed, and its instances are reified on-demand by the execution engine by the Virtual Machine implementation. This means that the APIs described above cannot be refined in subclasses to modify the behavior of method execution. Currently, such fine-grained intercession is achieved by bytecode rewriting using frameworks such as reflectivity [**?**].

Additionally, the fact that essential language features such as exceptions are built on top of it makes this support mandatory. Optional stack manipulation requires a major redesign such as a re-implementation of such essential features on the Virtual Machine, or at the extreme considering exceptions as optional reflective features too.

2.3 Discussions

In this section, we discuss various aspects ranging from the analysis we presented to the consideration to be taken into account.

2.3.1 Threats to Validity

False negatives in reflective methods. By construction, this study is susceptible to false negatives on reflective methods. As the identification of reflective methods was done manually, we might have missed some packages and methods as we could not read through the whole image. Therefore some reflective methods might not be identified and tagged as such. However, with 532 identified methods, we believe that the presented study is representative of the Pharo reflective API. Moreover, those tags have been submitted to Pharo, reviewed, and integrated for later versions.

2.3.2 General Concerns

About dual entities. On several occasions, the MOP proposes a kind of duplicated API: one for selectors and the other for compiled methods, or one for a variable and its name. Having only one API based on the object is not good because it forces the developer to have an object when it may not be possible. It means that using a name is a good approach. In particular, such metaobjects (compiled method, slot) expose their name. We suggest reducing the API spectrum by not proposing two APIs but instead favoring one based on the name for query and access. For modifications, the developer will query first based on the name to access an object and then perform the corresponding operation. In that regard, the question of the application of the Law of Demeter should be evaluated since it tends to produce larger APIs.

Absence of clear layers between base-level and meta-level. On several occasions, we see the need to identify the level of API. Indeed some methods require mere index (instVarAt:) while some others require names (instVarName:). While the first one is needed, we suggest (1) a clear naming convention that helps understand the level of the functionality, (2) a better naming (methods named instSize that returns the number of instance variables that feel outdated in a modern language). Finally, some APIs are large because basic functionality is augmented with helper behavior built on top of such basic functionality. While this is a good practice to promote code reuse and offer developers stronger APIs, we suggest laying off such APIs and making sure that high-level APIs are optional with clearly identified users.

About metaobject Protocol and piecemeal growth. In Smalltalk, reflection is exposed as methods of objects that modify the internals of the system and the causal connection makes sure that the modifications get in effect. We see this approach as an organic one and from this perspective we say that the metaobject Protocol of Smalltalk has been less designed than the one of CLOS [**?**].

We suggest that the design of a new MOP should consider how certain objects represent customization points and avoid piecemeal and accidental MOP growth. For example, in CLOS it is possible to specify at the metaclass level, the class of the executed method. The Method class is then a natural metaobject exposing a

method that can be specialized to for example count the number of executions of the methods. In Pharo, the hook to specify the class of a method is not clear. More important, when a method is executed no identified method is called before the method execution. Frameworks such as Method Proxy, MethodWrappers [**?**] build such functionality using VM hooks such the possibility to place any object in a system dictionary and that such an object receives the message run:with:in:.

A MOP may decide to expose customization points as dedicated objects and not necessarily objects that are currently been executed [**?**]. For example in CodA, different lifetime aspects of objects (message-send, message received, state access, execution,...) are reified via specific metaobjects.

Execution-Time Reflection. In our analysis, we have centered on the reflective API during the execution time. We analyzed the operations that are executed during code execution. In this sense, we have left outside operations performed outside the execution of the code. Operations such as static code analysis and rewriting, memory dump inspection and modification, refactoring, and on-load code rewriting or instrumentation are not performed during execution time. Those operations are outside our definition of reflective operations.

Compiler. In contrast to Rivard [**?**] who considers the compiler as part of the reflective API of the system, in our analysis we keep it out. We have taken this decision as the reflective API provides ways of creating and installing methods. In Pharo reflective API a method is created from its bytecode, literals, and header. The complexity of generating such a set of bytecode, literals, and header for the method is outside the reflective API. The compiler has as input the source code of the method. Through a series of complex transformations (such as parsing, AST building, AST rewriting, AST optimizations, Intermediate Representation Building, and bytecode generation) the compiler generates the bytecode, literals, and headers. A compiler is just one possible source of these elements. For example, in Pharo, we have a binary code loader that generates and installs methods. This binary code loader is used without the compiler to load code during the bootstrap process of the image. The compiler and the binary code loader both use the same reflective API, that allows them to create and install new methods in the running environment. Moreover, it is possible to have more alternative tools to generate methods profiting Pharo reflective API.

Package Loading / Unloading Missing. The existing reflective API does not present a clear metamodel to handle the concept of Packages. Even though this concept is used outside the execution of code. It is a key element in the metamodel of Pharo. It is used to load, unload, and version classes, methods, and extensions existing in Pharo. Moreover, it is the key element to support method extensions.

A clear reflective API is required to handle the loading, unloading, versioning, and modification of packages in Pharo. Also, clear modeling of the package allows for additional points of extension to the metamodel and the ability to improve existing tools (*e.g.* scoping extensions, dependencies).

We have left outside of this thesis the analysis of the features and a possible design of such Package API, but we recognize the importance of such reflective API.

Architecture for notification. In our analysis, we have found that there is no clear API for handling the notifications of changes. Tools working on the metamodel of Pharo require a good integration to be notified of changes. For example, a Code Browser requires a clear way of getting notifications when a new method or class is added to the system. Also, there are scenarios where the tools modify the system but this modification should not be notified. For example, when instrumenting a method, if the original method is replaced there is no need for the Code Browser to be notified.

A clear notification API should guarantee that the tools and libraries scope the notifications they want to produce and consume.

An extensive analysis of this notification architecture is outside the scope of this analysis, however, we realize that such a notification architecture is required.

About definition and method reification. In early versions of Squeak, a compiled method did not know its class or its selector. It was then expensive to ask a compiled for its selector since it required scanning all the methods installed in a class. Over the years compiled methods saw their API and representation improved. At the same time, there is a need to be able to represent methods that are not installed in a class, for example, to browse multiple versions of a method or perform branch analysis [**?**]. In this scenario, there is a need to represent a method with a source code that is not one of the currently installed compiled methods. Similarly, several meta-models such as Ring and Ring2 have been designed to support the analysis of code not loaded in an image (browsing, crossreferencing, remote browser...). There is a need to have method definitions as well as compiled methods. This raises the question of whether the tools should not manipulate method definitions and not compiled methods. Such method definitions could be connected to a compiled method when the compiled method is installed in the system. From a reflective API, compiled methods could be more executable objects and expose only information related to their execution and for all the other needs the tools could request the associated method definition. By making sure that the tools and reflective API always go from a method definition to the compiled method, we could restrain the compiled method API. Such architecture, however, should be built and validated because, in an image for development, it would double the number of objects representing methods or special caches should be done to support method cross-referencing.

2.4 Conclusion

In this chapter, we presented a new systematic and deep analysis of the reflective APIs revealing some often undocumented aspects such as memory scanning or method installation. In addition, the analysis proposes some potential improvements to the existing MOP. We started by giving an overview of the Pharo meta-object protocol. Then we presented each of the ten identified categories, their subcategories (with a total of 38 of them), what kind of possibilities they offered, how they are currently used in the system, and we highlighted areas of improvement. Finally, we discussed other concerns and peculiarities that were not addressed in the classification section.

The discussion raises the bar of the analysis because MOP designers should challenge the monolithic perception that a MOP should be omnipotent and cover all aspects all the time. We believe that a faceted MOP where on-demand reflective operations can be made available is the way to create a more versatile system that has different varieties of flavors depending on the kind of applications that one wants to deploy and their companion security properties.

This work serves as the basis for Chapter 3 and Chapter 4. By tagging reflective methods identified in this chapter with pragmas, we are now able to programmatically distinguish them from non-reflective methods. In the next chapter, we use this ability to study dependencies between reflective categories and between methods inside a specific category.

CHAPTER 3

Analyzing Dependencies Between Reflective APIs

Contents

3.1 Methodology 38

3.2 Reflective APIs Categories Interdependencies 39

3.2.1 Interdependencies Overview 39

3.2.2 Isolated Reflective Categories 42

3.2.3 Provider Categories 43

3.2.4 Client Categories 43

3.2.5 Iterating and Querying Hierarchy, a Hub Category 44

3.3 Visualising Reflective Operations Intra-category Dependencies and Their Layered Organization 45

3.3.1 Chasing and Swapping Pointers - Find Pointers to 45

3.3.2 Memory Scanning - Instances of a Class 46

3.3.3 Message Sending and Code Execution - Runtime and Evaluation & Message sending and Code Execution - Reflective Message Send 47

3.4 Threats to Validity 48

3.5 Conclusion 49

In this chapter, we leverage the categorization of reflective operations shown in Chapter 2 to study *inter-category* (Section 3.2.1) and *inter-method dependencies* (Section 3.3). Understanding the dependencies between reflective APIs will help redesign the current API. This means, for example, restructuring parts of the API to fit better in the system or, identifying optional APIs that could be packaged separately. This would lead to a simpler, lighter base language, with potentially less threats to the stability of the system.

Section 3.1 presents the methodology we used to study dependencies. Section 3.2 presents the *inter-category* dependencies of reflective APIs. Section 3.3 discusses the layered architecture of reflective operations we saw emerging when we studied

inter-method dependencies inside a specific category. Section 3.4 highlights the limits of our approach and the steps we took to mitigate those.

The content of this chapter has been published in the following article:

- "Pharo: a reflective language – Analyzing the reflective API and its internal dependencies" Thomas *et al.* [**?**].

3.1 Methodology

We identify dependencies between reflective methods using static analysis based on their selectors, given that Pharo is a dynamically-typed language without type annotations. We say that reflective selector A depends on reflective selector B if any method with the selector A sends a message with selector B. Unless specified otherwise, we focus only on users who are reflective methods themselves. When relevant, we extend our analysis to base-level users.

Unless stated otherwise, when analyzing categories interdependencies we leave out of our analysis some selectors that present both reflective implementors and not reflective implementors. This is the case of largely used selectors such as at:, at:put: and size that are implemented as reflective methods in Context but have non-reflective counterparts in collections. We also ignore from our analysis a dozen other selectors that have reflective implementors in different categories as we cannot statically pinpoint which version of the methods is meant to be called, and therefore cannot determine the appropriate reflective category for a dependency.

The following selectors are removed due to polymorphism with non-reflective selectors:

- at:
- at:put:
- value
- size

Thus, the uses of these selectors do not necessarily imply dependencies on the reflective version of the method.

The following selectors are removed due to polymorphism across multiple reflective categories:

- outerContext
- usingMethods
- receiver
- numArgs
- arguments
- arguments:
- instVarAt:put:
- selector
- selector:
- receiver:
- isClass
- sender
- valueWithEnoughArguments:

Section 3.2.1 presents a high-level view of the interdependencies in the reflective categories. For the sake of presentation, the dependency analysis removes, in addition to the previous list of selectors, the selector class as it is the one with the highest number of connections (28 other selectors have at least one method depending on class). As it is highly used, it creates a lot of noise in visualization, and removing it makes the rest of the information more readable. A specific analysis of its uses is in Section 3.2.3, in paragraph ***Object Inspection - Accessing object class***.

3.2 Reflective APIs Categories Interdependencies

3.2.1 Interdependencies Overview

Figure 3.1 presents a graph that illustrates how all reflective categories connect with their dependencies. The figure shows that most reflective categories are building on each other and highlight the high interconnectivity.

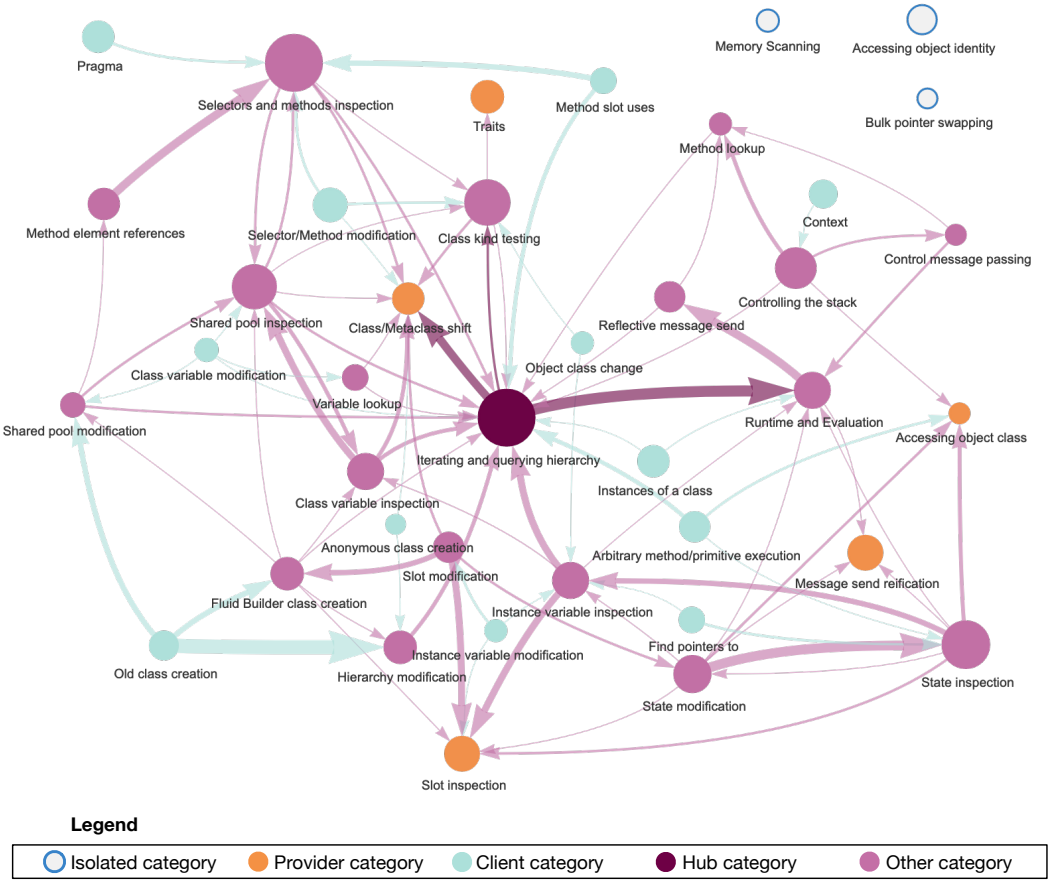


Figure 3.1: Reflective category dependency graph. The size of the circle corresponds to the number of selectors in the category. Line thickness depends on the number of dependent selectors. For more details see Figure 3.2.

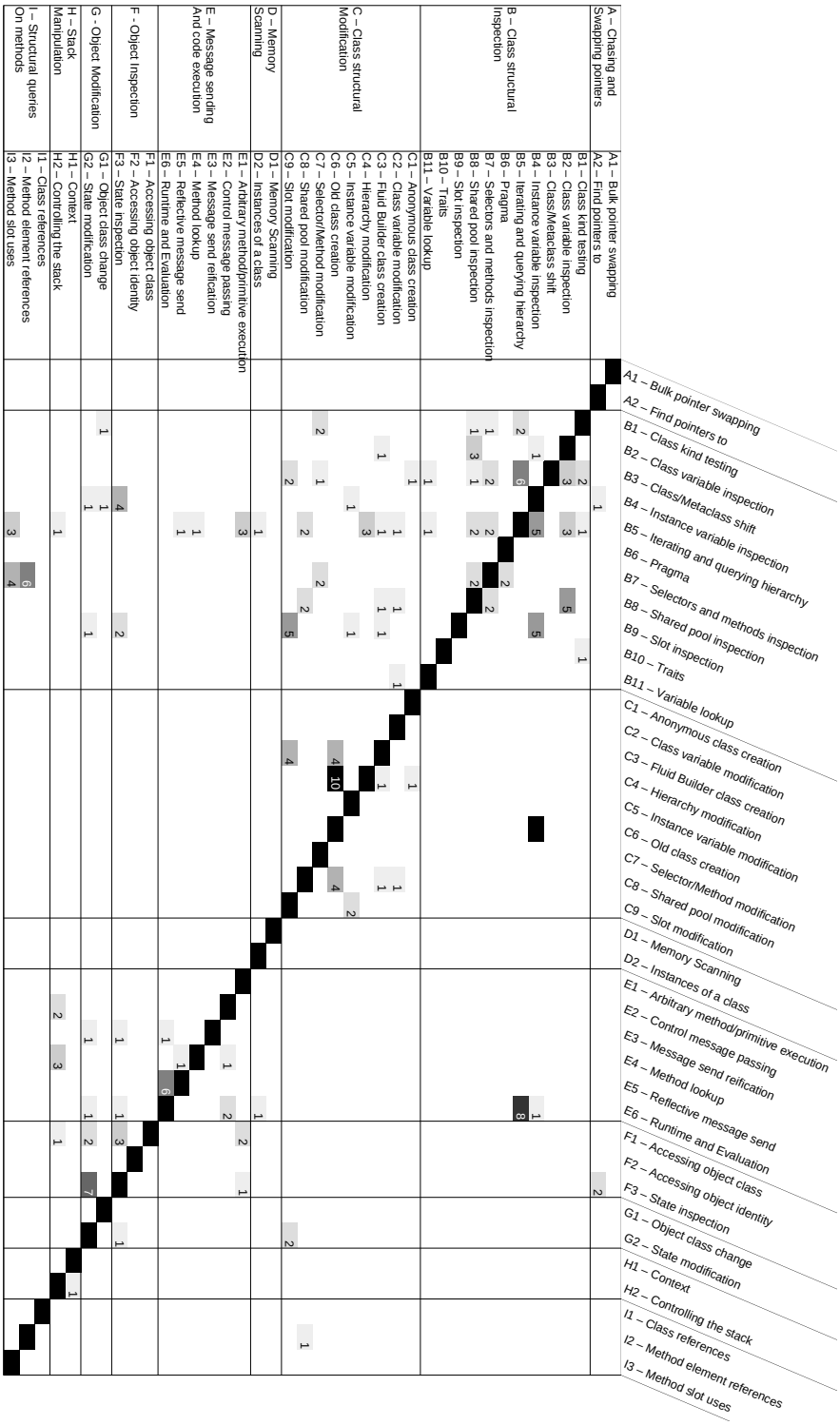


Figure 3.2: Matrix of dependencies between categories. The category in row X depends on the category in Column Y if there is a number at the intersection. The number corresponds to the number of different selectors depending on the other category.

Based on this data we manually identified four types of categories based on their dependency topology, shown in Figure 3.3:

- *Isolated categories*: These categories do not depend on any other categories and no other category depends on them.
- *Provider categories*: These categories do not depend on any other categories but they provide operations that are used by other categories.
- *Client categories*: These categories depend on other categories, but no other reflective category depends on them.
- *Hub categories*: These categories depend on a lot of other categories and many other categories depend on them.

The remaining categories both depend on some categories (outgoing dependency) and provide operations used by some other categories (incoming dependencies). They have a relatively low amount of each with at most 10 incoming and outgoing dependencies in total.

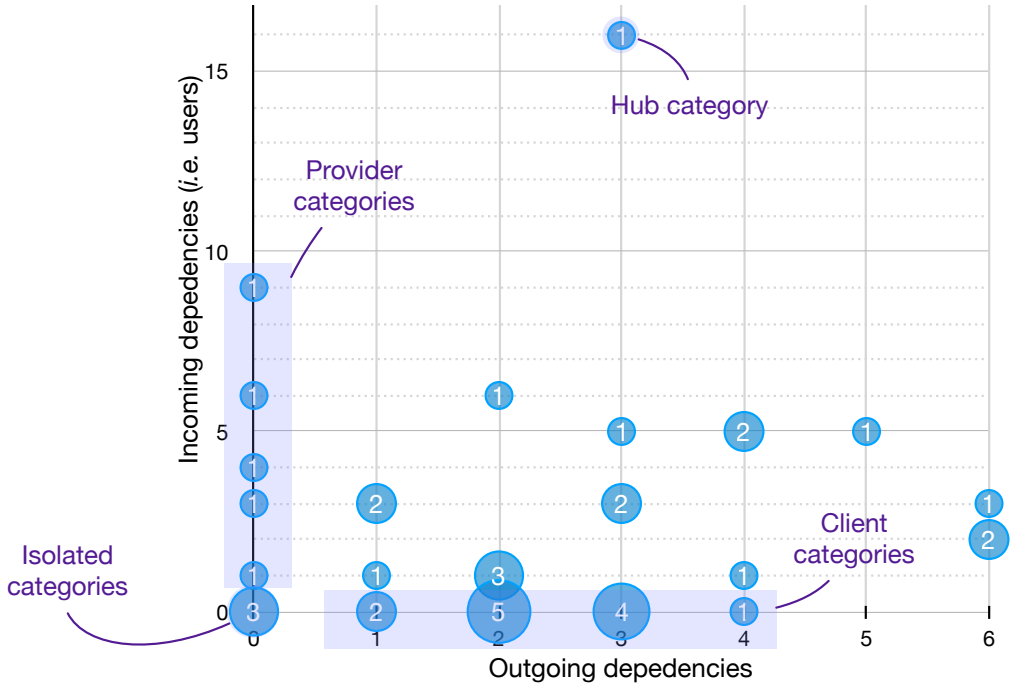


Figure 3.3: Categories according to the number of incoming and outgoing dependencies. The size of each bubble and the number inside of it correspond to the number of reflective categories matching that number of outgoing (x-axis) and incoming (y-axis) dependencies.

Isolated and *Client* categories are two types of categories that could be the first candidates for being packaged in a library. Indeed, those categories have no other category depending on them, which limits the impact of packaging them away on the rest of Pharo's reflective API. This would also require analyzing the base-level uses of their APIs in the base image, which is outside the scope of this chapter.

Figure 3.2 presents a table with all the categories and the interdependency of them. The intersection of categories shows the number of selectors used by the category on top of the category on the right. A higher number shows a higher level of dependency between the two categories. This provides an alternative view to the data in Figure 3.1 to facilitate readability when focusing on the dependencies of a single category.

3.2.2 Isolated Reflective Categories

We identified three categories of reflective operations that are neither relying on other categories nor relied on by other categories. The ***Memory Scanning - Memory Scanning***, ***Object Inspection - Accessing object identity***, and ***Chasing and swapping pointers - Bulk pointer swapping*** categories are not relying on any other categories because they rely on primitive operations. There is also no other reflective category relying on them. To understand how they fit in Pharo, in this section we analyze all users –reflective and not– in the base image, not only the ones that are reflective methods. They provide low-level APIs that are either not used in the base image (*i.e.* their selector appears neither in reflective nor non-reflective code) or used for low-level implementation details like defining equality and growing collections in memory.

- For ***Memory Scanning - Memory Scanning***, implementations of both nextObject and someObject directy call primitive operations. Those are two low-level methods that allow developers to iterate on the memory. The only sender of nextObject in the base image is a regression test on ProtoObject checking for a previous image crash. someObject is never used.

- For ***Object Inspection - Accessing object identity***, 7 out of 11 methods are calling primitive operations, and two have an internal dependency to basicIdentityHash. The two remaining methods are the overriding versions of basicIdentityHash and identityHash for SmallInteger which are based on the value of the integer itself. Those methods are used to implement hash functions and check for equality. They are not used by other reflective features.

- The three members of ***Chasing and swapping pointers - Bulk pointer swapping*** rely on three methods in Array that call the primitive operations. While the method become: is only used for some tests, becomeForward:copyHash: is used in a proxy implementation in the Iceberg package, a version control package. becomeForward: is the most used of the three, but for non-reflective

purposes: during the bootstrap to handle undeclared variables, to edit the specialObjectsArray, to manage internal representations of collections and method dictionaries, and to convert weak announcements into strong ones.

3.2.3 Provider Categories

We have four categories that are not relying on any other categories, but are providing APIs used by others :

- ***Class structural inspection - Class/Metaclass shift***. The implementations of classSide, instanceSide, and variants rely on the metamodel and its hierarchy. Among selectors that have been removed for clarity, it is only using the selector class once. Its messages are however used in a dozen of other categories.
- ***Class structural inspection - Slot inspection***. Slots are stored in the metaobjects of Pharo in a collection. Their implementation does not rely on primitive operations. Accessing the slots and querying them are used to access the state of an object or class and modify it, but also to create or modify the instance variables of a class. This covers seven categories of reflective operations.
- ***Object Inspection - Accessing object class*** Four methods rely on the same primitive operation, either directly or indirectly. It is usually used either for comparisons or to access the API of the class. In terms of reflective API Context≫objectClass: and MirrorPrimitives class≫classOf are used to access instance variables, send messages, and execute arbitrary methods/primitive operations. With more than 4800 senders in both the base-level and reflective methods, class is one of the selectors that is the most used in the Pharo image. This is why in the rest of this chapter, ProtoObject≫class is excluded when looking at the connections between categories. The class method is used directly by 15 out of 40 categories. 13 of those categories using class are only using that selector from the ***Object Inspection - Accessing object class*** category.
- ***Class structural inspection - Traits***. Like slots, traits do not rely directly on primitive operations. They are implemented in the metaobjects of Pharo. In the reflective API, they are only used in the ***Class structural inspection - Class kind testing*** to test for users of a class defining a trait.

3.2.4 Client Categories

Eleven categories rely on other categories and are not used by others:

- ***Chasing and swapping pointers - Find pointers to***
- ***Class structural inspection - Pragma***
- ***Class structural modification - Anonymous class creation***

- ***Class structural modification - Class variable Modification***
- ***Class structural modification - Instance variable modification***
- ***Class structural modification - Old class creation***
- ***Class structural modification - Selector/Method modification***
- ***Memory Scanning - Instances of a class***
- ***Message sending and code execution - Arbitrary method/primitive execution***
- ***Object Modification - Object class change***
- ***Stack Manipulation - Context***
- ***Structural queries on methods - Method slot uses***

As there are eleven client categories, we focus on highlighting commonalities instead of detailing each of them. Five of them are from the bigger category: ***Class structural modification***. Client categories usually provide higher-level APIs, like ***Structural queries on methods - Method slot uses***. For example, in ***Class structural modification - Instance variable modification*** , the methods relying on other categories are the ones offering to add or remove instance variables by their names. Those methods hide the complexity of the implementation with slots, which is powerful but more complicated to understand. Another case is the old class creation API. These API methods have been rewritten to rely on the new fluid class builder. We believe that it has no users because of the migration to the new API. This API is only present for retro-compatibility and class creation required by other reflective APIs like slot modification using the new API.

3.2.5 Iterating and Querying Hierarchy, a Hub Category

In Figure 3.1, we identify one hub category in the top left that presents many more connections than the others. Figure 3.3 shows that this category appears to rely on three other categories for its implementation and has 16 categories using it directly. The ***Class structural inspection - Iterating and querying hierarchy*** is a hub category. In particular, its user with the strongest connection is ***Class structural inspection - Instance variable inspection***. The operations to iterate and query the class hierarchy are used to look up for instance variable definitions. The three categories it relies on are:

- ***Class structural inspection - Class kind testing***. The two messages isTrait and isMetaclassOfClassOrNil are used respectively in the implementations of includesBehavior: and subclassesDo: to check for specific cases in the Metaclass class.

- ***Class structural inspection - Class/Metaclass shift***. Both instanceSide and classSide are used by three Metaclass methods: subclasses, subclassesDo: and obsoleteSubclasses. Those three methods rely on the instance side to get the subclasses: instances of Metaclass have a parallel hierarchy to the instances of class. The subclasses of the class side are the same as the class side of the instance side's subclasses.

- ***Message sending and code execution - Runtime and Evaluation***. The messages value: and value:value: are used to evaluate blocks in eight methods, including five enumeration and iteration methods, and a method looking for a superclass verifying a criteria passed as a block. These are false positives due to polymorphism between reified messages and blocks.

3.3 Visualising Reflective Operations Intra-category Dependencies and Their Layered Organization

To analyze the dependencies within a single category, we build visualizations showing each selector's dependencies. All selectors belonging to the studied categories are in black while selectors from other categories have other colors. To get a more detailed view in this analysis, we keep all selectors including the one excluded previously.

The graph is laid out to show the hierarchy of dependencies with dependent selectors placed below the ones they depend on. Two selectors with a dependency relationship are placed as close as possible while respecting the vertical positioning. (For example, Figure 3.4 shows that sender is just above pointersToExcept:among: instead of being higher on the graph). This leads to the apparition of some visual layers (indicated by the blue dashed lines on the following figures).

In this section, we will study three cases corresponding to four categories: ***Chasing and swapping pointers - Find pointers to***, **Memory Scanning - Instances of a class** and the pair of ***Message sending and code execution - Runtime and Evaluation*** and ***Message sending and code execution - Reflective message send***. We chose these three case studies as these categories have numerous intra-category dependencies, a relatively low number of selectors to facilitate readability, and present a variety of dependencies graph shapes. Some categories, such as ***Chasing and swapping pointers - Bulk pointer swapping*** do not have any intra-category dependencies, others such as ***Class structural inspection - Iterating and querying hierarchy*** have a higher number of selectors, making the graphs less legible.

3.3.1 Chasing and Swapping Pointers - Find Pointers to

The category ***Chasing and swapping pointers - Find pointers to*** is an example of the layers that emerge in some reflective APIs. When looking at the graph in Figure 3.4

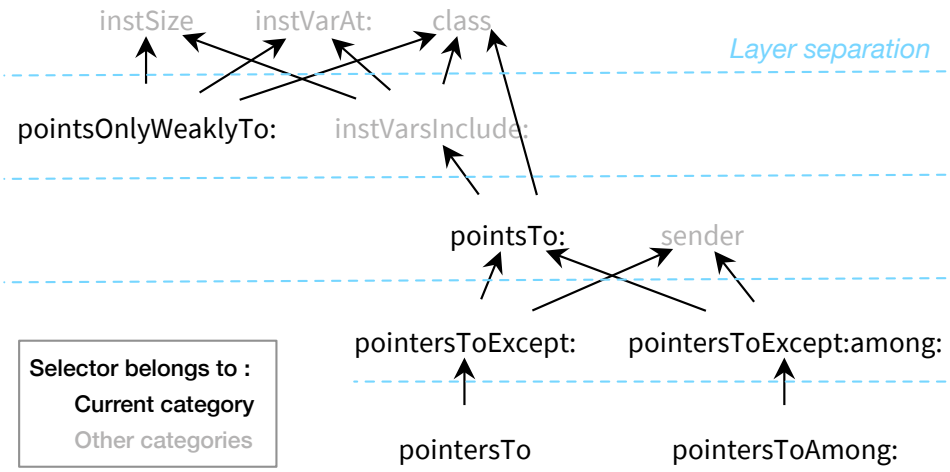


Figure 3.4: Subgraph of Chasing and swapping pointers - Find pointers to. The black selectors belong to the studied category.

we see that the method pointsTo: is the core one of this category, with all others except pointOnlyWeaklyTo: relying on it. This pointsTo: method tests for the presence of the parameter either as the class or in the instance variables of the receiver. By building on top of this method, we get more complex operations that allow one to get all pointers to an object.

We also notice that pointersTo: and pointersToAmong: rely respectively on pointersToExcept: and pointersToExcept:among:. The simpler APIs rely on the ones with more parameters for their implementations, therefore avoiding code duplication and facilitating code evolution. The presence of those simpler APIs in addition to the ones with more parameters reflects the absence of default parameters in Pharo.

pointsOnlyWeaklyto: is isolated in the category because it provides an independent operation: when calling it, a precondition is that the receiver is pointing to the parameter. This method tests if the references are weak or not. To do so it is using a lower level API, as information on the strength of references is not available at the abstraction level of pointsTo:.

3.3.2 Memory Scanning - Instances of a Class

The category **Memory Scanning - Instances of a class** is another example of a layered API. Here we got three leaf methods in the category allowing to access the instances of a class: allInstancesOrNil, someInstance, and nextInstance. More higher-level iteration methods and methods accessing sub-instances are built on top of those.

We notice in Figure 3.5 that allInstancesOrNil and allInstances look similar but do not depend on each other. This is due to both of them relying on the same primitive

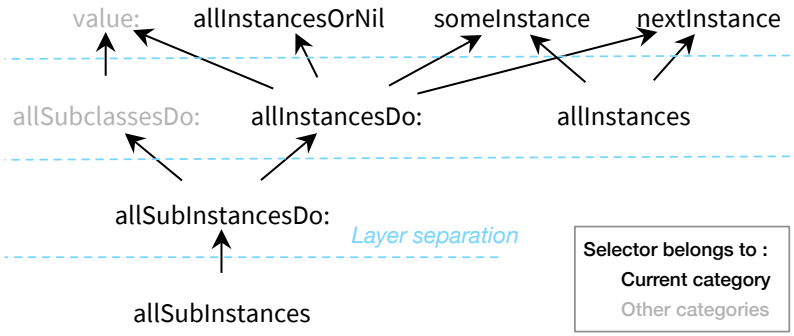


Figure 3.5: Subgraph of Memory Scanning - Instances of a class. Black selectors belong to the studied category.

operation. While allInstancesOrNil fails when running out of memory, allInstances has a backup implementation, which relies on someInstance to get the first instance of a class and nextInstance to iterate. The presence of both operations could be due to historical and retro-compatibility reasons.

Another noticeable point is the fact that while allInstancesDo: relies on allInstances, allSubinstances relies on allSubinstancesDo:. This is because allSubinstances requires iterating over instances of subclasses, and therefore if one wants to call a method on all subinstances, it is better to apply it directly rather than creating a new collection and then reiterate over it. This is an optimization for subinstances.

3.3.3 Message Sending and Code Execution - Runtime and Evaluation & Message sending and Code Execution - Reflective Message Send

Figure 3.6 shows in black the ***Runtime and Evaluation*** subcategory and teal operations of the ***Reflective message send*** subcategory. The former relies on the latter. Moreover, these categories are organized in different layers. In **Reflective message send**, the simpler APIs, with few arguments each as a different parameter, rely on the perform:withArguments: that takes an array of arguments in its second parameter. Finally perform:withArguments: itself relies on perform:withArguments:inSuperclass:. This is a similar structure as ***hashing and swapping pointers - Find pointers to*** category, seen in Section 3.3.1.

Runtime and Evaluation. contains value, cull: and their variants with one or more arguments. Variants of value assume that they are given the appropriate amount of arguments, while variants of cull ignore exceeding arguments. This leads to a horizontal layer in the API with all the variants of value relying on the perform: variant with the corresponding number of arguments. Variants of cull: rely both on

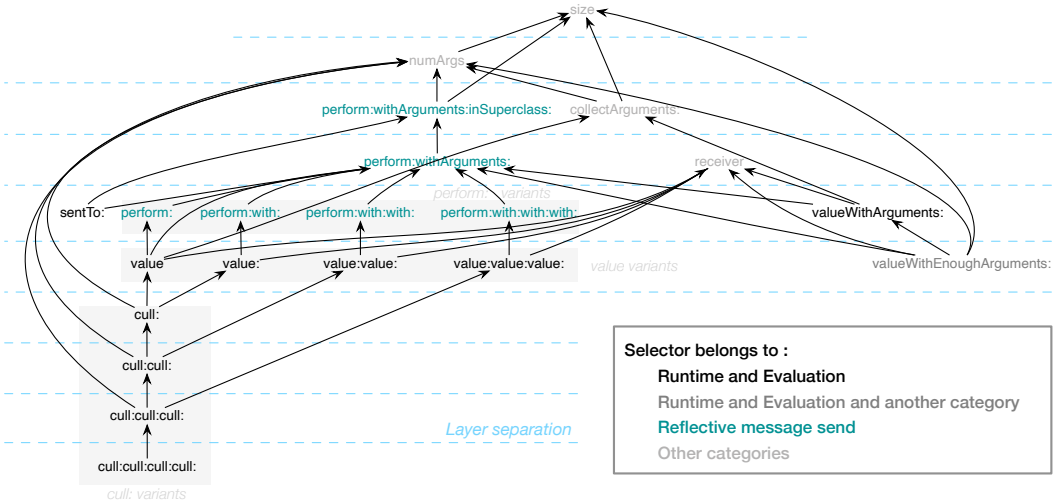


Figure 3.6: Subgraph of ***Message sending and code execution - Runtime and Evaluation*** and ***Message sending and code execution - Reflective message send***.

the value variant with the same number of arguments if there is the right number of arguments, and the cull: version with one less argument in case there are too many arguments. The cull: method with only one argument relies instead on the value message if no arguments are expected. This highlights the cost of using cull: variants when the number of arguments is known.

Once again here we observe that the absence of default arguments in Pharo leads to more methods being created to compensate. With empty default parameters, variants of cull:, value:, and perform: could be summarized by three methods with the maximum number of parameters.

3.4 Threats to Validity

By construction, this study is susceptible to false negatives on reflective methods as we are reusing the ones from Chapter 2, and both false positives and false negatives on dependencies.

False positives in dependency identification. Having base-level methods being polymorphic with reflective ones might lead to false positives during the static analysis. While we identified a few selectors (at:, at:put:, size and value) for which identified dependencies are not reliable, some others might have slipped through. This is due to the 44% of the reflective method's selectors that have also non-reflective implementors. While those that are implemented by and commonly used for collections can be identified as unreliable as collections are used to implement reflective methods, the others are harder to evaluate. We suppose that in the context

of reflective operation implementation, it is their reflective version that is used. If a reflective method actually calls the non-reflective version of a method with a selector not identified as unreliable, we will get a false positive dependency on the reflective version of the method. We choose not to exclude all 44% of reflective methods polymorphic with non-reflective ones, as this would exclude too much of the reflective APIs for our analysis.

In the dependency graph of selectors, we show selectors belonging to more than one category. However, we do not differentiate between the dependencies of methods belonging to different categories. This may lead to some false positives. For example, if a single method implementing the selector has a dependency, the categories of the other methods implementing this selector will show the dependency even if it is not their version of the method that is dependent.

False negatives in dependency identification. As we removed selectors belonging to several categories when drawing the category dependencies graph, some dependencies relationships are missing. However, only 13 selectors out of 344 are removed and they belong to varied categories. This leads us to the conclusion that no strong dependencies are going by unidentified.

One of the limits of the study is that we only look at direct dependencies between reflective methods. Therefore if a reflective method calls a non-reflective method, which itself calls a second reflective method, the dependency between both reflective methods will not be identified.

Non-formal definition of layers. While visual layers used in Section 3.3 offer a way to understand the hierarchy of dependencies, they do not have a formal definition. Slightly different visualizations of dependencies could generate other layers. However, as the organization of the selectors by the visualizations is specified, this makes those more reproducible.

3.5 Conclusion

In this chapter, we focused on the *inter-category dependencies* and *inter-method dependencies*. We identified chains of dependencies, which provide a starting point for the modularization of reflective APIs. Isolated categories such as *bulk pointer swapping* are interesting starting points, followed by client categories. Indeed, as those categories are not necessary for other reflective APIs, they could be removed or hidden behind a permission system without breaking the rest of the reflective APIs. For example for *bulk pointer swapping*, we identified the following uses: memory leak analyzers and dynamically updating existing instances to new class shapes (see Section 2.2.9). Both of those might not be necessary in a production environment. A deeper analysis of the kernel would be required, to identify which could actually be removed and which one would still be required.

In the next chapter, we will present RAPIM, our approach to dynamically identify dependencies to reflective operations inside an application. We will reuse the categories for this analysis. This could be later used on the Pharo kernel and combined with dependencies identified in this chapter to further study how the reflective APIs can be modularized.

CHAPTER 4

Mutation Analysis for reflection

Contents

4.1 RAPIM: Reflective API Mutation Analysis 52
4.1.1 Reflective Mutation Analysis Approach 53
4.1.2 Reflective Method Cancelling Operator 53
4.2 RAPIM by Example: a Developer Perspective 54
4.3 Evaluation of Reflective API Identification 59
4.3.1 Chosen Projects 59
4.3.2 Answering RQ1. MUTATION COMPARISON 60
4.3.3 Answering RQ2. EFFECTIVE POLYMORPHISM 62
4.4 Discussion 62
4.4.1 Limitations 62
4.4.2 Threats to Validity 63
4.5 Related Work for Mutation Analysis 63
4.5.1 Overcoming the Limits of Static Analyses 64
4.5.2 Program Analysis and Validation via Mutation Testing . . . 64
4.6 Conclusion 65

Although powerful, reflective features are usable as security exploits. For example, they allow malicious users to violate encapsulation and execute methods that were not intended to be executed [**?**, **?**, **?**]. This means that *developers must assess how much they rely on reflection*. It is important to understand if a reflective functionality is central to an application or if it is only confined to peripheral parts.

These problems become exacerbated in dynamically-typed languages. Reflective features defeat static analysis [**?**, **?**] because the actual attributes/methods that are being used are decided at runtime. Moreover, reflective APIs are often designed as normal methods and are often polymorphic with non-reflective operations. For example, in Javascript reflectively accessing an attribute (object['attribute']) is syntactically equivalent to array/dictionary accesses (dictionary[index]).

Chapter 2 illustrates the wide range of reflective operations available in Pharo and how they are a key part of the language kernel and libraries (See Section 1.5).

In Smalltalk and many of its derivatives, reflective facilities are mixed with the non-reflective API of objects and classes [**?**, **?**, **?**].

In this chapter, we propose to augment static reflection analysis with mutation analysis. In Section 4.1, we design reflection-specific mutations to obtain dynamic runtime information. We named this approach RAPIM. In Section 4.2, we use this information to assess the dependencies on reflective features and the degraded modes of an application. Then, in Section 4.3, we evaluate our approach on a selection of projects. We show that in four out of five projects RAPIM disambiguates more *potentially reflective call-sites* than static analysis. When the code coverage is good, the disambiguation rate is up to three times higher or more. We also challenge the relevance of polymorphism between reflective and non-reflective APIs based on those analyses. We show that the polymorphism is very rarely used, with only one project leveraging it for 1.4% of its *potentially reflective call-sites*. Section 4.4 presents the limitations of RAPIM and threats to validity. Finally, Section 4.5 summarizes related works.

The content of this chapter has been published in the following article:

- "Assessing Reflection Usage with Mutation Testing Augmented Analysis" Thomas *et al.* [**?**]

4.1 RAPIM: Reflective API Mutation Analysis

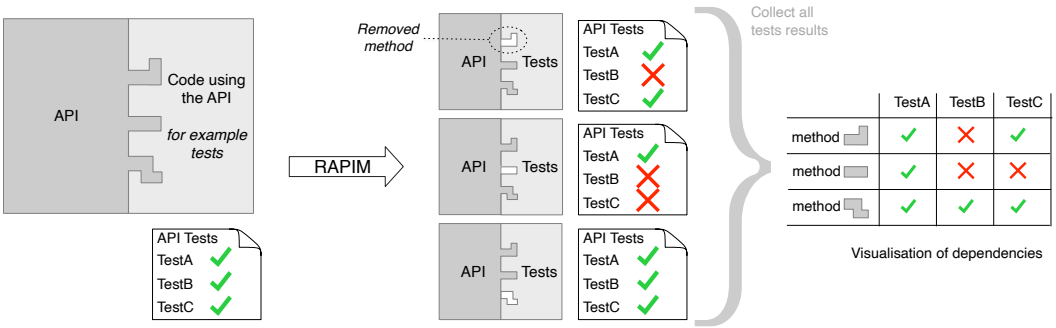


Figure 4.1: RAPIM: Using mutation analysis to assess reflection usage. For each reflective method removed, all tests are run and results are collected.

Modern development methodologies such as Agile Programming and Test-Driven Design [**?**, **?**, **?**, **?**] advocate the systematic use of tests. Tests are an automated way to verify an executable specification. While a suite of tests usually aims at testing various inputs, especially boundary values, and to cover as much code and paths as possible for the tested application, they are rarely randomly written but centered around the validation of application features. We propose to leverage these tests to assess dependencies to reflective operations.

Our approach, RAPIM, extends mutation analysis to assess reflective features use as illustrated by Figure 4.1. Mutation testing is a technique that allows one to evaluate the coverage of a test suite by modifying the code and checking if at least one test breaks. Indeed, if a test breaks, the test suite detects the modification, called a *mutant*. Here we remove reflective methods and we assess the impact by running the tests.

4.1.1 Reflective Mutation Analysis Approach

Mutation testing works by applying a non-semantic-preserving mutation and checking if at least a test breaks after such a change. We extended MUTALK, a mutation testing framework for the Pharo programming language, and designed a reflection-specific mutation operator to assess the use of reflection, explained in Section 4.1.2. This operator works on pairs of call-site and reflective methods to identify which one is called, if any. We rely on the list of reflective methods defined previously. We use this list to consider that a call-site is a *potentially reflective call-site*.

We use the list of reflective methods defined by Thomas et al. [**?**]. We use this list to consider that a call site is a *potentially reflective call-site*.

Since we want to provide a full assessment of the reflective API use, we configured the mutation framework to run all the tests covering a mutation instead of stopping at the first one that failed. This allows us to report the percentage of working tests after a mutation. Our operator fails tests using reflective features and this allows us to identify per test:

- **Surviving Mutant.** The code covered by the test does not use a specific reflective feature.
- **Killed Mutant.** The code covered by the test uses a specific reflective feature.

4.1.2 Reflective Method Cancelling Operator

For the mutation analysis, we want tests to fail when a specific reflective method is called from a specific *potentially reflective call-site*. To cancel a reflective method (so that calling it makes the current test fail), it is not sufficient to rewrite its call-site to invoke a different (potentially failing) method. Canceling a call-site in such a way will indeed fail the test when calling reflective methods, but will also fail when calling non-reflective methods. Remember that in Pharo, for example, the message at:put: is both implemented by collections (non-reflective setter) and the execution stack (reflective). Call-sites calling at:put: on collection should not fail.

To analyze the usage of reflection, we designed a reflection-specific operator that cancels all pairs (reflective method, call-site), one by one, taking into account for each *potentially reflective call-site* all reflective methods that could be invoked

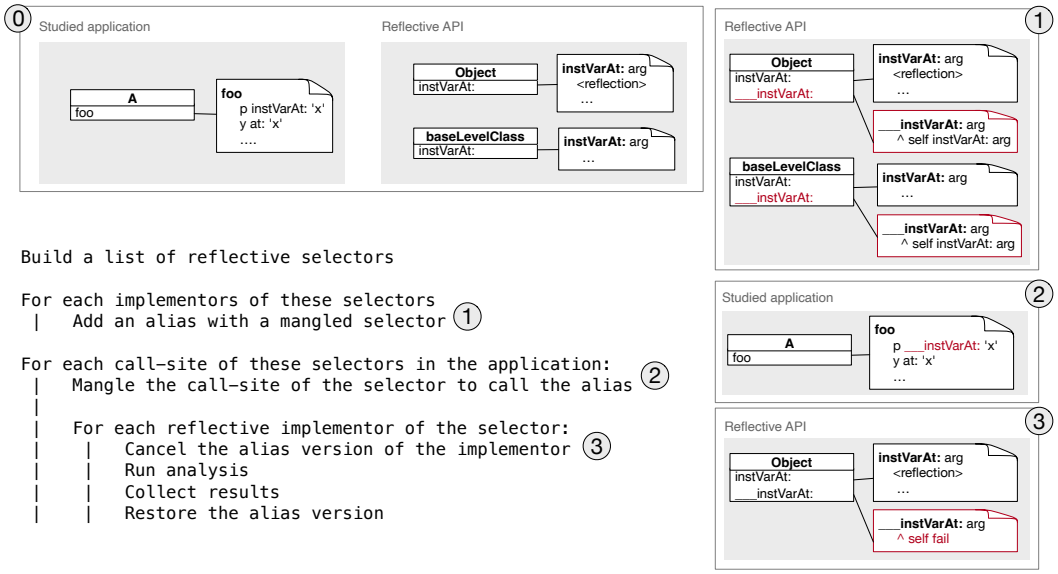


Figure 4.2: Code transformation for *Reflective method canceling*.

by that call site. We call this the *reflective method canceling operator*, illustrated in Figure 4.2. For each pair, the operator:

1. rewrites the *potentially reflective call-site* to an alias of the original method (see step 2, Figure 4.2)

2. introduces a canceled alias for the method (see step 3, Figure 4.2)

Then all tests covering this *potentially reflective call-site* are run. If the call-site calls the canceled reflective method the test fails, if it calls any other implementation, the code runs normally, with only an additional layer of indirection.

Our implementation further optimizes this approach by pre-installing the aliases for all implementors of reflective selectors instead of doing it for each call-site (see step 1, Figure 4.2).

4.2 RAPIM by Example: a Developer Perspective

In this section we illustrate RAPIM with a use case: Pharo's STON serialization framework. This application is built out of 14 classes, has 310 tests, and 54 *potentially reflective call-sites* present in the project. Through this use case, we show how it helps the developer better understand reflection and answer questions presented in 1.4. Here is a short summarized list of those questions:

- *General use.* How much does a project rely on reflection in general?

- *Faceted analysis.* How much does it rely on specific reflective features?

Percentage of:	Non covered	Refl.	Non-Refl.	Polymorphic
potentially reflective call-sites	13%	26%	61%	0

Table 4.1: STON's *potentially reflective call-sites* classification with RAPIM.

- *Spatial distribution.* Which application parts are impacted by a given reflective use?
- *Coupled uses.* Are there some reflective features that are often used together?
- *Degraded modes.* How much of the application still works without a given reflective use?

General use. Table 4.1 shows a first glance at the distribution of the 54 *potentially reflective call-sites*: 13% *(7)* of the call-sites are not covered by the tests, 26% *(14)* of the call-sites are only calling reflective methods and 61% *(33)* are only calling non-reflective methods.

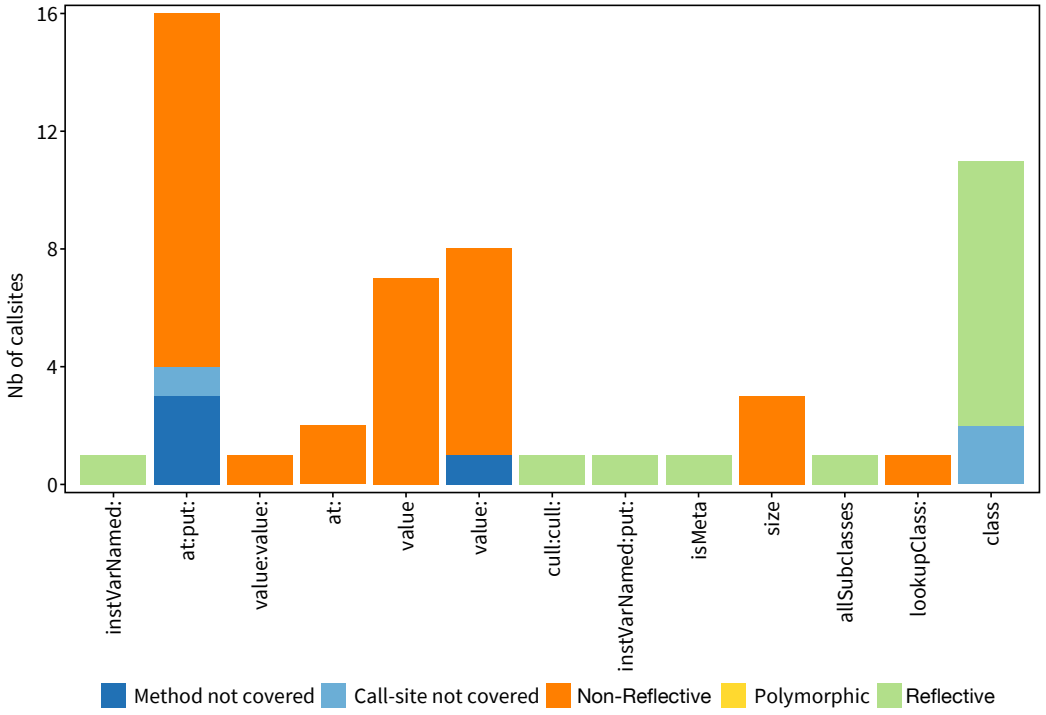


Figure 4.3: Proportion of call-sites types by selector for STON.

Faceted analysis. Figure 4.3 shows that out of the 13 selectors used by these call sites:

- 5 of them only have call-sites calling non-reflective methods (#value:value: #at: #value #size #lookupClass)
- 5 of them only have call-sites calling reflective methods (#instVarNamed: #cull:cull: #instVarNamed #isMeta #allSubclasses). Those have only one call-site each, which means that these uses of reflection are not too spread around in the application.
- #class has two out of eleven call-sites that are not covered by tests. However, a static analysis reveals that #class has only one implementor. Therefore, all of those call-sites are reflective too.
- All #at:put: and #value: covered call-sites are non-reflective, but some are not covered which means that we cannot conclude anything about them.

Spatial distribution and Degraded modes. The mutation matrix on Figure 4.4 gives fine-grained information on the same scenario: the percentage of broken tests in test classes if we actually remove each reflective method. STON relies on seven reflective methods. MySTONCStyleCommentsSkipStreamTest is the only test class whose tests do not use reflection. Assuming serialization is tested by MySTONWriterTest, it only relies on #instVarNamed: and #class which is expected. For deserialization, MySTONReaderTest requires more reflective methods, as it relies on six reflective methods (See Figure 4.4 for the details).

Table 4.1, Figures 4.3 and 4.4 highlight different levels of detail that lead to a better understanding of the dependency of a project on reflection. The matrix visualization provides also other levels, without grouping the tests by class for more fine-grained information (See Figure 4.6), or by grouping reflective methods by categories of reflective methods for coarser ones (See Figure 4.5).

Coupled uses. The most fine-grained matrix on Figure 4.6 displays which reflective methods are used by each test, allowing us to visually identify patterns. Each line corresponds to a reflective method that could be used by the application. We notice repeating patterns hinting at coupled uses. The blue squares on the first line corresponding to the uses of Object»#instVarNamed: seem to mostly coincide with the ones on the tenth line for using Object»#instVarNamed:put:. By counting the occurences, out of 61 tests using Object»#instVarNamed: and/or Object»#instVarNamed:put, 56 use both. Similarly, ClassDescription»#isMeta and Behavior»#allSubclasses are used by the exact same tests, suggesting that the same piece of code is using both.

In a later iteration of this implementation, metrics specific to the coupled uses could be introduced and automatically calculated.

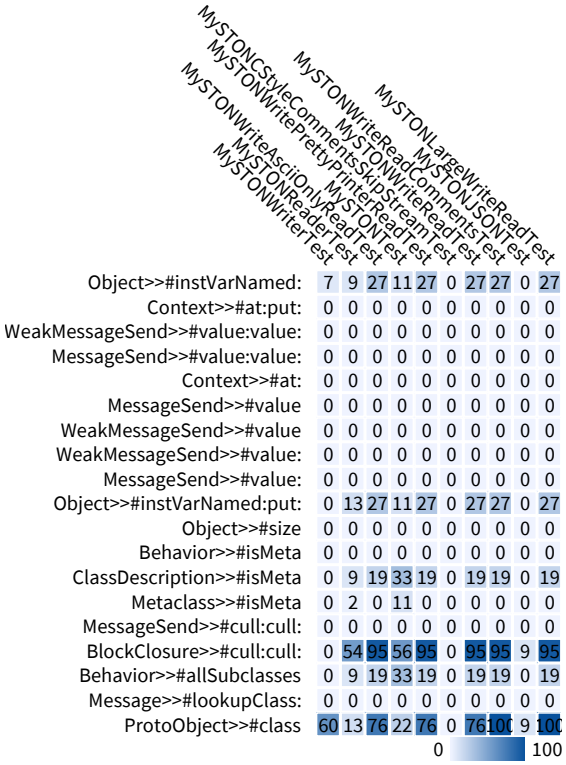


Figure 4.4: Table of the percentage of tests in a given class depending on a reflective method.

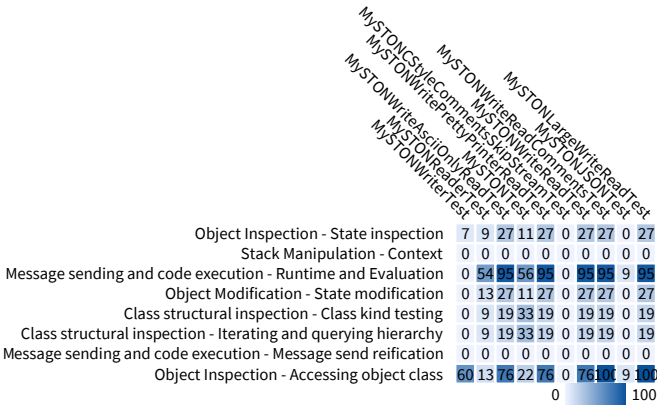


Figure 4.5: Table of the percentage of tests in a given class depending on a reflective category.

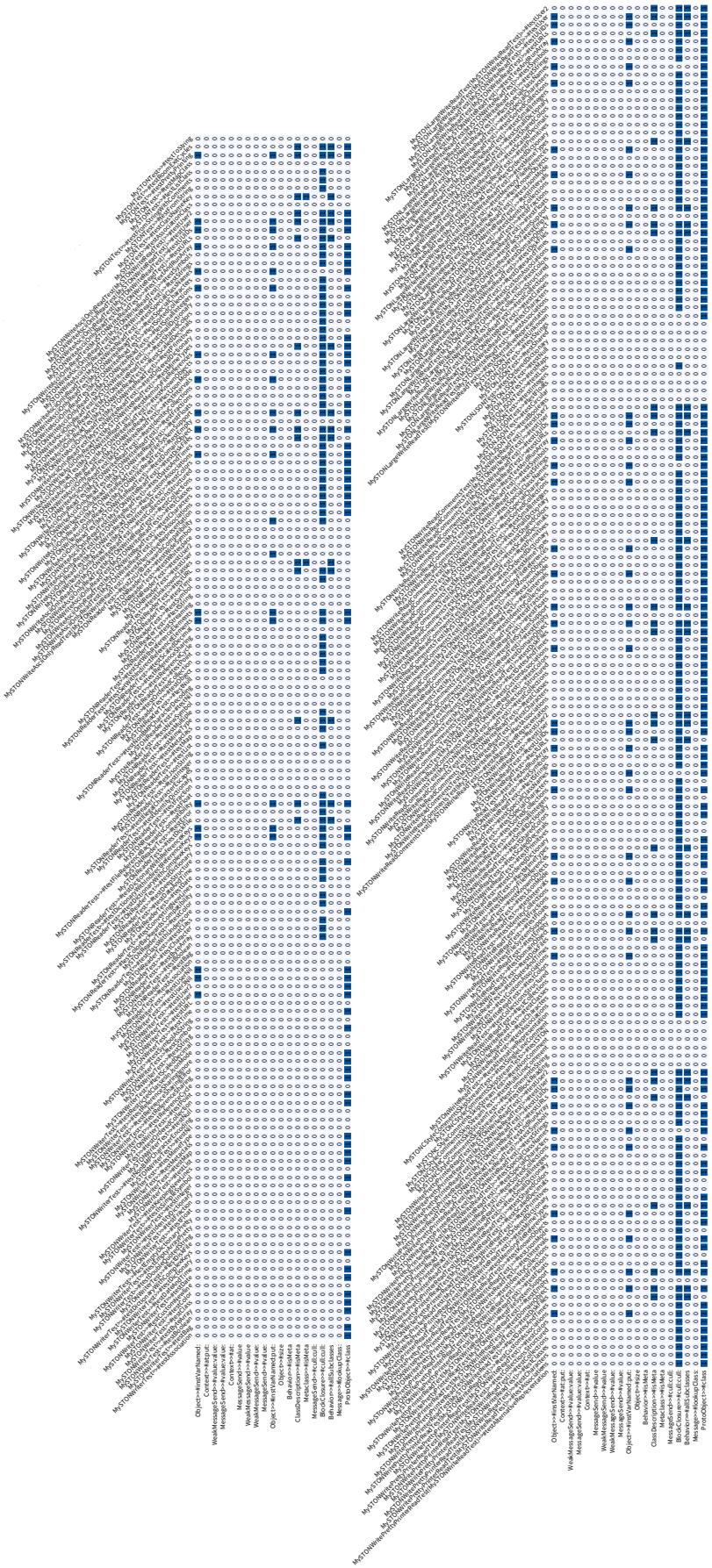


Figure 4.6: Table of Tests depending on a reflective method. A 1 (blue cell) means that the test has failed when this reflective method is removed.

4.3 Evaluation of Reflective API Identification

In the previous section, we reported how our analysis supports the developers in assessing the dependencies of the system to reflective features. In this section, we answer the research questions that drove our experiments:

- RQ1. MUTATION COMPARISON Does mutation analysis improve the detection of reflective API usages in comparison to static analysis?
- RQ2. EFFECTIVE POLYMORPHISM Is the polymorphism between non-reflective and reflective methods actually used by applications? Could we safely rename reflective methods to disambiguate *potentially reflective call-sites*?

4.3.1 Chosen Projects

Project	Url Link
STON	https://github.com/svenvc/ston
Microdown	https://github.com/pillar-markup/Microdown
Refactoring (a part of Pharo)	https://github.com/pharo-project/pharo
MuTalk	https://github.com/pharo-contributions/mutalk
Seaside	https://github.com/SeasideSt/Seaside

Table 4.2: Links to project repositories used for this analysis.

Following is the list of the projects we studied, and our expectations regarding the use of reflection in those projects. Table 4.3 summarizes the results of applying RAPIM on these projects. See Table 4.2 for links to repositories.

- *STON.* SmallTalk Object Notation. A textual object serializer/deserializer inspired by JSON. We expect reflection to be used for reading class information and instance variables for serialization and setting instance variables for deserialization.
- *Microdown.* A parser for a markup language derived from Markdown [**?**]. We do not expect many reflection uses in this application.
- *Refactoring.* The refactoring framework used in production by the Pharo development environment [**?**,**?**]. We studied both the *Refactoring-Core* and the *Refactoring-Transformations* packages. We expect a heavy use of reflection as it is manipulating methods and classes to perform code rewritings.
- *MuTalk.* A mutation testing framework, the same we use for this analysis. We worked on a copy of these packages to be able to execute our approach

Project	version	#classes	#tests	#call-sites	Non-cov.	Refl.	Non-Refl.	Poly.
STON	bbe8f5f	14	310	54	13%	26%	61%	0
Microdown	72f4ac7	168	583	576	77%	3%	20%	0
Refactoring	Pharo12 build.1386	211	815	1022	39%	7,4%	52,2%	1,4%
MuTalk	e712ac5	150	357	238	48%	11%	41%	0
Seaside	56286ac	547	1004	1314	74%	5%	21%	0

Table 4.3: Projects statistics and their *potentially reflective call-site* classification with RAPIM.

on its framework. We expect the use of reflection to edit methods to install mutations.

- *Seaside.* A web application framework maintaining sessions using continuations [**?**]. We expect the use of reflection in the continuation management.

4.3.2 Answering RQ1. MUTATION COMPARISON

Does Mutation Analysis improve the detection of reflective API usages in comparison to static analysis?

Static analysis classifies *potentially reflective call-sites* according to the implementors of the method called. If all implementors are reflective, the call-site is identified as a reflective call-site. Otherwise, if there is at least one non-reflective implementor, the call-site is ambiguous.

Table 4.4 shows the number of *potentially reflective call-sites* that static analysis identifies as reflective or ambiguous, and whether they are covered by tests or not. As explained in Section 1.6, reflective call-sites have only reflective implementations, and ambiguous call-sites' implementors have a subset that is reflective and a subset that is non-reflective.

Static analysis only identifies reflective call-sites for which all implementors are reflective. On the other hand, RAPIM analysis only includes covered call-sites. Ambiguous and covered *potentially reflective call-sites* are analyzed by RAPIM to identify the reflective ones. Non-covered reflective call-sites are not detected by the dynamic analysis of RAPIM because this approach is based on code coverage. The ambiguous and not covered call-sites are the ones neither approach can disambiguate.

As shown in Table 4.4, all projects have ambiguous *potentially reflective call-site* that are covered by tests. Those call sites cannot be disambiguated by static analysis, but RAPIM allows one to classify them. For example in the STON project, 87% of the *potentially reflective call-sites* are disambiguated by RAPIM, and 24% of

Project	Static analysis	Not Covered	Covered
STON	Statically Reflective	2 *(4%)*	11 *(20%)*
	Statically Ambiguous	5 *(9%)*	36 *(67%)*
Microdown	Statically Reflective	174 *(30%)*	16 *(3%)*
	Statically Ambiguous	272 *(47%)*	114 *(20%)*
Refactoring	Statically Reflective	108 *(10.5%)*	32 *(3%)*
	Statically Ambiguous	292 *(28.5%)*	590 *(58%)*
MuTalk	Statically Reflective	20 *(8%)*	15 *(6%)*
	Statically Ambiguous	95 *(40%)*	110 *(46%)*
Seaside	Statically Reflective	197 *(15%)*	779 *(59%)*
	Statically Ambiguous	52 *(4%)*	286 *(22%)*

24% disambiguated by static analysis

87% disambiguated by RAPIM

Table 4.4: Number of *potentially reflective call-sites* by projects, split by coverage and reflective ambiguity according to static analysis.

the *potentially reflective call-sites* are identified as reflective by the static analysis (See annotation on Table 4.4). Our analysis shows that 67% of the *potentially reflective call-sites* are ambiguous and covered. The 20% of statically reflective and covered call-sites are identified by both the static analysis and RAPIM. The 4% of not covered and statically reflective call sites are not taken into account by the dynamic analysis. Conversely, the 67% of ambiguous and covered call-sites are not disambiguated by the static analysis. The 9% remaining not covered and ambiguous call-sites are the ones neither approaches can disambiguate. The projects Refactoring and MuTalk have similar profiles.

The situation is slightly different in Microdown where the coverage of *potentially reflective call-sites* is lower (23%). There is only 3% of covered and reflective call-sites. This means that the overlap between static analysis and RAPIM is small. RAPIM disambiguates 20% of the *potentially reflective call-sites*. This is the lowest among the five studied projects. 47% of the call sites are never disambiguated. In addition, the static analysis disambiguates 33% of *potentially reflective call-sites* (which is the highest among our 5 projects) while RAPIM only 23%. This tilts the balance in favor of static analysis for these projects. The Seaside project exhibits the same profile, but the lower amount of statically reflective call-sites gives RAPIM the advantage.

Conclusion. On four out of five projects, RAPIM disambiguates more *potentially reflective call-sites* than the static analysis. When projects have good coverage (STON, MuTalk, Refactoring), our approach disambiguates **three times more** *potentially reflective call-sites* than the static analysis. For projects with low coverage, the disambiguation is on par with static analysis, but they present a high percentage of call sites that are still ambiguous.

4.3.3 Answering RQ2. EFFECTIVE POLYMORPHISM

Is the polymorphism between non-reflective and reflective methods used by applications? Could we safely rename reflective methods to disambiguate potentially reflective call-sites*?*

Table 4.3 shows that only one project uses polymorphism between reflective and non-reflective methods. This is expected because the refactoring engine has its meta-model of the code that mimics part of Pharo's reflective API [**?**]. Moreover, it mixes Pharo meta-objects with its own code model. Even in this case, Table 4.3 shows that polymorphic usage concerns only 1.4% of the *potentially reflective call-sites*.

Conclusion. Out of the five projects, only the refactoring engine leverages the polymorphism between reflective and non-reflective APIs. Even in this case, this is a very rare usage (1.4% of *potentially reflective call-sites*). This is **strong evidence** showing that reflective APIs could be re-designed to be mostly non-ambiguous in dynamically-typed languages.

4.4 Discussion

This section discusses the limitations of our approach and the threats to the validity of our evaluation.

4.4.1 Limitations

Code Coverage. RAPIM cannot disambiguate *potentially reflective call-sites* that are not covered. We saw in Section 4.3 that we get better results with a higher coverage. This is a limitation inherited from mutation analysis in general. However, our evaluation shows that when tests are available, mutation analysis complements static analyses.

Other meta-object. RAPIM only identifies dependencies to reflective methods. It does not cover access to global variables or stack reifications with the pseudo-variable thisContext. However, many methods that could be called on thisContext are identified as reflective methods (*i.e.* while accessing the reified stack is not detected by RAPIM, the use of many methods on the stack reification is detected).

Indirect dependencies. If the studied application relies indirectly on reflection (*i.e.* it uses a library that uses reflection), these dependencies will not be identified. RAPIM is designed to detect direct calls from the studied application to the reflective API scoped to a package.

Studying core packages. RAPIM cannot be used directly on the standard libraries of the Pharo programming language, as it could break the system. As in the case of MUTALK, this can be solved by creating copies of the studied packages and running RAPIM on this copy.

4.4.2 Threats to Validity

Internal Validity.

Code Coverage. Studying real-life applications is necessary to evaluate effective polymorphism, but our selection comes with a wide range of code coverage. Our analysis relies on the fact that tests cover both reflective and non-reflective cases. Having some projects with higher coverage mitigates the risk that reflective cases are overlooked in tests. The fact that we do not have a higher polymorphism percentage in those projects supports our conclusion.

Pharo exception handling. We left out of the analysis several tests (13) related to exception handling in the Seaside project. Running RAPIM on Seaside introduced a bug leading to infinite loops in these tests. The impact of removing this is mitigated by the amount of other tests as removed tests only represent 1,5% of the total tests one this project.

Construct Validity.

Studying core packages. RAPIM cannot be used directly on the core libraries of the Pharo programming language, as it could break the system. To study such packages (*e.g.* STON), we duplicate the package and run RAPIM on the copy. To ensure that results on the copy would be informative about the original package, we made sure that the duplicated version still runs well, and that all its tests are green.

External Validity.

Project selection for the evaluation. The evaluation of our approach relies on the results of RAPIM on five applications. To run this evaluation, we chose a set of various projects that we were expecting to use different quantities and features of reflection. We specifically aimed for variety to mitigate the bias introduced by selecting only a few projects.

4.5 Related Work for Mutation Analysis

This section presents related work in two different axes. First, the existing approaches to overcome the limitations of using only static analysis in dynamic languages. Then, the usage of mutation testing for program analysis and validation.

4.5.1 Overcoming the Limits of Static Analyses

Ruf [?] proposes the use of partial evaluation to detect the meta-level operations, once these meta-level operations are detected they are replaced by equivalent code as they are not present at run-time. Braux and Noyé [?] improve the results of static analysis for Java programs by applying partial evaluation to reflection resolution to apply optimizations. Their paper describes extensions to a standard partial evaluator to offer reflection support. The idea is to *compile away* reflective calls in Java programs, turning them into regular operations on objects and methods, given constraints on the concrete types of the object involved. The type constraints for performing specialization are manually provided.

Tip *et al.* propose Jax [?]: an application extractor and compactor for Java. In their solution, Jax performs a static analysis of the program and builds a call graph of the application. It then removes unused methods and compacts the application. It is also affected by the limitations of static analysis. To handle the missing information the authors propose three alternatives: (1) it requires user intervention to handle the dynamic loading and execution of code, (2) it performs a conservative selection of possible methods for a given call site, and (3) assumes that all methods in external library interfaces are called.

Bodden *et al.* [?] approach the limitations of static analysis by integrating static analysis tools with runtime information. Their tool inserts runtime checks into the code. These checks warn the user in case the program is performing reflective calls that were not identified by the static analysis. More recently, Liu *et al.* [?] improved the approach of Bodden *et al.* by taking benefit from runtime information obtained from code coverage. They automatically generate test cases to improve the code coverage and the detection of reflective call targets. Similar to us, this work uses runtime information, and more specifically tests, to augment static analysis. However, their objective differs: while they intend to obtain information on reflective call targets, our main goal is to understand the usage of reflective operations and de-ambiguate reflective from non-reflective operations in dynamic languages.

4.5.2 Program Analysis and Validation via Mutation Testing

Mouelhi *et al.* [?] propose using mutation testing to detect security issues. Loise *et al.* [?] based on this idea propose a series of mutation operators to detect specific security issues. They propose 15 mutation operators that are applied to Java programs. Using this technique they detect security issues that are ignored by static analysis. Our solution is not designed specifically for detecting security issues, but it is possible to identify misuses of reflective calls that introduce them.

Wen *et al.* [?] propose using mutation testing to detect misuses of library APIs. They represent API misuses as mutation operators applied in the code base. Then the mutant killing tests and their associated stack traces are collected to detect API misuses. We approach the identification of reflective usage in a similar way showing

that a single mutation operator is enough for reflective usage analysis.

4.6 Conclusion

In this chapter, we started from the reflective APIs identified in Chapter 2 to build RAPIM, a mutation analysis approach based on a new mutation operator to understand dependencies on reflective APIs. This approach handles the fact that reflective features are often a core part of a language (Python, Pharo...) and cannot be simply pulled off. We then present the identified dependencies with several levels of details for STON, the object serialization library used in Pharo.

To evaluate our approach, we used RAPIM on five different projects relying on reflection. On four out of five projects, RAPIM disambiguates more *potentially reflective call-sites* than the static analysis. When projects have good coverage (STON, MuTalk, Refactoring), our approach disambiguates three times more *potentially reflective call-sites* than the static analysis. For projects with low coverage, the disambiguation is on par with static analysis, but they present a high percentage of call sites that are still ambiguous.

Out of the five projects, only the refactoring engine leverages the polymorphism between reflective and non-reflective APIs. Even in this case, this is a very rare usage (1.4% of *potentially reflective call-sites*). This supports the idea that reflective API could be renamed to avoid accidental polymorphism with non-reflective methods.

We believe this approach applies to domains other than reflection and could help to sort out critical dependencies and further security analysis, such as dependencies to file access APIs or user inputs.

In the next part, we will shift our focus from analyzing reflection and its uses to how to control it, using the knowledge acquired in this first part to inform our decisions.

Part II

First Steps in Control for Reflection

CHAPTER 5

State of the Art for Controlling Reflection and Visibility Modifiers

Contents

5.1 Controlling Reflection 70
5.1.1 Mirror Architecture 70
5.1.2 Ownership-Based Control 71
5.1.3 Secure MOP 71
5.1.4 Optional Reflective Features 72
5.1.5 First-Class References 72
5.1.6 Reflectogram Reification 72
5.1.7 External Reflection 73
5.2 Existing Visibility Modifiers 73
5.2.1 Ruby 74
5.2.2 PHP 75
5.2.3 Python 75
5.2.4 Javascript 75
5.2.5 Statically-Typed Languages: Java, C#, and C++ 76
5.3 Object Encapsulation 76
5.3.1 ConstrainedJava 76
5.3.2 MUST 76
5.3.3 Encapsulation Policies and Visibility Semantics 76
5.4 Conclusion 78

As seen in the introduction, reflective methods are powerful operations allowing to alter language architectures and build tools such as debuggers and IDEs. However, as they can bypass many restrictions, they can lead to issues such as breaking encapsulation, breaking proxies and membranes, arbitrary code execution, memory scanning for forging references...

While in the first part of this thesis, we focus on understanding the available reflective operations and their uses, in this second part we focus on controlling

reflection and the relevance of using protected modifiers to do so. This chapter presents the existing state of the art regarding both the control of reflection, visibility modifiers, and encapsulation.

In Section 5.1 we present existing ways to control reflection and put them in perspective with Pharo. This notably includes layered designs using mirrors and packaging unnecessary reflective methods outside of the standard library.

In Section 5.2 we discuss existing visibility modifiers. The protected modifier supports encapsulation [**?**, **?**] while still allowing overriding [**?**, **?**]. Although the protected method modifier is present in many mainstream object-oriented languages, their semantics are slightly different in each of them. It generally conveys that a method is hidden from clients and accessible from subclasses, several variations exist in mainstream languages. Their exact visibility semantics, redefinition semantics, and enforcing mechanisms vary. In Section 5.2 we provide a comparison of modifier semantics across a selection of object-oriented languages (Java, C++, C#, Ruby, Ruby, PHP, Python, JavaScript).

Today, most statically-typed object-oriented languages such as Java, C++, and C# provide relatively good support for module encapsulation, and many proposals have been made for augmenting the static type systems of such languages so that they can also express object encapsulation [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**]. Note that none of these was adopted into an existing mainstream language. Adding a protected method mechanism is closely tied to the implementation of encapsulation, therefore Section 5.3 reports the work related to encapsulation, especially in dynamically-typed languages.

Some of the content of this chapter has been published in the following article:

- "A VM-Agnostic and Backwards Compatible Protected Modifier for Dynamically-Typed Languages" Thomas *et al.* [**?**].

5.1 Controlling Reflection

Controlling the run-time behavior of reflective facilities introduces several challenges, such as computational overhead, the possibility of meta-recursion, and an unclean separation of concerns between the base and the meta-levels [**?**, **?**]. In this section, we present some existing ways to control reflection.

5.1.1 Mirror Architecture

At the language level, Mirrors [**?**, **?**] aims for stratification of meta-level facilities and gives access to reflective capabilities based on a reference to a mirror factory. Mirrors were implemented in several languages, for example, Self [**?**], StrongTalk [**?**], Newspeak [**?**], and AmbientTalk [**?**].

In [?] the authors advocate that mirrors should also address structural decomposition. Mirrors should not only be the entry points of reflective behavior but also be the storage entities of meta-information.

Pharo offers a first implementation of Mirrors (with APIs to read/write fields, check the class and the identity of an object, and execution of a method with another receiver) with the class MirrorPrimitives. It is not systematically used and adds another mechanism to the existing reflective API. Given that MirrorPrimitives is a class, it is registered in the global environment, making access to this facility possible from anywhere. This defeats the idea of restricting access to reflection through the use of references to mirror factories.

5.1.2 Ownership-Based Control

Teruel *et al.* [?] devised a fine-grained access control policy. This policy allows one to specify which objects can use which reflective operation and on which target object. These permissions are based on ownership.

Using ownership allows the control policy to take into account dynamic relationships between objects rather than static criteria like classes or packages. The notion of object ownership was previously studied [?, ?, ?] and comes from the idea that in practice objects are not autonomous but work in aggregates.

In Teruel's ownership system, each object has a reference to its *direct owner* which is by default the object that created it. This creates an *ownership tree*. Depending on the execution of the host model the root of this tree can be either the first object instantiated or a special object. The *ownership* relationship is the transitive reflexive closure of the *direct ownership relationship*. This transitivity of ownership comes from the fact that reflective operation could be used to access objects transitively anyway. The control policy grants an object the permission to perform any reflective operations on an object it owns. If an object does not own the other one, the only reflective operations available are those that could have been done without reflection (*e.g.* sending a message reflectively).

The root of the *ownership tree* has the permission to perform any operation on any object. Therefore, it can be used to develop tools and dynamic analysis requiring access to reflective operations on the whole system.

5.1.3 Secure MOP

According to Caromel *et al.* [?], Meta-Object Protocols (MOP) can be compatible with security. They implemented a MOP for Java with proxies that differentiate the base level from the meta level. In this MOP, the meta level has all permissions, while the base level has restricted ones. This MOP relies on a Java built-in mechanism for security context that makes sure that permissions are preserved when a reified call crosses the boundary between the base level and meta level. Thanks to this property, Caromel *et al.* prove that this MOP makes the meta level transparent to the

base-level code in terms of security. This work was however not further evaluated and not used by mainstream languages.

5.1.4 Optional Reflective Features

In Java, many reflective operations are packaged in an optional package of the JDK. The documentation [?] identifies the "components of core reflection" as the content of the java.lang.reflect package as well as the classes Class, Package, and Module. This limits the reflective operations available by default in the language.

As of today, in Pharo, all reflective APIs are part of the standard library and available by default. However, the first part of this thesis identified some reflective behavior that could be packaged outside of the standard library. We suggest continuing this effort to obtain a minimal core with modular and optional extensions. This is particularly interesting because the expectations of a development session should be automatically the same as the one of a deployed application. Of course, this is handy to apply reflective actions on deployed applications to debug them, but this is important that MOP designers consider other scenarios such as more constrained application deployment setup.

5.1.5 First-Class References

Arnaud et al. [?] [?] designed first-class references, called handles, allowing instance-specific behavior on a per-reference basis. These handles allow one to create read-only references, revocable references, and software transactional memory. These handles have per-reference behavior shadowing, state shading, and control propagation for objects accessed through the reference. They are implemented at the VM level. This prevents them from being corrupted or tampered with through reflective operations.

These could be used to limit the reflective operations available on a given object reference. This provides a very fine-grained way to control available methods. However, in Pharo, this requires a specific runtime and cannot control reflective operations available on classes such as MirrorPrimitives as those are global variables.

5.1.6 Reflectogram Reification

The reflectogram represents "the control flow between the base level and the meta level during execution" [?]. Papoulias *et al.* [?] proposes a model for the reification of this concept.

They present five dimensions of meta-level control. The temporal control is the ability at run-time to add or remove reflective facilities. The spatial control is the ability to scope the reflection to a set of entities. The placement control allows one to control when a triggered action is executed in relation to the semantic trigger event (*e.g.* accessing a variable). Level control allows one to have different behaviors for

different meta levels. Finally, identity control is the ability to differentiate the target of the reflective operation from the receiver of the reflective message.

This reification is implemented for Pharo and for a specific VM via a VM extension. This allows one to unify the control of reflection.

5.1.7 External Reflection

Another approach is to separate the implementation of the language from the reflective API. Lorenz proposes a pluggable reflection [**?**], in which the reflective API should be external to the language. This solution allows tools using reflection to process information coming from different sources (source code, live environment) as long as the same external API is available. This solution aims at flexibility and interoperability. While this might seem to not fit the model of the Smalltalk/Pharo live environment with all its embedded tools, having such an external reflective API could allow remote debugging while removing the reflection inside the image. Again, this solution remained at the stage of a prototype and it was unclear how such design can be implemented in a real system.

5.2 Existing Visibility Modifiers

Method visibility modifiers such as public, protected, and private are present in mainstream statically-typed object-oriented languages such as Java, C#, and C++. These modifiers specify if and how a method should be used in other parts of an application. It is well-known that some methods may be defined for internal usage only and therefore should be differentiated from the ones that are available publicly [**?**, **?**, **?**, **?**]. Visibility modifiers also define whether a method can be overridden.

In dynamically-typed object-oriented languages, visibility modifiers are not as common and not as mature. Python implements private methods through name-mangling and does not enforce protected methods [**?**]. Ruby dynamically enforced access modifiers have changed their semantics recently in 2019 [**?**]. In 2022, private fields were included in the standard of ECMAScript [**?**]. All methods are public in Smalltalk and several of its descendant [**?**, **?**, **?**].

Protected method modifiers in their general form offer two interesting facets: (1) they hide methods from external objects while allowing the class and its subclasses to invoke protected methods, and (2) they authorize the redefinition in subclasses to support reuse [**?**]. This dual aspect makes them an interesting concept that fits well with late-bound object-oriented languages. This raises the question of their introduction in object-oriented dynamically-typed languages (*i.e.* without static type checking).

Table 5.1 summarizes the semantics of the protected modifiers for some mainstream languages regarding three aspects: their visibility semantics, visibility enforcement mechanism, and whether narrowing the visibility is allowed. The en-

forcement mechanism, if there is one, is the mechanism that imposes respecting the visibility rules to the developer. Narrowing visibility means reducing it in subclasses. For example, if a method can be public in a superclass and protected in a subclass, then narrowing is allowed.

Visibility semantics vary in each language. In the following sections, we first focus on Ruby's method visibility modifiers, as it is one of the few dynamic languages offering them. We then discuss Python, which uses name mangling for private modifiers. Then, we present method encapsulation in Javascript and Java, C#, and C++. Finally, we discuss related work on encapsulation in dynamically-typed languages.

	Solution	Visibility	Enforcement	Narrowing
static	Java protected	hierarchy, package	compile time	Forbidden
	C++ protected	hierarchy, friends	compile time	Forbidden
	C#protected	hierarchy	compile time	Forbidden
dynamically-typed	Smalltalk	all send-sites	N/A	N/A
	Ruby private	hierarchy using self or implicit receiver	run time	Allowed, checked at run time
	Ruby protected	hierarchy	run time	Allowed
	PHP private	class	run time	Forbidden
	PHP protected	hierarchy	run time	Forbidden
	Python private	hierarchy	static (mangling)	N/A
	Python protected	class	convention only	N/A
	JavaScript	class	run-time	N/A

Table 5.1: Accessibility of methods according to visibility modifiers in different languages. N/A means "Not Applicable"

5.2.1 Ruby

Ruby is one of the few dynamic languages that offers method modifiers: methods can be qualified as public, protected, or private

Ruby *syntactically* distinguishes private from public methods [**?**]. Modifiers can be changed in subclasses: a private method may be made public in subclasses opening further the API, or a public method can be restricted to protected or private leading to errors when not called properly. If a subclass uses a stricter modifier, its instances can not be used polymorphically with instances of the superclass.

Private. In Ruby, private methods can only be called in the class and its subclasses, and only if the receiver is syntactically self or the implicit receiver. In addition, a private method can also be overridden in subclasses. Originally, private methods in Ruby could only be invoked by messages to the *implicit* receiver (*i.e.* no *self*),

restriction removed in Ruby 2.7. Calls to private methods are not statically bound and can be overridden in subclasses. The visibility is checked and enforced dynamically with flags on the methods.

Protected. Ruby's protected methods can be invoked by sending a message from the same class where it has been defined or from its descendants. The receiver can be implicit, self, or another instance from the same family. This means that instances of sibling classes can call protected methods defined in a common ancestor on each other.

5.2.2 PHP

PHP supports also three visibility modifiers which are public, protected, and private. In PHP, private methods can only be invoked from the class where it is defined, but they can be invoked on another instance of the same class. This private semantic is class-based, rather than self-send-based like Ruby's. Note that the official description of the language semantics is rather vague [**?**]. We looked into the PHP Zend Virtual Machine [**?**] and found that the visibility mechanism is based on method flags as in Ruby.

5.2.3 Python

Private. Python supports public and private visibility modifiers through name mangling [**?**]. By default, all attributes and methods are public, and the addition of a double underscore in front of their names marks them as private. Private attributes and methods are mangled to include the name of the class before code generation. Each send-site inside a class using a private selector is rewritten using the mangled name, restricting the visibility to within the same class. private methods and attributes are not acessible in subclasses.

Protected. Protected methods are marked by convention with a simple underscore, but are not enforced.

5.2.4 Javascript

In 2022, private fields were included in the standard of ECMAScript [**?**] Private fields are prefixed with a # and accessible only from inside the class defining it, encapsulating both properties and methods. Referring to # names outside the scope results in a syntax error at run time.

5.2.5 Statically-Typed Languages: Java, C#, and C++

In Java, C#, and C++ visibility modifiers are based on the type of the sender and receiver as well as on which package/assemblies they are defined in, they are not based on the identity of each instance. All three support the redefinition of virtual protected methods as public methods in subclasses.

C++ protected modifier has a visibility that is partially independent of the hierarchy. The notion of friend allows one to share the visibility of certain methods and attributes with specific external classes and/or methods outside the hierarchy while hiding them from the other external classes and methods. This gives more flexibility to the developers.

5.3 Object Encapsulation

5.3.1 ConstrainedJava

In ConstrainedJava [**?**], the authors extend BeanShell, an extension of Java, to explore dynamic ownership in the context of a dynamically-typed language. Their model for dynamic ownership provides alias protection and encapsulation enforcement by maintaining a dynamic notion of object ownership at run time. It places restrictions on messages sent between objects based on their ownership. Their model further classifies message sends as *internal* (if the receiver of a message is `this` or is owned by `this`), or *visible* (sends to other visible objects). In addition, Dynamic Ownership recognizes two kinds of externally independent messages — pure messages that do not access the object state, and one-way messages that do not return results.

5.3.2 MUST

In the programming language MUST [**?**], methods can be private (visible only in the current class with `here`), public (visible everywhere), subclass-visible (callable with `super`), superclass-visible (callable with `self`), or both latest visibility. This encapsulation is based on a syntactic distinction: message-sends to any object, to `self`, to `super`, or to `here` are treated differently. This approach is revisited by Schärli *et al.* as explained thereafter.

5.3.3 Encapsulation Policies and Visibility Semantics

Schärli *et al.* proposed encapsulation policies as a way to constrain the interface of an object [**?**]. With Object-Oriented Encapsulation (OOE), two cases are distinguished: (1) an inheritance perspective where a class changes the way the superclass methods are bound from the subclass perspective and (2) an object perspective where the interface of an object itself is changed by associating encapsulation policies

with object references. From the inheritance perspective, an encapsulation policy associated with a subclass changes how methods in the superclass are bound.

Schärli *et al.* define three different rights to define the encapsulation of a method: the right to **{o}**verride, to r**{e}**-implement, and to **{c}**all a method. OOE semantics are also based on the syntactic distinction of three different messages: super-sends, self-sends, and object-sends. Only self-sends can be early bound.

Schärli *et al.* [**?**] studied different semantics for a modifier's visibility: based on static types, class-based, identity-based, and self-send based. Schärli's analysis considers only locally-bound private methods, but the different syntactic options are still relevant for protected methods.

Static type. The first option relies on a static type system. In terms of scope, a visibility modifier can be implemented to restrict access to instances of a single class. However, the static type requirement is opposed to the nature of the dynamically-typed languages that we are targeting.

Dynamic Class-based. The second option is a class-based modifier: the visibility of a method is assessed at run time according to the class of the sender object and the class where the method has been defined. This can be used to restrict visibility to a single class, or a class hierarchy (the class where the method is defined and its subclasses). With this option, overriding a public method as protected makes reasoning on the program execution more difficult.

Dynamic Identity-based. The third option is an identity-based modifier: a sender object can send a protected message *only to itself*. The receiver's identity of a protected method is compared at run time with the sender's identity. Such semantics bring subtle differences in method executions that may make programs difficult to predict. It can lead to a program working with some instance of a class and failing with other instances of the same class. In addition, using the same selector on the same object can produce a different result if a protected method is accessed in one case, and a public method for the same selector in another case.

Self-send-based. The final option is self-send-based visibility: protected messages can only be sent to `self`. As this does not rely on dynamic information on the receiver, it allows one to decide at compile time which message-send sites can access methods with restricted visibility. Self-send-based visibility makes it easier for the programmer to understand quickly where protected methods can be accessed

and where they can not. It makes reasoning on code easier.

Schärli *et al.* still point out some issues with symmetry properties. For example, self = arg might not be equivalent with arg = self even if arg and self point to the same object. This happens for the same reason as the problems with the two previous visibility semantics, but at least, in this case, the developer can identify syntactically whether a protected method will be invoked.

5.4 Conclusion

In this chapter, we have presented existing ways to control reflection, visibility modifiers and encapsulation semantics. In the next chapter, we will present PROTECTEDLITE, a protected visibility modifier and its implementation in Pharo. This is inspired by several of the modifiers and semantics we discussed in this chapter.

The model uses the same distinction between object-sends and self-sends as Schärli *et al.* [**?**] (This technique has also been chosen in [**?**, **?**].)

The closest semantics is the one from Ruby's private methods: Ruby's private methods can only be called in the class and its subclasses, and only if the receiver is syntactically self or the implicit receiver. In addition, a private method can also be overridden in subclasses.

The implementation relies on a similar name-mangling process as Python's protected methods. Self and super-sends messages are prefixed and protected methods are only registered with the prefix.

CHAPTER 6

Protected Modifier

Contents

6.1 PROTDYN: A Protected Modifier Model 80
6.1.1 Properties and Chosen Semantics for a PROTDYN 80
6.1.2 Object-Send and Self-Send Lookup Semantics by Example . 81
6.1.3 Changing Visibility in Subclasses 84
6.1.4 PROTECTEDLITE: PROTDYN Semantics 85
6.2 #PHARO Implementation 87
6.2.1 Design Principles 88
6.2.2 Implementation Overview 88
6.2.3 Double Public Registration 89
6.2.4 Selector Mangling for Self-Sends Sites 89
6.2.5 Preventing Selector Mangling Propagation to the Whole System 90
6.3 Discussing Alternative Implementations 91
6.3.1 Lookup Mechanism Modification 91
6.3.2 Run-Time Visibility Checks 92
6.4 Conclusion 92

In object-oriented languages, method visibility modifiers hold a key role in separating internal methods from the public API. Protected visibility modifiers offer a way to hide methods from external objects while authorizing internal use and overriding in subclasses. In the previous chapter, we presented the visibility modifiers in several languages: Smalltalk, Ruby, Javascript, Python, PHP, Java, C++, and C#. We discussed their varying semantics and enforcement mechanisms.

Protected method modifiers in their general form offer two interesting facets: (1) they hide methods from external objects while allowing the class and its subclasses to invoke protected methods, and (2) they allow redefinition in subclasses to support reuse [**?**]. This dual aspect makes them an interesting concept that fits well with late-bound object-oriented languages. This raises the question of their introduction in object-oriented dynamically-typed languages (*i.e.* without static type checking).

Section 6.1 presents PROTDYN, a visibility modifier model for dynamically-typed languages. It is calculated at compile time and relies on name-mangling and syntactic differentiation of self-sends (self doSomething) vs non-self-sends(anObject doSomething). This choice of using self-sends vs non-self-sends follows Schärli *et al.* analysis [**?**, **?**] as seen in the previous chapter. It offers a VM-agnostic and backward-compatible design to introduce protected semantics in dynamically-typed languages. We define PROTECTEDLITE, a corresponding extension of the formal semantics SMALLTALKLITE [**?**] expressing the PROTDYN model.

In Section 6.2, we present #PHARO, a PROTDYN implementation of this model that is backward-compatible with existing programs and computes method visibility at compile time. This implementation is (1) optionally loadable, (2) does not require any changes to the default method lookup supported by a virtual machine, and (3) is backward-compatible with existing code. A Python version was implemented demonstrating the applicability to other languages [**?**].

Section 6.3 highlights possible alternative implementations and discusses the reasons behind our choices.

The content of this chapter has been published in the following article:

- "A VM-Agnostic and Backwards Compatible Protected Modifier for Dynamically-Typed Languages" Thomas *et al.* [**?**].

6.1 PROTDYN: A Protected Modifier Model

In this section, we discuss the chosen semantic and introduce an informal description of PROTDYN, a `protected` modifier model based on the syntactical differentiation of object-sends and self-sends [**?**]. Then we provide examples to explore the ramifications of this model. We highlight why we chose to forbid reducing the visibility of a method in a subclass. Finally, we present a formal definition of this model.

6.1.1 Properties and Chosen Semantics for a PROTDYN

Based on our previous analysis of existing visibility modifiers, we want a protected modifier with the following properties:

- A **protected** method is visible from a message-send site if the receiver is the same as the current method receiver (*i.e.* being of the same class is not enough).
- A **protected** message-send site is statically determined using syntactic information.
- A **protected** method is overridable from the subclasses of the defining class.

We chose a self-send-based visibility semantics as it creates fewer ambiguities than the identity-based one (See section 5.2).

ProtDyn in a nutshell. ProtDyn is a self-send-based visibility model where:

- A **protected** method is visible from a self-send site.
- A **public** method (*i.e.* non-protected) is visible from all send sites (object-send and self-send sites).
- A **protected** method is overridable from the subclasses of the defining class.

Notice that this model trades off programmers' flexibility for a compile-time mechanism based on syntactic information. Protected methods are never visible from a non-self send (anObject doSomething), even though the receiver object may be identical to the current receiver (anObject == self). For example, in a statement sequence such as:

```
temp := self. temp foo
```

... the foo send will have more restricted visibility than directly doing self foo. However, syntactic differentiation is a simple rule that can be easily learned by developers and makes program understanding easier [**?**].

We say this semantics is self-send-based because it does not take into account lexical scoping such as the class, package, or namespace as a design choice. It only relies on the fact that the receiver is syntactically self/super or not. We also add the following rule:

- A **public** method can not be overridden by a **protected** method in any subclass.

This rule supports subtyping between a class and its subclasses. In addition, it ensures that symmetry issues will not arise. For example self = arg and arg = self will give the same result as long as arg and self point to the same object.

6.1.2 Object-Send and Self-Send Lookup Semantics by Example

In this section, we will go over a series of examples to better understand how PROTDYN semantics works. As it differentiates send sites by their syntactic receiver, object-sends and self-send behave differently:

- Object-sends see only *public* methods.
- Self-sends see *public and protected* methods.

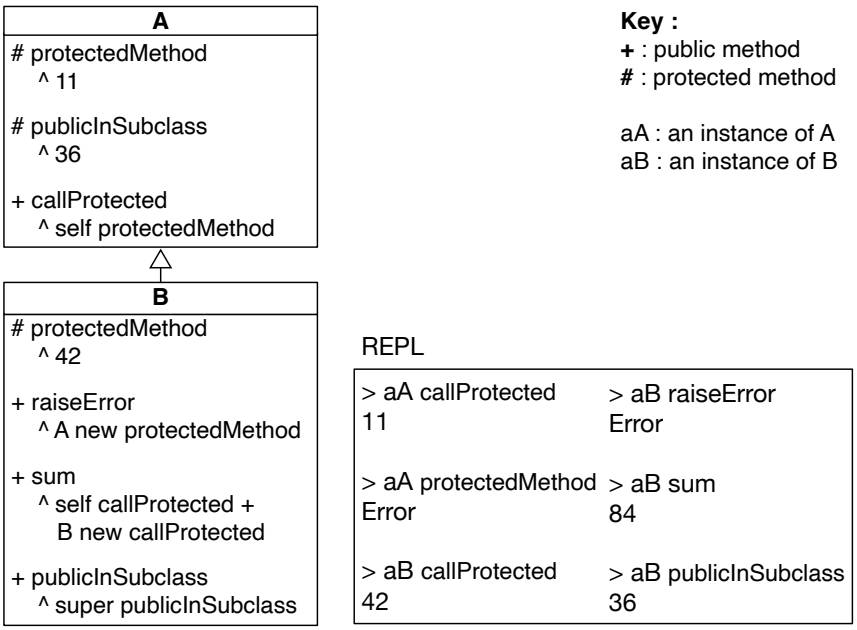


Figure 6.1: Message sending is modified to distinguish between object-sends and self-sends: only object-sends can invoke protected methods which can also be overridden and taken into account by default method lookup.

Figure 6.1 illustrates, with six scenarios, the distinction between object-sends and self-sends. In those scenarios, aA represents an instance of the A class, and aB, an instance of the B class. We also denote with A»foo the method with selector foo in class A.

An object-send targetting a protected method produces an error.

- Expression: aA protectedMethod – an instance of the class A is sent the message protectedMethod.
- Result: Runtime Error
- Explanation: The object-send aA protectedMethod does not find any public method with the selector protectedMethod and produces an error. Note that the method is not visible to this message send site even if at run time the receiver of protectedMethod is an instance of A because the receiver is not syntactically self nor super.

A self-send targetting a protected method activates the method.

- Expression: aA callProtected. – an instance of the class A is sent the message callProtected.
- Result: 11
- Explanation: The object-send aA callProtected finds the public method A»callProtected. From A»callProtected the self-send self protectedMethod finds and activates the protected method A»protectedMethod.

Method lookup semantics are not modified by self-sends.

- Expression: aB callProtected. – an instance of the class B is sent the message callProtected.
- Result: 42
- Explanation: The object-send aB callProtected finds the public method A»callProtected and activates it on aB. From A»callProtected the self-send self protectedMethod finds and activates the overridden protected method B»protectedMethod because aB the receiver is an instance of B.

Protected method visibility is not lexically bound.

- Expression: aB raiseError. – an instance of the class B is sent the message raiseError.
- Result: Error
- Explanation: The object-send aB raiseError finds the public method B»raiseError and activates it on aB. From B»raiseError the object-send A new protectedMethod looks for a public method with selector protectedMethod, finds none, and produces an error. Notice again that protected methods are not visible to this message send site even if the lexical scope contains a definition of protectedMethod.

Any message-send targetting a public method activates the method.

- Expression: aB sum. – an instance of the class B is sent the message sum.
- Result: 84
- Explanation: The object-send aB sum finds the public method B»sum and activates it on aB. From B»sum (a) the self-send self callProtected finds and activates the superclass' public method A»callProtected and (b) the object-send B new callProtected finds and activates the superclass' public method A»callProtected.

Increasing visibility in subclasses.

- Expression: aB publicInSubclass. – an instance of the class B is sent the message publicInSubclass.
- Result: 36
- Explanation: The object-send aB publicInSubclass finds the public method B»publicInSubclass and activates it on aB. From B»publicInSubclass the self-send super publicInSubclass finds and activates the superclass' protected method

  A»publicInSubclass. This example shows that subclasses can redefine and increase the visibility of protected methods in subclasses. As explained in the next section, restricting visibility is not allowed by construction.

6.1.3 Changing Visibility in Subclasses

Reducing the visibility of a method in a subclass makes programs difficult to reason about. Figure 6.2 shows two different examples where reducing visibility creates ambiguities when the method lookup in object-sends skips protected methods. In Figure 6.2(a), aB sum returns 108. The self size self-send lookup finds the B»size method as expected, but the object-send B new size cannot find the B»size method as it is protected. The lookup finds the A»size method which is public. In Figure 6.2(b), reducing the visibility of the sum: method makes the result of a sum: send asymmetrical:

- aB sum: aA returns 77 as expected. The self-send finds the protected B»size method and the object-send lookup finds the A»size method.
- aA sum: aB returns 132. The self-send lookup finds the A»size method as expected, but the object-send cannot find the B»size method as it is protected. Therefore, the lookup finds the A»size method a second time.

There are two different alternatives to avoid this unintuitive behavior, left as an implementation choice for language designers:

- *Raising an error at run time.* For object-sends, the semantics of this solution is that the lookup throws an exception if it finds a protected method. This is the strategy chosen by the Ruby language.
- *Statically forbidding the visibility change in subclasses.* This solution avoids the problem by construction and is used by statically-typed languages such as Java.

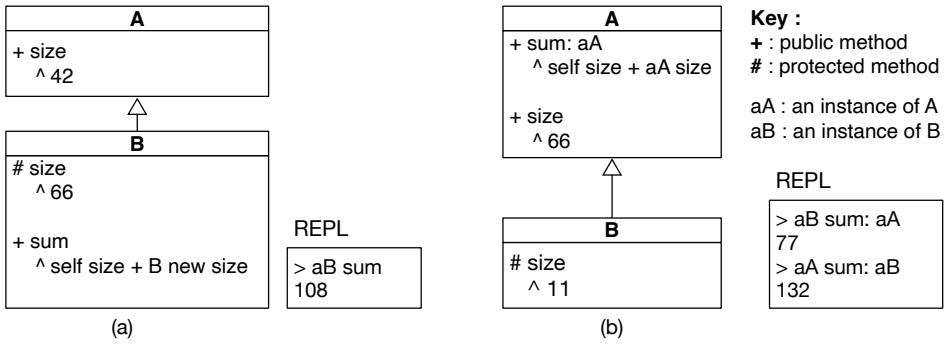


Figure 6.2: Overriding a public method by a protected one leads to buggy situations where self-sends and object-sends can yield two different results. (a) superclass method is used, (b) asymmetrical results.

The other way around, opening up the API by making a protected method public in a subclass does not lead to the discussed issues because public methods were visible from self-sends too. This provides additional flexibility by giving developers a better way to expose previously protected methods. This ability is present for example in the Java language.

6.1.4 PROTECTEDLITE: PROTDYN Semantics

To specify the semantics of self-sends versus object-sends, we define PROTECTEDLITE, an extension of SMALLTALKLITE [**?**].

SMALLTALKLITE is a dynamic language calculus featuring single inheritance, message-passing, field access and updates, and self/super sends. The syntax used in the calculus is presented in Figure 6.3. SMALLTALKLITE is heavily inspired by CLASSICJAVA defined by Flatt *et al.* [**?**]. Although not a contribution of this thesis, we repeated the full description of SMALLTALKLITE in the Appendix B to help the reader.

$$
\begin{array}{rcl}
P & = & \mathit{defn}^* e \\
\mathit{defn} & = & \textbf{class}\ c\ \textbf{extends}\ c'\ \{\ f^*\mathit{meth}^*\mathit{protectmeth}^*\ \} \\
e & = & \textbf{new}\ c \mid x \mid \textbf{self} \mid \mathsf{nil} \\
 & \mid & f \mid (f{=}e) \mid e.m(e^*) \\
 & \mid & \textbf{super}.m(e^*) \mid \textbf{let}\ x{=}e\ \textbf{in}\ e
\end{array}
\qquad
\begin{array}{rcl}
\mathit{meth} & = & m(x^*)\ \{\ e\ \} \\
\mathit{protectmeth} & = & \#m(x^*)\ \{\ e\ \} \\
c & = & \text{a class name} \mid \mathsf{Object} \\
f & = & \text{a field name} \\
m & = & \text{a method name} \\
x & = & \text{a variable name}
\end{array}
$$

Figure 6.3: Protected Pharo syntax.

METHODONCEPERCLASS(P)	Method names are unique within a class declaration $\forall m, m', \#m, \#m'$ **class** $c \cdots \{\cdots m(\cdots)\{\cdots\} \cdots m'(\cdots)\{\cdots\} \cdots$ $\#m(\cdots)\{\cdots\} \cdots \#m'(\cdots)\{\cdots\}\}$ is in P $\Rightarrow Set(m, m', \#m, \#m')size = 4$
OVERRIDINGPROTECTEDMETHOD(P)	Protected methods can be overridden by public or protected methods $\forall m, \langle m, x^*, e\rangle \widehat{\in_P} c, \langle m, x^*, e\rangle \in_P c', c \leq_P c'$ *or* $\forall m, \langle m, x^*, e\rangle \widehat{\in_P} c, \langle m, x^*, e\rangle \widehat{\in_P} c', c \leq_P c'$
OVERRIDINGPUBLICMETHOD(P)	Public methods can be overridden only by public methods $\forall m, \langle m, x^*, e\rangle \in_P c, \langle m, x^*, e\rangle \in_P c', c \leq_P c'$
$\in_P$	Public method defined in class $\langle m, x^*, e\rangle \in_P c \iff$ **class** $\cdots \{\cdots m(x^*)\{e\} \cdots\} \in P$
$\widehat{\in_P}$	Protected method defined in class $\langle m, x^*, e\rangle \widehat{\in_P} c \iff$ **class** $\cdots \{\cdots \#m(x^*)\{e\} \cdots\} \in P$
$\in_P^*$	Public method lookup starting from c $\langle c, m, x^*, e\rangle \in_P^* c' \iff c' = min\{c'' \mid \langle m, x^*, e\rangle \in_P c'', c \leq_P c''\}$
$\widehat{\in_P^*}$	Protected method lookup starting from c $\langle c, m, x^*, e\rangle \widehat{\in_P^*} c' \iff c' = min\{c'' \mid \langle \#m, x^*, e\rangle \widehat{\in_P} c'', c \leq_P c''\}$

Figure 6.4: Relations and predicates for PROTECTEDLITE.

Each class in PROTECTEDLITE has a list of protected methods after the public ones (see Figure 6.3). The MethodOncePerClass predicate presented in Figure 6.4 specifies that two methods shall not have the same name, even if they have different modifiers.

The OverridingPublicMethod and OverridingProtectedMethod predicates guarantee that public methods can only be overridden by public methods and that protected methods can be overridden by public or protected methods. These predicates guarantee by construction that is not possible to restrict the visibility of a public method.

The next four predicates express the lookup mechanism used in PROTECT-

P	$\vdash$	$\langle E[o.m(v^*)], \mathcal{S}\rangle \hookrightarrow \langle E[o[\![e[v^*/x^*]]\!]_{c'}], \mathcal{S}\rangle$ where $\mathcal{S}[o] = \langle c, \mathcal{F}\rangle$ and $\langle c, m, x^*, e\rangle \in_P^* c'$	[*object-send*]
P	$\vdash$	$\langle E[$**self**$\langle o, c\rangle.m(v^*)], \mathcal{S}\rangle \hookrightarrow \langle E[o[\![e[v^*/x^*]]\!]_{c'}], \mathcal{S}\rangle$ where $\mathcal{S}[o] = \langle c, \mathcal{F}\rangle$ and $\langle c, m, x^*, e\rangle \widehat{\in_P^*} c'$ or $\langle c, m, x^*, e\rangle \in_P^* c'$	[*self-send*]
P	$\vdash$	$\langle E[$**super**$\langle o, c\rangle.m(v^*)], \mathcal{S}\rangle \hookrightarrow \langle E[o[\![e[v^*/x^*]]\!]_{c''}], \mathcal{S}\rangle$ where $c \prec_P c',$ and $\langle c', m, x^*, e\rangle \widehat{\in_P^*} c''$ or $\langle c', m, x^*, e\rangle \in_P^* c'',$ and $c' \leq_P c''$	[*super-send*]

Figure 6.5: Message passing reductions for PROTECTEDLITE.

$$
\begin{array}{rcll}
\textbf{new}\ c\ [m'/m]_{self} & = & \textbf{new}\ c & \textit{instance creation} \\
\textbf{let}\ x = e\ \textbf{in}\ e'\ [m'/m]_{self} & = & \textbf{let}\ x = e[m'/m]_{self}\ \textbf{in}\ e'[m'/m]_{self} & \\
x\ [m'/m]_{self} & = & x & \textit{variable access} \\
f\ [m'/m]_{self} & = & f & \textit{field access} \\
(f{=}e)\ [m'/m]_{self} & = & (f{=}e[m'/m]_{self}) & \textit{field assignment} \\
\mathsf{nil}\ [m'/m]_{self} & = & \mathsf{nil} & \\
 & & & \\
\textbf{self}.m(e_i^*)\ [m'/m]_{self} & = & \textbf{self}.m'(e_i^*[m'/m]_{self}) & \textit{self/super sends} \\
\textbf{self}.n(e_i^*)\ [m'/m]_{self} & = & \textbf{self}.n(e_i^*[m'/m]_{self}),\ \text{if } n \neq m & \\
\textbf{super}.m(e_i^*)\ [m'/m]_{self} & = & \textbf{super}.m'(e_i^*[m'/m]_{self}) & \textit{can call protected methods} \\
\textbf{super}.n(e_i^*)\ [m'/m]_{self} & = & \textbf{super}.n(e_i^*[m'/m]_{self}),\ \text{if } n \neq m & \\
e.m(e_i^*)\ [m'/m]_{self} & = & e[m'/m]_{self}.m(e_i^*[m'/m]_{self}) & \textit{object-sends only} \\
 & & & \textit{call public methods}
\end{array}
$$

Figure 6.6: Self and super-send call site renaming (also called selector mangling). Method names of object-sends are not renamed to protected method name mangling, while self and super sends are.

EDLITE. Predicates $\in_P^*$ and $\widehat{\in_P^*}$ express the lookup of public and protected methods respectively. The lookup mechanism finds the closest superclass (*c'*) of the receiver class (c) that contains a method with the given selector (*m'*). The main difference between both mechanisms is that the public lookup only searches in public methods and the protected lookup only in protected ones.

The public and protected lookup mechanisms are then used in Figure 6.5 to define the self-send, super-send, and object-send. When a message is sent to an object (without using self or super), we look up the method body, starting from the class of the object *c* and only look at public methods. For both self-send and super-send, we use both lookups. Super-sends use the same mechanism as self-sends, but the lookup starts from the superclass (*c'*) of the class defining the method.

Selector mangling for self-sends sites. During compilation, all self-sends are mangled (See Section 6.2.4). Figure 6.6 formalizes this transformation. The expression $[m'/m]_{self}$ renames all the self-sends to m to m', where m' is the mangled selector obtained from the $\rho_{[]}$ hiding function. The scope of this hiding function is global.

6.2 #PHARO Implementation

In this section, we present #PHARO, an implementation of PROTDYN for the Pharo programming language. We first present the design principles behind our implementation and an overview of our solution. We then delve into the two key

aspects of our implementation: double public method registration and self-send selector mangling.

6.2.1 Design Principles

The design behind our protected modifier implementation follows the principles:

- **Backward compatible:** *Existing* programs (*i.e.* not using protected modifiers) should continue to work and expose the same behavior under the presence and absence of protected method support.
- **Not requiring a new runtime:** The solution should not be based on changing the virtual machine execution logic, to ease portability and deployment.
- **No run-time penalty when...**
  - **not using protected methods:** A program not using protected modifiers should not be impacted by the presence of the protected modifier implementation.
  - **using protected methods:** A program using protected methods should not have a run-time penalty compared to using public methods only.

6.2.2 Implementation Overview

Our implementation is based on two techniques: *double registration of public methods* and *selector mangling of self-sends*. Protected methods are identified by developers using method annotations using the Pharo annotation system [**?**], illustrated in Figure 6.1.

```
1     B >> protectedMethod [
2         <protected>
3         ^ 42 ]
4     B >> raiseError [
5         ^ A new protectedMethod ]
6     B >> sum [
7         ^ self callProtected + B new callProtected ]
8     B >> publicInSubclass [
9         ^ super publicInSubclass ]
10
```

Listing 6.1: Code excerpt from Figure 6.1.

Protected methods are registered in the method dictionary of their class with a mangled selector. Public methods are registered two times, once with their original selector and once with their mangled selector. All self-send sites are rewritten by mangling the message selector. These two transformations are shown in Figure 6.7.

Section 6.2.5 presents how to limit the recompilation to classes using protected methods and their subclasses.

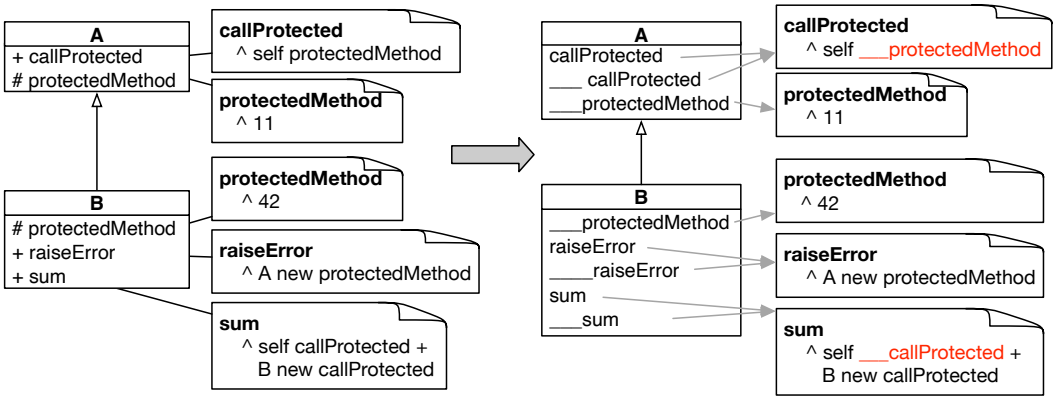


Figure 6.7: From the model to the actual implementation: Selector mangling and public method double addition to their class method dictionary.

6.2.3 Double Public Registration

Protected methods are added to method dictionaries with a mangled selector (*e.g.* prefixed with __ in the following examples and Figure 6.7). To avoid conflicts, mangled selectors are forbidden by the compiler and are not meant to be used directly. Self-sends are modified at compile time to look for methods with this prefix (see Section 6.2.4). Protected methods are therefore visible from self-sends, but not from object-sends.

In addition, public methods are visible from both self-sends and object-sends (*i.e.* all sends). Self-sends see the methods installed with the mangled selector while object-sends see methods installed with their original selector. As illustrated in Figure 6.7, in class A:

- The protectedMethod appears only once in the method dictionary : at the __protectedMethod selector.
- The public method callProtected appears twice : at callProtected without prefix and __callProtected with the prefix.

6.2.4 Selector Mangling for Self-Sends Sites

During compilation, all self-sends are mangled. In Appendix subsection 6.2.4, we formalize this transformation. Here are key points related to selector mangling for self-sends:

- Self-send sites are rewritten to use mangled selectors to see protected methods. For example, method A»callProtected contains a self-send to protectedMethod, that is rewritten as a send to __protectedMethod. The same goes for super-sends.

- Mangled self-sends see public methods because they are installed with the mangled selector in addition to the original selector. For example, B»sum definition contains a self-send to callProtected, hence it is rewritten as a send to __callProtected. The same goes for super-sends.

The rewrite unit of our implementation is a class hierarchy. If a class is using the protected modifier for the first time, we choose to recompile all methods in the class and its subclasses and add all the duplicated mangled entries when we add a protected method for the first time. This avoids recalculating the exact set of methods to be recompiled for each new protected method which would be required for lazy recompilation.

6.2.5 Preventing Selector Mangling Propagation to the Whole System

Our implementation rewrites all self-sends in a whole *descending* hierarchy to avoid unnecessary rewrites of the existing system. Superclasses without protected methods do not have double registration of public methods, thus self-sends will not find mangled entries in their method dictionaries. Mangling selectors upwards in the hierarchy would propagate the transformation to the full system.

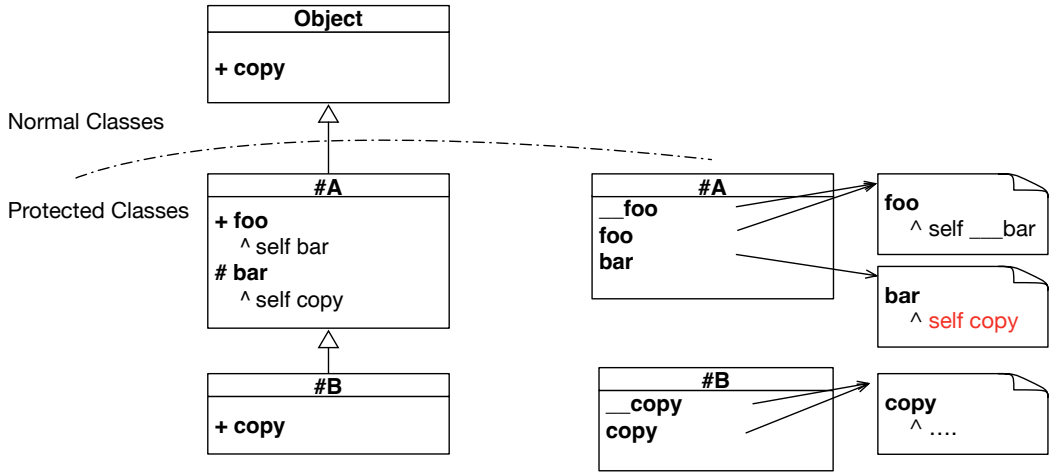


Figure 6.8: Limiting the propagation of recompilation with selector mangling for protected to the top of the hierarchy.

To avoid this problem, our rewriting strategy performs an additional check at compile-time when mangling self-sends. We handle public methods defined above classes with protected methods differently from other public methods. This is the case of the method copy in Figure 6.8: it is publicly defined in Object and used as a self-send in class A. For such a method, we do not mangle the self-send in the classes defining protected methods. In Figure 6.8, the method protectedMethod invokes copy without mangling as we do otherwise (and explained above). This way we do not have to recompile Object and therefore avoid the propagation of the double registration to the entire system. Applying double registration only on classes using the protected modifier limits the memory overhead (See Section 7.2).

As the modifier of a method can only change from protected to public in subclasses, any redefinition of copy will be public (as in class B). This ensures that invoking protectedMethod on an instance of B correctly finds the redefined version of copy in B.

A special case appears when a subclass does a self-send of a non-existing method such as the message self unknown in method anyMethod of class A in Figure 6.9. In that case, our implementation assumes it is a public send and the message-send site will be recompiled when (if) that method is installed later.

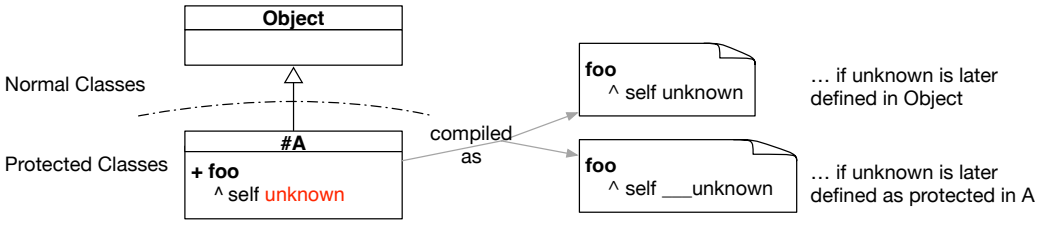


Figure 6.9: Compiling a method with an undefined selector assumes a public message.

6.3 Discussing Alternative Implementations

This section reports our implementation decisions and some alternatives.

6.3.1 Lookup Mechanism Modification

In our implementation, we avoided modifying the method lookup used in the VM by using selector mangling. An alternative implementation is to split the lookup into two different operations, one for public methods and the other for protected methods. To identify the methods we consider the following alternatives:

- Marking a method with its visibility, or;

- Splitting methods into two different collections (*e.g.* method dictionaries).

Marking methods with their visibility requires the lookup method to iterate all possible methods in a class and differentiate them when using the public or protected lookup. This approach simplifies the structure of the classes but it may affect the performance of the lookup mechanism, as it has to iterate more methods than required. The second alternative, splitting methods into two different method dictionaries, simplifies the lookup mechanism but requires maintaining two different collections per class.

All in all, a modified method lookup algorithm affects all classes in the system, not only the ones using protected methods. Our implementation only affects the classes using protected methods.

6.3.2 Run-Time Visibility Checks

In our implementation, when the lookup for a message send is performed from an object-send site and the corresponding method is protected, it will not be found and an error will be raised. This error is equivalent to the one produced when a message is not implemented, or in Smalltalk terminology, a *Message Not Understood* error.

Alternatively, a more specific error could be raised when a protected method is found from an object-send site. This is the strategy chosen by Ruby. This alternative requires performing a check on each method activation. We chose to use the lookup solution because it keeps backward compatibility, it profits from existing lookup optimizations, and because *Message Not Understood* errors are already commonly used in Pharo.

6.4 Conclusion

In this chapter, we presented PROTDYN, a self-send-based visibility model calculated at compile-time for dynamically-typed languages. Our model restricts protected methods' activation to self/super-sends and makes them available only from an instance of the class defining them and their subclasses. We formalized our model by introducing the formal semantics PROTECTEDLITE to describe it.

We presented #PHARO, a PROTDYN implementation that: is backward compatible with existing programs, does not require a new runtime (*i.e.* a new VM), and has a low run-time penalty. We discussed its implementation, and in particular how to limit the impact on the whole system by limiting the recompilation necessary. Our model was also ported to Python by one of our colleagues [**?**], demonstrating the applicability of this solution to other languages. Our implementation is designed to be loadable as libraries and add negligible run-time costs.

In the next chapter, we will evaluate the performances of #PHARO, both from a speed performance and from a memory footprint point of view. We will run benchmarks to measure these and compare them to the standard Pharo implementation.

CHAPTER 7

Protected Pharo Evaluation

Contents

7.1 Performance Evaluation 94
7.1.1 Experimental Design 94
7.1.2 Methodology and Setup 95
7.1.3 Selected Benchmarks 96
7.1.4 Scenario 1: Lookup Performance 97
7.1.5 Scenario 2: Global Lookup Cache Performance 98
7.1.6 Scenario 3: Lookup Cache Behaviour 99
7.1.7 Scenario 4: Polymorphic Inline Cache PIC Performance . 101
7.2 Memory Use Analysis of #PHARO 102
7.2.1 Methodology and Setup 102
7.2.2 Memory Cost: Results 103
7.3 Applicability to Reflection 104
7.3.1 Applicability to Reflective Methods 104
7.3.2 The Case of doesNotUnderstand: 105
7.3.3 Applicability to Internal Methods of Reflective Classes . . . 108
7.4 Conclusion 109

In this chapter, we evaluate the performance of PROTECTEDLITE, the protected modifier implementation for Pharo we presented in Chapter 6. PROTECTEDLITE is impacting the lookup, *i.e.* the algorithm that is responsible for finding at runtime the appropriate method to execute. As Pharo is a dynamically-typed language, this is an operation that is performed each time a message is sent to an object (or "a method is called" in Java vocabulary). It is therefore optimized with a global lookup cache [**?**] and polymorphic inline caches [**?**].

For our solution to be viable, as we discussed in the **Design Principles** (Section 6.2.1), we want no run-time penalty both when protected methods are not used (*i.e.* PROTECTEDLITE is just installed in the system) and when they are used. In

Section 7.1, we present the methodology we used to measure the impact of PROTECTEDLITE on the speed performance of Pharo. While having no run-time penalty at all is unrealistic, we will show that the impact is low enough for this solution to be viable.

Section 7.2 highlights the impact of installing and/or using PROTECTEDLITE on a Pharo image. We discuss why PROTECTEDLITE introduces additional memory usage and measure its impact.

In Section 7.3 we focus on the applicability of PROTECTEDLITE to Pharo's reflective API. As we have seen in Chapter 3 there are existing dependencies between reflective APIs. We want to see whether PROTECTEDLITE could be used to restrict access to some low-level reflective APIs without breaking higher-level ones.

The content of this chapter, except for Section 7.3, has been published in the following article:

- "A VM-Agnostic and Backwards Compatible Protected Modifier for Dynamically-Typed Languages" Thomas *et al.* [**?**].

7.1 Performance Evaluation

In this section, we will evaluate the speed performances of PROTECTEDLITE, and its ability to take advantage of different lookup optimization, such as global lookup cache and Polymorphic Inline Cache (PIC).

7.1.1 Experimental Design

Informally speaking, the main performance impact of our solution would come from an increase in the number of entries in method dictionaries, which would introduce a negative impact on CPU caches, method lookup algorithms, and message sends. The goal of this evaluation is to assess such an impact. In our evaluation, we compare several benchmarks on three different scenarios using three different virtual machine configurations derived from the Pharo VM v8.1.0-alpha-335-g70b7e3542.

Scenario 1: Method lookup impact. Using our implementation, public methods increase the size of the selector namespace by installing each method once with the non-protected and once with the protected selector. In the *worst case* where all methods are public, they double the method dictionary number of keys. This is the case we measure in all performance benchmarks. Thus, we measure the performance impact on lookup in a token-threaded interpreter implementation of Pharo [**?**], with lookup caches disabled [**?**].

Scenario 2: Lookup cache impact. Global lookup caches store method lookup results avoiding subsequent lookups of the same $(receivertype, selector)$ pair [**?**]. This scenario uses the same token-threaded interpreter above with a global lookup cache enabled. The global lookup cache is a hash table with 1024 entries and performs up to three lookups in the cache per message-send before doing a slow method lookup. This scenario is two-fold: we compare the impact on run-time performance and cache behavior (*i.e.* # hits and misses).

Scenario 3: Polymorphic inline cache impact. Polymorphic inline caches further avoid lookups by localizing a lookup cache on each message send-site, typically implemented with machine code stubs and code patching [**?**]. This scenario uses a mixed-mode Pharo implementation combining a token-threaded interpreter, a 1024-entry global lookup cache, and a non-optimizing method JIT compiler with polymorphic inline caches.

All scenarios are run in three different configurations:

- **Baseline** #PHARO not installed (and thus not used).
- **Case 1** #PHARO is installed but not used.
- **Case 2** #PHARO is installed and used in a *worst-case scenario*. As we want to estimate the maximum run-time penalty using #PHARO, all our benchmarks enable #PHARO but let all methods as public. This forces a *double registration for all methods*

7.1.2 Methodology and Setup

We designed our benchmarks following the guidelines of Georges *et al.* [**?**]. We did our best to minimize the system's noise [**?**], close all non-related non-essential applications and services, and shut down the internet connection. The machine was plugged in and there was no user interaction until the benchmark finished. We ran our benchmarks for #PHARO on a MacBook Pro 17.1 with an Apple M1 processor (8 cores including 4 performance cores and 4 efficiency cores) and 16GB of LPDDR4 RAM, running on macOS 12.0.1.

We use a fixed number of in-process iterations to determine steady-state instead of dynamically detecting it because none of our VM configurations includes profile-guided optimizations that require a long warmup time. Our methodology is as follows:

- 200 VM invocations.
- 55 benchmark iterations for each VM invocation.

- The first 5 benchmark iterations of each VM invocation are discarded (done to warm up the caches), leaving 50 measures per invocation.
- Package loading is done beforehand and saved in a snapshot.

For each benchmark, we report the average run times of each VM invocation. Figures plot numbers relative to the baseline configuration instead of absolute numbers for readability. For each benchmark, we show the distributions with violin graphs complemented with a whisker boxplot showing the median, lower, and upper quartiles. The upper whisker is the minimum between the max relative run time value and the value of the $upper\_quartile + 1.5 \times interquartile\_range$. The lower whisker is the minimum between the min relative runtime value and the value of the $lower\_quartile - 1.5 \times interquartile\_range$. Assuming a Gaussian distribution, 99 % of the values should be inside the whiskers. Any data points outside the whiskers are shown by black dots.

The average relative run times are shown by red dots within each violin shape. This is the main performance indicator used in the analyses.

7.1.3 Selected Benchmarks

We selected four different benchmarks, avoiding on-principle microbenchmarks because they would not exercise message sends and our protected implementation:

Microdown. Microdown is an implementation of a Markdown superset defining a parser, document tree, and several exporters. The parser uses a delegation-based approach using many polymorphic calls. We wrote a benchmark implementation that parses the README.md file of the Pharo GitHub repository.

Delta Blue. The DeltaBlue one-way constraint solver. We used the implementation available in the SMark benchmark library [**?**].

Richards. An OS kernel simulation originally written in BCPL by Martin Richards. We used the implementation available in the SMark benchmark library [**?**].

Bytecode Compiler Benchmark. The Pharo bytecode compiler exercises various compilation aspects: parsing, AST generation, semantic analysis, linear IR generation, and bytecode generation. During the benchmark, we perform several method compilations with source code larger than typical Pharo methods. We used the benchmark implementation available in the SMark benchmark library [**?**]. Notice that this benchmark exhibits different behavior than the previous three since our protected method implementation introduces modifications in the compiler.

Full results are available in Appendix C.

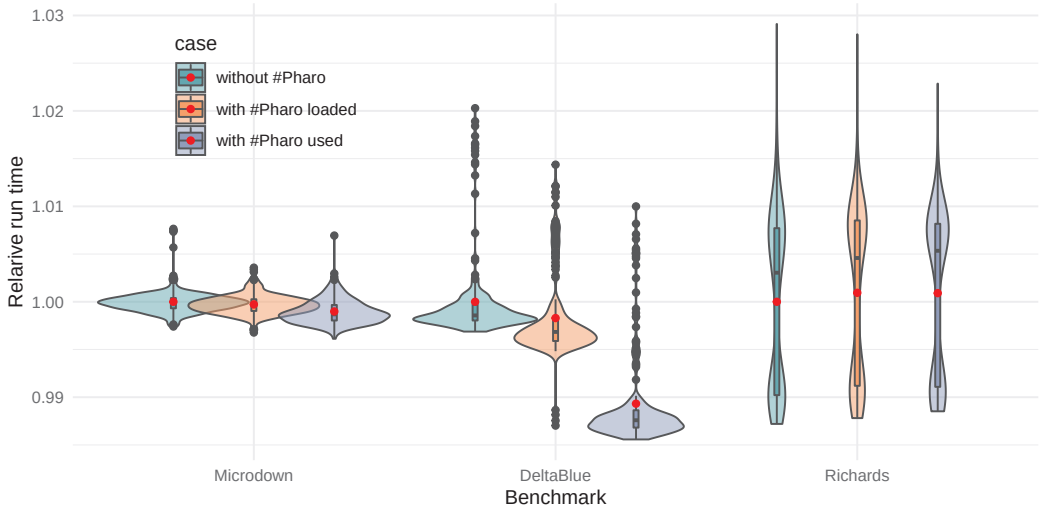


Figure 7.1: Relative run time performances with global cache disabled on Microdown, Deltablue, and Richards benchmarks. Lower is better. A red dot marks the average.

7.1.4 Scenario 1: Lookup Performance

We evaluate the impact of #PHARO on lookup performance using the VM with a disabled global lookup cache. This means that each message-send instruction produces a lookup in the class hierarchy. Figure 7.1 shows the distribution of the relative run times of three benchmarks (Microdown, Deltablue, and Richards). Figure 7.2 shows the performance of the Compiler benchmark.

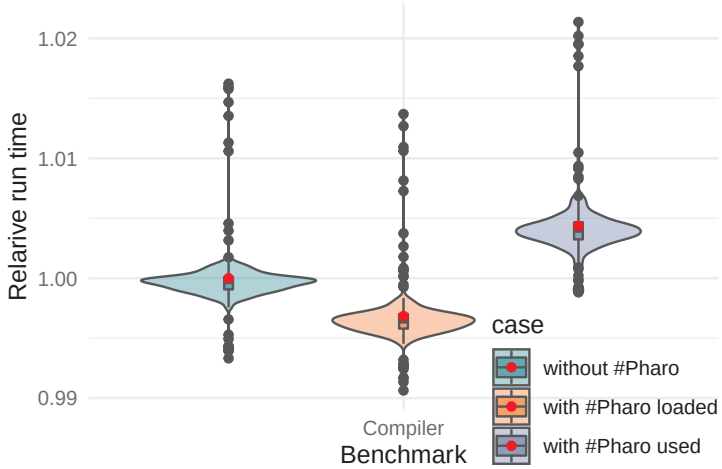


Figure 7.2: Relative run time performances on the VM without global cache for the Smark Compiler benchmark. Lower is better. The red dot marks the average.

For Microdown and Richards benchmarks, the average relative variation is less than $0.1\,\%$ when #PHARO is loaded and when it is used in the worst-case scenario. For the DeltaBlue benchmark, we have an average run time with #PHARO used that is at $98.9\,\%$ of the average run time without #PHARO. The run-time distribution is consistent for each benchmark.

The compiler benchmark has been set aside because our approach is implemented as a compiler plugin, introducing variations between #PHARO loaded and unloaded. When #PHARO is loaded the average impact at compile time is a $0.3\,\%$ speed-up.

When the compiler package itself uses #PHARO, we observe a $0.4\,\%$ slowdown. This is the biggest slowdown compared to the other benchmarks on the VM without a global cache.

We observe small differences in the run time performance for which we do not have an explanation yet. When these small performance changes were significant, their impact was usually below $0.5\,\%$ of the average speed without #PHARO and always below $1.1\,\%$. *This shows that our prototype is a viable solution and that we do not introduce significant run-time costs related to the lookup.*

7.1.5 Scenario 2: Global Lookup Cache Performance

This experiment evaluates the impact of the global lookup cache on performance. We run the same benchmarks as Experiment 1, enabling the global lookup cache.

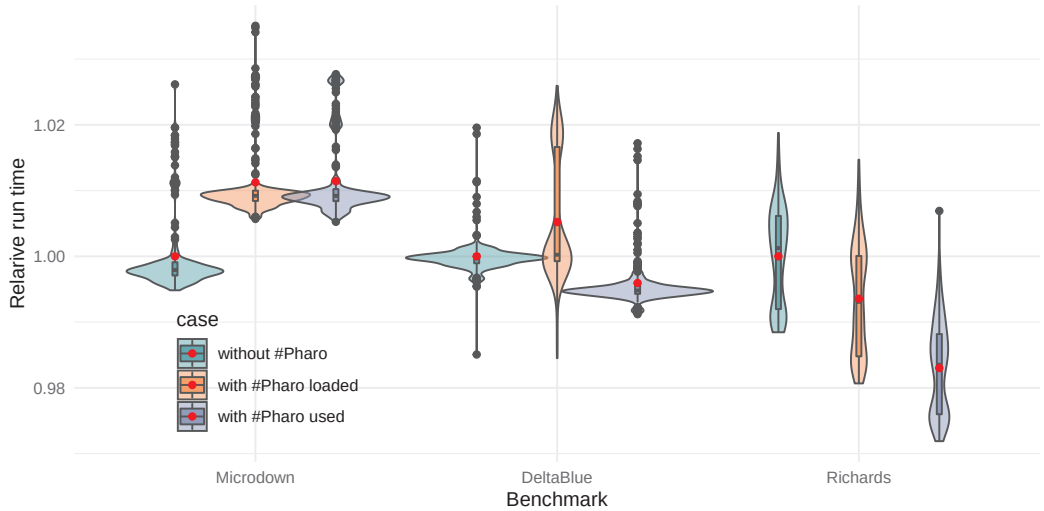


Figure 7.3: Relative run time performances on the VM with global cache only for Microdown, Smark Deltablue, and Smark Richards benchmarks. Lower is better. A red dot is the average.

Our results show that the benchmark that is slowed down the most is Mi-

crodown (see Figure 7.3). With #PHARO only loaded and #PHARO used, there is a $1.1\,\%$ slowdown in the average run time compared to #PHARO unloaded. Delta Blue with #PHARO loaded is also slowed down by $0.5\,\%$. However, Delta Blue with #PHARO used is sped up by $0.4\,\%$. There is also a speedup on Richards, both with #PHARO loaded ($0.6\,\%$) and used ($1.7\,\%$).

Regarding the compiler Benchmark presented in Figure 7.4, #PHARO loaded does not introduce slowdowns as we observed in Experiment 1. The speedup using #PHARO increases, from $0.3\,\%$ to over $1\,\%$ with the global lookup cache.

These changes in relative run-time performances can be related to the number of hits from the global lookup cache. See subsection 7.1.6, where we study the percentage of cache hits and misses.

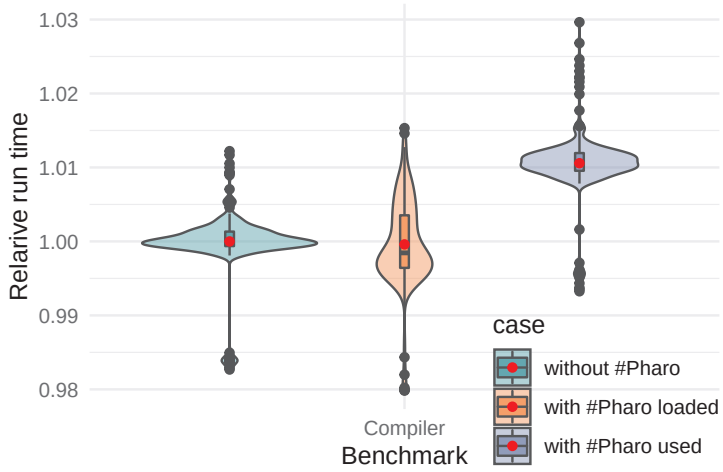


Figure 7.4: Results with lookup cache enabled on the Compiler benchmark. Lower is better. Red dots mark the average.

In summary, in this experiment, we observe slowdowns of at most $1.1\,\%$ with a global lookup cache. *This confirms that our prototype performs well in the presence of a global lookup cache.*

7.1.6 Scenario 3: Lookup Cache Behaviour

This experiment evaluates the behavior of our implementation on global lookup caches, given that name mangling introduces a larger set of selectors (with and without prefixes). We run the same benchmarks as Experiment 1, enabling the global lookup cache and recording cache hits and misses. Notice that Pharo's lookup cache works by probing up to three times in the hash table. A lookup in the table is considered a miss if the third probe fails. Figure 7.5 shows our results as percentages of hits and misses, discerning between the three hit probes. The

higher the percentage of hits the faster the program will run because cache misses will trigger a costly lookup. These measurements are done after the five warm-up iterations.

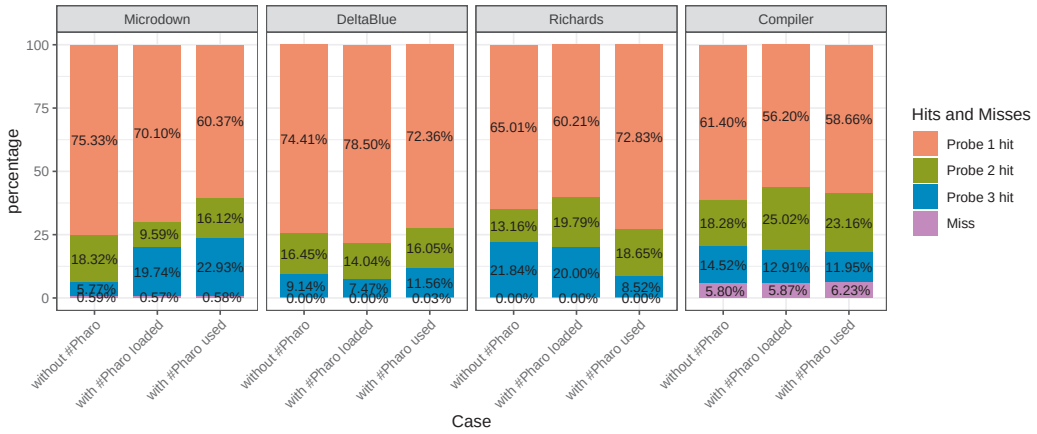


Figure 7.5: Cache hits and misses.

In Figure 7.5, for the Microdown benchmark, we see $5\,\%$ fewer first probe hits with #PHARO loaded compared to when it is not, and again $10\,\%$ less when it is used. Considering the first and second probe's hits, there are $14\,\%$ fewer hits on the third one with #PHARO installed and $3\,\%$ less when used. These differences contribute to slowing down the average run time of the Microdown Benchmark seen above.

On the contrary, for the Richards benchmarks with first and second probes, we have $2\,\%$ more hits with #PHARO installed and $11.5\,\%$ more again with #PHARO used. This explains the observed speedups of the Richards Benchmark shown before.

For the Delta Blue benchmark, we see $4\,\%$ more first probe hits with #PHARO installed and $2\,\%$ more first and second probe hits. When #PHARO is used we have $2\,\%$ fewer first probe hits and $2.5\,\%$ fewer first and second probe hits compared to without #PHARO. It matches the run-time variation on this specific VM.

For the Compiler benchmark, we see around $6\,\%$ misses for all configurations, probably due to hash collisions. There is a slight increase in the percentages in misses: $+0.07\,\%$ with #PHARO installed and $+0.4\,\%$ with #PHARO used compared to without #PHARO. However, considering we doubled the number of selectors in the method dictionary of all the classes of the targeted package, this is a relatively small increase.

Overall there is little to no variation in the number of misses. Performance variations depend on which probe hits: the faster a probe hits, the faster the benchmark will run. Depending on the benchmark, the percentages of probe hits increase or decrease when #PHARO is installed or used. While we cannot extract general rules

on when collisions happen while comparing without #PHARO, with #PHARO loaded and with #PHARO used, we can observe that probe hits are related to the changes in run time performances. Therefore, *the variations in run time performances are linked to global cache hits and* #PHARO *can have a small positive or negative impact on those*.

7.1.7 Scenario 4: Polymorphic Inline Cache PIC Performance

This experiment evaluates the impact of our implementation on Polymorphic Inline Caches (PICs). The informal assumption is that name mangling should not affect the runtime behavior of PICs. Figure 7.6 shows the results for the Microdown, Delta Blue, and Richards benchmarks on a VM with JIT and the Polymorphic Inline Cache (PIC) enabled. Figure 7.7 shows the results for the Compiler benchmark. For the Microdown benchmark, we observe speedups of $0.15\,\%$ and $0.35\,\%$ with #PHARO respectively installed and used. The distribution of each case's relative run times is similar. For the Delta Blue benchmark, the distribution of each case's relative run times is also similar, but there are slowdowns of $0.51\,\%$ and $0.64\,\%$ with #PHARO respectively installed and used.

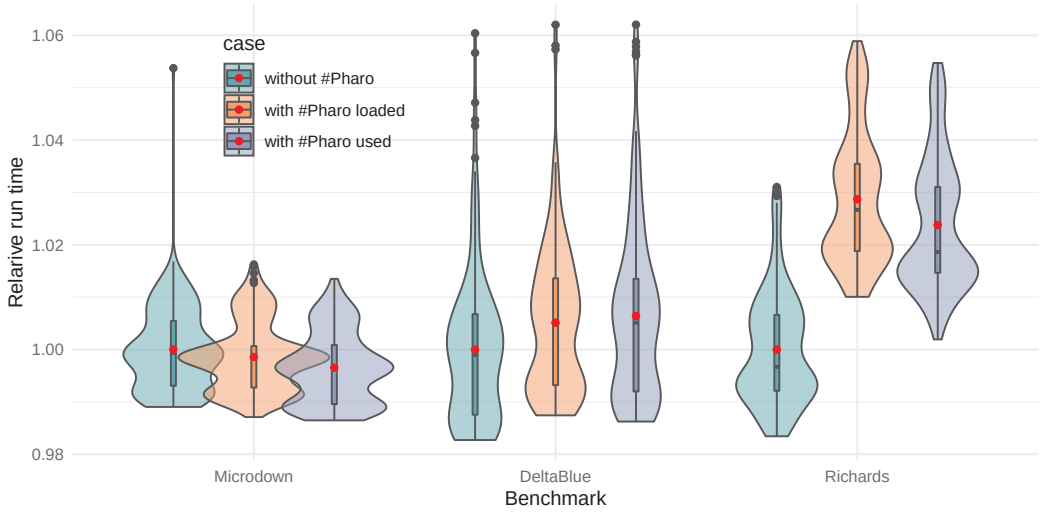


Figure 7.6: Results on the VM with JIT and PICs enabled for Microdown, Deltablue, and Richards. Lower is better. A red dot is the average.

In the compiler benchmark, we observe an average run time that is respectively $0.8\,\%$ and $1.4\,\%$ slower compared to the version without #PHARO.

The Richards Benchmark presents the biggest slowdowns, with a run time $2.9\,\%$ higher when #PHARO installed and $2.4\,\%$ higher when used.

These results reflect the performance of #PHARO on the default Pharo VM with the polymorphic inline cache. In the worst-case scenario, we have an average run

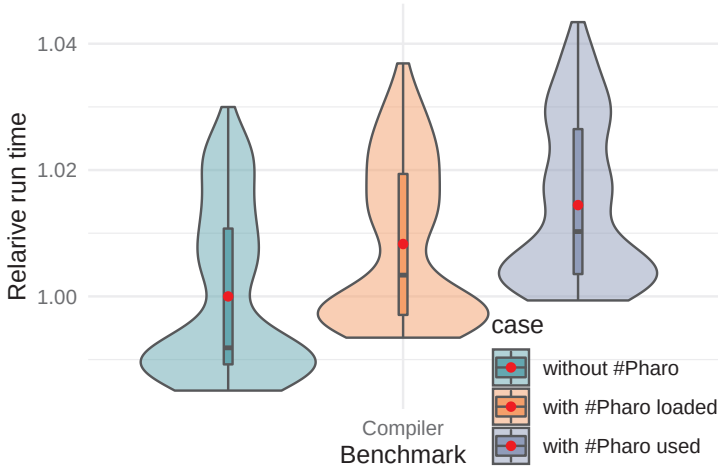


Figure 7.7: Results on the default VM for the Compiler benchmark. Lower is better. A red dot is the average.

time that is less than 3 % longer on the most optimized VM. However, in most cases, average run time variations with #PHARO installed and used are around 1 % or less. As we have seen in previous experiments, those variations can be partially due to the global cache performances and are not directly linked to the installation or the use of #PHARO.

There are variations in average run times, so we cannot claim that there is **no run-time penalty when not using protected** and the **no run-time penalty when using protected** constraint is met with our implementation. However, we believe that the run time performance variations in both cases are low enough to prove that this is a viable implementation.

7.2 Memory Use Analysis of #PHARO

Our solution relies on double registration of methods and name mangling, stressing method dictionaries and selector tables. This section assesses the memory cost of protected Pharo. We use the Microdown application as our benchmarking program (see Section 7.1.3) and measure the amount of memory used before and after introducing #PHARO and protected methods.

7.2.1 Methodology and Setup

We measure the use of memory using the "Space and time" [?] package. This package traverses the graph of objects starting from Microdown classes and measures the amount of memory taken in the heap in bytes. The traversal stops on global

variables to avoid propagating to the whole system. The rest of the setup is the same as the one described in Section 7.1.1.

We make measurements in three benchmark variations on Microdown which defines 257 classes and 2683 methods.

#PHARO Unused. Not using #PHARO is used as a comparison baseline.

Manual Tagging. We manually tagged 28 methods as protected in 8 classes, belonging to 6 different hierarchies with depths ranging from 1 to 5. Our 28 tagged methods led to 61 classes using our protected modifier, either directly or by inheritance. Few methods could be declared as protected in Microdown because most of its code uses the visitor pattern. Notice that those 28 methods were found by an automatic static analysis of the project. Our static analysis is conservative due to the lack of type information, some potential candidates may have been missed because of unintended polymorphism. Moreover, we made sure that all of the project tests stayed green after tagging: incorrect tagging can overprotect a method and change the application semantics. We leave for future work the analysis of the profitability of protected methods and the automatic migration of applications.

Worst case. To get the worst case, all methods are declared as public. This makes our implementation register each method both with mangled and not mangled selectors in all classes, taking more space in method dictionaries and selectors.

7.2.2 Memory Cost: Results

Measure	Unused	Manual Tagging	Worst Case
Total size (bytes):	1 023 216	1 044 840 (1.02x)	1 159 368 (1.13x)
Method dictionaries size (bytes):	249 592	263 592 (1.06x)	326 040 (1.30x)
Symbols size (bytes):	94 920	102 528 (1.08x)	154 560 (1.62x)
Number of instances:	19 563	19 835 (1.01x)	21 321 (1.08x)
Number of Symbols:	3 323	3 595 (1.08x)	5 081 (1.52x)

Table 7.1: Microdown memory use with and without #PHARO. Absolute numbers in bytes. Results relative to the baseline are in parentheses.

Table 7.1 shows an increase of 2.1 % of the memory used by Microdown in the case where #PHARO is used selectively and 13 % in the worst-case scenario. This increase is due to two main things: first, all the new symbols corresponding to mangled selectors, and second, the size increase of the method dictionaries. By looking at the number of instances, *i.e.* the number of objects in the studied graph, we can see that all the new instances actually correspond to the new symbols which

are created by name-mangling. As the double registration of the public method creates references to the same compiled method with the selector mangled and not mangled, the number of compiled methods does not increase. We only have a small overhead (48 bytes total in the worst case) in the space taken by compiled methods and compiled blocks. We consider that this is a reasonable increase in memory use that does not threaten the viability of #PHARO.

7.3 Applicability to Reflection

In this section, we discuss the applicability of #PHARO to Pharo's reflective APIs. First, we identify reflective methods that could be protected and discuss the relevance of protecting them. We discuss in more detail the case of doesNotUnderstand:. Then, we conclude with the applicability of #PHARO to methods in the private protocols of reflective classes such as Behavior.

7.3.1 Applicability to Reflective Methods

We have previously tagged reflective methods with pragmas and built a static analysis tool to detect which messages are only sent to self or super. Now we combine both to assess the relevance of #PHARO to reflective APIs.

Table 7.2 lists the 17 out of 329 reflective selectors that we identified with static analysis. All methods corresponding to those selectors could be protected without breaking anything in the standard libraries.

We ignore 18 selectors that do not present users. All send-sites are considered when looking for any object-send. Therefore, we might have some false negatives, *i.e.* methods that are not identified as being protectable due to polymorphism (*i.e.* they have the same name as a method in another branch of the class hierarchy that is public and has object-send sites). We could also have some false positives due to reflective message sends not being detected.

However, the highly open and reflective nature of Pharo leads to very few methods that can be surely identified as protectable. While the corresponding methods could be protected, some of them are actually part of APIs that are designed to be public, meaning their visibility should not be restricted. This includes: #selectSuperclasses:,#allInstancesOrNil, #pointersToExcept:among:, #localSendsAnySelectorOf:.

Some methods have clear comments such, as object:basicAt: and object:basicAt:put: which are implemented on the Context class in a mirror primitives protocol. They have the following comment: "This mimics the action of the VM when it indexes an object. Used to simulate the execution machinery by, for example, the debugger."

For some, even the manual evaluation is not straightforward, as the line between low-level reflective access and internal methods of reflective implementation is quite blurry. Moreover, not all methods have comments as clear as the ones from

selector	number of senders
#activateMethod:withArgs:receiver:class	2
#allInstancesOrNil:	2
#collectArguments:	3
#doesNotUnderstand:	22
#instVarsInclude:	2
#localSendsAnySelectorOf:	1
#object:basicAt:	2
#object:basicAt:put:	2
#objectSize:	1
#perform:orSendTo:	2
#pointersToExcept:among:	1
#printStackOfSize:	2
#receiver:withArguments:executeMethod:	2
#selectSuperclasses:	1
#send:to:with:lookupIn:	3
#send:to:with:super:	5
#stepUntilSomethingOnStack:	1

Table 7.2: Table of reflective methods selectors that could be protected according to static analysis of the Pharo standard library/base image.

object:basicAt:(put:).

Therefore, The direct applicability of #PHARO to reflective methods is existing but limited. We identified a very small amount of reflective methods for which this is relevant, but we do have some cases such as doesNotUnderstand: where it is applicable.

7.3.2 The Case of doesNotUnderstand:

The selector doesNotUnderstand: is one of the selectors that was detected as protectable by static analysis, meaning no explicit object-send could be detected in the standard library. In this section, we go over why it is a relevant case, and cover all the senders of doesNotUnderstand: and call-sites to study the applicability of #PHARO.

After looking up through the whole hierarchy, if no method is found, the doesNotUnderstand: (DNU) is sent to the object. To be able to be raised on any object, this is implemented in the class ProtoObject, which is the root of the inheritance tree in Pharo. Redefining this method allows each class to specify a behavior different from the default one when receiving an unknown message.

The message doesNotUnderstand: is a mechanism present since the beginning of Smalltalk. It is one of the most common exceptions raised in the current Pharo

environment, given the existing practice of coding in the debugger. As the developer starts coding and executing the code, when the execution reaches a place where a method is missing, it will raise a DNU exception. That will open a debugger. This allows the developer to interact with live objects and keep coding. This hook was notably used to implement one of the first general proxies in Pharo: This is done by creating a subclass of ProtoObject which has very few methods, defining a minimal amount of methods for the proxy and redirecting unknown message to the proxied object.

As it is commonly used while having mainly self-sends senders and a few reflective senders, it is a relevant case to study. In the rest of this section, we discuss its senders regarding the applicability of #PHARO.

In the standard library, senders of doesNotUnderstand: are the redefinitions of doesNotUnderstand: in the following classes :

- CollectionValueHolder
- DebugPoint
- DictionaryValueHolder
- FileLocator
- IceTipTreeNode
- MetaLink
- SpDynamicPresenter
- SpUIThemeDecorator
- RubParagraphDecorator
- ShiftClassBuilder
- ThemeIcons
- ToolRegistry

and the following methods:

- Context»#send:to:with:lookupIn:
- Context»#send:to:with:super:
- SmalltalkImage»#newSpecialObjectsArray
- Context»#printOn:
- ToolRegistry»#inspector
- DebugSession»#isInterruptedContextDoesNotUnderstand
- StMockDebuggerActionModel»#initialize
- InstructionStreamTest»#testStepThroughInAMethodWithMNU
- SindarinCommandsTest»#testStepToNextCallInClassWithError
- StDebuggerActionModelTest»#testDynamicShouldFilterStackUpdate

Redefinitions. There are twelve redefinitions of this method that send super doesNotUnderstand:. Applying a protected modifier on this method does not break those implementations. This would however lead to the propagation of our protected modifier to the whole system as ProtoObject is the single root of the inheritance tree.

Reified message passing. Two other reflective methods that are using DNU: Context»#send:to:with:lookupIn: and Context»#send:to:with:super:. Both these methods are reifying message sends and providing the reflective APIs to send those messages reflectively. Their implementation includes the lookup of the message sent. In case this lookup fails they recursively call themselves with the new message doesNotUnderstand: being sent to the same receiver object. These are actually not self-sends. Applying blindly the protected modifier on doesNotUnderstand: would lead to infinite loops in those methods: once the DNU method has been prefixed in the class hierarchy, sending the message doesNotUnderstand: without the prefix will never find an implementation, leading to it being sent again and again.

This can however be rewritten to accommodate for a protected method. In a system where #PHARO would be installed on the whole class hierarchy, two solutions could be either prefixing manually those two calls or having a specific reflective API to call the protected method reflectively.

Other non-test senders. They are four different methods that call doesNotUnderstand: without being reflexive and without being related to tests:

- **SmalltalkImage»#newSpecialObjectsArray**. In this method the selector doesNotUnderstand: is put into the specialArray. This an array with well-known objects used by the VM, and #doesNotUnderstand: is in this array so that if the lookup implemented at the VM level fails, it knows which method to call instead. As for the reflective implementation of the lookup, the selector should be prefixed in the specialArray.

- **Context»#printOn:** uses the selector to check if the Context is in a specific state. This is never sent. Alternative checks could be implemented or the selector could be prefixed.

- **DebugSession»#isInterruptedContextDoesNotUnderstand** . The selector is also used to check for the interrupted context's specific state and is never sent.

- **ToolRegistry»#inspector** explicitly sends a DNU saying that inspector is not understood by ToolRegistry. This is a workaround so that its sub-hierarchy can answer with an instance of the inspector tool that can be used to inspect something else rather than trying to open an inspector on themselves which would lead to a loop. As this is a regular self-send, protecting doesNotUnderstand: has no impact on it.

Testing. The four remaining users of the doesNotUnderstand: selector are tests and mocks. Once again either manually prefixing or a specific reflective solution for

accessing protected methods solves those cases.

To conclude, the case of doesNotUnderstand: is a good starting point to apply #PHARO to Pharo's reflective API. It is relevant as it is often triggered by users and has mainly self-send senders and a few reflective senders. However, its location at the root of the inheritance hierarchy leads to #PHARO being applied to the whole system.

7.3.3 Applicability to Internal Methods of Reflective Classes

While the applicability of #PHARO directly to reflective methods is limited, we identified other methods relevant to reflection. Some reflective classes have protectable methods that probably emerged after refactoring to avoid code duplication.

The method collectArguments: is a typical example of an internal method that can be protected. This method is implemented and used in classes MessageSend and WeakMessageSend to get the arguments of the reified messages. It is called from MessageSend»#value, MessageSend»#valueWithArguments: and WeakMessageSend»#valueWithArguments:, only with self-sends. This method is part of the private protocol, denoting the fact it is for internal use only.

As most reflective methods are parts of APIs designed to be public even if they are not used in the standard library, it is difficult to know if a given reflective method is a good candidate for reflection. However, some reflective classes such as Class, Behavior or Object do have a private protocol. In Table 7.3, we show how many methods those classes have in their private protocol and how many could be protected. Out of 80 methods in private protocols, 28 can be protected.

Class	number of private methods	number of protectable methods
Behavior	**9**	**5**
BlockClosure	8	0
Class	2	0
ClassDescription	**6**	**1**
CompiledMethod	5	0
Context	**25**	**7**
Context class	1	0
Message	3	0
MessageSend	**1**	**1**
Object	**7**	**5**
Slot	**4**	**3**
WeakMessageSend	**9**	**6**

Table 7.3: Number of protectable methods in reflective classes' private protocol.

As previously, we may have false negatives due to polymorphism. Some false negatives could also be due to the fact that those private protocols do not have an

enforcement mechanism. For example, one of the methods in the private protocol of Class is getName. The method name is a non-private alternative checking that the return variable is not nil. This one should ideally be the one used in other classes that do not belong to the hierarchy. Out of three non-self sends of getName, one is in a test class and two in other classes. If the private protocol was respected, getName could be protected without breaking anything.

This shows that #PHARO can be beneficial to the reflective architecture of Pharo. It also raises the question of how reflective APIs could evolve to accommodate this protected modifier if it were integrated into the system.

7.4 Conclusion

In this chapter, we evaluated PROTDYN, a self-send-based visibility model calculated at compile-time for dynamically-typed languages presented previously. This evaluation is done along three axes:

- speed performance,
- memory footprint,
- applicability to Pharo's reflective infrastructure.

To evaluate the performance, we studied the runtime of four different benchmarks with three different VM configurations. Those three VMs have varying levels of optimizations. We also measured the number of cache hits and misses, to study whether the increased number of selectors would lead to worse performances. We show that the overhead introduced by our solution based on name-mangling is usually below $1\,\%$ and profits from common lookup optimizations such as global lookup caches and polymorphic inline caches.

To evaluate memory footprint, we studied the space taken by an application with three different configurations: without using #PHARO, with #PHARO in a regular case, and in the worst case. We show also that the introduction of protected methods increases memory consumption of static code structures by 13 % in a worst-case scenario, but 2.1 % in a realistic scenario. Our solution is a viable approach to introduce a visibility modifier in dynamic languages.

Finally, we studied the applicability of #PHARO to Pharo's reflective infrastructure. While we identified 17 reflective selectors that could be protected, there are even fewer for which it is relevant, as several are parts of public APIs. The highly open and reflective nature of Pharo leads to very few methods that can be surely identified as protectable. However, doesNotUnderstand: is one such example we studied in detail. We also identified that several reflective classes have private protocols. Among those methods we identified 28 out of 80 that could be protected,

providing an enforcement mechanism for the expected senders.

In the future, we will study the automatic rewriting of programs to use protected modifiers. This could build up on the static analysis we used to identify protectable methods.

CHAPTER 8

Conlusion

Contents

8.1 Contributions 111

8.2 Future Work 112

To conclude this thesis dissertation, we summarise the contributions of this thesis and discuss leads for future work.

In this thesis, we explored reflective APIs, particularly in the Pharo environment. Reflective APIs are mixed with non-reflective ones, especially in the kernel of the language. When analyzing library and application dependencies, static analysis results are limited, so we proposed to use mutation analysis. If the code coverage is good enough, it provides the dynamic information missing for the static analysis. We then designed a protected modifier model that is performant and backward-compatible. We analyzed its applicability to Pharo's reflective infrastructure.

8.1 Contributions

The contributions follow three main directions:

Analysing existing reflective features. Through this thesis we provided an up-to-date inventory of the reflective features in Pharo (over 500 identified reflective methods). We proposed a classification with nine categories and 40 subcategories. For each category we presented what kind of reflective methods were included, the possibilities they offered, how they are currently used, and discussed some potential re-designs.

We analyzed dependencies between the reflective methods and between reflective categories. We noticed that only three subcategories are actually independent: *Memory scanning, Accessing object identity, and bulk pointer swapping*. All the other are part of a network of dependencies as reflective operations build on each other. *Iterating and querying the hierarchy* is a hub of this network. We also noticed a layered organization inside some categories, where high-level operations rely on lower-level ones.

Analysing dependencies of application/library on reflection. We designed an approach based on mutation testing to assess the dependencies of an application or library on certain reflective APIs. We then implemented it for Pharo. It handles the fact that reflective features are often a core part of a language (Python, Pharo...) and cannot be simply removed. We reported the dependency analysis results in a structured manner, demonstrating the information one could get with different levels of granularity.

We compared our approach to static analysis on a selection of projects to evaluate its performance. On four out of five projects, our approach disambiguates more *potentially reflective call-sites* than the static analysis. In case of high code coverage, mutation analysis disambiguates three times more.

We also questioned the polymorphism between the reflective and non-reflective operation: it is very rarely used, with only one project leveraging it for 1.4% of its *potentially reflective call-sites*. We argue that reflective APIs could be re-designed to avoid ambiguities.

Once we had a better understanding of reflective infrastructures, we focused on control strategies and explored the idea of using visibility modifiers to limit access to the reflective API.

First steps in control the reflection. We proposed a protected method model for dynamically-typed object-oriented languages. We presented an implementation in Pharo that relies on default method lookup. It is based on the syntactical difference between self-sends and non-self-send. It relies on double method registration and name-mangling.

It presents a negligible run-time overhead usually below 1 % and takes advantage of common lookup optimizations such as global lookup caches and polymorphic inline caches. It has a low additional memory footprint of 2.1 % in a realistic scenario. It is retro-compatible, does not require a specific VM, and is applicable to other languages.

We discussed the applicability of such a protected modifier to Pharo's reflective API. There are very few reflective methods that are currently protectable. However, doesNotUnderstand: could be a good starting point and we analyzed this case in detail. Reflective classes also have internal methods that could be protected (we identified 28 of them), which a protected modifier would clearly separate from the public reflective APIs.

8.2 Future Work

This thesis sets up the stage for future research, both in Programming language design and secure application development.

Comparing reflective APIs accross languages. We established a classification of reflective APIs that is based on what is available in Pharo. A next step for this work is to proceed to a similar analysis of reflective APIs for other languages such as Python, Ruby, and Java, and compare the available reflective operations. Such a study could back up the current categorizations and allow for a comparison of available reflective facilities across languages. It could also be extended to include instrumentation libraries and APIs that were excluded from this analysis.

Modularising Pharo's reflective APIs. Reflection is commonly used for developers' tools, but most of the time domain applications do not require it to work. In order to limit the number of available reflective methods in a production environment, it requires a redesign. For example, to gather the non-essential reflective operations into one or several packages separated from the kernel. These packages could then be loaded only when necessary. This would limit the impact of reflection by preventing some inadvertent uses. This raises the question: Which operation can be separated from the kernel? Which ones can not be removed? What could be done to limit the impact of the ones that cannot be packaged away? What is a minimally reflective kernel that still can support the full range of reflective API once they are loaded?

Designing application/libary-based access control. We designed a tool allowing developers to assess how much they rely on reflection. Could we design an access control mechanism that would allow us to control which application/library has access to which reflective operatives while not necessitating a specific runtime? By making explicit the request to access reflection, one could know which library or application has access to which reflective features. This would also limit inadvertent uses.

Security risk evaluation of reflective operations. While we pointed out some issues caused by reflection, those were not tied to specific reflective operations. We inventoried and classified reflective operations based on their semantics, *i.e.* what kind of manipulation they allow. A next step is to analyze more precisely the potential issues that reflective operations may generate in productive applications and their impact on the safety and security of the application: Which categories of reflective operations should be avoided to reduce performance issues? Which ones can lead to security issues for an application running in production? Which ones create more technical debt?

Providing meaningful feedback for reflection users. Currently, Pharo's development environment displays a "questionable message" warning as soon as a reflective message is used. This is not very informative. This raises the following question: Could we provide more meaningful messages, based on what kind of operation

is used, the issues it is known to cause (ex: slow reification), and alternatives to consider? This could rely on the existing classification, and on the risk evaluation mentioned in the previous paragraph.

Combining RAPIM information with static analysis. We concluded that for four out of five projects, RAPIM disambiguates more *potentially reflective call-sites* than the static analysis. However, in four cases, the overlap between what is disambiguated and what is not disambiguated is below 20% of the *potentially reflective call-sites* (See Table 4.4). This shows that there is room for improvement by combining the information from RAPIM with the information from static analysis.

Studying the larger applicability of #PHARO. We studied the applicability of #PHARO to the reflective infrastructure. We could widen this study to the whole language standard library to better study the relevance of such modifiers. This would provide a deeper insight to answer the question: Is this semantics adapted to Pharo in practice?

APPENDIX A

Exhaustive classification of runtime reflective operations in Pharo

Contents

A.1 Object Inspection 115
A.2 Object Modification 116
A.3 Class Structural Inspection 117
A.4 Class Structural Modification 123
A.5 Method Creation 127
A.6 Structural Queries on Methods 127
A.7 Message Sending and Code Execution 128
A.8 Chasing and Atomic Pointer Swapping 131
A.9 Memory Scanning 131
A.10 Stack Manipulation 132

A.1 Object Inspection

The first family of reflective operations is centered around object inspection. Rivard describes these operations in the *Meta-Operations* category, but he groups together inspection and modification. Our category is composed of three subcategories:

State inspection. These operations allow one to access the number of indexable variables, and the values of instance variables, if they include a specific object

- Context»objectSize:
- Object»instVarAt:
- Object>instVarNamed:
- ProtoObject»instVarsInclude:

Accessing object identity. These operations allow one to identify one object

- ProtoObject»basicIdentityHash
- ProtoObject»identityHash

Accessing object class. This operation allows one to access the class of an object :

- ProtoObject»class
- Context»objectClass:

A.2 Object Modification

The second family of reflective operations is centered around object modification. It is the counterpart of the first one and it is composed of **State modification**, **Manipulating object identity**, and **Object's class change**.

State modification. These operations allow one to set the value of an instance variable.

- Object»instVarAt:put: Write an instance variable using its index
- Object»instVarNamed:put: Write an instance variable using its name.

Manipulating object identity. As a side effect of reference swapping, the reflective operation ProtoObject»becomeForward:copyHash: manipulates the hash of an object.

Object's class change. These operations allow one to change the class of an object to another one:

- Behavior»adoptInstance:
- Metaclass»adoptInstance:from:
- Object»primitiveChangeClassTo:

A.3 Class Structural Inspection

This category groups reflective APIs on the query over the class structure and its constituents: methods, variables (instance/class/slots). It is composed of the following APIs:

- **Class/metaclass shift** to support the navigation from a class to its metaclass and the inverse,
- **Iterating and querying hierarchy** to support hierarchy analysis with, in particular, testing for belonging to a specific class or class hierarchy (*e.g.* isKindOf:),
- **Instance variable inspection**, **Class variable inspection**, **Shared pool inspection**, **Slot inspection** all deal about variables. **Slot inspection** provides a higher level view compared to instance variables. Some slots can come from traits. Therefore there is a difference between localSlots (not coming from traits) and slots (taking into account that they may come from traits),
- **Selector and method inspection** supports if a class has (abstract) methods, an object responds to a specific selector, and all the classes implementing a set of selectors.
- **Variable lookup** supports the access to the binding of a variable.
- **Pragmas** supports the query and iteration of class annotations (named pragmas in Pharo).
- **Class kind testing** supports the testing of the state of a class from a system perspective (obsolete, anonymous...).

Class/metaclass shift. Those operations deal with testing and navigating between the class and instance side of classes.

- ClassDescription»classSide / Class»classSide / Metaclass»classSide
- ClassDescription»instanceSide / Class»instanceSide / Metaclass»instanceSide
- ClassDescription»isClassSide / CompiledMethod»isClassSide
- ClassDescription»isInstanceSide
- Metaclass»isMetaclassOfClassOrNil
- ClassDescription»hasClassSide / Class»hasClassSide / Metaclass»hasClassSide
- Object»isClass / Class»isClass / Trait»isClass / Metaclass»isClass
- Behavior»isMeta / ClassDescription»isMeta / Metaclass»isMeta

***Iterating and querying hierarchy*.** These operations allow one to query the hierarchy of a class and iterate over those. They allow one to access superclasses and subclasses, testing for common ancestry in the class hierarchy, and testing that an object is an instance or subinstance of a class.

- Class»subclasses / MetaClass»subclasses / UndefinedObject»subclasses
- Behavior»allSubclasses
- Behavior»withAllSubclasses
- Behavior»obsoleteSubclasses / Metaclass»obsoleteSubclasses
- Class»hasSubclasses
- Behavior»superclass
- Behavior»allSuperclasses
- Behavior»withAllSuperclasses
- Behavior»allSuperclassesIncluding:
- Behavior»allSubclassesWithLevelDo:startingLevel:
- Class»subclassesDo: / Metaclass»subclassesDo: / UndefinedObject»subclassesDo:
- Behavior»allSubclassesDo:
- Behavior»withAllSubclassesDo:
- Behavior»allSuperclassesDo: / UndefinedObject»allSuperclassesDo:
- Behavior»withAllSuperclassesDo:
- Behavior»withAllSuperAndSubclasses
- Behavior»withAllSubAndSuperclassesDo:
- Object»isKindOf:
- Object»isMemberOf:
- Behavior»kindOfSubclass
- Class»commonSuperclassWith: / UndefinedObject»commonSuperclassWith:
- Behavior»whichSuperclassSatisfies:
- Behavior»inheritsFrom:

- Behavior»includesBehavior:
- Behavior»isRootInEnvironment
- Behavior»selectSuperclasses:
- Behavior»selectSubclasses:

Instance variable inspection. These operations allow one to query and get instance variable names, iterate over those, and get their index and the class defining a specific instance variable.

- Behavior»instVarNames / ClassDescription»instVarNames
- Behavior»allInstVarNames / ClassDescription»allInstVarNames
- ClassDescription»instanceVariableNamesDo:
- ClassDescription»hasInstVarNamed:
- Behavior»definedVariables / Class»definedVariables
- ClassDescription»allInstVarNamesEverywhere
- ClassDescription»classThatDefinesInstVarNamed:
- Behavior»whichClassDefinesInstVar:
- Behavior»instSize
- Object»basicSize / Context»basicSize
- Behavior»instVarNames

Class variable inspection. These operations allow one to query and test class variables, get class variable names, get their values, iterate over those, class defining a specific instance variable and who is using them.

- Class»classVariables / Metaclass»classVariables
- Behavior»allClassVarNames
- Behavior»classVarNames / Class»classVarNames / Metaclass»classVarNames
- Class»hasClassVariable:
- Class»hasClassVarNamed: / Metaclass»hasClassVarNamed:
- Class»classVariableNamed:ifAbsent:

- Class»definesClassVariable:
- Class»definesClassVariableNamed:
- Class»readClassVariableNamed:
- ClassDescription»classThatDefinesClassVariable:
- Behavior»whichClassDefinesClassVar:
- Class»usesClassVarNamed:

The methods ClassDescription»classThatDefinesClassVariable: and Behavior»whichClassDefinesClassVar: looks the same and the mix between variable and var is prejudicial.

Shared pool inspection. These operations allow one to query and test on sharedPools and where there are used. They are important to navigate with the IDE.

- ClassDescription»sharedPools / Class»sharedPools
- Behavior»allSharedPools / ctClassDescription»allSharedPools / ctClass»allSharedPools
- Class»sharedPoolNames / Metaclass»sharedPoolNames
- ClassDescription»hasSharedPools / Class»hasSharedPools
- ClassDescription»sharedPoolOfVarNamed / Class»sharedPoolOfVarNamed
- Class»sharedPoolsDo:
- Class»classPool / Metaclass»classPool
- ClassDescription»usesLocalPoolVarNamed: / Class»usesLocalPoolVarNamed:
- ClassDescription»usesPoolVarNamed: / Class»usesPoolVarNamed:
- ClassDescription»includesSharedPoolNamed:

Slot inspection. These operations allow one to query and test on slots in a class, or read those in an object. This is a higher-level view compared to instance variables. Some slots can come from traits. Therefore there is a difference between localSlots (without traits) and slots (with).

- Behavior»instanceVariables (returns a collection of slots)
- Behavior»slots / ClassDescription»slots / TraitedMetaclass»slots
- ClassDescription»localSlots

- Behavior»allSlots / ClassDescription»allSlots
- ClassDescription»slotNames
- ClassDescription»hasSlotNamed:
- ClassDescription»slotNamed:
- ClassDescription»slotNamed:ifFound:
- ClassDescription»slotNamed:ifFound:ifNone:
- Object»readSlot:
- Object»readSlotNamed:
- ClassDescription»definesSlot:
- ClassDescription»definesSlotNamed:

Selector and method inspection. These operations allow one to test if a class has (abstract) methods, an object responds to a specific selector, and all the classes that implement a set of selectors.

- Behavior»hasMethods / Class»hasMethods
- Behavior»hasAbstractMethods / Class»hasAbstractMethods
- Object»respondsTo:
- ClassDescription»classesThatImplementAllOf:

Getting the selectors and local selectors, iterating over them, and querying them :

- Behavior»includesSelector:
- Behavior»includesLocalSelector: / TraitedClass»includesLocalSelector: / Traited-Metaclass»includesLocalSelector:
- Behavior»isDisabledSelector:
- Behavior»isLocalSelector: / TraitedClass»isLocalSelector: / TraitedMetaclass»isLocalSelector:
- Behavior»selectors
- Behavior»selectorsDo:
- Behavior»selectorsWithArgs:

- Behavior»whichClassIncludesSelector:

Getting the methods, iterating over them, and querying them :

- Behavior»methods
- Behavior»methodNamed:
- Behavior»includesMethod:
- Behavior»methodsDo:
- Behavior»selectorsAndMethodsDo:

Variable lookup. These operations allow one to trigger a lookup for a variable in the scope of the receiver.

- Behavior»classBindingOf: / SharedPool class»classBindingOf:
- Object»bindingOf: / Behavior»bindingOf: / Class»bindingOf: / MetaClass»bindingOf:
- Behavior»lookupVar: / ctContext»lookupVar: / ctSystemDictionnary»lookupVar:
- Behavior»lookupVar:declare: / ctContext»lookupVar:declare: / ctSystemDictionnary»lookupVar:declare:
- Behvior»lookupVarForDeclaration:

Pragmas. Pragmas are method annotations that were introduced both in VisualWorks and Pharo over the year. These operations (Class»pragmasClass»pragmasDo:)allow one to get all the pragmas of a class and iterate over them.

Class kind testing. Test various properties of classes: usage, anonymity, obsolescence.

- Behvior»isAnonymous / Class»isAnonymous /MetaClass»isAnonymous
- Object»isClassOrTrait / Class»isClassOrTrait
- Behavior»isUsed / Class»isUsed / Trait»isUsed / Metaclass»isUsed
- Behavior»isObsolete / Class»isObsolete / Metaclass»isObsolete

A.4 Class Structural Modification

This category is the counterpart of the previous one. It is composed of the following APIs whose objectives are clear: *Hierarchy modification*, **Instance variable modification**, **Shared pool modification**, **Slot modification**, **Selector/Method modification**, **Old class creation**, **Fluid class creation**, and **Anonymous class creation**. It focuses on the modification of the structural relation a class has with its constituents.

Hierarchy modification. These operations allow one to edit the class hierarchy either by adding/removing/changing subclasses, or changing the superclass.

- Class»subclasses:
- Behavior»superclass:
- Behavior»basicSuperclass:
- Class»addSubclass: / Metaclass»addSubclass: / UndefinedObject»addSubclass:
- Class»removeSubclass: / Metaclass»removeSubclass: / UndefinedObject»removeSubclass:
- Behavior»removeAllObsoleteSubclasses
- Behavior»addObsoleteSubclass: / Metaclass»addObsoleteSubclass:

Instance variable modification. It allows one to add or remove an instance variable (by name).

- ClassDescription»addInstVarNamed: / Class»addInstVarNamed: / Metaclass»addInstVarNamed:
- ClassDescription»removeInstVarNamed:

Class variable modification. These operations allow one to add or remove a class variable.

- Class»addClassVariable:
- Class»addClassVarNamed:
- Class»removeClassVariable:
- Class»removeClassVarNamed:

Shared pool modification. These operations allow one to add, change, or remove shared pools.

- Class»sharedPools:
- Class»addSharedPool:
- Class»addSharedPoolNamed:
- Class»removeSharedPool:
- Class»classPool:

Slot modification.

- ClassDescription»addSlot: / Class»addSlot: / Metaclass»addSlot:
- ClassDescription»removeSlot: / Class»removeSlot: / Metaclass»removeSlot:
- Class»addClassSlot:
- Class»removeClassSlot:
- Object»writeSlot:value:
- Object»writeSlotNamed:value
- Class»writeClassVariableNamed:value:

Selector/Method modification. These operations allow one to add or remove selectors, or query the class including one or more selectors.

- Behavior»removeSelector: / ClassDescription»removeSelector: / TraitedMetaclass»removeSelector: / TraitedClass»removeSelector:
- Behavior»removeSelectorSilently:
- Behavior»addSelectorSilently:withMethod: / ClassDescription»addSelectorSilently:withMethod:
- Behavior»addSelector:withMethod: / ClassDescription»addSelector:withMethod: / TraitedMetaclass»addSelector:withMethod:
- addSelector:withMethod: / TraitedClass»>addSelector:withMethod: / TraitedClass class»>addSelector:withMethod:
- Behavior»addSelector:withRecompiledMethod: / TraitedMetaclass»addSelector:withRecompiledMethod: / TraitedClass»addSelector:withRecompiledMethod:

Old class creation. The following lists present the partial old API of Class for class creations. It ignores the message supporting optional arguments. It shows clearly why it is about to be deprecated.

- subclass:instanceVariableNames:classVariableNames:category:
- subclass:instanceVariableNames:classVariableNames:package:
- subclass:instanceVariableNames:classVariableNames:poolDictionaries:category:
- subclass:instanceVariableNames:classVariableNames:poolDictionaries:package:
- subclass:layout:slots:classVariables:package:
- subclass:layout:slots:classVariables:poolDictionaries:package:
- subclass:slots:classVariables:package:
- subclass:slots:classVariables:poolDictionaries:package:
- variableByteSubclass:instanceVariableNames:classVariableNames:category:
- variableByteSubclass:instanceVariableNames:classVariableNames:package:
- variableByteSubclass:instanceVariableNames:classVariableNames:poolDictionaries:category:
- variableByteSubclass:instanceVariableNames:classVariableNames:poolDictionaries:package:
- variableDoubleByteSubclass:instanceVariableNames:classVariableNames:package:
- variableDoubleWordSubclass:instanceVariableNames:classVariableNames:package:
- variableSubclass:instanceVariableNames:classVariableNames:category:
- variableSubclass:instanceVariableNames:classVariableNames:package:
- variableSubclass:instanceVariableNames:classVariableNames:poolDictionaries:category:
- variableSubclass:instanceVariableNames:classVariableNames:poolDictionaries:package:
- variableWordSubclass:instanceVariableNames:classVariableNames:category:
- variableWordSubclass:instanceVariableNames:classVariableNames:package:
- variableWordSubclass:instanceVariableNames:classVariableNames:poolDictionaries:category:
- variableWordSubclass:instanceVariableNames:classVariableNames:poolDictionaries:package:
- weakSubclass:instanceVariableNames:classVariableNames:category:

- weakSubclass:instanceVariableNames:classVariableNames:package:
- weakSubclass:instanceVariableNames:classVariableNames:poolDictionaries:category:
- weakSubclass:instanceVariableNames:classVariableNames:poolDictionaries:package:
- ephemeronSubclass:instanceVariableNames:classVariableNames:package:
- ephemeronSubclass:instanceVariableNames:classVariableNames:poolDictionaries:category:
- ephemeronSubclass:instanceVariableNames:classVariableNames:poolDictionaries:package:
- immediateSubclass:instanceVariableNames:classVariableNames:package:
- immediateSubclass:instanceVariableNames:classVariableNames:poolDictionaries:category:
- immediateSubclass:instanceVariableNames:classVariableNames:poolDictionaries:package:

Fluid class creation. The following list presents the equivalent of the previous API in the new fluid class creation API:

- Beharior»« / Metaclass»« / Trait class»«
- FluidClassBuilder»layout:
- FluidBuilder»traits:
- FluidClassBuilder»sharedVariables:
- FluidClassBuilder»superclass:
- FluidClassBuilder»sharedPools:
- FluidBuilder»slots:
- FluidBuilder»package:
- FluidBuilder»tag:

Anonymous class creation. Pharo introduced the notion of anonymous classes that support instance-specific behavior.

- Class»newAnonymousSubclass / Metaclass»newAnonymousSubclass

A.5 Method Creation

The API **Compiled method creation** is a low-level API that supports the definition of compiled methods. Such an API is often ignored but it is central because it is responsible for the creation of new compiled methods that can then be further installed and executed.

Compiled method creation These operations support the creation of compile methods.

- CompiledMethod class»newMethod:header:
- CompileMethod»literalAt:put:
- CompiledMethod»classBinding:
- CompiledMethod»at:put:

A.6 Structural Queries on Methods

This category supports the cross-referencing between methods, instance variables, and classes. It is composed of two APIs: **Method slot uses** (which supports the access to a variable and modification of a variable) and **Method element references** (which is a low-level API querying the internal implementation of methods).

Method slot uses. Query on which method is using some specific slot and how.

- Behavior»allMethodsAccessingSlot:
- Behavior»allMethodsReadingSlot:
- Behavior»allMethodsWritingSlot:
- Behavior»methodsAccessingSlot:
- Behavior»methodsReadingSlot:
- Behavior»methodsWritingSlot:
- Behavior»hasMethodAccessingVariable:

Method element references. Query which method/selector is using a given literal.

- Behavior»whichMethodsReferTo:
- Behavior»whichSelectorsReferTo:
- Behavior»thoroughHasSelectorReferringTo:
- Behavior»thoroughWhichMethodsReferTo:
- Behavior»thoroughWhichMethodsReferTo:specialIndex:
- Behavior»thoroughWhichSelectorsReferTo:
- Behavior»hasSelectorReferringTo:

Class references. Behavior»usingMethods returns methods is referencing a given class.

A.7 Message Sending and Code Execution

This category of operations allows us to explicitly send messages, handle lookup failures, or execute compiled methods. It is composed of different APIs: **Reflective message send** (which supports the lookup and execution of a method), **Arbitrary method/primitive execution** (which supports the execution of methods without lookup), **Method lookup** (which simulate the method lookup), **Control message passing** (which supports the possibility to control message sent), and **Message send reification** (which provides ways to access message information).

Reflective message send

- Object»perform:
- Object»perform:orSendTo:
- Object»perform:with:
- Object»perform:with:with:
- Object»perform:with:with:with:
- Object»perform:with:with:with:with:
- Object»perform:withArguments:
- Object»perform:withArguments:inSuperclass:
- Object»perform:withEnoughArguments:

Arbitrary method/primitive execution. While the message perform: supports message sending, the following messages bypass the method lookup and execute a given compiler method.

- ProtoObject»withArgs:executeMethod:
- ProtoObject»tryPrimitive:withArgs:
- CompiledMethod»valueWithReceiver:arguments:
- ProtoObject»executeMethod:
- CompiledMethod»receiver:withArguments:executeMethod:

Method lookup. These operations look up for selectors, and query about this lookup.

- Behavior»lookupSelector:
- Behavior»canPerform:
- Behavior»canUnderstand:

Control message passing.

- ProtoObject»cannotInterpret: Is sent to the receiver if the lookup finds a nil method dictionary. Use a special lookup starting above the class with the nil method dictionary. Has been implemented for proxy implementation.
- ProtoObject»doesNotUnderstand: / Object»doesNotUnderstand: Is sent to the receiver of the message if the lookup reaches the root of the class hierarchy without finding the receiver.
- ReflectiveMethod»run:with:in:

Message send reification. These operations allow one to query a message send to know if it is its arguments, receiver and selectors, and to get a corresponding message.

- MessageSend»isValid
- MessageSend»arguments / Message»arguments
- MessageSend»numArgs / Message»numArgs
- MessageSend»collectArguments:
- MessageSend»receiver

- MessageSend»selector / Message»selector
- MessageSend»message
- Message»lookupClass

These operations allow one to modify a message.

- MessageSend»arguments:
- Message»argument:
- MessageSend»receiver:
- MessageSend»selector:
- Message»setSelector:
- Message»lookupClass:
- Message»setSelector:arguments:

Runtime and Evaluation These operations allow one to evaluate a messageSend or convert it to a messageSend.

- Message»asSendTo:
- Message»sends:
- Message»sentTo:
- MessageSend»value BlockClosure»value
- MessageSend»value: BlockClosure»value:
- MessageSend»value:value: BlockClosure»value:value:
- MessageSend»value:value:value: BlockClosure»value:value:value:
- MessageSend»valueWithArguments: BlockClosure»valueWithArguments:
- MessageSend»valueWithEnoughArguments: BlockClosure»valueWithEnoughArguments:
- MessageSend»cull: BlockClosure»cull:
- MessageSend»cull:cull: BlockClosure»cull:cull:
- MessageSend»cull:cull:cull: BlockClosure»cull:cull:cull:

The following methods perform explicit message sending (do the lookup and then apply a compiled method if any is found):

- Object»perform:(with: with: with: with:)
- Object»perform: orSendTo:
- Object»perform:withArguments:
- Object»perform:withArguments: inSuperclass:
- Object»perform:withEnoughArguments:

A.8 Chasing and Atomic Pointer Swapping

The APIs in this category are **Find pointers to** and **Bulk pointer swapping** (supports the atomic swapping to references). The first one is rarely mentioned but Pharo supports the possibility of identifying pointers to a given object (*e.g.* ProtoObject»pointersTo and ProtoObject»pointsTo:).

***Find pointers to*.**

- ProtoObject»pointersTo
- ProtoObject»pointersToAmong:
- ProtoObject»pointersToExcept:
- ProtoObject»pointersToExcept:among:
- Object»pointsOnlyWeaklyTo:
- ProtoObject»pointsTo:

***Bulk pointer swapping*.**

- ProtoObject»become: swaps the references in both ways,
- ProtoObject»becomeForward: swaps only towards a given object.
- ProtoObject»becomeForward:copyHash:

A.9 Memory Scanning

This category contains two APIs *Memory scanning* that support the traversal of the complete heap and *Instances of a class* that give access to all the instances of a class.

Memory scanning. The operations allow one to traverse the heap.

- Object»someObject
- ProtoObject»nextObject

Instances of a class. These operations allow one to get or iterate over the instances and subinstances of a class.

- Behavior»someInstance
- ProtoObject»nextInstance
- Behavior»allInstances
- Behavior»allInstancesOrNil
- Behavior»allInstancesDo:
- Behavior»allSubInstancesDo:
- Behavior»allSubInstances

A.10 Stack Manipulation

This category groups together all APIs that support traversing and modifying the execution stack. These APIs are accessible from two main entry points: the Process class that provides access to the existing processes and their suspended execution stack, and the thisContext pseudo-variable that provides access to the current method execution. Both these entry points provide instances of Context that represent a method execution and that represent the execution stack as a linked list.

Context These operations allow one to get information on the execution context (stack, sender, receiver, ...). The thisContext pseudo-variable supports access to the current execution context. This supports the creation of continuations and co-routines.

- Context»selector
- Context»sender
- Context»activeOuterContext
- Context»arguments
- Context»at:

- Context»at:put:
- Context»method
- Context»methodNode
- Context»outerContext
- Context»outerMostContext
- Context»receiver
- Context»tempAt:
- Context»tempAt:put:

Controlling the stack These operations allow one to control the execution of the current program.

- Context»top
- Context»stepUntilSomethingOnStack
- Context»runUntilErrorOrReturnFrom:
- Context»resume:through:
- Context»activateMethod:withArgs:receiver:class:
- Context»terminate
- Context»send:to:with:lookupIn:
- Context»resumeEvaluating:
- Context»jump
- Context»terminateTo:
- Context»send:to:with:super:
- Context»return
- Context»pop
- Context»shortDebugStack
- Context»return:
- Context»push:

- Context»step
- Context»return:from:
- Context»resume
- Context»stepToCallee
- Context»return:through:
- Context»resume:

APPENDIX B

SMALLTALKLITE Formal Semantics

This is here as a reference for the reader, to facilitate the understanding of Chapter 6, this is a reproduction of a section from Bergel *et al.* [**?**] introducing a formal semantics. This is not part of the contributions of this thesis.

In this appendix we present SMALLTALKLITE [**?**], a Smalltalk-like dynamic language featuring single inheritance, message-passing, field access and update, and **self** and **super** sends. SMALLTALKLITE is similar to CLASSICJAVA, but removes interfaces and static types. Fields are private in SMALLTALKLITE, so only local or inherited fields may be accessed.

B.1 SMALLTALKLITE Reduction Semantics

The syntax of SMALLTALKLITE is shown in Figure B.2. SMALLTALKLITE is similar to CLASSICJAVA while eliding the features related to static typing. We similarly ignore features that are not relevant to a discussion of traits, such as reflection or class-side methods.

To simplify the reduction semantics of SMALLTALKLITE, we adopt an approach similar to that used by Flatt *et al.* [**?**], namely we annotate field accesses and **super** sends with additional static information that is needed at "run-time". This extended redex syntax is shown in Figure B.1. The figure also specifies the evaluation contexts for the extended redex syntax in Felleisen and Hieb's notation [**?**].

Predicates and relations used by the semantic reductions are listed in Figure B.4. (The predicates CLASSESONCE(P) *etc* are assumed to be preconditions for valid programs, and are not otherwise explicitly mentioned in the reduction rules.)

$P \vdash \langle \epsilon, \mathcal{S} \rangle \hookrightarrow \langle \epsilon', \mathcal{S}' \rangle$ means that we reduce an expression (redex) ϵ in the context of a (static) program P and a (dynamic) store of objects $\mathcal{S}$ to a new expression ϵ' and (possibly) updated store $\mathcal{S}'$. A redex ϵ is essentially an expression e in which field names are decorated with their object contexts, *i.e.* f is translated to $o.f$, and **super** sends are decorated with their object and class contexts. Redexes and their subexpressions reduce to a value, which is either an object identifier or nil. Subexpressions may be evaluated within an expression context E.

The store consists of a set of mappings from object identifiers $oid \in \text{dom}(\mathcal{S})$ to tuples $\langle c, \{f \mapsto v\} \rangle$ representing the class c of an object and the set of its field values. The initial value of the store is $\mathcal{S} = \{\}$.

$$
\begin{array}{rcl}
\epsilon & = & v \mid \textbf{new}\ c \mid x \mid \textbf{self} \mid \epsilon.f \mid \epsilon.f{=}\epsilon \\
& \mid & \epsilon.m(\epsilon^*) \mid \textbf{super}\langle o, c\rangle.m(\epsilon^*) \mid \textbf{let}\ x{=}\epsilon\ \textbf{in}\ \epsilon \\
E & = & [\,] \mid o.f{=}E \mid E.m(\epsilon^*) \mid o.m(v^*\ E\ \epsilon^*) \\
& \mid & \textbf{super}\langle o, c\rangle.m(v^*\ E\ \epsilon^*) \mid \textbf{let}\ x{=}E\ \textbf{in}\ \epsilon \\
v, o & = & \mathsf{nil} \mid \mathit{oid}
\end{array}
$$

Figure B.1: Redex syntax.

$$
\begin{array}{rcl}
P & = & \mathit{defn}^*\, e \\
\mathit{defn} & = & \textbf{class}\ c\ \textbf{extends}\ c\ \{\ f^*\mathit{meth}^*\ \} \\
e & = & \textbf{new}\ c \mid x \mid \textbf{self} \mid \mathsf{nil} \\
& \mid & f \mid f{=}e \mid e.m(e^*) \\
& \mid & \textbf{super}.m(e^*) \mid \textbf{let}\ x{=}e\ \textbf{in}\ e \\
\mathit{meth} & = & m(x^*)\ \{\ e\ \} \\
c & = & \text{a class name} \mid \mathsf{Object} \\
f & = & \text{a field name} \\
m & = & \text{a method name} \\
x & = & \text{a variable name}
\end{array}
$$

Figure B.2: SMALLTALKLITE syntax.

Translation from the main expression to an initial redex is specified by the $o[\![e]\!]_c$ function (see Figure B.3). This binds fields to their enclosing object context and binds **self** to the *oid* of the receiver. The initial object context for a program is nil. (*i.e.* there are no global fields accessible to the main expression). So if e is the main expression associated with a program P, then $\mathsf{nil}[\![e]\!]_{\mathsf{Object}}$ is the initial redex.

The reductions are summarised in Figure B.5.

new c [*new*] reduces to a fresh *oid*, bound in the store to an object whose class is c and whose fields are all nil. A (local) field access [*get*] reduces to the value of the field. Note that it is syntactically impossible to access a field of another object. The redex notation $o.f$ is only generated in the context of the object o. Field update [*set*] simply updates the corresponding binding of the field in the store.

When we send a message [*send*], we must look up the corresponding method body e, starting from the class c of the receiver o. The method body is then evaluated in the context of the receiver o, binding **self** to the receiver's *oid*. Formal parameters to the method are substituted by the actual arguments (see Figure B.6). We also pass in the actual class in which the method is found, so that **super** sends have the right context to start their method lookup.

$$
\begin{array}{rcl}
o[\![\textbf{new } c']\!]_c & = & \textbf{new } c' \\
o[\![x]\!]_c & = & x \\
o[\![\textbf{self}]\!]_c & = & o \\
o[\![\mathsf{nil}]\!]_c & = & \mathsf{nil} \\
o[\![f]\!]_c & = & o.f \\
o[\![f{=}e]\!]_c & = & o.f{=}o[\![e]\!]_c \\
o[\![e.m(e_i^*)]\!]_c & = & o[\![e]\!]_c.m(o[\![e_i]\!]_c^*) \\
o[\![\textbf{super}.m(e_i^*)]\!]_c & = & \textbf{super}\langle o, c\rangle.m(o[\![e_i]\!]_c^*) \\
o[\![\textbf{let } x{=}e \textbf{ in } e']\!]_c & = & \textbf{let } x{=}o[\![e]\!]_c \textbf{ in } o[\![e']\!]_c
\end{array}
$$

Figure B.3: Translating expressions to redexes.

super sends [*super*] are similar to regular message sends, except that the method lookup must start in the superclass of the class of the method in which the **super** send was declared. When we reduce the **super** send, we must take care to pass on the class c'' of the method in which the **super** method was found, since that method may make further **super** sends. **let in** expressions [*let*] simply represent local variable bindings.

Errors occur if an expression gets "stuck" and does not reduce to an *oid* or nil. This occurs if a non-existent variable, field, or method is referenced (for example, when sending a message to nil). For the purpose of this paper, we are not concerned with errors, so we do not introduce any special rules for these cases.

$\prec_P$ Direct subclass
$c \prec_P c' \iff \textbf{class}\ c\ \textbf{extends}\ c' \cdots \{\cdots\} \in P$

$\leq_P$ Indirect subclass
$c \leq_P c' \equiv$ transitive, reflexive closure of $\prec_P$

$\in_P$ Field defined in class
$f \in_P c \iff \textbf{class} \cdots \{\cdots f \cdots\} \in P$

$\in_P$ Method defined in class
$\langle m, x^*, e\rangle \in_P c \iff \textbf{class} \cdots \{\cdots m(x^*)\{e\} \cdots\} \in P$

$\in_P^*$ Field defined in c
$f \in_P^* c \iff \exists c', c \leq_P c', f \in_P c'$

$\in_P^*$ Method lookup starting from c
$\langle c, m, x^*, e\rangle \in_P^* c' \iff c' = \mathit{min}\{c'' \mid \langle m, x^*, e\rangle \in_P c'', c \leq_P c''\}$

CLASSESONCE(P) Each class name is declared only once
$\forall c, c',\ \textbf{class}\ c \cdots \textbf{class}\ c' \cdots$ is in $P \Rightarrow c \neq c'$

FIELDONCEPERCLASS(P) Field names are unique within a class declaration
$\forall f, f',\ \textbf{class}\ c \cdots \{\cdots f \cdots f' \cdots\}$ is in $P \Rightarrow f \neq f'$

FIELDSUNIQUELYDEFINED(P) Fields cannot be overridden
$f \in_P c, c \leq_P c' \implies f \notin_P c'$

METHODONCEPERCLASS(P) Method names are unique within a class declaration
$\forall m, m',\ \textbf{class}\ c \cdots \{\cdots m(\cdots)\{\cdots\} \cdots m'(\cdots)\{\cdots\} \cdots\}$ is in $P \Rightarrow m \neq m'$

COMPLETECLASSES(P) Classes that are extended are defined
range$(\prec_P) \subseteq$ dom$(\prec_P) \cup \{\mathsf{Object}\}$

WELLFOUNDEDCLASSES(P) Class hierarchy is an order
$\leq_P$ is antisymmetric

CLASSMETHODSOK(P) Method overriding preserves arity
$\forall m, m', \langle m, x_1 \cdots x_j, e\rangle \in_P c, \langle m, x'_1 \cdots x'_k, e'\rangle \in_P c', c \leq_P c' \implies j = k$

Figure B.4: Relations and predicates for SMALLTALKLITE.

$P \vdash \langle E[\textbf{new}\ c], \mathcal{S}\rangle \hookrightarrow \langle E[\mathit{oid}], \mathcal{S}[\mathit{oid} \mapsto \langle c, \{f \mapsto \mathsf{nil} \mid \forall f, f \in_P^* c\}\rangle]\rangle$ [*new*]
where $\mathit{oid} \notin$ dom$(\mathcal{S})$

$P \vdash \langle E[o.f], \mathcal{S}\rangle \hookrightarrow \langle E[v], \mathcal{S}\rangle$ [*get*]
where $\mathcal{S}(o) = \langle c, \mathcal{F}\rangle$ and $\mathcal{F}(f) = v$

$P \vdash \langle E[o.f{=}v], \mathcal{S}\rangle \hookrightarrow \langle E[v], \mathcal{S}[o \mapsto \langle c, \mathcal{F}[f \mapsto v]\rangle]\rangle$ [*set*]
where $\mathcal{S}(o) = \langle c, \mathcal{F}\rangle$

$P \vdash \langle E[o.m(v^*)], \mathcal{S}\rangle \hookrightarrow \langle E[o[\![e[v^*/x^*]]\!]_{c'}], \mathcal{S}\rangle$ [*send*]
where $\mathcal{S}[o] = \langle c, \mathcal{F}\rangle$ and $\langle c, m, x^*, e\rangle \in_P^* c'$

$P \vdash \langle E[\textbf{super}\langle o, c\rangle.m(v^*)], \mathcal{S}\rangle \hookrightarrow \langle E[o[\![e[v^*/x^*]]\!]_{c''}], \mathcal{S}\rangle$ [*super*]
where $c \prec_P c'$ and $\langle c', m, x^*, e\rangle \in_P^* c''$ and $c' \leq_P c''$

$P \vdash \langle E[\textbf{let}\ x{=}v\ \textbf{in}\ \epsilon], \mathcal{S}\rangle \hookrightarrow \langle E[\epsilon[v/x]], \mathcal{S}\rangle$ [*let*]

Figure B.5: Reductions for SMALLTALKLITE.

$$
\begin{array}{rcl}
\textbf{new}\ c\ [v/x] & = & \textbf{new}\ c \\
x\ [v/x] & = & v \\
x'\ [v/x] & = & x' \\
\textbf{self}\ [v/x] & = & \textbf{self} \\
\mathsf{nil}\ [v/x] & = & \mathsf{nil} \\
f\ [v/x] & = & f \\
f{=}e\ [v/x] & = & f{=}e[v/x] \\
e.m(e_i^*)\ [v/x] & = & e[v/x].m(e_i^*[v/x]) \\
\textbf{super}.m(e_i^*)\ [v/x] & = & \textbf{super}.m(e_i^*[v/x]) \\
\textbf{let}\ x{=}e\ \textbf{in}\ e'\ [v/x] & = & \textbf{let}\ x{=}e[v/x]\ \textbf{in}\ e' \\
\textbf{let}\ x'{=}e\ \textbf{in}\ e'\ [v/x] & = & \textbf{let}\ x'{=}e[v/x]\ \textbf{in}\ e'[v/x]
\end{array}
$$

Figure B.6: Variable substitution.

APPENDIX C

Numerical Results for Benchmarks

The following table shows the results of the three runtime performance experiments described in Section 7.1.3

Experiment	Benchmarks	Without #Pharo	Loaded	Used
1	Microdown	1880.8ms ± 0.4 ms	1880.3ms ± 0.3ms	1878.9ms ± 0.4ms
	Delta Blue	8286ms ± 5ms	8272ms ± 5ms	8197ms ± 6ms
	Richards	3519ms ± 4ms	3522ms ± 4ms	3522ms ± 4ms
	Compiler	4225ms ± 2ms	4211ms ± 2ms	4243ms ± 2ms
2	Microdown	788.2ms ± 0.7ms	797.1ms ± 0.6ms	797.2ms ± 0.6ms
	Delta Blue	4197ms ± 2ms	4219ms ± 5ms	4180ms ± 2ms
	Richards	2148ms ± 2ms	2134ms ± 2ms	2111ms ± 2ms
	Compiler	2891ms ± 2ms	2889ms ± 2ms	2921ms ± 2ms
4	Microdown	441.4ms ± 0.5ms	440.8ms ± 0.4ms	439.9ms ± 0.4ms
	Delta Blue	1275ms ± 3ms	1281ms ± 2ms	1283ms ± 3ms
	Richards	448.2ms ± 0.7ms	461.1ms ± 0.8ms	458.9ms ± 0.8ms
	Compiler	906.6ms ± 1.6ms	914.2ms ± 1.6ms	919.7ms ± 1.6ms

Table C.1: Average run times and standard interval with confidence of 95 % for the different benchmarks: without using the protected modifier library, with the protected modifier library loaded, and with the protected modifier library used.