Vrije Universiteit Brussel

Faculty of Science and Bio-Engineering Sciences
Department of Computer Science
Software Languages Lab

Supporting Concurrency Abstractions in High-level Language Virtual Machines

Dissertation Submitted for the Degree of Doctor of Philosophy in Sciences

Stefan Marr

Promotor: Prof. Dr. Theo D'Hondt
Copromotor: Dr. Michael Haupt

January 2013

Print: Silhouet, Maldegem

2013 Uitgeverij VUBPRESS Brussels University Press
VUBPRESS is an imprint of ASP nv (Academic and Scientific Publishers nv)
Ravensteingalerij 28
B-1000 Brussels
Tel. +32 (0)2 289 26 50
Fax +32 (0)2 289 26 59
E-mail: info@vubpress.be
www.vubpress.be

ISBN 978 90 5718 256 3
NUR 989
Legal deposit D/2013/11.161/010

DON’T PANIC

ABSTRACT

During the past decade, software developers widely adopted JVM and CLI as multi-language virtual machines (VMs). At the same time, the multicore revolution burdened developers with increasing complexity. Language implementers devised a wide range of concurrent and parallel programming concepts to address this complexity but struggle to build these concepts on top of common multi-language VMs. Missing support in these VMs leads to tradeoffs between implementation simplicity, correctly implemented language semantics, and performance guarantees.

Departing from the traditional distinction between concurrency and parallelism, this dissertation finds that parallel programming concepts benefit from performance-related VM support, while concurrent programming concepts benefit from VM support that guarantees correct semantics in the presence of reflection, mutable state, and interaction with other languages and libraries.

Focusing on these concurrent programming concepts, this dissertation finds that a VM needs to provide mechanisms for *managed state*, *managed execution*, *ownership*, and *controlled enforcement*. Based on these requirements, this dissertation proposes an *ownership-based metaobject protocol* (OMOP) to build novel multi-language VMs with proper concurrent programming support.

This dissertation demonstrates the OMOP's benefits by building concurrent programming concepts such as agents, software transactional memory, actors, active objects, and communicating sequential processes on top of the OMOP. The performance evaluation shows that OMOP-based implementations of concurrent programming concepts can reach performance on par with that of their conventionally implemented counterparts if the OMOP is supported by the VM.

To conclude, the OMOP proposed in this dissertation provides a unifying and minimal substrate to support concurrent programming on top of multi-language VMs. The OMOP enables language implementers to correctly implement language semantics, while simultaneously enabling VMs to provide efficient implementations.

SAMENVATTING

Over de laatste jaaren hebben softwareontikkelaars de JVM en CLI beginnen gebruiken als multi-language virtual machines (VM). Gelyktydig werd door de multicore revolutie de taak van de softwareontwikkelaar vermoeilijkt. Programmeertaalontwerpers ontwikkelden een grote variëteit aan concurrente en parallelle programmeerconcepten, maar het implementeren van deze concepten bovenop de multi-language VM's blijft een penibel probleem. Gebrekkige ondersteuning hiervoor in de VM's leidt tot afwegingen in de programmeertaalimplementaties tussen simpliciteit, correctheid en performantie.

Vertrekkende van de traditionele verdeling tussen concurrent en parallel programmeren vindt deze verhandeling dat parallelle programmeerconcepten voordeel halen uit performantie-gerelateerde VM ondersteuning, gelykaardig halen concurrente programmeerconcepten voordeel halen uit correctheidsgaranties van semantiek onder reflectie, mutable state en interactie met andere programmeertalen en libraries.

Door het toe te spitsen op deze concurrente programmeerconcepten vindt deze verhandeling dat een VM mechanismen moet aanbieden voor *managed state*, *managed execution*, *ownership* en *controlled enforcement*. Daarop gebaseerd stelt deze verhandeling een *ownership-based metaobject protocol* (OMOP) voor om vernieuwende multi-language VM's te bouwen met fatsoenlijke ondersteuning voor concurrente programmeerconcepten.

We demonstreeren de voordelen van de OMOP door er concurrente programmeerconcepten bovenop te bouwen, zoals agents, software transactional memory, actors, active object en communicating sequential processes. De performantieëvaluatie toont aan dat implementaties bovenop de OMOP van deze concurrente programmeerconcepten de performantie kan evenaren van conventionele implementaties, zolang de OMOP ondersteund is door de VM.

In conclusie, de OMOP biedt een verenigd substraat aan om concurrent programmeren te ondersteunen bovenop multi-language VM's. De OMOP laat programmeertaalontwikkelaars toe om op een correcte manier de semantiek van een taal te implementeren, maar het laat ook deze VM's toe om hiervoor een efficiënte implementatie te voorzien.

ACKNOWLEDGMENTS

First and foremost, I would like to heartily thank my promotors Theo D'Hondt and Michael Haupt for their advice and support. I am truly grateful to Theo for the opportunity and freedom to pursue my research interests at his lab. I am greatly indebted to Michael for sparking my interest in virtual machines with his lectures and for his continuous support throughout the years.

I sincerely thank the members of my Ph.D. committee for the time and effort they put into the evaluation of this work: David Ungar, Shigeru Chiba, Tom Van Cutsem, Wolfgang De Meuter, and Jacques Tiberghien.

I am earnestly thankful to David Ungar and IBM Research for their collaboration as part of the Renaissance project. His persistence in searching for the fundamental roots of a problem greatly influenced my work and thinking.

To all of my colleagues of the Software Languages Lab, I am equally grateful. Your feedback guided my investigations for this dissertation. Moreover, I am truly indebted for your support from the very beginning. Your help was essential for me to successfully write and defend my scholarship proposal. Special thanks goes to the Parallel Programming Group for all of our insightful debates. I would especially like to thank Charlotte Herzeel and Bruno De Fraine for their feedback on the very first drafts of this dissertation.

My personal thanks go to Stijn Timbermont and Elisa González Boix for "getting ze German started" at the lab and helping me to find my way in Brussels. I also want to thank Andoni, Andy, Carlos, Christophe, Coen, Eline, Engineer, Jorge, Lode, Nicolás, Wolf, et al. for their friendship and support.

Ganz besonders möchte ich mich auch bei meinen Eltern und Schwestern für ihre bedingungslose Liebe, stetige Unterstützung und beständigen Rückhalt bedanken.

I want to thank my friends in Brussels from all over the world. Brussels is a wonderful place to learn about cultural differences and I would not want to miss this experience, nor the beach volleyball during rainy summers. Thanks!

Last but not least, I would like to thank Kimberly for her patience and understanding and the time we have together.

This work is funded by a Ph.D. scholarship of IWT, for which I am grateful.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

1

INTRODUCTION

During recent years, multicore processors have become available in a majority of commodity hardware systems such as smartphones, laptops, and workstations. However, threads, locks, message passing, and other concurrent and parallel programming techniques continue to remain the tools of specialists for system programming or high-performance computing, because they are considered to be too complex and too hard to manage for application developers. However, these techniques become increasingly important for the development of mobile and desktop applications, since it becomes mandatory to exploit parallelism at the application level in order to achieve desired performance levels on modern hardware.

At the same time, managed languages, i. e., high-level programming languages on top of virtual machines (VMs), have become ubiquitous. The range of devices that utilize such general-purpose platforms grew steadily over the last decade, enabling the same application to run on wristwatches, phones, tablet devices, laptops, workstations, servers, and clusters. In addition, improvements in runtime technologies such as just-in-time compilation and automatic memory management widened the range of possible applications on top of these VMs. Eventually, this led to ecosystems emerging around multi-language VMs such as the Java Virtual Machine (JVM) and the Common Language Infrastructure (CLI). Supported by growing ecosystems, these VMs became the target platform of choice in many application domains.

However, research has not reconciled these two trends have so far. While, the complexity of concurrent and parallel programming inspired a wide range

of solutions for different application domains, none of the VMs provide sufficient support for these techniques, and thus, application developers cannot benefit from these solutions. Furthermore, current research [Catanzaro et al., 2010; Chafi et al., 2010] indicates that there is no *one-size-fits-all* solution to handle the complexity of concurrent and parallel programming, and therefore, application developers are best served with access to the whole field of solutions.

Domain-specific languages are one promising way to alleviate the complexity of concurrent and parallel programming. While the academic community continuously proposes abstractions that facilitate certain use cases, reduce the accidental complexity, and potentially provide improved performance, language implementers struggle to build these abstractions on top of today's VMs, because the rudimentary mechanisms such multi-language VMs provide are insufficient and require the language implementers to trade off implementation simplicity, correctly implemented language semantics, and performance.

The goal of this dissertation is to improve the support VMs provide for concurrent and parallel programming in order to enable language implementers to build a wide range of language abstractions on top of multi-language VMs.

1.1. Research Context

The research presented in this dissertation touches upon the domains of *concurrent and parallel programming* as well as on *virtual machine construction*. The main concerns of these domains are the following:

Concurrent and Parallel Programming Today, an overwhelmingly large body of literature describes a wide range of different concepts to manage concurrency, coordinate parallel activities, protect shared resources, describe data dependencies for efficient computation, etc. Already with the first computers, i. e., the Zuse Z3 [Rojas, 1997] and the ENIAC [Mitch and Atsushi, 1996], researchers have experimented with such concepts for concurrent and parallel programming, but never came close to a *one-size-fits-all* solution.

Virtual Machine Construction Software developers widely adopted high-level language VMs such as the JVM and the CLI as multi-language runtimes, because the research on just-in-time compilation [Aycock, 2003]

and garbage collection [Jones et al., 2011] led to highly efficient VM implementations that can support a wide variety of use cases. The diversity in supported use case as well as availability turned these VMs into relevant targets for language implementation. With the INVOKEDYNAMIC bytecode [Rose, 2009; Thalinger and Rose, 2010], the JVM improved its support for dynamic languages even further. Moreover, other language paradigms may benefit from improved support.

1.2. Problem Statement

While many domains, e. g., web applications or single-use scripts, are sensitive to programmer productivity, the field of concurrent and parallel programming frequently requires a tradeoff in favor of performance. In practice, the need for parallel execution only arises when certain performance properties like minimal latency, responsiveness, or high throughput are required and these requirements cannot be fulfilled with sequential implementations. Sequential performance improvements eventually level off because of physical and engineering limitations, known as the *power wall*, *memory wall*, and *instruction-level parallelism wall* [Asanovic et al., 2006]. Additionally, the rise of devices that are sensitive to energy efficiency further increases the need for parallelization. Compared to today's personal computers, the restricted energy budget of mobile and autonomous devices effectively reduces available sequential processing power. Thus, it becomes more common for application developers to consider performance, further increasing the necessity to utilize parallel and concurrent programming techniques.

When exploiting parallel hardware, the diversity of concurrent and parallel programming concepts and the absence of a one-size-fits-all solution suggest using problem-specific abstractions to enable application developers to address the inherent complexity. However, today's VMs support only a small and often low-level set of such concepts. While the rise of dynamic languages led, for example, to explicit support of customizable method lookup strategies in the JVM through the INVOKEDYNAMIC bytecode, VMs do not provide similar mechanisms to enable library and language implementers to build custom abstractions for concurrent and parallel programming. On the contrary, platforms such as the JVM and CLI, which feature shared memory semantics and thread-based concurrency models, make it hard to faithfully implement abstractions like the actor model [Karmani et al., 2009]. On such platforms, implementation simplicity or correct language semantics are often

traded in for better performance, which hinders the development of domain-specific abstractions.

An additional problem arises from the fact that the field of concurrent and parallel programming is largely uncharted, i. e., there is no widely accepted taxonomy that covers the wide range of different concepts. While a number of surveys provide an overview of programming concepts and language constructs [Briot et al., 1998; De Bosschere, 1997; Gupta et al., 2001; Skillicorn and Talia, 1998], they are far from complete and do not cover more recent work in the field. Thus, it is not even clear which concepts a VM should support in order to be a viable platform for a wide range of different problems from the field of concurrent and parallel applications. Supporting all possible concepts directly would not be possible, because of the resulting complex feature interactions within VMs. Since there is no *one-size-fits-all* solution either, one problem this dissertation needs to solve is to identify which of all these concepts a virtual machine should support in order to enable library and language implementers to provide domain-specific solutions for a relevant range of concurrent and parallel programming.

To conclude, VMs such as the JVM and the CLI lack sufficient support for parallel and concurrent programming. The goal of this dissertation is to identify a unifying substrate for concurrent and parallel programming that allows efficient implementation in a VM and provides the necessary abstractions to enable language and library implementers to implement custom abstractions. In summary, the two problems that need to be addressed are:

Insufficient Support for Concurrent and Parallel Programming in VMs Today's VMs do not provide sufficient support for concurrent and parallel programming in order to enable library and language implementers to build domain-specific abstractions.

Set of Required Concepts Unknown Since supporting all possible concepts is prohibitively complex, a VM needs to make abstractions of concrete programming concepts or support a subset of them. However, the subset of concurrent and parallel programming concepts that would enable domain-specific solutions is currently unknown.

1.3. Research Goals

The thesis of this dissertation is:

> *There exists a relevant and significant subset of concurrent and parallel programming concepts that can be realized on top of a unifying substrate. This substrate enables the flexible definition of language semantics that build on the identified set of concepts, and this substrate lends itself to an efficient implementation.*

This dissertation pursues the following research goals in support of this thesis:

Identify a Set of Requirements First, this dissertation has to examine how concurrency and parallelism are supported in VMs today, how the underlying concepts for concurrent and parallel programming relate to each other, and which problems occur when building higher-level abstractions on top of today's VMs. The resulting understanding of the state of the art and common problems enable the establishment of a set of requirements that guide the implementation of such concepts on top of a VM.

Define a Unifying Substrate Based on the set of requirements identified in the first step, this dissertation has to defines an abstraction that can serve as a unifying substrate for the implementation of a significant subset of concurrent and parallel programming concepts. This abstraction has to enable library and language implementers to customize semantics and guarantees for the programming concepts they want to provide, while preserving acceptable performance. Since complexity is an inherent issue for VM implementations, the abstraction has to demonstrate *unifying characteristics*. Thus, it has to generalize over a set of programming concepts to achieve abstraction, while avoiding adding independent, i. e., separate, support for each of the programming concepts to the VM.

Demonstrate Applicability This dissertation has to demonstrate the applicability of the proposed unifying substrate as an extension to high-level language VMs in order to show its benefits for building multi-language runtimes. The evaluation is based on the implementation of common abstractions for concurrent programming on top of the proposed substrate. The goal is to show the substrate's potential compared to classic

ad hoc implementations. Therefore, this dissertation need to show that the substrate fulfills the posed requirements, that it enables enforcement of the desired language semantics, and that it gives rise to an efficient implementation.

Note that the investigation of security aspects, reliability, distribution, and fault-tolerance is outside of the scope of this dissertation. This dissertation primarily focuses on improving support for concurrent and parallel programming for multi-language runtimes in the form of high-level language virtual machines.

1.4. Dissertation Outline

This dissertation is structured as follows:

Chapter 2: Context and Motivation
This chapter outlines the rationale for the assumptions stated above. It argues that VMs are target platforms for a wide range of applications, and thus, require better support for concurrent and parallel programming abstractions as a responds to the multicore revolution. Furthermore, it defines concurrent and parallel programming, and introduces common concepts for it as background for this dissertation. The chapter concludes with a vision for constructing applications in the presence of appropriate abstractions for concurrent and parallel programming to motivate the goal of this dissertation.

Chapter 3: Which Concepts for Concurrent and Parallel Programming does a VM need to Support?
This chapter establishes the requirements for VM support for a wide range of different abstractions for concurrent and parallel programming. First, it surveys contemporary VMs, assessing the state of the art by identifying which concepts the VMs support and how they realize these concepts. Second, the chapter surveys the field of concurrent and parallel programming in order to identify its basic concepts, concluding that parallel programming concepts benefit from VM support for a wide range of different optimizations, while concurrent programming concepts benefit from extended support for their semantics. This leads to the observation that both sets of programming concepts have distinct requirements. This dissertation focuses on VM support for concurrent

programming concepts. The third part of the chapter identifies common problems in implementing concurrent programming concepts on top of today's VMs. Based on survey results and identified problems, this chapter concludes with requirements for comprehensive VM support for concurrent programming.

Chapter 4: Experimentation Platform

This chapter discusses the motivation and choices for the platforms used for experiments and evaluation. First, it introduces SOM (Simple Object Machine), a minimal Smalltalk dialect, which is used throughout this dissertation for code examples and the discussion of the OMOP's semantics. This section includes an introduction to general Smalltalk syntax and its semantics. Second, it motivates the choice of Smalltalk as platform for this research. Finally, it discusses RoarVM as a choice for the VM implementation experiments and detailed its implementation.

Chapter 5: An Ownership-based MOP to Express Concurrency Abstractions

This chapter introduce this dissertation's main contribution, an ownership-based metaobject protocol (OMOP). The OMOP is a unifying substrate for the implementation of concurrent programming concepts. First, the chapter discusses the foundational notions of open implementations and metaobject protocols. Second, it presents the OMOP itself. Third, it discusses examples of how to apply the OMOP to enforce immutability and how to implement Clojure agents with it. Finally, the chapter defines the OMOP's semantics based on SOM's bytecode interpreter and discusses the OMOP in the context of related work.

Chapter 6: Evaluation – The OMOP as Unifying Substrate

In order to evaluate the OMOP, this chapter discusses how it fulfills the identified requirements. First, the chapter discusses the evaluation criteria. Second, it examines the case studies implementing Clojure agents, software transactional memory (STM), and AmbientTalk actors. Third, the chapter discusses concepts identified in Chapter 3 and argues that the OMOP supports all of the concepts that require VM support for guaranteeing correct semantics. Fourth, the chapter shows that the using OMOP does not have a negative impact on the implementation size of agents, actors, and STM, by comparing their OMOP-based implementations against their ad hoc implementations. Finally, the limitations of the OMOP are discussed.

Chapter 7: Implementation Approaches
This chapter details OMOP implementation strategies. First, it discusses the OMOP implementation based on program transformation with abstract syntax trees. Secondly, it describes the implementation in the RoarVM bytecode interpreter and the chosen optimizations.

Chapter 8: Evaluation – Performance
This chapter evaluates the performance of the OMOP implementations. Furthermore, it compares the performance of an STM and an actor implementation based on the OMOP with the performance of their corresponding ad hoc, i. e., conventional, implementations. To that end, the chapter first details the methodology used for the performance evaluation. Second, it assesses the performance of the VMs used for the experiments. Third, it compares the performance of the ad hoc with the OMOP-based implementations. Fourth, it evaluates the performance of different aspects of the OMOP implementation, such as inherent overhead and the impact of the optimizations. Finally, it compares the absolute performance of the two OMOP implementations.

Chapter 9: Conclusion and Future Work
The last chapter revisits the dissertation's problem and thesis statement to argue that the OMOP is an appropriate unifying substrate for implementing a wide range of concepts for concurrent and parallel programming on top of a VM. It summarizes the OMOP as well as this dissertation's research contributions. Finally, it discusses the OMOP's current limitations and outlines future work. For example, it speculates how the OMOP could be supported on VMs with statically typed languages, and how just-in-time compilation could improve performance.

1.5. Supporting Publications and Technical Contributions

A number of publications, exploratory activities, and technical contributions directly support this dissertation. This section discusses them briefly to highlight their relevance to this work.

Main Idea The main idea, i. e., the design of an ownership-base MOP and initial experiments were presented at TOOLS'12 [Marr and D'Hondt, 2012].

Chapter 5 and part of the evaluation in Chapter 6 and Chapter 8 are based on the material presented in:

- Stefan Marr and Theo D'Hondt. Identifying a unifying mechanism for the implementation of concurrency abstractions on multi-language virtual machines. In *Objects, Models, Components, Patterns, 50th International Conference, TOOLS 2012*, volume 7304 of *Lecture Notes in Computer Science*, pages 171–186, Berlin / Heidelberg, May 2012. Springer. ISBN 978-3-642-30560-3. doi: 10.1007/978-3-642-30561-0_13.

This publication builds on gradual development based on an initial idea of abstracting from concrete concurrency models, presented at the PLACES'09 workshop [Marr et al., 2010a]. The evolving ideas were also presented at several other occasions: as posters [Marr and D'Hondt, 2010], which once resulted in a *Best Poster Award* [Marr and D'Hondt, 2009], or as a contribution to the SPLASH'10 Doctoral Symposium [Marr, 2010].

Surveys Chapter 3 relies on three surveys conducted during the preparation of this dissertation. The main part of Sec. 3.2 was also part of Marr and D'Hondt [2012]. The other two surveys have been presented at the VMIL'09 and VMIL'11 workshops:

- Stefan Marr, Michael Haupt, and Theo D'Hondt. Intermediate language design of high-level language virtual machines: Towards comprehensive concurrency support. In *Proc. VMIL'09 Workshop*, pages 3:1–3:2. ACM, October 2009. ISBN 978-1-60558-874-2. doi: 10.1145/1711506.1711509. (extended abstract)

- Stefan Marr, Mattias De Wael, Michael Haupt, and Theo D'Hondt. Which problems does a multi-language virtual machine need to solve in the multicore/manycore era? In *Proceedings of the 5th Workshop on Virtual Machines and Intermediate Languages*, VMIL '11, pages 341–348. ACM, October 2011a. ISBN 978-1-4503-1183-0. doi: 10.1145/2095050.2095104.

Exploring Programming Models While the surveys provided a good overview of the field, the practical insights gathered during exploring and experimenting with the different technologies provided valuable additional experience that enabled a proper classification of the obtained knowledge. Experiments were conducted with barrier-like synchronization [Marr et al., 2010b], resulting in a *Best Student Paper Award*, different notions of Actor languages

and systems [De Koster et al., 2012; Renaux et al., 2012; Schippers et al., 2009] were explored, experience was gathered by teaching Erlang and Clojure [Van Cutsem et al., 2010], as well as with the general exploration of concurrent language implementations [Marr et al., 2012].

- Stefan Marr, Stijn Verhaegen, Bruno De Fraine, Theo D'Hondt, and Wolfgang De Meuter. Insertion tree phasers: Efficient and scalable barrier synchronization for fine-grained parallelism. In *Proceedings of the 12th IEEE International Conference on High Performance Computing and Communications*, pages 130–137. IEEE Computer Society, September 2010b. ISBN 978-0-7695-4214-0. doi: 10.1109/HPCC.2010.30. Best Student Paper Award.

- Joeri De Koster, Stefan Marr, and Theo D'Hondt. Synchronization views for event-loop actors. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 317–318, New York, NY, USA, February 2012. ACM. doi: 10.1145/2145816.2145873. (Poster)

- Hans Schippers, Tom Van Cutsem, Stefan Marr, Michael Haupt, and Robert Hirschfeld. Towards an actor-based concurrent machine model. In *Proceedings of the Fourth Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, pages 4–9, New York, NY, USA, July 2009. ACM. ISBN 978-1-60558-541-3. doi: 10.1145/1565824.1565825

- Thierry Renaux, Lode Hoste, Stefan Marr, and Wolfgang De Meuter. Parallel gesture recognition with soft real-time guarantees. In *Proceedings of the 2nd edition on Programming Systems, Languages and Applications based on Actors, Agents, and Decentralized Control Abstractions*, SPLASH '12 Workshops, pages 35–46, October 2012. ISBN 978-1-4503-1630-9. doi: 10.1145/2414639.2414646

- Tom Van Cutsem, Stefan Marr, and Wolfgang De Meuter. A language-oriented approach to teaching concurrency. Presentation at the workshop on curricula for concurrency and parallelism, SPLASH'10, Reno, Nevada, USA, 2010. URL http://soft.vub.ac.be/Publications/2010/vub-tr-soft-10-12.pdf.

- Stefan Marr, Jens Nicolay, Tom Van Cutsem, and Theo D'Hondt. Modularity and conventions for maintainable concurrent language implementations: A review of our experiences and practices. In *Proceedings of*

the 2nd Workshop on Modularity In Systems Software (MISS'2012), MISS'12. ACM, March 2012. doi: 10.1145/2162024.2162031.

Technical Contributions The work of this dissertation has only been made possible by starting from existing research artifacts. These research artifacts enabled this dissertation's experiments and provide the foundation for the implemented prototypes. The main artifact used for the experiments is the RoarVM, designed and implemented by Ungar and Adams [2009]. It was later documented in a so-far unpublished report [Marr et al., 2011b]. The following list briefly discusses these key research artifacts and their relation to this dissertation. The source code and an overview of all artifacts is available online.[1]

RoarVM The RoarVM is a Smalltalk interpreter compatible with Squeak and Pharo Smalltalk. It was designed to experiment with manycore architectures such as the Tilera TILE64 and runs on up to 59cores on these machines. This dissertation relies on the RoarVM as an experimentation platform to study the OMOP's performance in a VM with a bytecode interpreter.

URL: `https://github.com/smarr/RoarVM`

AST-OMOP This dissertation's first implementation of the OMOP is based on AST transformation of Smalltalk code and can be used with standard Smalltalk VMs. Thus, it enables experimentation with the OMOP's basic mechanisms without requiring VM changes.

URL: `http://ss3.gemstone.com/ss/Omni.html`

RoarVM+OMOP Using the RoarVM as a foundation, the RoarVM+OMOP adds support for the complete ownership-based metaobject protocol in the interpreter. The current implementation changes the bytecode to support the OMOP's semantics (cf. Chapter 7).

URL: `https://github.com/smarr/OmniVM`

SOM+OMOP Building on SOM (Simple Object Machine), which was used for previous research projects [Haupt et al., 2010, 2011a,b], SOM+OMOP is a simple Smalltalk interpreter implementing and documenting the OMOP's semantics as part of this dissertation.

URL: `http://ss3.gemstone.com/ss/Omni.html`

[1] `http://www.stefan-marr.de/research/omop/`

ReBench This dissertation's performance evaluation requires rigorous preparation and a proper experimental setup to yield reliable results. ReBench is a benchmarking tool that documents the experiments and facilitates their reproduction. It uses configuration files for all benchmarks, documenting the benchmark and VM parameters used for the experiments, and thus providing the necessary traceability of results and a convenient benchmark execution.

URL: https://github.com/smarr/ReBench

SMark The benchmarks used for the performance evaluation have been implemented based on SMark, which is a framework inspired by the idea of unit-testing that allows the definition of benchmarks in the style of SUnit.

URL: http://www.squeaksource.com/SMark.html

2

CONTEXT AND MOTIVATION

This chapter introduces the context for this dissertation. It motivates the need for multi-language virtual machines (VMs) and argues that their rudimentary support for parallel and concurrent programming needs to be extended to maintain the versatility of these VMs in the multicore era. Furthermore, it defines *concurrent programming* and *parallel programming* deviating from existing literature to overcome the shortcomings of the existing definitions. The newly proposed definitions enable a classification of programming concepts based on intent and purpose to facilitate the discussion in the later chapters. Based on that, this chapter gives a brief overview of concurrent and parallel programming concepts as a foundation for the remainder of this dissertation. Finally, the chapter concludes by combining the different elements to envision how applications are built when developers are able to utilize appropriate abstractions to tackle the challenges of concurrency and parallelism.

2.1. Multi-Language Virtual Machines: Foundation for Software Ecosystems

High-level language VMs are used as general purpose platforms with large software ecosystems. The role of high-level language VMs has been shifting over the past decades. Starting out as interpreted VMs for languages that offer high productivity, they became VMs that use highly efficient just-in-time compilation technology [Aycock, 2003] and well tuned garbage collectors [Craig, 2006; Smith and Nair, 2005]. The resulting performance improvements opened the door for a wide range of application domains. Consequently, VMs are now the platform for many applications that earlier on would have been implemented in native languages such as C or C++ and targeted a particular operating system. With the increased adoption of VMs came additional support from a wide range of parties. Software and tool vendors, as well as the various open source communities started to build large software ecosystems [Gregor, 2009] around VMs. Reasons for this adoption are availability of libraries, code reuse, portability, and the desire to integrate different systems on the same platform. Studies such as the one of Ruiz et al. [2012] show that code reuse in such diverse ecosystems is not merely a theoretical opportunity, but realized in practice. In addition to code reuse, tooling is an important motivation. For instance, the availability of IDEs, performance analysis tools, testing frameworks, and continuous integration techniques are important factors. Eclipse[1] and Netbeans[2] as IDEs, and tools like VisualVM[3] enable developers to use multiple languages, while relying on the same common tools.

Language implementers target VMs to provide appropriate abstractions for specific problem domains. While the adoption of VMs like JVM and CLI grew over the years, the desire to use different kinds of languages interoperating with the existing ecosystem, grew as well. Both, JVM and CLI, were originally tailored towards either one specific language, i. e., Java for the JVM, or a set of closely related languages, i. e., VB.NET and C# for the CLI. The resulting VMs were however efficient enough to attract a vast number of lan-

[1] http://www.eclipse.org/
[2] http://www.netbeans.org/
[3] http://visualvm.java.net/

guage designers and implementers, who built hundreds of languages on top of them.[4,5]

Often the motivation for such language implementations is to fill a particular niche in which the language-specific properties promise higher productivity, even when the performance might be sacrificed. Another motivation might be adoption. While a new language with its own runtime typically lacks tooling and libraries for productive use, targeting an existing platform such as the JVM and CLI can ease the integration with existing systems, and thus, facilitate adoption. To give a single example, Clojure[6] integrates well with the ecosystem of the JVM, which enabled adoption and brought concurrent programming concepts such as agents, atoms, and software transactional memory to a wider audience.

Support for dynamic languages was extended to strengthen VMs as general purpose platforms. Over the years, the JVM and CLI grew into fully adopted general purpose platforms and with their success grew the adoption of new JVM and CLI languages. This motivated efforts to reduce the performance cost of dynamic languages. The various method dispatch semantics of dynamic languages were one of the largest performance concerns. To improve the situation, the JVM specification was extended by infrastructure around the new INVOKEDYNAMIC bytecode [Rose, 2009; Thalinger and Rose, 2010], which gives language designers a framework to specify method dispatch semantics and enables the just-in-time compiler to optimize the dispatch for performance. With the *Dynamic Language Runtime* for the CLI, Microsoft went a different route and provides a common dynamic type system and infrastructure for runtime code generation. However, both approaches have in common that they extend the reach of the underlying platform to new languages, and thus application domains.

Multi-language VMs are targets for library and language implementers. With the additional infrastructure for dynamic languages in place, the JVM and the CLI became *multi-language VMs*. While the notion has been used

[4] *A list of languages targeting the .NET Framework*, Brian Ritchie, access date: 28 September 2012 http://www.dotnetpowered.com/languages.aspx

[5] *Programming languages for the Java Virtual Machine JVM and Javascript*, Robert Tolksdorf, access date: 28 September 2012 http://www.is-research.de/info/vmlanguages/

[6] http://clojure.org

in literature [Folliot et al., 1998; Harris, 1999] and industry,[7] the exact meaning remains undefined. This dissertation assumes a multi-language VM to be a high-level language runtime that supports language and library implementers actively in their efforts to implement a wide range of programming abstractions for it.

With the advent of multicore hardware, mechanisms for concurrency and parallelism became important concerns for general-purpose platforms. However, the JVM and CLI provide only minimal support. Most importantly, they lack features that enable language designers to build efficient abstractions for concurrent and parallel programming. An equivalent to invokedynamic *for concurrency* becomes highly desirable to enable these VMs to remain general purpose platforms and *multi-language VMs*, facilitating applications that need to utilize concurrency and parallelism for various purposes.

Complex feature-interaction in VMs requires a minimal set of unifying abstractions. The complexity of VMs is not only a challenge for their implementation [Haupt et al., 2009], but also presents difficulties for their evolution and extension. The dependencies between the wide range of supported mechanisms lead to situations where it is unclear whether a desired feature can be supported without breaking the existing features.

One prominent example in the context of the JVM is support for tail call elimination, which is particularly desirable for functional languages [Schinz and Odersky, 2001]. First, it was assumed that it cannot be supported because of Java's use of stack inspection for its security features [Fournet and Gordon, 2003]. Later, theoretical [Clements and Felleisen, 2004] and practical [Schwaighofer, 2009] solutions were found. However, the solutions have tradeoffs and tail call elimination is still not included in the JVM specification because of the complex interplay of VM features.

In conclusion, multi-language VMs need to offer a limited number of abstractions that have unifying characteristics. That is to say, the offered abstractions need to enable a wide range of language features. From our perspective, it would be infeasible to add every possible language feature directly to a VM. The complexity of their interactions would be unmanageable. Thus, the abstractions offered by a VM need to provide a unifying substrate that facilitates the implementation of concrete language features on top of the VM.

[7] *The Da Vinci Machine Project: a multi-language renaissance for the Java Virtual Machine architecture*, Oracle Corp., access date: 28 September 2012
http://openjdk.java.net/projects/mlvm/

2.2. The Multicore Revolution

Performance improvements for sequential processors are tailing off. Before 2005, processor designers could use more transistors and higher clock frequencies to continuously increase the performance of new processor generations. Moore's Law, which states that with improved manufacturing techniques and the resulting miniaturization the number of transistors doubles approximately every two years without extra cost, enabled them to increase the amount of logic that processors could contain. They used the additional logic to improve techniques such as out-of-order execution, branch prediction, memory caching schemes, and cache hierarchies. Unfortunately, these techniques are optimizations for specific usage patterns and eventually, their returns diminish [Hennessy and Patterson, 2007; Michaud et al., 2001].[8] With the shrinking transistor sizes, it was possible to steadily increase the clock frequency as well.

Higher clock speeds become impractical. Around 2005, processor designers reached a point were it was no longer feasible to keep increasing the clock speed with the same pace as in previous decades. An increase in clock speed corresponds directly to higher energy consumption and more heat dissipation [Hennessy and Patterson, 2007, p. 18]. However, handling the lost heat beyond a certain limit requires cooling techniques that are impractical for commodity devices. For mobile devices, the necessary increase in power consumption and the resulting heat make clock speed increases beyond a certain limit prohibitive. Processor designers worked around the issue by applying various engineering techniques to handle the heat. This led to processors that can over-clock themselves when the thermal budget permits it. However, none of these techniques could deliver the performance improvements software developers have gotten accustomed to over the decades.

Increasing transistor budgets led to multicore processors. Still, the transistor budget for processors keeps increasing. In an attempt to satisfy the software industry's need for constant performance improvement, processor designers started to explore the design space of parallel processors. The increasing transistor budget can be spent on a wide range of different features of a modern processor [Hill and Marty, 2008]. Duplicating functional units,

[8] *Welcome to the Jungle*, Herb Sutter, access date: 27 June 2012
http://herbsutter.com/welcome-to-the-jungle/

e. g., arithmetic units enables higher degrees of instruction-level parallelism. In combination with duplicated register files, techniques such as simultaneous multithreading can hide memory latency and utilize the available functional units more efficiently. Duplicating whole cores enables more heterogeneous parallel workloads. However, workloads such as data-intensive graphic operations are better served by large data-parallel units. These can come in the form of vector processing units embedded into a traditional processor, or modern GPU computing processors that consist of vast numbers of very simple but highly parallel processing elements. Depending on target applications, processor vendors mix and match these design options to offer more computational power to their customers [Hennessy and Patterson, 2007]. However, exploiting available hardware parallelism remains a task for software developers.

To conclude, multicore processors will play a major role for the foreseeable future and software developers will need to use the offered parallelism to fulfill the performance requirements for their software.

2.3. Concurrent vs. Parallel Programming: Definitions

The goal of this section is to clarify the notions of *concurrency* and *parallelism* in order to accurately categorize the programming concepts of this field later in this dissertation. To this end, this dissertation introduces the additional notions of *concurrent programming* and *parallel programming*, which provide stronger distinguishing properties and facilitate the discussions in Chapter 3.

2.3.1. Concurrency and Parallelism

Distinction often avoided Defining the two terms *concurrency* and *parallelism* is often avoided in literature. Subramaniam [2011, p. xiv] even claims *"there's no clear distinction between these two terms in industry[...]"*. Others such as Axford [1990, p. 4,6] and Lea [1999, p. 19] mention the terms, but do not make a clear distinction between them. However, they indicate the difference between systems with physical parallelism and systems with merely time shared execution.

Distinction by execution model from historical perspective Lin and Snyder [2008, p. 21] explain the situation within its historic context. The term *concurrency* was used in the operating system and database communities, which

were mostly concerned with small-scale systems that used time shared execution. Following their argumentation, the term *parallelism* on the other hand comes from the supercomputing community, which was mostly concerned with large-scale systems and physically parallel execution of programs. However, for the purpose of their book, they conclude the discussion by saying: *"we will use the terms [concurrent and parallel] interchangeably to refer to logical concurrency"*.

Definitions of Sottile et al. Overall, the field distinguishes the two terms based on the underlying execution model. A set of concrete definitions is given by Sottile et al. [2010, p. 23-24]. They write: *"We define a* concurrent program *as one in which multiple streams of instructions are active at the same time."* Later, they go on and write: *"Our definition of a* parallel program *is an instance of a concurrent program that executes in the presence of multiple hardware units that will guarantee that two or more instruction streams will make progress in a single unit of time."* While these definitions are seldom spelled out precisely in literature, the definitions of Sottile et al. seem to reflect the common consensus of what *concurrency* and *parallelism* mean. Therefore, parallel programs are considered to be a subset of concurrent programs. For instance, Sun Microsystems, Inc. [2008] uses very similar definitions.

Not all parallel programs are concurrent programs. The major issue of characterizing parallel programs as a subset of concurrent programs is the implication that all parallel programs are concurrent programs as well. Thus, all parallel programs can be mapped on a sequential execution. However, there are useful algorithms such as the elimination stack proposed by Shavit and Touitou [1995b] that are not strictly linearizable [Herlihy and Wing, 1990]. Which means, parallel programs can have executions that cannot be replicated by any concurrent program, i. e., they cannot be mapped on a sequential execution without losing semantics.

Current definitions do not add value to explain programming concepts. In addition to the conceptual problem of characterizing parallel programs as a subset of concurrent programs, the notion of a subset relation does not add explanatory value in practice, either.

For example, Cilk's *fork/join* with a work-stealing scheduler implementation [Blumofe et al., 1995] is designed for parallel programming. The idea of fork/join is to recursively divide a task to enable parallel execution of sub-

tasks with automatic and efficient load-balancing. When the execution model provides only time shared execution, the overhead of fork/join is in most implementations prohibitive [Kumar et al., 2012], especially when compared to a sequential recursive implementation. Furthermore, the problem of load-balancing that is solved by work-stealing does not exist in the first place. To conclude, parallel programming needs to solve problems that do not exist for concurrent programs, and parallel programs require solutions that are not necessarily applicable to concurrent programs.

Conversely, low-level atomic operations such as *compare-and-swap* have been conceived in the context of few-core systems. One important use case was to ensure that an operation is atomic with respect to interrupts on the same core. One artifact of the design for time shared systems is that Intel's compare-and-swap operation in the IA-32 instruction set (`CMPXCHG`) requires an additional `LOCK` prefix to be atomic in multicore environments [Intel Corporation, 2012]. Similarly to the argumentation that parallel programming concepts do not necessarily apply to time shared systems, the usefulness of low-level atomic operations originating in concurrent systems is limited. Using them naively restricts their scalability and their usefulness diminishes with rising degree of parallelism [Shavit, 2011; Ungar, 2011]. They are designed to solve a particular set of problems in concurrent programs but are not necessarily applicable to the problems in parallel programs.

Concluding from these examples, it would be beneficial to treat *concurrent programming* and *parallel programming* separately to properly reflect the characteristics and applicability of the corresponding programming concepts.

2.3.2. Concurrent Programming and Parallel Programming

This section defines the notions of *concurrent programming* and *parallel programming* to create two disjoint sets of programming concepts. Instead of focusing on the execution model as earlier definitions do, the proposed definitions concentrate on the aspect of programming, i. e., the process of formalizing an algorithm using a number of programming concepts with a specific intent and goal. The distinction between the execution model and the act of programming is made explicitly to avoid confusion with the common usage of the terms concurrency and parallelism.

One inherent assumption for these definitions is that they relate the notions of concurrent and parallel programming with each other on a fixed level of abstraction. Without assuming a fixed level of abstraction, it becomes easily confusing because higher-level programming abstractions are typically built

on top of lower-level abstractions, which can fall into a different category. For instance, as discussed after these definitions, parallel programming abstractions are often implemented in terms of concurrent programming abstractions. Thus, these definitions have to be interpreted on a fixed and common abstraction level.

Definition 1. Parallel programming *is the art of devising a strategy to coordinate collaborating activities to contribute to the computation of an overall result by employing multiple computational resources.*

Definition 2. Concurrent programming *is the art of devising a strategy to coordinate independent activities at runtime to access shared resources while preserving the resources' invariants.*

This means that parallel programming is distinct from concurrent programming because it provides techniques to employ multiple computational resources, while concurrent programming provides techniques to preserve semantics, i. e., the correctness of computations done by independent interacting activities that use shared resources.

Furthermore, an important aspect of parallel programming is the decomposition of a problem into cooperating activities that can execute in parallel to produce an overall result. Therefore, the related concepts include mechanisms to coordinate activities and communicate between them. This coordination can be done by statically planing out interactions for instance to reduce communication, however, it usually also needs to involve a strategy for the communication at runtime, i. e., the dynamic coordination.

In contrast, concurrent programming concepts include techniques to protect resources, for instance by requiring the use of locks and monitors, or by enforcing properties such as isolation at runtime, preventing undesirable access to shared resources. The notion of protecting invariants, i. e., resources is important because the interacting activities are independent. They only interact based on conventions such as locking protocols or via constraint interfaces such as messaging protocols to preserve the invariants of the share resources.

The nature of *activities* remains explicitly undefined. An activity can therefore be represented for instance by a light-weight task, a thread, or an operating system process, but it could as well be represented by the abstract notion of an actor.

Note that these definitions do not preclude the combination of concurrent and parallel programming. Neither do they preclude the fact that programming concepts can build on each other, as discussed in the beginning of this

section. For instance, the implementation of barriers that are used in parallel programming rely on concurrent programming concepts [Marr et al., 2010b]. Thus, in many cases developers need to combine parallel and concurrent programming techniques to account for their requirements.

It remains to be mentioned that similar definitions have been proposed before. For instance, in the teaching material of Grossman [2012].[9] The definitions given by this dissertation use a different wording to avoid the implication of performance and the concrete notions of threads, since concurrent and parallel programming are more general.

Rationale These two definitions are partially based on the following observations: Classic parallel programming approaches such as *single program multiple data* (SPMD) techniques based on MPI [Message Passing Interface Forum, 2009] or shared memory approaches such as OpenMP [OpenMP Architecture Review Board, 2011] are used to decompose problems to use multiple computational resources such as processor cores. All activities in such programs collaborate to calculate the overall result. They are coordinated by the use of barriers and collective operations to make an abstraction of concrete data dependencies requiring all activities to actively participate, which is trivial in a *single program* model. However, even contemporary APGAS languages such as X10 (cf. Sec. 2.4.4) provide barriers [Shirako et al., 2008], while advocating for fork/join-like programming models that encourage a much higher degree of dynamics than in the SPMD model.

In addition to the use of barriers, the fork/join-based programming model in the style of Cilk [Blumofe et al., 1995] is strongly based on the notion of collaborating activities. It uses the notion of recursive divide-and-conquer to expose the parallelism in a problem. One important assumption that is inherent to this model is that the recursive devision makes an abstraction of all data dependencies. Thus, it is assumed that the resulting program is data-race-free when it synchronizes correctly on completion of its sub-tasks. The fork/join model itself does not provide any means to coordinate access to shared resources other than by synchronizing on the completion of sub-tasks. Thus, all activities have to collaborate and it is assumed that no independent activities in the system interact in any way with the fork/join computation. Therefore, correctness of the computation is ensured by construction and does not need to be enforced at runtime.

[9] The key notions are also mentioned in Grossman and Anderson [2012].

The observations for concurrent programming concepts are different. Conventional approaches based on mutexes require sequential execution of activities that work on a shared resource in order to enforce correctness. Programs using these concepts typically consist of multiple activities that have different purposes but require a common resource. In such systems, interactions are resource-centric, without a common purpose, and require that the invariants for the shared resources hold. Reader/writer locks for instance exclude only conflicting operations from using the same resource. Software transactional memory (cf. Sec. 2.4.3) goes further by removing the need for managing resources manually. In contrast, event-loop concurrency models (cf. Sec. 2.4.4) promote resources to active entities that are responsible for managing their consistency and allow clients to interact via an asynchronous interface only. Thus, the main commonality of these concepts is the protection of invariants of shared resources to guarantee correctness, while permitting interaction of independent activities at runtime.

The following section discusses these approaches in more detail.

2.3.3. Conclusion

To conclude, the definitions of *concurrency* and *parallelism* as found in the literature can be inappropriate when it comes to categorizing concepts.

In contrast, the alternative definitions for *concurrent programming* and *parallel programming* given here categorize concepts in two disjoint sets. Instead of focusing on the execution model, these definitions focus on the aspect of programming and relate to the intent and goal of a programming concept. Therefore, concurrent programming concepts coordinate modifications of shared resources, while parallel programming concepts coordinate parallel activities to compute a common result.

2.4. Common Approaches to Concurrent and Parallel Programming

This section gives an overview of common concepts in the field of concurrent and parallel programming to provide a foundation for our later discussions. Note that this section introduces Clojure agents, which are used in later chapters as running examples and therefore discussed here in more detail.

Table 2.1.: Flynn's taxonomy

		Data Stream	
		Single	*Multiple*
Instruction Stream	*Single*	SISD	SIMD
	Multiple	MISD	MIMD

2.4.1. Taxonomies

To structure the discussion of the vast field of concurrent and parallel programming, this section reviews two taxonomies proposed in the literature. These two taxonomies unfortunately do not match the focus of this dissertation. Therefore, a partial taxonomy is introduced to structure the discussion here and in later chapters of this dissertation.

Flynn's Taxonomy

Flynn's taxonomy can be used as a coarse categorization of programming concepts [Flynn, 1966]. Originally, the proposed taxonomy is meant to categorize computer organizations, but it is abstract enough to be used in a more general context. Flynn based it on the notions of *instruction streams* and *data streams*, where a set of instructions forms a program to consume data in a given order. Using these notions, Flynn identifies four possible categories (cf. Tab. 2.1): *single instruction stream - single data stream* (SISD), *single instruction stream - multiple data streams* (SIMD), *multiple instruction streams - single data stream* (MISD), and *multiple instruction streams - multiples data streams* (MIMD). For the purpose of this dissertation, SISD represents classic single threaded programs. SIMD corresponds to single threaded programs that use vector instructions to operate on multiple data items at the same time. MISD can be understood as a multi-threaded program that runs on a single core with time shared execution without exhibiting real parallelism. Thus, each thread of such a program corresponds to a distinct instruction stream (MI), but the observable sequence of memory accesses is a single data stream coming from a single data store. MIMD corresponds then to applications with multiple threads executing in parallel, each having its distinct stream of memory access, i. e., separate data streams and data stores. Since the concurrent and parallel programming concepts relevant for this dissertation are variations of multiple instruction streams (MI), Flynn's taxonomy is too coarse-grained to structure the discussion.

Almasi and Gottlieb's Taxonomy

Almasi and Gottlieb [1994] use a different taxonomy to classify *"most parallel architectures"*. Their taxonomy (cf. Tab. 2.2) is based on two dimensions *data* and *control*, much like Flynn's. However, the interpretation of these dimensions is very different. The *data mechanism* divides the parallel architectures into being based on shared or private memory. The *control mechanism* on the other hand, classifies based on the way control flow is expressed. Their classification starts with *control driven* as the category with the most explicit control, and ends with *data driven* for mechanism that depend the least on explicit control flow.

Table 2.2.: Classification of computational models, with examples. [Almasi and Gottlieb, 1994, p. 25]

	Data Mechanism	
Control Mechanism	*Shared Memory*	*Private Memory* (message passing)
Control driven	von Neumann	Communicating processes
Pattern driven	Logic	Actors
Demand driven	Graph reduction	String reduction
Data driven	Dataflow with I-structure	Dataflow

While this taxonomy covers a wide range of concurrent and parallel programming concepts, it does so on a very abstract level. Especially its emphasis on the control mechanism is of lesser relevance for this dissertation. Since this research targets contemporary multi-language VMs, all computational models have to be mapped to a control-driven representation.

A Partial Taxonomy of Contemporary Approaches

Neither Flynn's nor Almasi and Gottlieb's taxonomy reflect common concurrent and parallel programming concepts in a way that facilitates their discussion in the context of this dissertation. Therefore, this dissertation uses a partial taxonomy on its own. The following section discusses this taxonomy and its four categories *Threads and Locks*, *Communicating Threads*, *Communicating Isolates*, and *Data Parallelism*. The categorization focuses on how concurrent and parallel activities interact and coordinate each other, since these aspects are the main mechanisms exposed to the programmer, and one point which makes the various concepts distinguishable, even with the necessarily brief discussions of the concepts in this dissertation.

The remainder of this section gives a brief overview over the four categories and then discuss them and the corresponding concepts in more detail.

Threads and Locks
: The main abstraction for computation in this case is *threads of execution* that use mechanisms like locks, semaphores, and condition variables as their means to coordinate program execution in a shared memory environment.

Communicating Threads
: The main abstraction for computation in this case is *threads of execution* in a shared memory environment. The concepts in this category use higher-level means than basic locks and condition variables for coordination and communication. Message sending and channel-based abstractions are examples for such higher-level abstractions. Threads that coordinate with barriers, clocks, or phasers, fall into this category as well. Programming models can be based on *active objects* or for instance variants of *communicating sequential processes* without the isolation property.

Communicating Isolates
: Communicating isolates are similar to *communicating threads*, but with a strict enforcement of memory isolation between *threads of execution*. Thus, the main distinguishing feature is the absence of an inherent notion of shared memory between different threads of execution. The means of communication in this category vary substantially. They range from message or channel-based communication to restricted on-demand shared-memory data structures. Programming models in this category are for instance *actors* and *communicating event-loops*, which require isolation as one of their main characteristics.

Data Parallelism
: This category combines abstractions for data parallel loops, fork/join, map/reduce, and data-flow. The common key idea is the focus on abstractions for parallel programming instead of concurrent programming.

2.4.2. Threads and Locks

This category is currently believed to represent the mainstream of concurrent and parallel programming concepts applied in practice. Languages such

as Java [Gosling et al., 2012], C [ISO, 2011], C++ [ISO, 2012], C# [ECMA International, 2006], and Smalltalk [Goldberg and Robson, 1983] come with the notion of threads and mechanisms that enable the protection of shared resources. Threads are often directly based on the abstraction provided by the underlying operating system, however, for instance Smalltalk implementations instead tend to use green threads that are managed by the Smalltalk implementation.

In addition to threads, the mentioned languages often provide support for monitors, mutexes, locks, condition variables, and atomic operations. These mechanisms facilitate the coordination of threads at the level of resources, and are considered as low-level abstractions requiring careful engineering to prevent data races and deadlocks.

Systems software had to deal with concurrency for decades, and since these parts of the software stack are performance sensitive, the available abstractions have been threads and locking mechanisms. While experts are able to build stable systems on top of these low-level abstractions, the engineering effort is high [Cantrill and Bonwick, 2008]. Consequently, the proposed higher-level abstractions are discussed in the following sections.

2.4.3. Communicating Threads

Message or channel-based communication, as well as high-level synchronization mechanisms such as barriers [Gupta and Hill, 1989], clocks [Charles et al., 2005], and phasers [Shirako et al., 2008] fall into this category. A few selected abstractions are detailed below.

Active Objects The *active objects* pattern [Lavender and Schmidt, 1996] is one such abstraction for object-oriented languages. It distinguishes active and passive objects. Active objects are objects with an associated execution thread. Methods on these objects are not executed directly, but asynchronously by reifying the invocation and deferring its execution using a queue. The execution thread will process one such invocation at a time. Passive objects do not have their own thread of execution, but are used by the active objects. The active object pattern does not introduce any restrictions on how passive objects are to be used, but it is implied that an active object takes responsibility for its own subgraph of the overall object graph. In the case where such a design is not feasible, synchronization still needs to be done explicitly using lower-level mechanisms.

Clojure Agents The Clojure programming language offers a variation inspired by the active objects pattern called agents.[10] An agent represents a resource with a single atomically mutable state cell. However, the state is modified only by the agent itself. The agent receives *update functions* asynchronously. An update function takes the old state and produces a new state. The execution is done in a dedicated thread, so that at most one update function can be active for a given agent at any time. Other threads will always read a consistent state of the agent at any time, since the state of the agent is a single cell and read atomically.

Specific to agents is the integration with the Clojure language and the encouraged usage patterns. Clojure aims to be *"predominantly a functional programming language"*[11] that relies on immutable, persistent data structures. Therefore, it encourages the use of immutable data structures as the state of the agent. With these properties in mind, agents provide a more restrictive model than common active objects, and if these properties would be enforced, it could be classified as a mechanism for *communicating isolates*. However, Clojure does not enforce the use of immutable data structures as state of the agent, but allows for instance the use of unsafe Java objects.

The described set of intentions and the missing guarantees qualify agents as an example for later chapters. Thus, it is described here in more detail.

```
1  (def cnt (agent 0))
2  (println @cnt)      ; prints 0
3
4  (send cnt + 1)
5  (println @cnt)      ; might print 0 or 1, because of data race
6
7  (let [next-id (promise)]
8    (send cnt (fn [old-state]
9          (let [result (+ old-state 1)]
10           (deliver next-id result) ; return resulting id to sender
11           result)))
12   @next-id)         ; reliably read of the update function's result
```

Listing 2.1: Clojure Agent example

Lst. 2.1 gives a simple example of how to implement a counter agent in Clojure, and how to interact with it. First, line 1 defines a simple counter agent named `cnt` with an initial value of `0`. Printing the value by accessing it directly

[10] http://clojure.org/agents

[11] *Clojure*, Rich Hickey, access date: 20 July 2012 http://clojure.org/

with the `@`-reader macro would result in the expected `0`. At line 4, the update function `+` is sent to the agent. The `+` takes the old state and the given `1` as arguments and returns the updated result `1`. However, since update functions are executed asynchronously, the counter might not have been updated when it is printed. Thus, the output might be `1` or `0`. To use the counter for instance to create unique identifiers, this is a serious drawback, especially when the counter is highly contended. To work around this problem, a *promise* can communicate the resulting identifier ensuring synchronization. Line 8 shows how a corresponding update function can be implemented. The `next-id` promise will be used in the anonymous update function to deliver the next identifier to the sender of the update function. After delivering the result to the waiting client thread, the update function returns, and the state of the agent will be set to its return value. When reading `next-id`, the reader will block on the promise until it has been delivered.

Software Transactional Memory For a long time, various locking strategies have been used in low-level software to address the tradeoff between maintainability and performance. A fine-granular locking scheme enables higher degrees of parallelism and can reduce contention on locks, but it comes with the risks of programming errors that can lead to deadlocks and data corruption. Coarse-grained locking on the other hand is more maintainable, but may come with a performance penalty, because of the reduced parallelism.

To avoid these problems Shavit and Touitou [1995a] proposed *software transactional memory* (STM), taking concepts from the world of database systems with transactions and apply them to a programming language level. The main idea is to avoid the need for having to deal with explicit locks, and the complexity of consistent locking schemes. Thus, locking is done implicitly by the runtime system when necessary. Critical code sections are not protected by locks, but are protected by a transaction. The end of a critical section corresponds to aborting or committing the transaction. The STM system will track all memory accesses during the execution of the transaction and ensure that the transaction does not lead to data races. For the implementation of STM, a wide range of different approaches have been proposed [Herzeel et al., 2010]. Theoretically, these can provide the desired engineering benefits, however, STM systems proposed so-far still have a significant performance penalty [Cascaval et al., 2008]. The performance overhead forces developers to optimize the use and thus, has a negative impact on the theoretically expected engineering benefit.

To achieve the desired modularity with STMs, nesting of transactions is a solution that comes with additional complexity and performance trade-offs [Moravan et al., 2006]. Similarly, transactional memory systems have a number of known problems with pathological situations such as starvation of transactions and convoying of conflicting transactions [Bobba et al., 2007]. These can lead to performance problems and require additional effort to debug. These problems lead to STM being one option for handling concurrent programming, but prevent it from being a general solution.

PGAS (Partitioned Global Address Space) programming languages like Co-Array Fortran [Numrich and Reid, 1998] and Unified Parallel C [UPC Consortium, 2005] have been proposed to increase the programmer productivity in the field of *high-performance computing* applications, running on supercomputers and large clusters. Thus, they are meant for large scale distributed memory clusters, but are also used in smaller systems with strong non-uniform memory access (NUMA) [Herlihy and Shavit, 2008] characteristics.

They are commonly designed for *Single Program Multiple Data* (SPMD) scenarios providing the illusion of shared memory, while indicating to the developer the additional cost of this abstraction by differentiating between local and remote memory. The dominating synchronization mechanisms for these kind of languages are barrier-like constructs and data-oriented parallel reduce operators.

Note that PGAS and with it some of the SPMD approaches are categorized as *communicating threads* instead of *data parallel*. The main reason for this decision is that common realizations of these approaches are highly imperative and control-flow focused, while data parallel programming models tend to deemphasize the control-flow aspect.

2.4.4. Communicating Isolates

The main difference with communicating threads is the strict isolation of *threads of execution*. Thus, the basic model does not provide shared memory between isolates, which requires them to make any form of communication explicit. Especially in the context of concurrent applications, this approach is considered to be less error-prone and more high-level than *communicating threads*. In the field of high-performance computing (HPC) however, the use of explicit communication in the form of MPI [Message Passing Interface Forum, 2009] is considered more low-level than the model offered by PGAS languages. One possible explanation is that concurrent applications are often

task-oriented, while HPC applications are mostly data-oriented, which leads to different tradeoffs in the design and implementation of algorithms.

Actors The *actor model* is a formalism to describe computational systems. Hewitt et al. [1973] distilled it from the many ideas of that time in the field of artificial intelligence to provide a *"coherent manageable formalism"*. This formalism is based on a single kind of object, an actor. These actors are computational units that respond to messages by sending messages to other actors, create new actors, or designate how to handle the next message they receive. Communication is done via addresses of other actors. An actor has to be introduced explicitly to another actor or it has to create the actor to know its address. Later, Hewitt [2012] clarified that the actor model assumes that messages are processed concurrently, and more importantly, that actors do not require any notion of threads, mailboxes, message queues, or even operating system processes.

The benefit of the actor model is that it provides a concise formalism to describe systems. By using the simplest possible notion of the actor model in the way Hewitt [2012] proposes, the actor model achieves desirable theoretical properties such as locality, safety of communication, and unbounded nondeterminism.

However, the minimalism of the model comes with a significant engineering tradeoff. Similarly to the discussion of the right object granularity for code reuse, which component-based software engineering tries to address, the granularity of actors becomes important when actors are used to build software systems. The idea of *event-loop concurrency* tries to address these granularity issues.

Event-Loop Concurrency E and AmbientTalk propose the idea of actors integrated with object-oriented programming languages [Miller et al., 2005; Van Cutsem et al., 2007]. E uses the term *vat* to denote a container of objects, which in AmbientTalk corresponds to an *actor*, that holds an object graph. Inside a vat, objects can refer to each other with *near references*. References between different vats are only possible via *far references*. While near references allow synchronous access to objects, far references only allow asynchronous interaction. The semantics of far references is that of asynchronous message sends. Every vat executes an infinite loop that processes incoming messages. Based on these ideas, Van Cutsem describes the three main properties of the event-loop concurrency model as follows: *Serial execution* is guaranteed inside a vat.

Each vat processes a maximum of one event at a time. *Non-blocking communication* is ensured by construction. All operations that have blocking semantics are realized by deferring the continuation after that operation asynchronously, freeing the event-loop to enable it to process another event, and rescheduling the continuation once the blocking condition is resolved. *Exclusive state access* is guaranteed by vat semantics to ensure freedom from low-level data races and provide strong encapsulation of actors.

Event-loop concurrency is often seen as an extension of the actor model proposed by Hewitt et al. Vats, or *event-loop actors*, provide a higher level of abstraction and extend the notion of actors from elementary particles to coarser-grained components.

Communicating Sequential Processes Hoare [1978] proposed the idea of *Communicating Sequential Processes* (CSP) as a fundamental method for structuring programs. The idea is that input and output are basic programming primitives that can be used to compose parallel processes. Later the idea was developed further into a process algebra [Hoare, 1985], which is also the foundation for the *occam* programming language [May, 1983]. The main concepts of the algebra and occam are blocking communication via channels and parallel execution of isolated processes. They support the notion of instruction sequences, non-deterministic writing and reading on channels, conditions, and loops. This provides an algebra and programming model that is amenable to program analysis and verification.

The idea of channel-based communication has been used and varied in a number of other languages and settings such as JCSP [Welch et al., 2007], the Go programming language,[12] occam-π [Welch and Barnes, 2005], and the Dis VM [Lucent Technologies Inc and Vita Nuova Limited, 2003] for the Limbo programming language. Notable is here that the strong isolation of processes is often not guaranteed by the programming languages or libraries that provide support for CSP. Thus, depending on the actual implementation, many incarnations of CSP would need to be categorized as *communicating threads* instead.

MPI (Message Passing Interface [Message Passing Interface Forum, 2009]) is a widely used middleware for HPC applications. Programs are typically written in an SPMD style using MPI to communicate between the isolated, and often distributed parts of the application. MPI offers send and receive oper-

[12] http://golang.org/

ations, but also includes collective operations such as scatter and gather. In general, these operations have rendezvous semantics. Thus, every sending operation requires a matching receive operation and vice versa. MPI-2 [Message Passing Interface Forum, 2009] added operations for one-sided communication to introduce a restricted notion of shared memory in addition to the main message passing, for situations where rendezvous semantics are not flexible enough.

While MPI is widely used, and still considered the standard in HPC applications when it comes to performance, it is also criticized for not being designed with productivity in mind [Lusk and Yelick, 2007]. Its low-level API does not fit well with common HPC applications. In addition to offering comparably low-level APIs and data types, there is a potential misfit between message-passing-based programming and data-centric applications.

APGAS (Asynchronous PGAS) languages were conceived with the same goals as PGAS languages to solve the problems MPI has with regard to programmer productivity. The distinguishing feature of APGAS languages compared to PGAS languages is the notion that all operations on remote memory need to be realized via an asynchronous task that executes on the remote memory. Thus, APGAS language try to make the cost of such remote memory accesses more explicit than in PGAS languages. This programming model is supposed to guide developers to utilize data locality efficiently and to structure data and communication to reduce costly operations.

Languages such as X10 [Charles et al., 2005] and Habanero [Cavé et al., 2010] realize that idea by making locality explicit as part of the type system. X10's specification [Saraswat et al., 2012] goes so far as to define remote accesses as having a by-value semantics for the whole lexical scope. This results in a programming model very similar to message-passing. X10 combines this with a fork/join-like task-based parallelism model, which makes is a hybrid language in terms of our categorization. X10 differentiates between different *places* as its notion of locality. Across places, it enforces isolation, but inside a single place it provides a programming model that corresponds to our definition of *communicating threads*.

2.4.5. Data Parallelism

Approaches for data parallelism provide abstractions to handle data dependencies. In general, the tendency in these approaches is to move from control

driven to data driven computation. However, control driven programming, i. e., imperative programming remains important.

Fork/Join utilizes the inherent parallelism in data-oriented problems by using recursion to divide the computation into steps that can be processed in parallel. It thereby makes an abstraction of the concrete data dependencies by using recursive problem decomposition and relying on explicit synchronization points when the result of a subproblem is required. While it is itself a control-driven approach, relying on control-flow-based primitives, it is typically used for data-parallel problems. However, it leaves it to the programmer to align the program with its data-dependencies.

Cilk [Blumofe et al., 1995] introduced fork/join as a novel combination of the classic recursive divide-and-conquer style of programming with an efficient scheduling technique for parallel execution. Nowadays, it is widely known as *fork/join* and available, e. g., for Java [Lea, 2000] and C/C++ with libraries such as Intel's Threading Building Blocks[13]. Primitives of this parallel programming model are the spawn, i. e., fork operation, which will result in a possibly parallel executing sub-computation, and the sync, i. e., join-operation, which will block until the corresponding sub-computation is finished. Fork/join is a model for parallel programming in shared memory environments. It enables developers to apply divide-and-conquer in a parallel setting, however, it does not provide mechanisms to handle for instance concurrency on global variables. Such mechanisms have been proposed [Frigo et al., 2009], but the original minimal model focuses on the aspect of parallel execution.

With *work-stealing*, Cilk also pioneered an efficient scheduling technique that makes parallel divide-and-conquer algorithms practical for situations in which a static schedule leads to significant load imbalances and thus suboptimal performance.

MapReduce Functional programming languages have introduced the notion of mapping a function on a sequence of values to produce a result sequence, which then can be reduced to some result value with another function. Based on this simple notion, distributed processing of data has become popular [Lämmel, 2008]. For companies like Google, Microsoft, or Yahoo, processing of large amounts of data became a performance challenge that required the use of large clusters and resilient programming models. The model

[13] http://threadingbuildingblocks.org/

proposed by Dean and Ghemawat [2004] includes mechanisms to provide fault tolerance and scalability to utilize large clusters. It also extends the basic map/reduce idea with notions of combiner functions and they describe how to support side-effects for map and reduce operators. The side-effects are however restricted to being idempotent and atomic to ensure deterministic results in the case of failure and recomputation. Side-effects on the input data themselves are however not supported. Instead, the input data are considered to be immutable for the overall process.

Compared to fork/join, MapReduce exposes the developer much less to the aspect of control flow. Instead, it requires only input data and a set of operators that are applied in a predefined order, without making any promises about the order in which the input data are processed.

Data-flow Programming languages attempt to move entirely to data-dependency based program representation. Languages such as Lucid [Ashcroft and Wadge, 1977] do not regard the sequential notation as imperative to the order of program execution. Instead, programs are evaluated in a lazy, demand-driven manner.

Other languages such as StreamIt [Thies et al., 2002] make data dependencies even more explicit by reifying the notion of data streams to which a set of kernel functions is applied. These programming languages enable the explicit encoding of data dependencies, which can then be used by optimizing compilers to generate highly efficient code that exploits the available data parallelism in the application.

2.4.6. Summary

This section introduced a partial taxonomy to categorize concurrent and parallel programming concepts, because Flynn's and Almasi and Gottlieb [1994]'s taxonomies do not reflect the common approaches to concurrent and parallel programming in a way that facilitates their discussion in the context of this dissertation. Therefore, this dissertation proposes to categorize concurrent and parallel programming concepts into *communicating threads*, *communicating isolates*, and *data parallelism*. For each of these categories, this section discussed a number of common approaches.

Clojure agents are highlighted and discussed in more detail, because Chapter 5 and Chapter 6 rely on them as a running example.

2.5. Building Applications: The Right Tool for the Job

Researchers often make the case that the implementation of parallel and concurrent systems is a complex undertaking that requires the right tools for the job, perhaps more so than for other problems software engineering encountered so far [Cantrill and Bonwick, 2008; Lee, 2006; Sutter, 2005]. Instead of searching for a non-existing *silver bullet* approach, this dissertation argues, like other research [Catanzaro et al., 2010; Chafi et al., 2010], that language designers need to be supported in building domain-specific concurrency abstractions.

Typical desktop applications such as the e-mail application sketched in Fig. 2.1 combine several components that interact and have different potential to utilize computational resources. The user interface component is traditionally implemented with an event-loop to react to user input. In a concurrent setting, it is also desirable to enforce encapsulation as in an actor model, since encapsulation simplifies reasoning about the interaction with other components. Thus, a programming model based on event-loop concurrency might be the first choice.

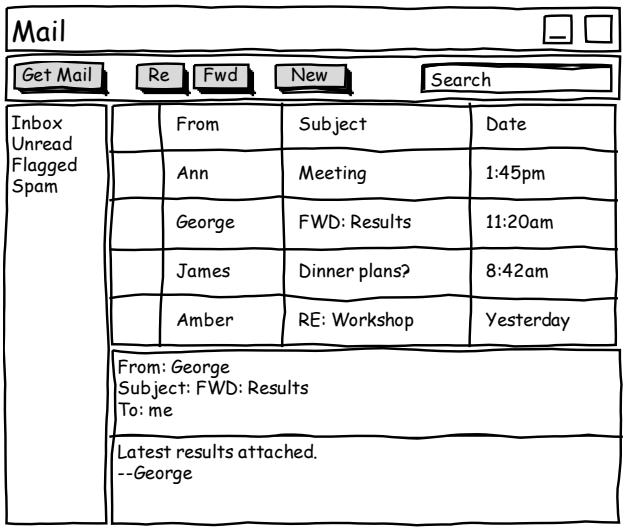

Figure 2.1.: Mockup of an E-Mail Application

Another part of the application is data storage for emails and address book information. This part traditionally interacts with a database. The natural way to implement this component is to use an STM system that extends the transaction semantics of the database into the application. This allows for unified

reasoning when for instance a new mail is received from the network component and needs to be stored in the database.

A third part is a search engine that allows the user to find emails and address book entries. Such an engine can typically exploit data-parallel programming concepts like map/reduce or parallel collection operations with PLINQ[14] for performance.

However, supporting the various parallel and concurrent programming concepts on top of the same platform comes with the challenge to identify basic commonalities that allow to make abstractions of the particularities of specific constructs and languages. Today's VMs such as JVM and CLI provide direct support for threads and locks only. While some concepts such as fork/join [Lea, 2000], concurrent collections [Budimlic et al., 2009], or PLINQ can be implemented as libraries without losing any semantics or performance, concepts such as the actor model are typically implemented with weaker semantics than originally proposed, losing for instance the engineering benefits of encapsulation [Karmani et al., 2009].

While it is desirable to support different kinds of models, it is not clear how a single language can support them directly. In his PLDI'12 keynote, Doug Lea[15] emphasized the point that *"effective parallel programming is too diverse to be constrained by language-based policies"*. While he combines the notions of concurrent and parallel programming (cf. Sec. 2.3), others have raised similar opinions before. The problems that need to be tackled by concurrent and parallel programming techniques are too diverse to be tackled appropriately by a fixed set of abstractions [Catanzaro et al., 2010; Chafi et al., 2010]. Instead, domain-specific abstractions are necessary to be able to achieve an appropriate level of abstraction and achieve the desired performance.

The goal of this dissertation is to identify a way to realize this vision and enable library and language implementers to provide domain-specific abstractions for concurrent and parallel programming.

2.6. Summary

This chapter gave an overview over the context and motivation behind the research in this dissertation.

[14] http://msdn.microsoft.com/en-us/library/dd460688.aspx

[15] *Parallelism From The Middle Out*, Doug Lea, access date: 16 July 2012 http://pldi12.cs.purdue.edu/sites/default/files/slides_pldi12-dlea.pdf

First, it discussed the notion of multi-language VMs as general purpose platforms, which are used for a wide range of applications. Language implementers need to be supported in implementing abstractions for concurrent and parallel programming to enable VMs to remain general purpose platforms. Furthermore, VMs need unifying abstractions for that support, because supporting a wide range of independent features in a VM is infeasible.

Second, this chapter briefly revisited the background behind the multicore revolution, which increases the need to support concurrent and parallel programming in VMs. Multicore processors will play a major role for the foreseeable future and therefore, software developers will need to utilize them to satisfy their application's performance requirements.

Third, this chapter proposed the notions of *concurrent programming* and *parallel programming* to enable the categorization of the corresponding programming concepts based on their intent, realizing two distinct sets of concepts. Concurrent programming concepts are meant to coordinate modification of shared resources, while parallel programming concepts are meant to coordinate parallel activities to compute a common result.

Fourth, a partial taxonomy is proposed to categorize concurrent and parallel programming concepts into *threads and locks*, *communicating threads*, *communicating isolates*, and *data parallelism*. This chapter discusses a number of concepts based on this categorization and details Clojure agents, because they are used in the remainder of this dissertation as a running example.

Finally, the chapter concludes with a vision on how to build applications in the multicore era. Applications need to be able to exploit concurrency and parallelism, and software developers want to utilize appropriate programming abstractions for the different parts of an application to achieve their goals. With this vision in mind, the next chapter discusses the question of which concurrent and parallel programming concepts multi-language VMs needs to support.

3

WHICH CONCEPTS FOR CONCURRENT AND PARALLEL PROGRAMMING DOES A VM NEED TO SUPPORT?

The goal of this chapter is to identify requirements for a unifying substrate that supports parallel and concurrent programming. First, a survey of the state of the art in VM support finds that support for parallel programming is relegated to libraries, while concurrent programming is only supported selectively. Thus, VM support is currently insufficient for supporting the notion of a multi-language runtime for concurrent and parallel programming. Second, a survey of the field of concurrent and parallel programming identifies concepts that significantly benefit from VM support either for performance improvement or to guarantee correct semantics. Based on these concepts, this chapter derives general requirements for VMs, which cover two independent areas of research. This dissertation focuses on the research to enforce language semantics, and thus correctness, leaving the research on performance improvement for future work. With this focus in mind, this chapter discusses common problems of language implementers that need to be solved to improve support for concurrent programming on multi-language VMs. Finally, the survey results and discussed problems are used to extract concrete requirements for the design of a unifying abstraction for concurrent programming.

3.1. VM Support for Concurrent and Parallel Programming

The goal of this section is to investigate the state of the art in VM support for concurrent and parallel programming, in order to determine the requirements for multi-language VMs. To this end, the survey identifies for thirteen VMs*which* concepts they support and *how* the VMs expose them. First, this section details the survey design, i. e., the questions to be answered for each VM, the VMs to be discussed, and the survey approach to be taken. Then it categorizes the VMs based on the taxonomy used in Sec. 2.4.1 and reports on the support they provide. Finally, it discusses the threats to validity and presents the conclusions.

The main conclusion is that currently most VMs support only one or two categories of concurrent programming concepts as part of the VM. Moreover, the analyzed VMs support parallel programming in libraries only. Thus, the vision of multi-language VMs that offer support for a wide range of different approaches to concurrent and parallel programming is as yet not supported.

3.1.1. Survey Design

The survey is designed to answer the following question: *How do today's VMs support parallel and concurrent programming?*

Following the goal of this dissertation, this survey focuses on VMs that are used as multi-language VMs, i. e., VMs that have been designed as platforms for multiple languages or are contemporary targets for multiple language implementations. To complement these VMs, the survey includes a number of high-level language VMs that provide support for concurrent and parallel programming concepts or are known for contributions to VM implementation techniques.

3.1.1.1. Survey Questions

To focus and standardize the survey, the following concrete questions are answered for each survey subject, i. e., VM:

Which concepts are supported by the language runtime system, i. e., VM?

How are the supported concepts exposed by the VM?

The first question refers to the general set of concepts for concurrent and parallel programming that is supported by a subject, i. e., the VM under investigation. Note, the question refers explicitly to *language runtime systems* instead of VMs. Thus, the analysis includes concepts provided by the corresponding standard library as well. Hence, a wider range of concepts is covered and it becomes possible to determine how the concepts that are directly provided by the VM are used.

To answer the second question, each concept is assigned to one of the following four categories:

Implicit Semantics Concepts like the *memory models* guaranteed by the JVM and the CLI, or the *global interpreter lock* semantics used by Python, are realized by the combined underlying infrastructure instead of a single mechanism. Thus, overall semantics of the system implicitly supports the concept by coordinating a wide range of mechanisms.

Instruction Set Architecture (ISA) Concepts that are either supported by a specific set of opcodes, or concepts which are realized by the structures which the VM operates on are classified as being part of the instruction set architecture (ISA). Examples are opcodes to acquire and release locks, as well as flags in a method header that require the VM to acquire an object's lock before executing that method.

Primitive Concepts that require direct access to VM internals but do not fit into the ISA are typically realized by so-called primitives. They are routines provided as part of the runtime, implemented in the implementation language of the VM. Common examples are thread-related operations.

Library Other concepts can be delivered as an integral part of the language runtime system, but are implemented entirely in terms of other abstractions provided by the VM, i. e., without requiring new primitives for their implementation.

Appendix A documents the outline of the questionnaire for this survey in Lst. A.1. The appendix also includes an example of a completed questionnaire in Lst. A.2.

3.1.1.2. Survey Subjects

Based on the goal of this dissertation, which is improvementing concurrent and parallel programming support for multi-language VMs, the focus of this

survey is the set of VMs that are used as runtime platforms for a wide range of languages for a single system. Thus, distributed VMs such as the PVM [Geist et al., 1994] are not considered because they solve a distinct set of problems such as physical distribution and fault tolerance, which are outside of the scope of this dissertation.

This survey considers the Java Virtual Machine (JVM) [Lindholm et al., 2012], the Common Language Infrastructure (CLI) [ECMA International, 2010], and the Parrot VM to be multi-language VMs. The JVM is with the addition of the INVOKEDYNAMIC bytecode [Rose, 2009; Thalinger and Rose, 2010] explicitly designed to host a wider range of languages. The CLI and Parrot VM[1] have been explicitly designed as execution platforms for multiple languages from the beginning. Because of its close relation to the JVM (cf. Sec. 3.1.2.1), the survey includes the Dalvik VM as a multi-language VM, as well.

While JavaScript and its VMs have not been designed as multi-language platforms, wide availability led to their adoption as a platform for language implementation.[2] Therefore, this survey includes JavaScript as well. The discussion is based on JavaScript's standardized form ECMAScript [ECMA International, 2011] and the upcoming HTML5 standard to cover WebWorkers[3] as a mechanism for concurrent programming.

To complement these multi-language VMs, this survey includes additional VMs that either support relevant concurrent and parallel programming concepts, or have contributed to VM implementation techniques. The selected VMs are: DisVM [Lucent Technologies Inc and Vita Nuova Limited, 2003], Erlang [Armstrong, 2007], Glasgow Haskell Compiler (GHC), Mozart/Oz, Perl,[4] Python,[5] Ruby,[6] Self [Chambers et al., 1989], and Squeak [Ingalls et al., 1997]. An overview over the subjects, including the relevant version information is given in Tab. 3.1.

3.1.1.3. Survey Execution

For each subject of the survey, the analysis uses the VM specification if it is available and if it provides sufficient information for the assessment. Otherwise, it examines the source code of the implementation and the language

[1] *Parrot – speaks your language*, Parrot Foundation, access date: 4 December 2012 http://www.parrot.org/

[2] *altJS*, David Griffiths, access date: 4 December 2012 http://altjs.org/

[3] http://www.whatwg.org/specs/web-apps/current-work/multipage/workers.html

[4] http://perldoc.perl.org/perl.html

[5] http://docs.python.org/release/3.2.3/

[6] http://www.ruby-lang.org/en/documentation/

VM	Spec.	Src.	Version
CLI	X		5th edition
Dalvik		X	Android 4.0
DisVM	X		4th edition
ECMAScript+HTML5	X	X	ECMAScript5.1, HTML5
Erlang		X	Erlang/OTP R15B01
GHC		X	GHC 7.5.20120411
JVM	X	X	Java SE 7 Edition
Mozart		X	1.4.0.20080704
Perl		X	5.14.2
Python		X	3.2.3
Ruby		X	1.9.3
Self		X	4.4
Squeak	X	X	4.3

Table 3.1.: VM survey subjects, their version, and the availability of specification and source code.

documentation. For example, the CLI's specification describes the relevant parts of the standard library and is therefore deemed to be sufficient. For the JVM however, the analysis includes one of the actual implementations (HotSpot and OpenJDK[7]) and its accompanying documentation since the standard library is not covered by the specification. In the case of ECMAScript and HTML5, the specification is complemented with an analysis of the implementations based on V8[8] and SpiderMonkey[9]

For languages without specification such as PHP, Python, or Ruby, the analysis includes only the implementation that is widely regarded to be the official, i. e., standard implementation, as well as the available documentation.

To answer the survey questions, the analysis assesses whether a concept is realized with the help of VM support or purely as a library. For VMs where only the specification was examined, the assessment was deduced from this information. For VMs where the implementation was inspected, the decision was simpler. A concept is considered to be realized as a library if its implementation is written completely in the language provided by the VM. None of the VM's primitives, i. e., mechanisms directly provided by the VM are an essential part of the concept's implementation. This criterion was unambiguous in this survey because none of the inspected implementations relied on a

[7] http://openjdk.java.net/
[8] https://github.com/v8/v8/archive/3.9.24.zip
[9] http://ftp.mozilla.org/pub/mozilla.org/firefox/releases/12.0b5/

meta-circular implementation, which makes primitives and their use directly identifiable in the code.

3.1.2. Results

Structure of the Discussion This section first discusses the VMs supporting *Threads & Locks* (T&L), then the ones with support for *Communicating Threads* (ComT), and finally the ones supporting *Communication Isolates* (ComI) (cf. Sec. 2.4.1). In the case a VM supports more than a single category, the discussions for the different categories build on each other. The category of *Data Parallelism* (DPar) is not discussed because all VMs examined in this survey relegate support for parallel programming to libraries. These libraries build on lower-level mechanisms, which are concurrent programming concepts on their own and are used for other purposes as well.

The remainder of this section discusses the survey questions and highlights the concepts supported by the different VMs. A full overview of the survey is given in Appendix A, Tab. A.2.

General Remarks Tab. 3.2 provides an overview of the categorization of the various VMs. Furthermore, while three of the VMs can be categorized as belonging to two categories, most of the VMs focus on supporting a single category of concurrency-related mechanisms.

VM	T&L	ComT	ComI	DPar
CLI	X		X	Lib
Dalvik	X			Lib
DisVM		X		
ECMAScript+HTML5			X	
Erlang			X	
GHC		X		Lib
JVM	X			Lib
Mozart	X	X		
Perl			X	
Python	X		X	
Ruby	X			
Self	X			
Squeak	X			

Table 3.2.: Categories of approaches supported by VMs: Threads & Locks (T&L), Communicating Threads (ComT), Communicating Isolates (ComI), and Data Parallelism (DPar)

The Parrot VM is excluded from the remainder of the discussion, because the mismatch between implementation and documentation is too significant. While it is a interesting and relevant subject, a correct assessment of its feature set and capabilities was not possible (cf. Sec. 3.1.2.5).

3.1.2.1. Threads and Locks (T&L)

The group of VMs supporting concurrency abstractions based on *Threads and Locks* is the largest. Tab. 3.2 shows that eight of the selected VMs provide mechanisms for these abstractions. However, they expose them in different ways, e. g., as part of the ISA or as primitives.

Common Language Infrastructure (CLI) The Common Language Infrastructure is the standard describing Microsoft's foundation for the .NET Framework, and was developed as a reaction to the success of Java. Furthermore, it was designed as a common platform for the various languages supported by Microsoft. Initially, this included C#, Visual Basic, J#, and Managed C++. While JVM and CLI have many commonalities [Gough, 2001], the designers of the CLI benefited from the experiences gathered with the JVM.

The analysis of the Common Language Infrastructure (CLI) is solely based on its standardized ECMA-335 specification [ECMA International, 2010]. Since the specification includes a discussion of the relevant details, concrete implementations such as Microsoft's .NET or Mono were not considered.

The CLI specifies shared memory with threads as the standard abstraction used by applications. It further specifies that the exact semantics of threads, i. e., whether they are cooperative or pre-emptive, is implementation specific. To facilitate shared memory programming, the CLI specifies a memory model. The memory model includes ordering rules for read and write operations to have reliable semantics for effects that need to be observable between threads. The CLI further includes the notions of volatile variables as part of the instruction set. The ISA provides the instruction prefix '`volatile.`' to indicate that the subsequent operation has to be performed with cross-thread visibility constraints in mind, as specified by the memory model. The memory model further guarantees that certain reads and writes are atomic operations. For this survey, the memory model and the related mechanisms are considered to be realized by the implicit semantics implemented in the VM and its just-in-time (JIT) compiler.

The foundation for locks and monitor primitives exposed in the standard libraries is laid in the specification as well. Building on that, the object model

defines that each object is implicitly associated with a lock to enable synchronized methods. The metadata associated with every method, which is part of the ISA, has therefore to contain the `ImplFlags.Synchronized` flag, according to the specification. Other atomic and synchronization operations are defined in terms of primitives for instance for compare-and-swap, atomic update operations, and explicit memory barriers, i. e., fences.

Beyond these mechanisms, the specification also discusses the functions `Parallel.For`, `Parallel.ForEach`, and `Parallel.While`. They are part of the standard library to provide parallel looping constructs. With the notion of `AppDomains` it also includes a mechanism to facilitate *communicating isolates*, which is discussed in Sec. 3.1.2.3.

Java Virtual Machine (JVM) The analysis of the Java Virtual Machine (JVM) relies on the specification [Lindholm et al., 2012] and considers the OpenJDK7 for details about the standard library.

The JVM provides a programming model that is solely based on shared memory. It relies on the memory model defined by the Java Language Specification [Gosling et al., 2012] to establish the semantics of operations on this shared memory and the visibility of changes to the memory between operations and threads. The semantics are defined in terms of *happens-before* relationships between operations. For example, synchronization operations as well as reading and writing of volatile fields have specific semantics, which constrain optimizations of compilers and processors to guarantee that the observable ordering of operations is deterministic. Similar to the CLI, these semantics are realized in terms of the implicit semantics implemented in the VM and its JIT compiler.

In addition to the memory model, locking-based abstractions are a key concurrency feature, because every object is implicitly associated with a lock to enable synchronized methods and can also be used as a condition variable. Methods can carry the `ACC_SYNCHRONIZED` flag, and the two bytecode instructions `monitorenter` and `monitorexit` expose the object's lock on the ISA level to enable the implementation of synchronized blocks. A note in the specification also hints at `Object.wait` and `Object.notify` being realized as primitives. In contrast to the CLI, the JVM does not have the notion of volatile variables, because it does not have closures. Instead, it provides only volatile object fields, which are realized in the ISA by a the `ACC_VOLATILE` flag in the object field's metadata.

The JVM does not provide any high-level mechanisms for communication between threads. While the standard library provides futures, a set of concurrent objects/data structures, barriers, and the necessary abstractions for fork/join programming, these libraries completely rely on the abstractions provided by the VM in terms of volatile fields, locks, and atomic operations.

Dalvik VM Google develops the Dalvik VM for its Android platform. While there is no specification available, Google stated its goal to build a VM that implements the semantics of the Java Language Specification (JLS) [Gosling et al., 2012].

An inspection of the source code reveals that Dalvik provides `monitorenter` and `monitorexit` instructions to interact with an object's lock, which are similar to the instructions in the JVM. Furthermore, Google intends to provide a Java-compatible standard library as well. With these similarities in mind, Dalvik is classified for the purpose of this discussion as a JVM derivative. It differs in certain points from the JVM, e. g., using a register-based bytecode format and a different encoding for class files, but does not deviate from the JVM/JLS when it comes to concurrency and parallelism-related aspects.

Mozart The Mozart VM is an implementation of the Oz language. The multi-paradigm approach of Oz also has an impact on the VM and the concepts it exposes. The Oz language provides a variety of different abstractions for concurrent and distributed programming. This includes a shared memory model extended by different adaptable replication semantics and remote references for distributed use. The Mozart VM combines the notions of green threads, locks, and data-flow variables. It exposes `Ports` as a channel-based abstraction, and additionally reifies data-flow variables to be used as futures/promises. Similarly to the CLI and JVM, lock support is provided at the instruction set level with a parameterized `LOCKTHREAD` instruction that represents operations on reentrant locks. In addition to this VM support, the standard library and documentation discuss concepts built on top of these abstractions, e. g., monitors and active objects.

Sec. 3.1.2.2 discusses the abstractions related to *Communicating Threads*.

Python Traditionally, Python offers a thread-based shared memory programming model. In contrast to the CLI and JVM, its memory model is defined by *global interpreter lock*-semantics. The global interpreter lock (GIL) prevents parallel execution of Python code, however, it for instance enables the use of I/O

or computational intensive operations provided by the underlying system or primitives. Thereby, it enables a certain degree of parallel execution and latency hiding. C modules for Python are able to use `PyEval_AcquireLock()` and `PyEval_ReleaseLock()` to release the lock before starting I/O or computational intensive operations that do not require interpreter intervention and thereby enable other threads to use it.

The `threading` module provides threads, locks, recursive locks, barriers, semaphores, and condition variables. However, only thread operations, locks, and recursive locks are implemented as primitives. Barriers, semaphores, and condition variables are built completely on top of them.

Similarly to Mozart, Python's support for *Communicating Isolates* is discussed separately in Sec. 3.1.2.3.

Ruby Similarly to Python, Ruby's standard implementation is based on a GIL, called global VM lock (`gvl`). C modules that want to execute in parallel can do so by using the high-level `rb_thread_blocking_region(..)` function, by which they promise to take care of threading issues by themselves, and the VM will release the `gvl` during execution. However, this is of course not offered at the language level.

To the user, Ruby offers primitives for `Threads` and `Mutexes` combined with the shared memory model as part of VM support. The standard library builds on this basic support by offering for instance condition variables.

Self and Squeak Squeak is based on the ideas of Smalltalk-80 [Goldberg and Robson, 1983] and follows its specification closely (cf. sections 4.3 and 4.4). Self [Chambers et al., 1989] belongs to the Smalltalk family of languages as well, but has significant differences in terms of its object model. Furthermore, it pioneered important language implementation techniques such as dynamic optimization. Both Squeak and Self closely follow the standard Smalltalk model when it comes to concurrent and parallel programming. They offer the notion of green threads in combination with semaphores.

Squeak implements these concepts based on primitives. It provides for instance `yield`, `resume`, and `suspend` primitives to influence the scheduling of green threads (instances of the `Process` class). Furthermore, it provides the `signal` and `wait` primitives to interact with the semaphore implementation of the VM. Based on these, the libraries provide mutexes, monitors, and concurrent data structures such as queues.

In Self, the functionality is implemented completely in libraries. The VM provides only preemptive green threads and the so-called `TWAINS_prim` primitive. This provides sufficient VM support to build scheduling, semaphores, and other abstractions on top.

3.1.2.2. Communicating Threads (ComT)

Similarly to *Threads and Locks*, the *Communicating Threads*-based models offer shared memory and come with the notion of threads as a means of execution. However, they offer approaches to communicate between threads that are distinct from approaches that rely on locks for correctness. For instance, channel-based communication has software engineering tradeoffs different from the use of locks.

Dis VM The Dis VM [Lucent Technologies Inc and Vita Nuova Limited, 2003] is part of the Inferno OS and hosts the Limbo programming language. Since the source code does not seem to be available, the analysis relies solely on the specification and manuals.

Limbo, and consequently the Dis VM are strongly inspired by Hoare's CSP (cf. Sec. 2.4.4). However, Limbo and the Dis VM provide the notion of shared mutable memory between threads and do not follow the notion of isolated processes as proposed by CSP.

The instruction set of the Dis VM provides operations to mutate shared memory, spawn threads, and use channel-based communication with sending and receiving. Since the Dis VM uses a memory-to-memory instruction set architecture, instead of relying on registers, most operations mutate heap memory. While threads are available, there are no locking-like primitives or instructions available. Instead the `send`, `recv`, and `alt` channel operations are the only instructions in the specification that provide synchronization.

Nevertheless, the Limbo language manual demonstrates how monitors can be built on top of channels.

Glasgow Haskell Compiler The Glasgow Haskell Compiler (GHC) was one of the first language implementations to include a software transactional memory (STM) system in its standard distribution. While Haskell is designed as a side-effect free language with lazy execution semantics, its monadic-style can be used in an imperative way, which makes it necessary to coordinate side-effects when parallel execution is used. To that end, GHC introduced

STM based on mutable `TVars` [Harris et al., 2005]. TVars, i. e., transaction variables, are shared memory locations that support atomic memory transactions. It is implemented as part of the GHC runtime system, i. e., it is supported directly by the VM. One of the provided STM-implementation strategies uses fine-grained locking of TVars to minimize serialization of execution during commits.

Independently from the STM system, GHC supports explicit spawning of parallel executing *sparks*, which are lightweight threads. Haskell's `par` eventually results in a call to the corresponding `newSpark` primitive in the runtime.

With the support for `MVars`, GHC enables additional communication patterns, for instance channel-based communication, building on the ideas of M-Structures [Barth et al., 1991; Peyton Jones et al., 1996]. MVars are synchronization variables that can store values. Their basic operations are `takeMVar` and `putMVar`. When an MVar is *empty*, a `takeMVar` will block. Similarly, `putMVar` will block when it is *full*. MVars are implemented in the runtime (`StgMVar`).

The standard library provides a semaphore construct on top of MVars. Furthermore, the library includes proper channels and various abstractions to perform map/reduce and data parallel operations.

Mozart The data-flow variables of Oz, and their direct support in the Mozart VM enable algorithm designs that are significantly different from algorithms based on threads and locks. An `OzVariable` represents these data-flow variables inside the VM and for instance keeps track of all threads that are suspended on unresolved data dependencies. The directly exposed VM support for channels (`OzPort`) enriches the choice for programmers even further. Therefore, we conclude that the Mozart VM provides abstractions for *Communicating Threads* in addition to the *Threads and Locks*-based ones.

3.1.2.3. Communicating Isolates (ComI)

As defined in Sec. 2.4.1, the main concept introduced by communicating isolates is the strong *state encapsulation* between concurrent entities. This breaks with the typical shared memory model and enforces clear separation between components. Communication has to be performed either via value-based messaging or by using explicitly introduced and constrained shared memory data structures.

Common Language Infrastructure In addition to *Threads and Locks*-based abstractions, the CLI specifies so-called *application domains* that provide a no-

tion of isolated execution of different applications in the same VM process. This concept has to be supported by a VM implementation in order to provide isolation properties similar to operating system processes. Different application domains can only communicate using the *remoting* facilities.[10] *Remoting* is Microsoft's and the CLI's technique for inter-process communication and distributed objects. The remoting facilities enable remote pointers and marshaling protocols for remote communication in an object-oriented manner. They are realized with `System.MarshalByRefObject` which is according to the specification supposed to use a proxy object, which locally represents the remote object of another application domain.

Because application domains are isolated from each other, the static state of classes is not shared between them either. This isolation approach goes as far as to require types to be distinct in different application domains.

ECMAScript+HTML5 JavaScript as well as the current version of the corresponding ECMAScript standard [ECMA International, 2011] do not provide any abstractions for concurrent or parallel programming. Since browsers use event loops to react to user input, they process the corresponding JavaScript code at a later point in time to guarantee sequential execution. Furthermore, the execution model is nonpreemptive, and each turn of the browser's event loop executes completely and is not interrupted. In the absence of parallel execution on shared memory, ECMAScript/JavaScript does not provide any mechanisms for synchronization.

However, this survey includes the widely available *Web Worker*[11] extension, which is proposed for standardization as part of HTML5. Web workers introduce the notion of background processes that are completely isolated from the main program. The specification names the communication interface `MessagePort` and offers a `postMessage` method as well as an `onmessage` event handler to enable communication between web workers. They communicate solely via these channels with by-value semantics. The two JavaScript browser runtimes SpiderMonkey and V8 support processes and channels directly.

Erlang Erlang is known for its support of actor-based concurrent programming. It is implemented on top of the BEAM (Bogdan's Erlang Abstract Machine [Armstrong, 2007]), which exposes the main abstractions required for an

[10] *.NET Remoting Overview*, Microsoft, access date: 15 Sep. 2012
http://msdn.microsoft.com/en-us/library/kwdt6w2k(v=vs.71).aspx

[11] http://www.whatwg.org/specs/web-apps/current-work/multipage/workers.html

actor-based language directly to application programs. The main operations are directly provided by the instruction set. The `send` instruction implements asynchronous message sends to a specified actor, i. e., Erlang `process`. The `wait` and `wait_timeout` instructions are used to wait for incoming messages on the message queue. To facilitate Erlang's pattern matching, messages are only removed from the message queue when the explicit `remove_message` instruction is used. However, new Erlang processes are spawned by a primitive.

For use-cases like the Mnesia database,[12] Erlang also provides a set of primitives for mutable concurrent hash-tables (`ets`, Erlang term storage[13]) that can be shared between actors. While this introduces a restricted notion of shared state, it is not part of the core primitives suggested for general use.

Perl The Perl[14] VM offers support for so-called `threads`. However, this terminology is not in line with the commonly accepted notion of threads. Perl's threads are more directly comparable to OS processes as they provide a non-shared memory programming model by default. However, the VM provides primitives in the `threads::shared` module to enable the use of shared data structures between these threads. Data structures can only be shared between threads after they have been explicitly prepared for sharing with the `share()` primitive. To use this restricted form of shared memory, the VM provides the `Queue` data structure for channel-like communication, as well as a `lock` primitive, a `Semaphore` class, and primitives for condition variables.

Python As discussed in Sec. 3.1.2.1, Python's main programming model is based on threads and locks with a VM implementation that is constrained by global interpreter lock (GIL) semantics. Since the GIL hinders any form of parallel execution of Python bytecodes, VM support for process-based parallelism was added. The standard library provides the `multiprocessing` module to offer an abstraction for parallel programming similar to Perl. Python's VM includes primitives for channels in the form of a message queue (`Queue`), as well as primitives for semaphores on top of which for instance the `Lock` and `Condition` classes are built. These can operate on shared memory that has to be requested explicitly between communicating processes and is restricted to certain data structures. `Value` and `Array` are the shareable data structure primitives directly exposed by the library. `Value` is by default a synchronized

[12] http://www.erlang.org/doc/apps/mnesia/
[13] http://www.erlang.org/doc/man/ets.html
[14] http://perldoc.perl.org/perl.html

wrapper around an object, while `Array` wraps an array with implicit synchronization.

3.1.2.4. Data Parallelism (DPar)

Surprisingly, none of the VMs examined in this survey expose direct support for data parallelism. Instead, they are typically accompanied by standard libraries that build support for parallel programming on top of lower-level abstractions. Examples are the CLI with its library of parallel loop constructs, the JVM (incl. Dalvik) with its support for fork/join parallelism in the `java.util.concurrent` library,[15] and GHC with libraries for map/reduce parallelism. These libraries are built on top of concepts already available, e. g., threads, atomic operations, and locks, and therefore the analyzed VMs do not provide primitives that expose parallel programming concepts directly.

3.1.2.5. Threats to Validity

Completeness of Selected Subjects The main survey question was: *how do today's VMs support parallel and concurrent programming approaches?* The answer to this question is based on a set of thirteen VMs. Thus, the number is limited and the selected set may exclude VMs that provide other mechanisms and expose them in other ways to the application programmer. However, since the survey covers contemporary multi-language VMs, and a number of additional VMs respected for their support for parallel and concurrent programming, it covers the VMs that are directly relevant for the goal of this dissertation. Furthermore, the selected VMs reflect *common practice* in VM support for concurrent and parallel programming approaches, because the selection includes the most widely used VMs, as well.

Correctness and Completeness of Results With respect to the concrete survey questions of *which* concepts are exposed and *how* they are exposed, a number of threats to correctness and completeness of the results need to be considered.

General Threats to Accuracy As shown in Tab. 3.1, the analysis did not include the implementations of the CLI and DisVM. Instead, the quality of their

[15] http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/package-summary.html

specification is deemed sufficiently precise to answer the questions for *which* and *how* concepts are exposed accurately.

Other aspects have a stronger impact on the confidence in the results. In general, the survey is based on the specifications, language manuals, and inspection of actual implementations. While it initially included the Parrot VM[16] as an interesting subject, it was omitted, since the mismatch between documentation and implementation was major. It was not clear whether the documentation was outdated or visionary, or whether the understanding of the implementation was not sufficient, making a proper assessment impossible. This example raises the question of how valid the results are. Most implementations use customary names and make an effort to be comprehensible, which results in a high confidence in the results. However, it is not possible to eliminate the possibility that the analysis is to a lesser extent inaccurate or incomplete.

Exposure Assessment The confidence in the assessment that a given concept is supported directly by a VM is high, because the VM implementations provide direct evidence that a concept is supported as primitive or as part of the the instruction set. However, a concept may have been missed. Thus, some concepts may in fact be supported by the VM directly. Completeness is thus an issue for the question of whether concepts are directly supported by a VM. This assessment could also not by verify trivially. However, this form of completeness has only little impact on the overall results, since it includes a larger number of different VMs.

Concept Completeness The confidence in the completeness of the overall identified set of concepts that is exposed by a VM and its libraries is high, because the typically sufficiently extensive and well structured language documentations enable a trivial verification for completeness.

Problematic are concepts that are solely available in the implementations and remain undocumented. One example for such a case is the relatively well know `Unsafe` class in Oracle's JVM implementation.[17] Other similarly private APIs might have been missed. Arguably, such concepts are not meant to be used by programmers targeting the VM, and therefore do not need to be considered as concepts that are offered by the VM. Instead, these concepts are

[16] http://www.parrot.org/

[17] http://hg.openjdk.java.net/jdk7/jdk7/jdk/log/tip/src/share/classes/sun/misc/Unsafe.java

considered internal basic building blocks outside of the scope of this survey, and thus do not have an influence on our results.

3.1.3. Conclusion

Summary This survey investigated the state of the art in VM support for concurrent and parallel programming. It examined thirteen VMs, including contemporary multi-language VMs and a number of VMs selected for their support for concurrent and parallel programming. The analysis identified for each VM the concepts it exposes and whether it exposes them in terms of *implicit semantics*, as part of the VM's *instruction set architecture*, in terms of *primitives*, or merely as part of the standard libraries. The main insight is that the analyzed VMs only support one or two categories of concepts. Furthermore, they consistently relegate support for parallel programming to the standard library without providing explicit support for optimization.

Concept Exposure In answering the question of how concepts are exposed, the survey shows that very general concepts such as shared memory, memory models, and global interpreter lock semantics are realized by a combination of mechanisms in the VM, which were categorized as *implicit semantics* (cf. Sec. 3.1.1.1). Typically, they have an impact on most parts of a VM, because they require guarantees from a wide range of VM subsystems. More restricted, and often performance sensitive concepts are exposed as part of the overall *instruction set architecture*. Examples are monitors, synchronized methods, volatile variables, and in some cases also high-level concepts like channels, message sends, message receives, and threads or processes.

Primitives are used for a wide range of concepts. Design decisions differ between VMs, thus some concepts are supported either in the instruction set or as primitives, e. g., locks, channels, and threads, but also concepts like atomic operations, and condition variables. Other high-level concurrency concepts such as concurrent data structures are provided as libraries. With the definitions of concurrent and parallel programming of Sec. 2.3 in mind, the conclusion is that none of the concepts that are provided with implicit semantics, ISA support, or primitives is directly related to parallel programming. Instead, all the identified parallel programming concepts have been realized in the form of libraries.

Only limited support for concurrent and parallel programming. Since support for parallel programming is based on libraries, none of the VMs is cate-

gorized as providing abstractions for *Data Parallelism*. However, roughly half of the VMs provide abstractions around *Threads and Locks*. Three provide abstractions for *Communicating Threads*, and five provide abstractions for *Communicating Isolates*. Only three of the VMs provide abstractions for two of these categorizes. Therefore, most of the contemporary VMs concentrate on a single model for concurrent programming.

To conclude, it is uncommon to support multiple concurrent or parallel programming concepts. This is not surprising, since most VMs are built for a single language. Furthermore, support for parallel programming is relegated to libraries, and none of the examined VMs provides direct support for it.

Seeing the current state of the art and the design choice of supporting only a small number of specific concepts in VMs, it remains open how multiple concepts can be supported on a modern multi-language VM.

3.2. A Survey of Parallel and Concurrent Programming Concepts

After investigating the state of the art in VM support for concurrent and parallel programming, it can be concluded that the vision of this dissertation has not yet been realized. Thus, there is currently no unifying substrate that enables language and library developers to implement concurrent and parallel programming concepts on top of multi-language VMs. However, the field of these programming concepts is wide and it is not yet clear what exactly a VM needs to support. To discuss this question in its full generality, this section reports on a survey that covers as many of the field's programming concepts as possible to reach a more complete understanding of which concepts benefit from VM support. Specifically, this survey identifies concepts that significantly benefit from VM support. To this end, the survey divides the concepts of the field into the concepts that require VM support to guarantee their semantics, and the concepts that can achieve significantly improved performance using VM support.

The main result of this survey is a set of general requirements for the two groups of concepts. The concepts that benefit from improved performance require support for dynamic optimization and runtime monitoring that captures specific execution characteristics to enable adaptive optimization. The concepts that require VM support to guarantee their semantics benefit most from mechanisms to specify custom method execution and state access policies. Since the requirements for the two sets of concepts are independent

of each other and require significantly different research, this dissertation chooses to focus on guaranteeing the semantics, i. e., the correctness of implementations for concurrent programming concepts on top of multi-language VMs.

3.2.1. Survey Design

The goal of this survey is to *identify concepts* that are relevant for a multi-language VM in order to facilitate the implementation of concurrent and parallel programming abstractions. To that end, this section first discusses the selected questions that categorize the concepts. Then, it details the approach to identify parallel and concurrent programming concepts and finally, it presents the findings and concludes with general requirements for the support of these concepts.

3.2.1.1. Survey Questions

When features are considered for inclusion in a VM, one of the main goals is to avoid unnecessary complexity (cf. Sec. 2.1). From this it follows that a new concurrent or parallel programming concept needs to be added to a VM only if it cannot reasonably be implemented in terms of a library on top of the VM. Thus, our first question is:

Lib Can this concept be implemented in terms of a library?

Interpreting the question very broadly, it considers whether some variation of the concept can be implemented, i. e., a variation for instance with missing semantic guarantees such as isolation. Thus, such a library implementation can either suffer from losing semantic guarantees, or it has to accept performance drawbacks. The following two questions account for this variation:

Sem Does this concept require runtime support to guarantee its semantics?

Perf Are there indications that runtime support would result in significant performance improvements compared to a pure library solution?

The answers to Sem also consider interactions of different languages on top of a VM as well as the influence of reflection. This is relevant since language guarantees are often enforced by a compiler and do not carry over to the level of the VM. One example is the semantics of single-assignment variables, which is typically not transferred to the bytecode level of a VM.

The answers to PERF consider intuitive optimizations that become possible with knowledge of full language semantics. For instance, knowledge about immutability enables constant folding, and taking the semantics of critical sections into account enables optimizations like lock elision [Rajwar and Goodman, 2001]. Furthermore, the answers rely on the literature of proposed implementation strategies that require changes to a VM.

The last categorization criterion is whether the concept is already common to VMs that are used as multi-language platforms and should be regarded as prior art (PA). When it is already available to the major multi-language VMs identified in Sec. 3.1.1.2, i. e., the JVM or CLI, general VM support for the concept is considered to be feasible and well understood, and therefore, does not need to be included in further discussion. Only JVM and CLI are considered here, because these two are specifically designed as multi-language VMs.

PA Is the concept already supported by a VM like the JVM or CLI?

3.2.1.2. Selecting Subjects and Identifying Concepts

The survey of Briot et al. [1998] of concurrent and distributed systems as well as the survey of Skillicorn and Talia [1998] of parallel models and languages are the main subjects to identify concurrent and parallel programming concepts. They provide a broad foundation and an overview over a wide range of concurrent and parallel programming concepts. Since concurrent and parallel programming have been investigated for years in the field of logic programming as well, two surveys for parallel logic programming [De Bosschere, 1997; Gupta et al., 2001] complement the surveys from the imperative world. However, since all four surveys are dated and lack coverage of recent work, a number of languages used in research or industry and selected research papers from recent years are included as subjects to cover current trends. The full list of *subjects* is given in Tab. 3.3.

This survey *identifies* for each of these subject the basic concurrent and parallel programming concepts it introduces, i. e., the concepts that are presented by the paper or provided by the language. For languages, this includes concepts from the language-level as well as the implementation-level. Note that the identified concepts abstract necessarily from specific details that vary between the different subjects. Thus, this analysis does not regard minor variations of a concept separately. However, this leaves room for different interpretations of the survey questions. Furthermore, the analysis of subjects such as C/C++ and Java considers only the core language and standard libraries.

Table 3.3.: Survey Subjects: Languages and Papers

Active Objects [Lavender and Schmidt, 1996]	JCSP [Welch et al., 2007]
Ada	Java 7 [Gosling et al., 2012]
Aida [Lublinerman et al., 2011]	Java Views [Demsky and Lam, 2010]
Alice	Join Java [Itzstein and Jasiunas, 2003]
AmbientTalk [Van Cutsem et al., 2007]	Linda [Gelernter, 1985]
Ateji PX	MPI [Message Passing Interface Forum, 2009]
Axum	MapReduce [Lämmel, 2008]
Briot et al. [1998]	MultiLisp [Halstead, Jr., 1985]
C#	Occam-pi [Welch and Barnes, 2005]
C/C++11 [ISO, 2011, 2012]	OpenCL
Chapel	OpenMP [OpenMP Architecture Review Board, 2011]
Charm++	Orleans [Bykov et al., 2011]
Cilk [Blumofe et al., 1995]	Oz [Mehl, 1999]
Clojure	Parallel Actor Monitors [Scholliers et al., 2010]
CoBoxes [Schäfer and Poetzsch-Heffter, 2010]	Parallel Prolog [De Bosschere, 1997; Gupta et al., 2001]
Concurrent Haskell	Reactive Objects [Nordlander et al., 2002]
Concurrent ML [Reppy et al., 2009]	SCOOP [Morandi et al., 2008]
Concurrent Objects [Herlihy, 1993]	STM [Shavit and Touitou, 1995a]
Concurrent Pascal	Skillicorn and Talia [1998]
Concurrent Smalltalk [Yokote, 1990]	Sly [Ungar and Adams, 2010]
Erlang [Armstrong, 2007]	StreamIt [Thies et al., 2002]
Fortran 2008	Swing
Fortress	UPC [UPC Consortium, 2005]
Go	X10 [Charles et al., 2005]
Io	XC

Thus, common libraries and extensions for these languages are considered as separate subjects.

3.2.2. Results

The analysis of the subjects given in Tab. 3.3 resulted in 97 identified concepts. Since most of them are accepted concepts in the literature and major ones have been covered in Sec. 2.4, this section restricts their discussion to the results of the survey questions. As mentioned earlier, some concept variations have been considered together as a single concept. For example, the distinct concepts of monitors and semaphores, have been regarded as part of *locks* in this survey. Similarly, the concept of *parallel bulk operations* also covers the concept of *parallel loops* for the purpose of this discussion, because both are similar enough and have closely related implementation strategies. With these subsumptions, Tab. 3.4 needs to cover only the 66 major concepts and their respective survey results. See Tab. 3.5 for the details about which concepts have been considered together.

Table 3.4.: Identified concepts classified. PA: prior art, Lib: implemented as library, Sem: support for semantics required, Perf: support for performance

Prior Art	PA	Lib	Sem	Perf	*Prior Art*	PA	Lib	Sem	Perf
Asynchronous Operations	X	X		X	Join	X			
Atomic Primitives	X			X	Locks	X	X		X
Co-routines	X			X	Memory Model	X		X	X
Condition Variables	X	X		X	Method Invocation	X			X
Critical Sections	X	X		X	Race-And-Repair	X	X		
Fences	X		X		Thread Pools	X	X		
Global Address Spaces	X	X		X	Thread-local Variables	X	X		X
Global Interpreter Lock	X		X		Threads	X	X		
Green Threads	X				Volatiles	X		X	
Immutability	X		X	X	Wrapper Objects	X	X		X

Library Solutions	PA	Lib	Sem	Perf	*Library Solutions*	PA	Lib	Sem	Perf
Agents		X			Guards		X		
Atoms		X			MVars		X		
Concurrent Objects		X			Message Queue		X		
Event-Loop		X			Parallel Bulk Operations		X		
Events		X			Reducers		X		
Far-References		X			Single Blocks		X		
Futures		X			State Reconciliation		X		

Potential Perf. Benefits	PA	Lib	Sem	Perf	*Potential Perf. Benefits*	PA	Lib	Sem	Perf
APGAS		X		X	Implicit Parallelism		X		X
Barriers		X		X	Locality				X
Clocks		X		X	Mirrors		X		X
Data Movement				X	One-sided Communication		X		X
Data-Flow Graphs		X		X	Ownership				X
Data-Flow Variables		X		X	PGAS		X		X
Fork/Join		X		X	Vector Operations		X		X

Semantics req. Support	PA	Lib	Sem	Perf	*Semantics req. Support*	PA	Lib	Sem	Perf
Active Objects		X	X		Message sends		X	X	X
Actors		X	X	X	No-Intercession		X	X	X
Asynchronous Invocation		X	X	X	Persistent Data Structures		X	X	
Axum-Domains		X	X		Replication		X	X	
By-Value		X	X	X	Side-Effect Free			X	X
Channels		X	X	X	Speculative Execution			X	X
Data Streams		X	X	X	Transactions		X	X	X
Isolation		X	X	X	Tuple Spaces		X	X	
Map/Reduce		X	X		Vats		X	X	X

Table 3.5.: Subsumed concepts: These concepts are regarded together.

Main Concept	Subsumed Concepts
Atomic Primitives	atomic swap, compare-and-swap
By-Value	isolates
Fences	memory barriers
Futures	ivars, promises
Global Address Spaces	shared memory
Immutability	single assignment variables
Isolation	encapsulation, processes
Locks	monitors, reader-writer-locks, semaphore, synchronized methods
No-Intercession	no-reflection
Parallel Bulk Operations	data parallelism, parallel blocks, parallel loops, parallel prefix scans
Speculative Execution	speculative parallelism
Transactions	atomic operations
Volatiles	volatile fields, volatile variables

As Tab. 3.4 shows, with 34 concepts about half of the concepts are already available in JVM and CLI or can be implemented in terms of a library without sacrificing semantics or performance. Therefore, this section discusses only the remaining 32 concepts. It starts with detailing the assessment for the 14 concepts for which only potential performance benefits have been identified.

This discussion is completed by detailing the 18 concepts that require runtime support to enforce their semantics properly. This analysis briefly describes what exactly the VM needs to enforce and if applicable, how VM support could improve performance.

Improved Performance without Semantic Impact The concepts listed in Tab. 3.4 under *Potential Performance Benefits* could benefit from VM support in terms of performance. However, they do not require an enforcement of semantic guarantees by the runtime.

APGAS, One-sided Communication, PGAS The PGAS (cf. Sec. 2.4.3) and APGAS (cf. Sec. 2.4.4) languages make the distinction between local and remote operations or memory explicit. Thereby, they give the programmer a better intuition on the cost of an operation. However, this leads to engineering tradeoffs between defining general algorithms and optimizing for the local case [Barik et al., 2011].

Proposed compiler optimizations can be applied to reduce the need for communication and improve performance [Barik et al., 2011; El-Ghazawi et al.,

2006; Serres et al., 2011; Zhao and Wang, 2009]. However, such static compiler optimizations cannot take dynamic properties into account. Thus, additional runtime information could lead to further optimizations. Optimizations to reduce and optimize communication adaptively require information on communication patterns, for instance to batch and interleave them. Furthermore, since such optimizations could likely benefit from language-specific details, the VM would need to expose JIT compiler infrastructure to enable an interaction between language implementation and runtime.

Barriers, Clocks Barriers, X10's clocks, and phasers [Shirako et al., 2008] have traditionally been implemented as libraries, since they can be efficiently realized based on atomic operations only. One example is our design of an efficient and scalable phaser implementation [Marr et al., 2010b]. However, since the use of barriers varies widely, static analysis can select more efficient implementations based on the actual use in a program [Vasudevan et al., 2009].

Additionally, runtime information could complement static analysis and reduce the overall need for synchronization even further. Following the idea of phasers, a conservative barrier could be weakened dynamically to avoid over synchronization. Such an optimization would require information on data access and access patterns in high-level data structures. Furthermore, it would need to interact with the JIT compiler to adapt the generated code.

Data Movement, Locality Since multicore and manycore architectures have increasingly *Non-Uniform Memory Access* (NUMA) characteristics, data locality becomes a significant performance concern. While the mentioned PGAS and APGAS programming models tackle that problem in an explicit manner by exposing distribution on the language level like in X10 or Chapel, current NUMA support in VMs is often restricted to allocating objects in memory that is local to the core which performed the allocation. Here feedback-driven approaches to dynamic optimization could improve the locality by moving data or computation based on actual usage patterns. Such optimizations could be realized based on low-level hardware features taking the processor's cache architecture into account. Thus, they need precise information about concrete hardware properties, GC behavior, and heap layout. For instance, the approach proposed by Zhou and Demsky [2012] monitors memory accesses at the page level and changes caching policies and locality accordingly. Such

an optimization would be at the VM level and does not require interaction with the high-level language on top of the VM.

Data-Flow Graphs, Data-Flow Variables, Implicit Parallelism Optimizations for data parallel programs are often represented as data-flow graphs that enable more powerful static analyses than control-flow graphs (cf. Sec. 2.4.5). However, this representation is usually only an intermediate form that is used to compile to traditional imperative native code representations. In that step, parallelism is often coarsened up in terms of a sequential cut-off to reduce its overhead and make it practical on today's hardware architectures. These coarsening techniques are relevant for programs in data-flow representation, programs based on data-flow variables, or implicit parallelism. Furthermore, they could also be beneficial to models like fork/join.

Such optimizations typically require detailed knowledge about the semantics of the compiled language. Furthermore, they could benefit from profiling information during runtime to assess the cost of operations more precisely. Thus, they would benefit from access to the corresponding dynamic profiling information and the JIT compiler infrastructure.

Fork/Join While fork/join could benefit from classic compiler optimizations such as coarsening, Kumar et al. [2012] report that one of the most significant costs is the reification of tasks. Thus, they propose to integrate the work-stealing that is used for fork/join into the runtime and reify task objects only lazily. With such an optimization, the sequential overhead of fork/join would be significantly reduced. On the other hand, the runtime would need to provide facilities to inspect and modify the call stack during execution.

To support multiple different languages, with potentially different variations of fork/join, an extended VM interface to realize such optimizations to tailor the implementation to the specific language semantics would be beneficial.

Mirrors While mirrors [Bracha and Ungar, 2004] are not directly related to concurrency and parallelism, languages like AmbientTalk integrate them tightly with the concurrency model. Furthermore, their use can enable the enforcement of certain language guarantees at a library level for systems that use a capability-based security model. Problematic is the performance overhead of reflective operations (cf. Sec. 8.5.1). Approaches like INVOKEDY-

NAMIC [Thalinger and Rose, 2010] or partial evaluation [Shali and Cook, 2011] can reduce that overhead by enabling optimization of reflective calls.

Ownership The notion of object ownership is used in a number of concepts such as Clojure agents, actors, and CSP to define policies that distinguish between entities that access an object. The implementation of ownership in a library would require an extension of every object to keep track of its owner. Runtime support in the memory management system could optimize the memory overhead by keeping track of the owner implicitly for instance by splitting the heap in regions that can belong to different owners. Similar optimizations have been proposed to keep track of meta data for garbage collection, for instance whether objects contain pointers.[Jones et al., 2011]

Vector Operations VM support for vector operations enables efficient data-parallel processing of homogenous data vectors. However, they are not yet widely available in VM instruction sets, even though some argue that it would provide performance benefits [Parri et al., 2011].

Conclusion From our perspective, the discussed concepts can benefit from a wide range of different optimizations. The most notable commonality is that they could benefit from VM infrastructure that provides feedback for adaptive optimizations. This includes information on high-level communication behavior, order of data accesses to high-level data structures, information about low-level data access frequencies, and access to execution profiling data. Furthermore, it would be beneficial to have a VM interface to inspect and manipulate the runtime stack, control heap layout, and the location of objects.

Most of these optimizations would rely on a common access to the JIT compiler and could benefit from solutions such as Graal [Würthinger, 2011]. However, it seems likely that they would also require a significant amount of infrastructure to be added to a VM that is specific to a single programming concept.

To conclude, these concepts can benefit from advanced infrastructure for dynamic optimizations and the corresponding facilities for runtime feedback and monitoring.

Enforcement of Semantics The following concepts require VM support for enforcing correct semantics. Some of then could also show from improved

performance when they are supported by the VM. These concepts are listed in Tab. 3.4 under *Semantics require Support*.

Asynchronous Invocation, Active Objects *Active objects* and other concepts that rely on *asynchronous invocation* are typically designed to disallow any synchronous invocation in order to have control over the coordination between concurrent activations. Some actor systems for object-oriented languages combine the notion of method invocation and message sends while assuming asynchronous semantics to avoid low-level deadlocks. While these semantics can often be enforced by relying on proxies, proxies themselves introduce for instance the identity problem. Furthermore, the desired semantics are in general not protected against reflective operations.

Actors, Axum-Domains, Isolation, Vats The concepts of *actors*, *vats*, *Axum's domains*, and *isolation* (cf. Sec. 2.4.4) are in general based on proper state encapsulation. State encapsulation is often guaranteed by construction. Providing state encapsulation on top of a system that only has the notion of shared memory and protecting it against reflection and other languages cooperating on the same system becomes difficult. Approaches that wrap references exchanged via interfaces are possible but have performance drawbacks. Furthermore, they do not provide a general guarantee when reflective capabilities are available. Sec. 3.3.2 discusses the related problems in more detail.

By-Value, Channels, Message Sends The notion of state encapsulation for actors, CSP, and others also requires proper messaging semantics. Thus, messages send between different entities need to have value semantics to guarantee isolation between these entities. Independent of whether they communicate via *channels* or other forms of *message sends*, the *by-value* needs to be enforced. For performance reasons, copying is to be avoided. However, this requires different entities not to have pointers to the same mutable state at the same time, or that they cannot observe and mutate state concurrently. The mechanisms for reflective programming offered by VMs are often able to subvert such guarantees.

Sec. 3.3.2 discusses the problems related to safe messaging in more detail.

Data Streams, Immutability, Map/Reduce, No-Intercession, Persistent Data Structures, Replication, Side-effect Freedom, Tuple Spaces Notions such as *immutability* or the guarantee of *side-effect free* computation are relevant for

a wide range of concepts such as *persistent data structures*, *tuple spaces*, the data used in *data streams* and for *map/reduce*. As an example, a concept that provides the notion of consistent data *replication* needs to be able to track all modifications in order to propagate them correctly. The main obstacles to these guarantees are typically reflective capabilities of a VM that do not provide any powerful means to disable *intercession* or to scope or restrict them in some way (cf. Sec. 3.3.5).

Speculative Execution, Transactions Similarly to the required ability to restrict intercession is the need to integrate some of the programming concepts deeply into the system. *Transaction* systems often rely on the ability to track all loads and stores to memory locations, which is often problematic in the presence of primitive operations. These have to be adapted to provide the necessary support for such a software transactional memory system. Some programming concepts for implicit parallelism require similar capabilities to provide *speculative execution* [Hennessy and Patterson, 2007; Herzeel and Costanza, 2010], enabling them to cancel the side-effects of branches that are not taken. Here, a full integration with the underlying system can become unavoidable to enforce the language semantics and causality even in the presence of speculative execution.

Conclusions Following from our discussion, the concepts that require enforcement of semantics vary in only a small number of points, but with these variations they achieve different sets of language guarantees. One of the aspects is the semantics of executing code or invoking methods. A language can for instance require asynchronous execution or consider other constraints to allow execution. Another aspect is the accessibility and mutability of state. The different concepts have a wide range of rules. For some concepts the owner has the exclusive right to read or mutate state, and other concepts require full tracking of all state accesses. In addition, all these basic aspects need to be properly integrated with concepts such as reflection to achieve the desired semantics.

To conclude, these concepts benefit from a VM interface that enables customizable semantics for execution and state access as well as customizable enforcement against reflection.

3.2.3. Threats to Validity

For the validity of the conclusions drawn from this survey, two of its properties are of major interest: *completeness* and *correctness*. The *completeness* of this study cannot be guaranteed since it did not cover all existing literature in the field. Even if it would have covered all known literature, it would still suffer from the completeness problem since an analysis requires manual work in which certain concepts could be overlooked. However, since this study is based on existing surveys and is complemented with programming languages and a wide range of recent work, it covers a significant range of concepts and includes all major trends.

The *correctness* of this survey might be reduced by the subsumption of concepts, because relevant details may have been ignored that could have yielded additional problems that need to be tackled by a VM. However, since the discarded details are most likely specific to a single concept, they would not have yielded problems that are as general as the ones already discussed in this survey. Another aspect of the correctness of the results is their consistency with the discussion in Sec. 3.1. To prevent such consistency issues, the data for the survey is recorded in a machine readable format. Appendix A.2 details the used templates for the survey and gives the full list of concepts identified for each subject. By using a machine readable notation, it was possible to check the consistency between the two surveys automatically. For instance, the automation includes a cross-check of the assessment of the different surveys on whether a concept is supported by a VM, and whether a library implementation is possible. Thus, consistency between concepts and their implementation and implementability is given between the two surveys.

3.2.4. Summary

This survey discusses a wide range of concepts for concurrent and parallel programming. For each concept, it answers the question of whether the concept's semantics benefit from an enforcement in the VM and whether its performance could benefit significantly from VM support.

A flexible optimization infrastructure and monitoring facilities can be beneficial for performance. The first conclusion is that the various concepts would benefit from a wide range of different optimizations. They all could benefit from access to the JIT compiler infrastructure to improve optimization. However, to use it properly they would require additional mechanisms

in the VM to gather a wide range of runtime information for adaptive optimizations, e. g., high-level communication behavior and low-level data access frequency information. Further extensions, for instance to inspect and manipulate the runtime stack or the heap layout, would be useful as well. Thus a VM needs to provide advanced infrastructure for dynamic optimization as well as facilities for customizable runtime feedback and monitoring.

Policies for execution and state access require support from the VM to be enforced and need to be integrated with reflection. The second conclusion is that concepts that require VM support to guarantee their semantics show a number of commonalities. These concepts vary in only a small number of aspects, but with these variations they achieve different sets of language guarantees. The first aspect is the semantics of executing code or invoking methods, and the second aspect is the accessibility and mutability of state. An additional common problem is that these basic aspects need to be properly integrated with concepts such as reflection to achieve the desired semantics.

Improving performance and improving the support for language semantics are two distinct research projects. To improve execution performance, an investigation of dynamic compilation, adaptive optimization, and runtime monitoring techniques is necessary. The main focus is on finding efficient techniques to gather and utilize a wide range of different runtime information to adapt execution dynamically. Such research is expected to yield improvement for non-functional system requirements.

In contrast, improved support for language semantics requires an investigation of correctness issues and techniques for changing language behavior. While performance is an issue here as well, the focus is on providing more flexibility to language developers to adapt execution semantics as necessary. Thus, this research is expected to yield improvements for functional requirements. While performance is a relevant aspect for this research question as well, it is not in the focus. To conclude, the two research questions are distinct. Therefore, from our perspective, they should be tackled by two distinct research projects in order to provide the projects with a clear scope and research goal.

This dissertation concentrates on improving support for language semantics in VMs as the first step in order to achieve a full understanding of the corresponding problems and to treat the question comprehensively. The main reason is to concentrate on improving functional concerns first and therefore

defer research on the non-functional concerns to future work. The improvements in the area of language semantics promise to alleviate some of the difficulties that come with concurrent and parallel programming today. Supporting language implementers by simplifying the implementation of language guarantees can lead to languages that in turn support the application developer by providing desirable correctness and engineering properties, which is the main vision behind this dissertation.

However, performance improvements are an important topic as well and will be pursued as part of future work (cf. Sec. 9.5.1).

3.2.4.1. General Requirements

Derived from the observations of this survey, the requirements for a multi-language VM that supports a wide range of abstractions for concurrent and parallel programming are:

Flexible Optimization Infrastructure Concepts such as *barriers*, *data-flow execution*, *mirrors*, and *vector operations* would benefit from access to the JIT compiler infrastructure to optimize the generated code either based on static analysis or based on runtime feedback.

Flexible Runtime Monitoring Facilities To facilitate adaptive optimization of languages that use concepts such as *PGAS*, *barriers*, *locality*, *data-flow Graphs*, a VM would need to provide an infrastructure that enables efficient and flexible monitoring of a wide range of aspects. It would need to record information on low-level data access, high-level data structure usage, high-level communication patterns, and other information specific to a given concept.

Powerful VM Interface Related to the flexible optimization infrastructure is the desire for a powerful VM interface that enables customization of various aspects. Examples could be customizability of heap arrangements to support *ownership* or full support to *intercept primitives* for accurate tracking of memory load and store operations, and influence data *locality*. Another objective is complete access to the runtime stack to support efficient work-stealing for *fork/join*.

Custom Execution and State Access Policies The wide range of variations for method execution and state access policies suggests the need for variable semantic guarantees. A language and library implementer needs to be able to modify and adjust language semantics for a particular

purpose, for instance in the case of of domain-specific languages. The guarantees that at least need to be supported are *asynchronous invocation*, *isolation* in terms of state encapsulation and safe messaging, *scheduling policies*, and *immutability*. Thus, a VM needs to provide a mechanism for custom semantics of code execution and state access.

Semantic Enforcement against Reflection As mentioned in the previous discussions, these guarantees need to be enforceable also in the presence of reflection. Thus, a form of scoping or a way to differentiate language guarantees is required (cf. Sec. 3.3.5).

3.2.4.2. Connection with Concurrent and Parallel Programming

This survey indicates a strong correlation between the concepts that require semantic enforcement on the part of a VM and the category of concepts that fit the definition of *concurrent programming* (cf. Sec. 2.3). The definition says that concurrent programming is about devising a strategy to coordinate independent activities at runtime that access a shared resource in order to preserve the resource's invariants. Based on this definition, asynchronous invocation, active objects, actors, Axum-domains, isolation, vats, by-value, channels, message sends, immutability, persistent data structures, tuple spaces, and transactions can be categorized as being concepts that are inherently meant to facilitate concurrent programming.

On the other hand, parallel programming is defined to be the art of devising a strategy to coordinate collaborating activities in order to compute a common result by employing multiple computational resources. With this definition, PGAS, one-sided communication, barriers, data movement, data-flow graphs, implicit parallelism, fork/join, and vector operations can be categorized as concepts for parallel programming. None of these concepts is meant to coordinate arbitrary activities as the concepts for concurrent programming do. Instead, these concepts either make data dependencies explicit, or try to make abstractions of them, while facilitating parallel execution.

This distinction seems to be a useful one, because it enables a categorization of the wide field of concepts in two disjoint sets. While the implementation of concepts for parallel programming might require concepts for concurrent programming, on an abstract level they can be discussed as two independent groups.

The main insight of this survey is that the two identified sets of concepts require orthogonal support from a VM. Thus, this dissertation focuses on one

of them. As argued above in Sec. 3.2.4, this dissertation focuses on concepts that require support for an enforcement of their semantics on the part of the VM. Based on the distinction provided by the two definitions, this dissertation focuses on the set of concurrent programming concepts, i. e., the concepts that are meant to support algorithms to yield consistent and correct results.

3.2.4.3. Conclusions

This survey discussed a wide range of concepts for concurrent and parallel programming. It categorizes the concepts identified in literature and programming languages based on answering the question of whether their semantics benefit from an enforcement in the VM and the question of whether their performance could significantly benefit from VM support.

Parallel programming concepts benefit most from flexible infrastructure for performance optimizations, while concurrent programming concepts require support from the VM in order to enforce their semantics.

The remainder of this dissertation focuses on concurrent programming concepts, i. e., concepts that require support from the VM to guarantee correct semantics, and proposes a unifying substrate for concurrent programming.

3.3. Common Problems for the Implementation of Concurrency Abstractions

The previous section concluded that a wide range of concurrent programming concepts require runtime support to enforce their semantics in an efficient way. This section discusses the semantic issues of these concepts in more detail. The analysis is based on the work of Karmani et al. [2009] who identified and discussed a number of problems in the context of actor-oriented frameworks, as well as examples from our own experiences in implementing concurrent languages [Marr et al., 2010a, 2011a,b, 2012] that are problematic for multi-language VMs. The in-depth investigation of these problems prepares, as the main result of this chapter, the formulation of requirements for multi-language VMs.

3.3.1. Overview

Karmani et al. surveyed actor-oriented frameworks for the JVM to assess which semantics these frameworks provide and what the performance cost

of the different actor properties are. The analysis in this dissertation concentrates on the issues that are relevant for multi-language VMs and disregards problems that are relevant for distributed systems only. The relevant actor properties for this discussion are insufficient guarantees with regard to *isolation* of state as well as with regard to *scheduling guarantees*. While Karmani et al. concentrated on actor-oriented frameworks, the analysis in this section shows that these actor properties are relevant for a wider range of concurrency abstractions.

In addition to these problems, the analysis in this section includes previously discussed issues related to *immutability* and *reflection* [Marr et al., 2011a]. Supporting reliable immutability and enabling reflection while maintaining concurrency-related language guarantees lead to complex implementations of concurrency abstractions. This section demonstrates with examples that language implementers face a number of significant problems when they try to realize such concepts on top of today's VMs. Therefore, these problems need to be considered in order to determine the requirements for multi-language VMs with extended support for concurrent programming.

3.3.2. Isolation

Isolation, often also referred to as *encapsulation*, is a valuable property that enables local reasoning about the effects of operations since it clearly separates two entities from each other and enforces the use of explicit interfaces for interaction. It is of high relevance, because many concurrent programming concepts rely on strong isolation between entities in order to restrict the set of directly mutable entities and thus, limit the number of potential data races in a concurrent system. Karmani et al. make a distinction between *state encapsulation* and *safe messaging* to clarify the underlying problems.

State Encapsulation Integrating the actor model correctly with languages that have mutable state, e. g., conventional object-oriented languages, is a challenge. Sharing mutable state between actors violates the actor model, because communication and interaction are supposed to be based on message-based communication only. Therefore, mutable state has to be *owned* by a single actor and any form of access needs to be restricted to its owner.

Semantic Aspects See Lst. 3.1 for the example Karmani et al. use to illustrate the problem. The depicted Scala code is supposed to implement a basic

```
1  object semaphore {
2      class SemaphoreActor() extends Actor {
3          // ...
4          def enter() {
5              if (num < MAX) {
6                  // critical section
7                  num = num + 1; } } }
8
9      def main(args : Array[String]) : Unit = {
10         var gate = new SemaphoreActor()
11         gate.start
12         gate ! enter    // executes on gate's thread
13         gate.enter      // executes on the main thread
14 } }
```

Listing 3.1: Example of incomplete State Encapsulation: This semaphore has a race condition since Scala's actors do not enforce encapsulation and the actor as well as the main thread have access to the `num` field. [Karmani et al., 2009, Fig. 2]

semaphore. The `SemaphoreActor` has a counter `num`, which indicates the number of activities that entered the critical section.

However, Scala's actor implementation does not guarantee encapsulation. This makes the `gate` object simultaneously accessible in the main thread and the thread of `gate`, i. e., the `SemaphoreActor`. Both threads can execute `enter` at the same time, leading to a race condition on the `num` variable, undermining one of the main benefits of the actor model. If encapsulation would be guaranteed, as it is for instance in Erlang, this example would work as intended, because only the owning actor could access `num` and the data race would not be possible.

While actors require state encapsulation to yield full engineering benefits, implementing this guarantee comes either at the cost of a high implementation complexity or a significant performance impact. The main reason is that VMs such as the JVM do not provide sufficient abstraction for the notion of ownership and state encapsulation. AmbientTalk [Van Cutsem et al., 2007], JCoBox [Schäfer and Poetzsch-Heffter, 2010], and NAct[18] enforce the discussed encapsulation by construction. The compiler ensures that only so-called *far references* can be obtained to objects such as the `SemaphoreActor`. These far references enforce state encapsulation to guarantee isolation.

Kilim [Srinivasan and Mycroft, 2008] is an example for another approach to the problem. Kilim employs compile-time checking based on annotations to

[18] http://code.google.com/p/n-act/

guarantee isolation. The compiler ensures that references are only passed on in messages when they are no longer used within the sending actor.

Problematic with both types of approaches is that they rely on the compiler and statically determinable properties. Moreover, these properties neither carry over to other languages on the same VM, nor do they apply to the standard libraries. Thus, they are only guaranteed for a specific language.

Interacting with legacy libraries, code written in other languages, or the use of reflection typically break these guarantees. On a multi-language platform, this situation cannot be avoided and seriously restricts the language and library implementers flexibility.

Applicability beyond Actors Other concurrent programming concepts than the actor model also rely on *state encapsulation*. It applies to all models that provide *communicating isolates* (cf. Sec. 2.4.4), i. e., non-shared-memory models most notably CSP and APGAS languages that require mutation to be performed locally to a process or the state-owning place, i. e., region. Other concepts such as *Clojure agents* (cf. Sec. 2.4.3), have different semantics, but also restrict the capability of mutating state to its owner.

Safe Messaging Karmani et al. further discuss the issue of messaging. Non-shared memory models require that message passing has *by-value* semantics. Otherwise, shared mutable state is introduced by passing normal references. An example would be similar to Lst. 3.1. By passing any mutable Java object as argument to a message send in Scala, the object becomes shared between the sender and the receiver actor and thereby introduces shared mutable state. Scala does not enforce safe messaging, and thus, it does not handle the content of a message to ensure semantics. Instead, objects are simply passed by reference.

Performance Aspects Traditional solutions enforce by-value semantics either by copying the object graph of a message or by relying on a type or annotation system that enables static checks at compile-time as in Kilim. Copying the whole object graph referenced by a message often implies significant overhead as measured by Karmani et al. They report that a microbenchmark for passing on messages has a maximal runtime of ca. 190sec when naive deep-copying is used. An optimization that does not copy immutable objects reduces the runtime to 30sec. However, this is still significantly slower than

when the benchmark uses pass-by-reference, which brings the runtime down to ca. 17sec.

Approaches based on type systems or other forms of static verification enable the safe passing of references, and thus, bring significant performance advantages. However, a type system comes with additional implementation complexity, its guarantees do not reach beyond language boundaries, and are typically voided by the use of reflection.

Again, these problems are universal and apply to *communicating isolate* concepts, e. g., CSP [Hoare, 1978] and APGAS [Saraswat et al., 2010] concepts as well (cf. Sec. 2.4.4). For instance, JCSP [Welch et al., 2007] does not guarantee safe messaging for reference types. The by-value semantics is only given for primitive types, and channels that are based on network socket communication. The main concerns are performance, similar to actor-oriented frameworks on the JVM and frameworks like Retlang[19] on top of the CLI. X10 [Charles et al., 2005] as one example for an APGAS language explicitly specifies that deep copying is performed [Saraswat et al., 2012]. Similarly, the *application domains* specified by the CLI rely on marshalling, i. e., either deep copying or far references, to enforce safe messaging and state encapsulation. Thus, the implementation of properties such as *safe messaging* can become a performance issue when the VM does not provide mechanisms to support it, which often leads to language and library implementers giving up on desirable semantic properties.

Conclusion Following this discussion, the implementation of isolation on today's multi-language VMs is challenging and thus, it often remains unenforced or only partially supported. Solutions for enforcing isolation imply tradeoffs between performance and implementation complexity. Furthermore, mechanisms to handle reflection and the interaction with other languages remain uncommon.

Note that the notion of ownership is central for the definition of isolation. Furthermore, a mechanism for ownership transfer between entities can simplify the implementation of safe messaging greatly.

3.3.3. Scheduling Guarantees

The second issue identified by Karmani et al. is that the overall progress guarantee of an actor systems assumes fair scheduling. In systems without fair scheduling, actors can be starved of computation time and block the overall

[19] http://code.google.com/p/retlang/

```
1  object fairness {
2      class FairActor() extends Actor {
3      // ...
4      def act() { loop {
5          react {
6              case (v : int) => { data = v }
7              case (wait)    => {
8                  // busy-waiting section
9                  if (data > 0) println(data)
10                 else self ! wait
11             }
12             case (start)   => {
13                 calc ! (add, 4, 5)
14                 self ! wait
15             }
16         }}}}}
```

Listing 3.2: Example for missing Scheduling Guarantees: Without the guarantee of fair scheduling, *busy-waiting* can starve other actors forever. [Karmani et al., 2009, Fig. 3]

system from making progress. The example they give is shown in Lst. 3.2. Here the actor busy-waits by sending itself a message to await the arrival of the computation result. While the example is contrived, livelocks like this can be effectively avoided by fair scheduling.

An ad hoc solution to these problems is the use of a lightweight *task* representation for actors, which is used to schedule these tasks on top of the threading mechanism provided by the VM to guarantee the desired fairness property. To this end, the Java standard library provides the `ExecutorServices`[20] and the CLI offers a `TaskScheduler`.[21]

Ad hoc solutions to enforce scheduling policies have limitations and introduce tradeoffs. Since these solutions build on top of the VM's thread abstraction, the provided guarantees are restricted. Computationally expensive operations and blocking primitives can bind the underlying thread. This restricts the ability of the scheduler to enforce the desired guarantees, because as Karmani et al. note, the actor or task scheduler is prevented from running. It is possible to compensate to a certain degree for such effects by

[20] http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutorService.html

[21] http://msdn.microsoft.com/en-us/library/system.threading.tasks.taskscheduler.aspx

introducing additional preemptive OS threads that schedule actors, i. e., tasks. Karmani et al. propose to use a *monitoring thread*. This thread will spawn additional threads if it does not observe progress. However, there are limits to the approach and it requires tradeoffs between application responsiveness, overhead, and monitoring precision. Furthermore, it can add complexity to the implementation since it prescribes how actors can be represented.

Applicability beyond Actors Again, the problem of scheduling guarantees is not specific to actor-based concepts. Instead, it has to be carefully considered for any concurrent system that implies any notion of overall forward progress. The absence of such required guarantees is for instance visible in Clojure. Its agent construct makes a clear distinction between normal operations, and operations that result in long running computations or blocking I/O. For the latter kind of operations, Clojure offers the `(send-off)` construct, which will use a new thread to execute the operation to avoid starvation of other agents.

Conclusion While ad hoc solutions used today have drawbacks, they provide the flexibility to implement custom policies that can be adapted precisely to the characteristics of a specific concurrent programming concept. The main problem with these approaches is the missing control over primitive execution and computationally expensive operations, which can prevent these ad hoc solutions from enforcing their policies.

3.3.4. Immutability

Immutability is a desirable guarantee to simplify reasoning over programs in a concurrent setting, and it facilitates techniques such as replication, or constant propagation. However, in today's VMs, immutability is guaranteed with limitations only. Often it is provided at the language-level only and not preserved at the VM level. For instance, the JVM and CLI allow a programmer to change `final` or `readonly` object fields via reflection. Nonetheless, it is used for optimizations, and the JVM specification includes a warning on the visibility of such reflective changes, because JIT compilers for the JVM perform constant propagation for such fields.

Weak immutability is a workaround for missing functionality. While immutability by itself is desirable, notions of weak immutability have been used

for other purposes than optimization and strong guarantees for programmers. One use case for such weak immutability is provided by VisualWorks 7 [Cincom Systems, Inc., 2002]. It allows marking objects as immutable and raises an exception when mutation is attempted. The exception handler is then free to mutate such an object. This behavior is used to map objects to persistent data storage and enable efficient consistency management. In this case the programmer does not consider the object to be immutable, instead this technique is a workaround for the missing capability of tracking mutation. However, the example also illustrates the relation between immutability and reflection. In sequential programs, such workarounds do not necessarily have a negative impact on the guarantees the programmer relies on. In a concurrent setting however, such workarounds can easily lead to unintended race conditions.

Benefits of Immutability For a concurrency-aware VM, immutability can be used for performance optimization, either by avoiding copying of immutable objects, or replicating them to improve locality of access. In either case, immutability semantics have to be guaranteed to avoid semantic problems. Using it as sketched in the above example of VisualWorks is not an option.

When immutability for a particular language is guaranteed by its compiler only, interactions across language boundaries become problematic, similarly to the use of reflection. JCoBox and Kilim use the knowledge of immutability to avoid copying objects in their implementation of safe messaging, but they rely on the absence of reflection for their semantics. Clojure's approach to the problem is different. Its language model of mostly side-effect free concurrency on immutable data structures creates engineering benefits when direct access to the underlying Java ecosystem is avoided. However, it is not enforced. On the contrary, Clojure's close integration with Java is an important aspect for its adoption.

Conclusion Immutability is a basic property that needs to be preserved, especially in concurrent systems. Using it for other purposes such as the tracking of mutation in VisualWorks 7 should be avoided and the necessary facilities for such use cases should be provided directly. The main problem with immutability in a concurrent context is its relation to reflection. While there are use cases such as deserialization that can require the use of reflection and setting of final fields, in the context of concurrency, immutability requires strict guarantees.

3.3.5. Reflection

Reflection, i. e., metaprogramming is used to circumvent the restrictions of a particular language or to modify and extend its language semantics. When reflection is used, language guarantees are no longer enforced and developers have to take care not to violate inherent assumptions made by the rest of the program. In sequential programs this is common practice and used widely for a variety of use cases.

Bypassing Language Restrictions Many common use cases bypass restrictions imposed by language semantics. Examples are the modification of supposedly immutable objects or bypassing restrictions to access protected fields. These capabilities are often desirable to enable the implementation of frameworks that work on any given object by dynamically reflecting over it. Widely used examples are unit testing frameworks reflecting over classes to execute tests, mockup generators to facilitate testing without the need to explicitly define test classes, object-relational mappers (ORM) to bridge between application and database for persistence, and other frameworks for general marshalling and serialization.

Note that the characteristics of these use cases are very similar. The main reflective features used include inspecting objects, invoking methods, or changing fields reflectively. While language restrictions such as modifiers for private and protected fields need to be circumvented, concurrency-related language properties should not be circumvented in these use cases. Instead, in most situations reflection should be able to respect these language guarantees.

Concurrency properties need to be maintained during reflection. Since the described use cases are ubiquitous, it is impractical to ban the use of reflection to guarantee the semantics of a particular concurrent programming concept. Instead, a VM needs to provide a mechanism that makes it safe to use reflection for application purposes while maintaining the desired part of the language guarantees, in this case the concurrency properties.

Imagine an application implemented in an actor language that uses a reflective object-relational mapping (ORM) system such as Hibernate [Bauer and King, 2005], which by itself is not actor-aware. One actor tries to persist an object owned by another actor. Hibernate would use reflection to read the state. Since the Java reflection API[22] does not preserve concurrency-

[22] http://docs.oracle.com/javase/6/docs/api/java/lang/reflect/package-summary.html

related properties, Hibernate can observe inconsistencies because of race conditions caused by the other actor executing in parallel. In the case of Java, the `java.util.concurrent.atomic`[23] package introduced an API for updating fields specifically to maintain concurrency properties such as those given by volatile fields. However, this requires mixing different APIs and is only restricted to field semantics.

Other language guarantees that need to be maintained are for instance constraints on method invocations. For instance, active objects require that their methods are not executed directly but via an indirection that ensures that the method is executed asynchronously by the active object's thread. With Join Java [Itzstein and Jasiunas, 2003], execution constraints can be more complex. It allows the developer to define join patterns that need to be fulfilled before a method is activated. Using reflection to circumvent these constraints can have undesired effects. Thus, depending on the use case, even reflective invocation should be able to preserve the constraints specified by such methods.

Another complication comes from the vision of a multi-language VM. In this scenario it is insufficient to make reflection aware of a single set of desired guarantees. Instead, additional flexibility is required. Language semantics such as isolation for an actor language are to be implemented on top of the VM, based on a unifying abstraction. A multi-language VM will need to be able to distinguish between situations where an enforcement of guarantees is required and situations where enforcement is not desired. Kiczales et al. [1997] and Tanter [2009] discuss how to scope reflection. Similarly, in the context of multi-language VMs, it also needs to be possible to scope, i. e., restrict reflection to circumvent a small set of language guarantees only, while maintaining others.

Reflection and Security The power of reflection is also an issue for security. Therefore, the JVM and CLI provide infrastructure to manage reflection in a way that allows them to restrict the reflective capabilities depending on a security context. For the JVM the `SecurityManager`[24] is consulted for reflective operations to verify that the necessary permissions are given. The CLI provides similar mechanisms with its `System.Security` facilities.[25]

The `SecurityManager` provides the flexibility to customize the check that is performed for reflective operations. However, the offered interface is very

[23] http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/atomic/package-summary.html
[24] http://docs.oracle.com/javase/7/docs/api/java/lang/SecurityManager.html
[25] http://msdn.microsoft.com/en-us/library/stfy7tfc.aspx

minimal and does not provide the flexibility required for multi-language VMs. For instance, the JVM relies on the problematic method `setAccessible(bool)`, which needs to be called to disable the security checks to circumvent language restrictions. While `setAccessible(bool)` is covered by the `SecurityManager` and its handler can be customized with another security strategy, the object can be manipulated at will once `obj.setAccessible(true)` succeeded.[26] Thus, enforcing complex policies is not possible with this approach. Another weak point is that the security infrastructure only considers fields that are protected by Java language semantics. Thus, public fields of an object are not covered by the security manager, which therefore cannot be used to express concurrency policies for all objects.

Conclusion Reflection in the form it is supported on today's VMs is not designed to enable the required fine-grained control. Common approaches provide only limited security-related mechanisms to restrict the reflective capabilities. However, the provided abstractions are not expressive enough to distinguish between different parts of a language. For instance, is not possible to freely use reflection on objects inside of an actor to circumvent their private field restrictions, without risking to have concurrency issues with objects that belong to another actor, because the reflection would also circumvent these concurrency restrictions.

Thus, reflection needs to provide the ability to do metaprogramming with the possibility to circumvent only certain language restrictions, while adhering to other parts of the language, for instance concurrency-related semantics.

3.3.6. Summary

As argued in this section, the implementation of proper *isolation*, *scheduling guarantees*, and *immutability* is problematic with today's VMs. For the implementation of isolation, language designers have to make a tradeoff between semantic guarantees, performance, and implementation complexity. To ensure progress in models such as actors or CSP, language designers have to be able to rely on scheduling guarantees which are typically implemented on top of the VM, and thus, handling of computationally expensive or blocking operations undermines the required guarantees.

Immutability also requires proper enforcement to yield its full engineering benefit, especially in the setting of a multi-language VM for concurrent

[26] http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/reflect/Field.html#set(java.lang.Object,java.lang.Object)

programming. The main issue with immutability for the implementation of languages on top of today's VMs is its interaction with reflection, and the missing enforcement of guarantees.

One issue with guaranteeing such semantics is the presence of reflection. Today's VMs support reflection in a way that circumvents all language guarantees, while it would be beneficial to restrict it to a certain subset of the language guarantees that it bypasses. This way, for instance concurrency-related guarantees could still be enforced. In the context of reflection, the results of the survey in Sec. 3.2, i. e., the need for *custom execution and state access policies* (cf. Sec. 3.2.4.1) becomes relevant. Different *execution policies* and *state access policies* can require for correctness that they are enforced for reflective operations as well. In this context, the notion of *ownership* becomes relevant again, because reflection can be allowed with its full capability inside an entity such as an actor, but might be restricted with respect to other entities.

The result of this discussion is given in Tab. 3.6. The table lists the identified challenges language and library implementers are facing today when they target multi-language VMs such as the JVM and CLI.

3.4. Requirements for a Unifying Substrate for Concurrent Programming

This section concludes the preceding discussions and derives requirements for a unifying substrate for concurrent programming. This substrate is meant to support the vision of a multi-language VM that enables language and library developers to provide problem-specific abstractions for the multicore age.

Sec. 3.2 discussed a wide range of concepts from the field of concurrent and parallel programming. This dissertation focuses on the concepts for concurrent programming in order to construct a unifying substrate for them. Sec. 3.2.4.1 identified a general set of requirements. From this set, the need for flexible optimization and monitoring facilities are outside the scope of this dissertation. Instead, requirements for a *powerful VM interface*, *custom semantics for execution and state access*, as well as *enforcement of semantics against reflection* are in its focus.

Sec. 3.3 detailed common problems for the implementation of concepts for concurrent programming. Isolation in the form of state encapsulation and safe messaging requires support to ensure semantics and performance. Scheduling guarantees typically lack enforceability, too. Immutability as a specific

Table 3.6.: Common Challenges for the Implementation of Concurrent Programming Concepts on top of Multi-language VMs.

Enforcement of	
Isolation	Guaranteeing isolation between entities requires *state encapsulation* and *safe message passing*, to ensure that only the *owner* of an object can access it. *State encapsulation* relies on restricting access, while *safe message passing* relies on *by-value* semantics or the transfer of ownership of objects between entities. The absence of support for these notions leads to implementation challenges and incomplete isolation, high implementation complexity, or low performance.
Scheduling Policies	Guaranteeing scheduling policies such as *fairness* or dependency ordering requires control over executed code. Primitives and computationally expensive operations reduce the degree of control and reduce the guarantees that can be given. Ad hoc solutions based on monitoring require tradeoffs between overhead and precision.
Immutability	Only guaranteed immutability provides semantic benefits in a concurrent setting. The ability to reflectively change immutable objects limits its guarantees and reduces its engineering benefit. Using it as a workaround for missing functionality is therefore counterproductive. Instead, reflection must be able to respect such concurrency-related properties.
Execution Policies	Guaranteeing a wide range of execution policies is challenging. Policies such as for asynchronous invocation and guarded execution can be circumvented by reflection even if it is not desired. Furthermore, their implementation can be challenging, because they typically require the notion of *ownership*, which is not supported by today's VMs.
State Access Policies	Guaranteeing a wide range of state access policies is equally challenging. On today's VMs, they are hard to enforce, because of the presence of primitives and reflection. The implementation of such policies can be facilitate by enabling adaptation of primitives, and by enabling a higher degree of flexibility for the use of reflection with regard to its effect on semantic guarantees.

concept relevant for concurrency abstractions is hardly enforceable on any of the surveyed platforms today. The main reason for weak enforceability is the ubiquitous availability and use of reflection. Since immutability and other concurrency properties are essential for correctness of code, reflection needs to be flexible and allow developers to circumvent only the necessary subset of language guarantees for a given use case.

In conclusion, a unifying substrate needs to support some form of MANAGED STATE and MANAGED EXECUTION. This is necessary to enable *custom semantics for execution and state access* and a *powerful VM interface*. This dissertation refers to MANAGED STATE as the notion of reifying access to state, i. e., reading and writing of object fields or global variables. MANAGED EXECUTION is the notion of reifying execution of methods and primitives.

A mechanism satisfying these two notions is supposed to enable at least the implementation of *asynchronous invocation*, *isolation*, *scheduling policies*, *immutability*, *ownership*, and *interception of primitives*.

The discussion of for instance active objects, CSP, and actors, showed that concurrency policies, i. e., for instance access restrictions can be based on the notion of *ownership* of objects. To support such concurrency policies, a unifying substrate needs to complement MANAGED STATE and MANAGED EXECUTION with OWNERSHIP. A mechanism satisfying this requirement enables the implementation of *isolation* and supports concepts such as Clojure actors which allow reads from arbitrary entities but restrict write access to the owning entity.

To enforce *scheduling policies*, this dissertation requires a VM to support some form of preemptive scheduling that will enable a higher prior thread to execute when necessary to ensure fairness. Blocking primitives remain problematic however, because they can block the scheduling logic from executing. Thus, primitives have to be manageable as we required it for execution in general. This aspect is covered by MANAGED EXECUTION.

The required enforcement of guarantees against reflection needs to be flexible. As explained with the example of an ORM system, reflection needs to be enabled to obey concurrency semantics (cf. Sec. 3.3.5), while the ORM still needs to be able to circumvent access restrictions like private modifiers for fields to fulfill its purpose. Thus, such an implementation needs to be able to specify whether an operation is supposed to be executed with enforcement enabled or disabled. To conclude, a VM needs to provide the notion of CONTROLLED ENFORCEMENT. A mechanism that satisfies this notion has to enable for instance reflection over private fields of an objects while its concurrency properties are maintained. In the context of an STM system, this would

Table 3.7.: Requirements for a Unifying Substrate for Concurrent Programming

Requirement	
Managed State	Many concepts impose rules for when and how state can be accessed and modified. Thus state access and mutation must be manageable in a flexible manner.
Needs to facilitate:	isolation, immutability, reified access to object field and globals.
Managed Execution	Similarly, the activation of methods on an object needs to be adaptable. This includes the activation of primitives to be able to handle their effects and scheduling properties.
Needs to facilitate:	asynchronous invocation, scheduling policies, interception of primitives.
Ownership	One recurring notion is that mutation and execution are regulated based on and relative to an owning entity. Thus, ownership of objects needs to be supported in a manner that enables adaptable state access and execution rules.
Needs to facilitate:	definition of policies based on ownership.
Controlled Enforcement	To be applied safely, reflection still needs to follow the concurrency-related language semantics for many use cases. Thus, whether language guarantees should be enforced needs to be controllable.
Needs to facilitate:	Flexible switching between enforced and unenforced execution.

mean that in certain situations reflective state changes need to be tracked in the same way that normal state changes are tracked.

Tab. 3.7 summarizes these requirements briefly.

3.5. Conclusions

VMs support only concurrent programming concepts directly. Parallel programming concepts are provided in libraries only. Sec. 3.1 investigated the state of the art in VM support for concurrent and parallel programming. The survey examined thirteen VMs, including the contemporary multi-language VMs, and a number of VMs that are reputed for their support for concurrent and parallel programming. The analysis identified for each VM the concepts it exposes and whether it exposes them in terms of *implicit semantics*, as part of the VM's *instruction set architecture*, in terms of *primitives*, or merely as part of standard libraries. The major insight is that the surveyed VMs support only one or two categories of concepts. Furthermore, they consistently relegate support for parallel programming to their standard library without providing explicit support for optimization.

Another observation is that common language guarantees such as *isolation* for actors or the constraints around the STM system of Haskell are realized as part of these VMs. Thus, the VM has full control over the exact set of provided guarantees. however, providing support for all conceivable concepts is not desirable for a multi-language VM. Problems with complexity and feature interaction make it infeasible, as argued in Sec. 2.1.

Parallel programming concepts benefit from performance optimizations, while concurrent programming concepts require semantic enforcement. Sec. 3.2 discussed a wide range of concepts for concurrent and parallel programming. Concepts are categorized by answering the question of whether their semantics benefit from an enforcement in the VM and the question of whether their performance could benefit significantly from VM support. The result is that parallel programming concepts benefit most from performance optimization, while concurrent programming concepts require support from the VM to enforce their semantics.

The concepts that require support from the VM to guarantee their semantics were chosen as the focus of this dissertation. Therefore, the remainder of this dissertation investigates a unifying substrate for concurrent programming.

Proper support for isolation, scheduling guarantees, immutability, and reflection are an issue with contemporary VMs. Sec. 3.3 analyses existing literature and additional examples that detail five common problems for the implementation of concurrent programming concepts on top of multi-language VMs. Proper implementation of *isolation*, *scheduling guarantees*, and *immutability* on top of contemporary VMs is challenging since performance, implementation complexity, and an enforcement of desirable semantics have to be traded off against each other. The way *reflection* is supported in today's VMs makes it hard to use in concurrent settings, because it circumvents all language guarantees instead of circumventing only the required ones.

Note that later chapters use these problems as part of the evaluation.

Requirements Sec. 3.4 concludes the discussion with a set of concrete requirements. These requirements are derived from the surveys and from the discussion of common implementation problems. The requirements have been chosen to facilitate the design of a framework in which a unifying substrate for concurrent programming concepts can be defined. The basic requirements for such a unifying substrate are support for the notions of MANAGED STATE, MANAGED EXECUTION, OWNERSHIP, and CONTROLLED ENFORCEMENT.

These requirements guide the design of the proposed solution and are the foundation for its evaluation in the remainder of this dissertation.

4

EXPERIMENTATION PLATFORM

The goal of this chapter is to introduce the chosen experimentation platform. Furthermore, for each VM this chapter gives the rationale for its choice. First, it sketches the requirements for a platform on which the research for this dissertation can be performed. Based on these requirements, SOM (Simple Object Machine), Squeak/Pharo Smalltalk, and the RoarVM are chosen. Second, it introduces SOM and gives an overview of Smalltalk and the syntax used in this dissertation. Third, the chapter briefly introduces Squeak, Pharo, and the CogVM as the platform for the initial implementation of the evaluation case studies. Finally, the RoarVM and a number of its implementation details are described to aid in later chapters.

4.1. Requirements for the Experimentation Platform

It is desirable to define a precise and minimal executable semantics to explain and document the ownership-based metaobject protocol (OMOP), which is the proposed solution of this dissertation (cf. Chapter 5). Furthermore, it is desirable to choose a medium for the definition of this executable semantics that is similar to the platforms used for the implementation and evaluation of the OMOP, because the similarity facilitates their explanation as well.

To demonstrate that the OMOP satisfies the identified requirements, this dissertation evaluates its applicability and assesses its performance. This evaluation needs to assess the impact of the OMOP by comparing implementations of concurrent programming abstractions with classic ad hoc implementations, including an assessment of the implementation size. Thus, it must be possible to implement concurrent programming concepts based on common implementation strategies in addition to implementing them based on the OMOP. For the assessment of the OMOP's performance it is necessary to evaluate an implementation based on direct VM support.

This dissertation uses three Smalltalk systems for these tasks: SOM (Simple Object Machine), Squeak/Pharo Smalltalk, and the RoarVM. The remainder of this chapter motivates their choice and gives a brief overview of them to lay the foundation for the technical discussions in later chapters. The discussion of SOM includes an introduction to the essential concepts of Smalltalk and the syntax used throughout this dissertation. Note that these general ideas, i. e., language and execution model, apply to the other Smalltalk dialects and VMs discussed here as well.

4.2. SOM: Simple Object Machine

SOM[1] (Simple Object Machine) is a Smalltalk variant meant for teaching language implementation and VM techniques. Therefore, it is kept as simple as possible and its implementation focuses on accessibility and clarity of concepts rather than execution performance. It has been implemented in Java (SOM), C (CSOM), C++ (SOM++), and Smalltalk (AweSOM). Following the Smalltalk tradition, SOM is an object-oriented language based on classes and supports the Smalltalk-80 notion of closures called blocks.

[1] Originally developed at the University of Århus, SOM is now maintained at the HPI: http://hpi.uni-potsdam.de/hirschfeld/projects/som/. It was also the foundation for *Resilient Smalltalk* and the OOVM [Andersen et al., 2005].

SOM's focus on clarity and a minimal set of concepts makes it a good candidate to express the execution semantics of the OMOP. Essential semantics can be expressed while the accidental complexity that comes with the extensive feature set of common mainstream languages is avoided. Compared to basing the semantics on a calculus such as the $imp\varsigma$-calculus of Abadi and Cardelli [1996], SOM provides a platform that is closer to common language implementations. It includes common language concepts and their essential complexity. For instance, this directly allows a discussion of the special case of VM primitives, which would not be the case with an application of the $imp\varsigma$-calculus, without adding specific extensions to it.

AweSOM[2] is chosen over the other available SOM implementations, because its Smalltalk implementation is the most concise in the SOM family and enables the formulation of an executable semantics that is concise enough to serve for illustration as part of this dissertation.

4.2.1. Language Overview and Smalltalk Specifics

This section gives a brief overview of SOM's Smalltalk syntax and Smalltalk concepts used in the source code examples of this dissertation.

SOM's standard implementation relies on a textual syntax for class definitions. Lst. 4.1 gives a minimal definition of the class `Object`. Note that `Object` inherits from `nil` and thus, it is the root class of the class hierarchy. If neither a superclass nor `nil` are given, a class will implicitly have `Object` as superclass.

```
1  Object = nil (  "defines the class Object, superclass is 'nil'"
2    | class |     "object fields: each object has a field 'class'"
3
4    class       = ( ^class )      "method to return field 'class'"
5    =  other    = ( ^self == other )
6    == other    = primitive  "equality test implemented in the VM"
7    asString    = ( ^'instance of ' + (self class) )
8    value       = ( ^self )
9    yourself    = ( ^self )
10   ifNil: nilBlock ifNotNil: goBlock = (^goBlock value)
11
12   "Error recovering"
13   doesNotUnderstand: selector arguments: arguments = (
14     self error: 'Method not found: '
15                 + class name + '>>#' + selector ) )
```

Listing 4.1: SOM Language Example: Object class extending nil

[2] https://github.com/smarr/SOM/#readme

Classes and Methods Object fields are defined with a syntax similar to local variables in methods of other Smalltalk versions. In this case, every object has a `class` field, referring to the object's class.

Note that SOM uses object-based encapsulation. Thus, only the object itself has access to its fields. This is different from class-based encapsulation as used for instance in Java [Gosling et al., 2012]. In languages with class-based encapsulation, all objects of a class can access the private fields of other objects of the same class.

Methods in Smalltalk have either no arguments, one argument when the method selector is a special symbol, or they use so-called keyword messages, which indicate the arguments with colons.

The first method in line 4 is a simple method without arguments. It is an accessor to the `class` field and directly returns the value. Note the circumflex (`^`), which corresponds to the return keyword in other languages. Further note that the body text of this dissertation refers to methods in source code examples by their selector symbol, i. e., the method name with a preceding hashmark (`#`). Thus, line 4 defines the method `#class`.

Line 5 defines a second method `#=`, which takes the argument `other` to implement object equality by using the reference equality message `#==`. Examples for binary messages are `#=` and `#==`, which take an extra argument in addition to the receiver. In this particular case `#==` refers to its argument as `other`. Line 6 defines `#==` to be implemented via a VM primitive, which checks the receiver and the argument for reference equality. The `#ifNil:ifNotNil:` method defined in line 10 is a convenience method implementing a simple control structure based on blocks. It is a keyword message with two parameters, here called `nilBlock` and `goBlock`.

Blocks are anonymous functions, i. e., lambdas. Depending on the Smalltalk implementation, they are either full closures or classic restricted blocks as in Smalltalk-80 [Goldberg and Robson, 1983]. Classic blocks cannot be used once they left the dynamic extend in which they have been created. Thus, they cannot be returned from the method in which they were created.

SOM also supports class-side methods and static fields in terms of fields of the class object. Since classes are objects, the same rules apply and they are treated identically.

Handling of Globals References to classes are treated as lookups of global variables. Such global variables can also refer to objects other than classes. However, assignments to globals are ignored and not directly supported by

```
1 SOMUniverse = ( "..."
2  bootstrapFrameWithArguments: args = (
3    (interpreter pushNewFrameWithMethod: self bootstrapMethod)
4        push: (self globalAt: #system);
5        push: args;
6        yourself     "convenience method to return self" ) )
```

Listing 4.2: Cascaded Message Sends

the language. Instead, the binding of global variables can only be changed by the explicit use of a VM primitive. Thus, supposedly constant globals such as `true`, `false`, and class references cannot simply be assigned. While they might appear to be keywords or special literals, similar to how they are treated in languages such as C++ or Java, in SOM, they are treated like any other global and will be looked up in the globals dictionary. Following the Smalltalk spirit of *everything is an object*, `true`, `false`, and `nil` are objects as well. More precisely, they are the sole instances of their corresponding class. Consequently, they can be adapted and customized when necessary.

Cascaded Message Sends Some code examples use Smalltalk's *cascaded message sends* for conciseness, which is a language feature that is uncommon in other languages. The example in Lst. 4.2 defines a method that uses a complex expression in line 3 that yields a frame object. This frame object still needs to be populated with initial data, i. e., the method pushes a number of objects onto the frame. For conciseness, instead of assigning the frame object to a temporary variable, the method uses cascaded message sends. Thus, the resulting frame object of the expression becomes the receiver of the first regular message send in line 4, which pushes the `#system` global, and then indicated by the semicolons, also becomes the receiver of the two following message sends in lines 5 and 6. Note the last message `#yourself` in Lst. 4.2, it is a convenience method defined in `Object`. It is often used in cascaded message sends to return `self`. In this case it returns the frame object that was the result of the initial expression on line 3.

Dynamic Array Creation Some of the code examples in this dissertation use a notation for dynamically created arrays that is currently not part of the SOM implementation. However, it is commonly used in Squeak and Pharo Smalltalk to instantiate arrays in concise one-liners. The syntax uses curly braces to delimit a list of Smalltalk statements. The result value of each of

these statements is used as the value of the corresponding array index. As an example, the following line of code will create an array with three elements, the symbol `#foo`, the integer 42, and a new object:

```
anArray := {#foo. 21 * 2. Object new}
```

All statements are evaluated at runtime and can contain arbitrary code, including assignments.

Non-local Returns Smalltalk offers the notion of *non-local returns* to facilitate the implementation of custom control structures based on blocks. A non-local return from a block will not just return from the block's execution, as is done for instance with JavaScript's return statement inside a lambda, but it will return from the enclosing method of the block. This dissertation uses it as illustrated in Lst. 4.3 to avoid additional nesting of the else branch. Line 3 returns from `#foo:` when the argument `aBool` is `true`. Thus, the remainder of the method will only be executed if the argument was `false`.

```
1 Example = (
2   foo: aBool = (
3     aBool ifTrue: [ ^ #bar ].
4     " else: multiple statements without need to nest them "
5     ^ #baz ) )
```

Listing 4.3: Non-local Returns in Smalltalk

4.2.2. Execution Model and Bytecode Set

Since SOM focuses on clarity of concepts, its implementation is kept as accessible and straightforward as possible. This means that the VM implementation does not apply common optimizations. For instance, the interpreter loop in the AweSOM implementation uses a simple double dispatch for the bytecodes, which are represented as objects.

The execution model of SOM is based on a simple stack machine, which is a simplified variant of the one used in Smalltalk-80. The execution stack is built from linked *frame objects*, which are also called *context objects*. Each frame object corresponds to a method or block activation. Lst. 4.4 illustrates this with the basic definition of `SOMFrame`, including its instance variables. The field `previousFrame` is used to refer to the frame one level lower in the execution stack. If a block is executed, the outer frame, i. e., the `blockContextFrame`

needs to be set as well. This field refers to the frame that was active when the block object was created, and thus, constitutes the lexical environment of the block. Frames are the operand stacks for the execution of a method/block. They are initialized on activation, holding the arguments of the method/block at the bottom of the operand stack. `#bootstrapFrameWithArguments:` illustrates the calling convention with its creation of a bootstrap frame for the interpretation. The first element is always the receiver, in this case the global object for `#system`, and the following elements on the stack are the arguments to the activated method. In this case it is a single argument `args`, the array of arguments given to the initial program.

```
SOMObject = (
  | hash class |
  postAllocate = ()
"..." )

SOMFrame = SOMArray (
  | previousFrame blockContextFrame
    method
    bytecodeIndex stackPointer localsOffset |

  currentObject = ( ^ self at: 1 ) "the receiver object"
"..." )

SOMUniverse = (
  | globals symbolTable interpreter |

  bootstrapFrameWithArguments: args = (
    (interpreter pushNewFrameWithMethod: self bootstrapMethod)
                push: (self globalAt: #system);
                push: args;
                yourself ) )
```

Listing 4.4: SOM Implementation of stack frames and the initial bootstrap frame.

This section briefly goes over the implementation of essential bytecodes, because the discussion of the executable semantics for the solution in Sec. 5.4 relies on the basics described here.

The `SOMInterpreter` sketched in Lst. 4.5 manages three relevant instance variables. The `frame` instance variable refers to the currently active context object, `universe` refers to the object holding references to global and symbols table, and `currentBytecode` references the currently executing bytecode for convenience. The `currentObject` always represents the receiver, i. e., `self`

for the currently executing method and block. The `POP_FIELD` bytecode implemented by `#doPopField` in line 6, modifies the current object at the field index indicated by the bytecode. It stores the current top element of the stack into that field, and pops the top off the stack. The reverse operation is realized in line 11 by the `PUSH_FIELD` bytecode, which reads the field of an object and pushes the result onto the operand stack. The implementations of the `SEND` and `SUPER_SEND` bytecode given in lines 16 and 24 first determine the object that receives the message send, based on the number of arguments on the stack, and will then determine the class where the lookup starts to eventually delegate to the `#performSend:to:lookupCls:` method.

How the invocation is implemented depends on whether the message leads to an application of a SOM method representing bytecodes, or a primitive that is implemented in the implementation language. The application of a method, as shown in line 6 of Lst. 4.6, will result in the creation of a new frame (cf. line 29) that will be initialized with the receiver and the arguments. The invocation of a primitive on the other hand, is performed in AweSOM by taking arguments from the operand stack, executing the primitive with these arguments and pushing the result value back onto the stack. In other SOM implementations, and in typical Smalltalk VMs like the CogVM or RoarVM, primitives will obtain a reference to the context object, i. e., frame instead and will manipulate it directly, because it gives primitives more freedom and power in terms of what they can implement. AweSOM optimizes for the common case, reducing the burden on the primitive implementer.

The remaining bytecodes of SOM complement the ones already discussed. Since their details are beyond the scope of this dissertation, this section only gives a brief description of the operation each bytecode represents. Note however that bytecodes represent access to object fields and local variables with indexes. This is a typical optimization known as lexical addressing [Abelson et al., 1996].

```
SOMInterpreter = (
  | frame universe currentBytecode |

  currentObject = ( ^ (frame outerContext) currentObject )

  doPopField = (
    self currentObject
      fieldAtIndex: currentBytecode fieldIndex
               put: frame pop )

  doPushField = (
    frame push:
      (self currentObject
              fieldAtIndex: currentBytecode fieldIndex))

  doSend = (
    | receiver |
    receiver   := frame stackElementAtIndex:
                          currentBytecode selector numArgs + 1.
    ^ self performSend: currentBytecode selector
                    to: receiver
             lookupCls: receiver class )

  doSuperSend = (
    | receiver superClass |
    receiver   := frame stackElementAtIndex:
                          currentBytecode selector numArgs + 1.
    "Determine super in the context of the correct method."
    superClass := frame outerContext method holder superClass.
    ^ self performSend: currentBytecode selector
                    to: receiver
             lookupCls: superClass )

  performSend: selector to: receiver lookupCls: cls = (
    ^ self send: selector toClass: cls )

  send: selector toClass: cls = (
    (cls lookupInvokable: selector)
      ifNotNilDo: [:invokable | invokable invokeInFrame: frame]
      ifNil: [self sendDoesNotUnderstand: selector] )
"...")
```

Listing 4.5: Basics of the SOM Interpreter

```
SOMInvokable = ( | signature holder numArgs | )

SOMMethod = SOMInvokable (
  | numLocals bytecodes |

  invokeInFrame: frame (
    | newFrame |
    newFrame := self universe
                        interpreter pushNewFrameWithMethod: self.
    newFrame copyArgumentsFrom: frame.
    ^ newFrame ) )

SOMPrimitive = SOMInvokable (
  | realSignature |
  invokeInFrame: frame (
    ^ self invokePrimitiveInPlace: frame )

  invokePrimitiveInPlace: frame (
    | theSelf arguments result |
    "without self, self is first argument"
    arguments := frame popN: numArgs - 1.
    theSelf   := frame pop.
    frame push: (theSelf
      performPrimitive: realSignature
         withArguments: arguments) ) )

SOMInterpreter = ( "..."
  pushNewFrameWithMethod: method = (
    ^ frame := SOMFrame new
        method:             method;
        previousFrame:      frame;
        resetStackPointerAndBytecodeIndex;
        yourself ) )
```

Listing 4.6: SOM Method and Primitive invocation.

`HALT` return from the interpreter loop, without changing the interpreter's state.

`DUP` duplicates the element at the top of the operand stack.

`POP` removes the element from the top of the operand stack.

`RETURN_LOCAL` performs a return from the current method. The top of the current frame's operand stack is saved as return value, the current stack frame is removed from the interpreter's stack, and the return value is then pushed onto the operand stack of the calling method.

`RETURN_NON_LOCAL` performs a return from a block activation. The top of the current frame's operand stack is saved as return value, then all stack frames from the interpreter's stack are removed until the end of the *context frame* chain is reached, named the *target frame*; the target frame is removed, too, and the return value is then pushed onto the top frame's operand stack.

`PUSH_LOCAL` **and** `POP_LOCAL` either push or pop the value of a local variable onto or from the operand stack.

`PUSH_ARGUMENT` **and** `POP_ARGUMENT` either push or pop the value of a method argument onto or from the operand stack.

`PUSH_FIELD` **and** `POP_FIELD` either push or pop the value of an object's field onto or from the operand stack.

`PUSH_BLOCK` pushes a new block object onto the operand stack. The block object is initialized to point to the stack frame of the currently-executing method, so that the block method can access its arguments and locals.

`PUSH_CONSTANT` pushes a constant value object onto the operand stack.

`PUSH_GLOBAL` pushes the value of an entry from the global symbol table onto the operand stack.

`SEND` **and** `SUPER_SEND` send a message to the class or superclass of an object. The name of the message specifies how many arguments are consumed from the operand stack. For example, the `#ifNil:ifNotNil:` message uses 3 elements: the receiver object and two explicit arguments. Each send leads to the creation of a new frame, which takes the arguments and is used for the execution of the corresponding method. Arguments are popped from the operand stack of the original frame.

4.3. Squeak and Pharo Smalltalk

Squeak and Pharo are open source Smalltalk implementations in the tradition of Smalltalk-80 [Goldberg and Robson, 1983]. They provide an image-based development environment that offers good support for the prototyping of language and VM ideas. Both rely on the same VM for execution.

CogVM The *CogVM*[3] is primarily a VM with a just-in-time (JIT) compiler. It also includes a bytecode-based interpreter. Interpreter and JIT compiler are based on a bytecode set that differs only marginally from Smalltalk-80. One noticeable difference is the support for closures, which goes beyond Smalltalk-80 in that closure activation is legal even after the closure escaped from the dynamic extent in which it was created. Furthermore, the implementation applies a moderate number of optimizations for interpreters such as immediate integers based on tagged pointers, context-to-stack mapping to optimize representation of stack frames [Miranda, 1999], and threaded interpretation [Bell, 1973] to reduce the overhead of bytecode interpretation when the JIT is not yet warmed up, i. e., before a method was compiled to native code.

Rationale The OMOP needs to be evaluated against the state of requirements and its applicability needs to be demonstrated. For this dissertation, the use of Smalltalk is an advantage, since the available tools enable developers to implement experiments with a high level of productivity. Furthermore, while the language and its tools are somewhat different from mainstream environments like Java, Python, etc., it provides a concurrency model based on shared memory and threads, which reflects the most common approach to concurrent and parallel programming. One difference with other languages using threads is that Smalltalk's threads, i. e., processes, are green threads and scheduling is performed by the VM without relying on the operating system. In addition to that, Smalltalk has a strong track record of being used for research in concurrent programming models and VM implementation techniques [Briot, 1988, 1989; Gao and Yuen, 1993; Pallas and Ungar, 1988; Renggli and Nierstrasz, 2007; Thomas et al., 1988; Ungar and Adams, 2009; Yokote, 1990]. The high productivity of the development tools and the adequate performance of the underlying VM implementation make Squeak and Pharo based on the CogVM good candidates for experimentation.

[3] *Teleplace Cog VMs are now available*, Eliot Miranda, 20 June 2010
http://ftp.squeak.org/Cog/README

Used Software and Libraries Squeak and Pharo were used as development platforms for specifying the operational semantics of the OMOP using AweSOM and to develop the AST-transformation-based prototype (cf. Sec. 7.1). While Smalltalk typically uses a bytecode-based representation of the compiled code, transforming it is low-level and error-prone. To avoid the associated complexity, the first prototype implementation of the OMOP uses code transformations based on the AST (abstract syntax tree) instead. The main benefit is that the AST will be used to generate bytecode that corresponds to the expectations of the whole Smalltalk toolchain. Using direct bytecode transformation can result in perhaps more optimal bytecode, however, the used tools, i. e., the debugger and decompiler, accept only a subset of the legal bytecode sequences, rendering some correct bytecode sequences non-debuggable. The AST transformations uses the Refactoring Engine,[4] which generates ASTs from Smalltalk code and provides a transformation framework implementing the classic visitor pattern. With these tools it became possible to implement AST transformations and produce bytecode that was executable and debuggable.

4.4. RoarVM

The RoarVM is a Squeak and Pharo-compatible Smalltalk VM designed and initially implemented by Ungar and Adams [2009]. It is a platform for experimenting with parallel programming on the Tilera TILE64 manycore processor [Wentzlaff et al., 2007]. Building on top of the work of Ungar and Adams, we ported it to commodity multicore systems. It enables the parallel execution of Smalltalk code in a shared memory environment. Thus, Smalltalk processes, i. e., threads, of a given image can be scheduled and executed simultaneously depending on the available hardware parallelism.

Rationale for Choosing the RoarVM The RoarVM was chosen to experiment with VM implementations for several reasons. On the one hand, the complexity of the RoarVM is significantly lower than that of the CogVM, facilitating experiments with different implementation approaches. Furthermore, the RoarVM has a parallel execution model, which preserves the opportunity to investigate support for parallel programming as part of future work (cf. Sec. 9.5.1).

[4] *Refactoring Engine*, Don Roberts, John Brant, Lukas Renggli, access date: 17 July 2012 http://www.squeaksource.com/rb.html

Another reason for its choice are the experiments that were performed in addition to the work on this dissertation. Examples are the implementation of a pauseless garbage collector [Click et al., 2005] to avoid stop-the-world garbage collection pauses,[5] as well as a work-stealing scheduler to improve the performance of fine-grained parallelism.[6] Especially relevant for the work presented here was an experiment to use operating system processes instead of threads as the underlying abstraction to use multiple cores (cf. Sec. 4.4.3).[7]

These extensions to the RoarVM promise to be a good foundation to experiment with different optimizations (cf. Sec. 9.5.3).

4.4.1. Execution Model, Primitives, and Bytecodes

This section discusses technical details that are relevant for the implementation of the OMOP and its performance evaluation.

Execution Stack Representation Similarly to the execution model of SOM, the RoarVM uses the classic Smalltalk-80 model of context objects that represent stack frames. Each frame, i. e., context object is a standard Smalltalk object allocated on the heap and subject to garbage collection. They represent the method activation, the operand stack, and temporary variables. A context object also encapsulates the corresponding instruction and stack pointer. To reduce the pressure on the garbage collector (GC), context objects are cached and reused if possible, instead of leaving them for the GC.

While using objects to represent the execution state enables for instance metaprogramming, it comes with a performance cost. To reduce that cost by avoiding frequent indirections on the context objects, part of the execution state such as instruction pointer and stack pointer are replicated in the interpreter object and maintained there. The context object is only updated with the execution state when necessary. Thus, the execution state is written to the actual context object before garbage collection starts, before scheduling operations, such as resuming a processes or yielding execution, might change the currently active context, and before a message send activates the new context for its execution.

Primitives and Quick Methods Primitives are used to implement functionality that cannot be implemented directly in Smalltalk or for functionality

[5] https://github.com/smarr/RoarVM/tree/features/parallel-garbage-collection
[6] https://github.com/smarr/RoarVM/tree/features/scheduler-per-interpreter
[7] https://github.com/smarr/RoarVM/tree/features/processes-on-x86

that is performance critical and benefits from avoiding the interpretation overhead.

The RoarVM and CogVM are derived from the same original code base and share the implementation of primitives. While the implementations diverged over time, the RoarVM supports the same primitives and collections of primitives in the form of plugins to be compatible with the CogVM. Thus, on the language level the provided mechanism are identical.

Smalltalk has access to primitives via an encoding in the method header of a method object. If the primitive part of the method header is set to a value different from zero, the VM is asked to execute the primitive referenced by this method instead of executing the bytecodes encoded in the method. For the execution of a primitive the current context object is used, instead of creating a new one, as is done for standard message sends. This gives the primitive access to the current operand stack, the VM, and perhaps the underlying system. A number of primitive identifiers is however reserved for so-called *quick methods*. Quick methods do not encode bytecodes, but instead use the primitive identifier to encode presumably common short methods. This includes return of `self`, `true`, `false`, and `nil`. Furthermore, quick methods encode accessors, i. e., methods that only return the object stored in a field.

Tab. 4.1 shows the encoding of the method header in the RoarVM. It uses the same encoding as the CogVM for compatibility. To avoid issues with the garbage collector, and because the method header is a normal field in the method object, it is encoded using a `SmallInt`, which is indicated by the least significant bit set to one. The header encodes the number of arguments (`#args`) the method expects, the number of slots the context object should provide for temporary variables (`#temporaries`), whether a small or large context object should be allocated for the operand stack (`FS`: frame size), and the number of literals encoded in the method. For historical reasons, the primitive is encoded with 10 non-consecutive bits. Bit 29 was presumably added when the need arose. The 30th bit (`F`) is a flag bit that is reserved for custom use at the language side, and bit 31 remains unused.

Table 4.1.: RoarVM Method Header

	F	P.	#args	#temporaries	FS	#literals	Primitive	1
31	30	29	28 27 26 25	24 23 22 21 20 19	18	17 16 15 14 13 12 11 10	9 8 7 6 5 4 3 2 1	0

Bytecodes The bytecode set used by the RoarVM is very similar to the Smalltalk-80 bytecode set depicted in Tab. 4.2. It encodes common operations with single bytes, directly encoding a small range of parameters. Supposedly, this encoding was chosen to keep the code size of Smalltalk methods small and enable efficient implementation based on C's `switch/case` statement that will be mapped on a dispatch table by most compilers. The RoarVM also still uses `switch/case` instead of direct or indirect threading. While the RoarVM supports the closure extension that was introduced by the CogVM, it is not considered in this dissertation. Another difference in the used bytecode set is the changed meaning of bytecode 132. It has been changed to the unequivocal name: `doubleExtendedDoAnythingBytecode`. Three bits of the second byte are used to encode the operation, and the remaining 5 bits encode an argument. The third byte encodes a literal to be used.

The bytecodes for *arithmetic* and *special* methods are essentially *shortcut* bytecodes that reduce the number of bytes needed to encode presumably common message sends. Besides arithmetic operations such as *add*, *subtract*, and *multiply*, this includes comparisons such as *less than*, *greater or equal*, and *equal*. The special methods include for instance `#at:`, `#at:put:`, and `class`. The implementation of these bytecodes will first set the corresponding symbol for the message send, and then issue the actual message send. However, if possible the addition is executed directly without doing an actual message send, for instance if the receiver and the argument of the `#+` message are integers.

4.4.2. Memory Systems Design

The memory system of the RoarVM has been designed with simplicity in mind, facilitating the experimentation on manycore platforms like the Tilera TILE64 [Wentzlaff et al., 2007]. Therefore, the RoarVM uses a simple compacting mark-and-sweep GC that relies on a stop-the-world mechanism for safe memory reclamation. Beside the `SmallInt` immediate tagged integers, all objects are garbage collected.

One important artifact of Ungar and Adams' research that became part of the RoarVM is an additional memory word that is prepended to every object. For their implementation of the Ly and Sly language prototypes [Ungar and Adams, 2010], they changed the semantics of message dispatching. In Sly, an object can be part of an ensemble, i. e., a collection. If a message is sent to such an object, the message is reified and sent to the ensemble instead. This technique allows Ungar and Adams to unify to a certain degree the treatment of

Table 4.2.: The Smalltalk-80 Bytecodes [Goldberg and Robson, 1983, p. 596]

Range	Bits	Function
0-15	0000iiii	Push Receiver Variable #iiii
16-31	0001iiii	Push Temporary Location #iiii
32-63	001iiiii	Push Literal Constant #iiiii
64-95	010iiiii	Push Literal Variable #iiiii
96-103	01100iii	Pop and Store Receiver Variable #iii
104-111	01101iii	Pop and Store Temporary Location #iii
112-119	01110iii	Push (receiver, true, false, nil, -1, 0, 1, 2) [iii]
120-123	011110ii	Return (receiver, true, false, nil) [ii] From Message
124-125	0111110i	Return Stack Top From (Message, Block) [i]
126-127	0111111i	unused
128	10000000	Push (Receiver Variable, Temporary Location,
	jjkkkkkk	Literal Constant, Literal Variable) [jj] #kkkkkk
129	10000001	Store (Receiver Variable, Temporary Location,
	jjkkkkkk	Illegal, Literal Variable) [jj] #kkkkkk
130	10000010	Pop and Store (Receiver Variable, Temporary Location,
	jjkkkkkk	Illegal, Literal Variable) [jj] #kkkkkk
131	10000011	Send Literal Selector #kkkkk
	jjjkkkkk	With jjj Arguments
132	10000100	Send Literal Selector #kkkkkkkk
	jjjjjjjj	With jjjjjjjj Arguments
	kkkkkkkk	
133	10000101	Send Literal Selector #kkkkk To Superclass
	jjjkkkkk	With jjj Arguments
134	10000110	Send Literal Selector #kkkkkkkk To Superclass
	jjjjjjjj	With jjjjjjjj Arguments
	kkkkkkkk	
135	10000111	Pop Stack Top
136	10001000	Duplicate Stack Top
137	10001001	Push Active Context
138-143		unused
144-151	10010iii	Jump iii + 1 (i.e., 1 through 8)
152-159	10011iii	Pop and Jump On False iii +1 (i.e., 1 through 8)
160-167	10100iii	Jump (iii - 4) * 256 + jjjjjjjj
	jjjjjjjj	
168-171	101010ii	Pop and Jump On True ii * 256 + jjjjjjjj
	jjjjjjjj	
172-175	101011ii	Pop and Jump On False ii * 256 + jjjjjjjj
	jjjjjjjj	
176-191	1011iiii	Send Arithmetic Message #iiii
192-207	1100iiii	Send Special Message #iiii
208-223	1101iiii	Send Literal Selector #iiii With No Arguments
224-239	1110iiii	Send Literal Selector #iiii With 1 Argument
240-255	1111iiii	Send Literal Selector #iiii With 2 Arguments

ensembles and their elements. The technique of prepending additional words to objects is also useful for the experiments of this dissertation and will be detailed in later chapters (cf. Sec. 7.2.1).

Object Table for Manycore Architectures Since cache locality and the non-uniform memory access properties of the TILE64 were one of the fields of interest for Ungar and Adams, they decided to use an object table in the VM to reduce the necessary effort for moving objects between different parts of the heap.

As the number of cores grows, it becomes increasingly difficult for caches to provide the illusion of a single coherent memory with uniformly short access time. Consequently an application or VM seeking good performance on a manycore system may be required to dynamically move objects between different parts of the heap for the sake of improved locality. The TILE64, Ungar and Adams targeted, has only restricted support for cache coherency. Memory pages are *homed* to a certain core which means that only this core is able to cache memory from that page in its local cache. This means that it is desirable to be able to move objects easily between memory pages to allow them to be cached by a core that actually needs them.

When an object needs to be moved, a naive solution would require a full scan of the heap to adjust all references to it as well. While the cost is often amortized by moving multiple objects, it remains significant. Other approaches employ a read-barrier, which however can itself add overhead and complexity. An object table on the other hand allows an immediate update of the object's location. Using the infrastructure for flexible additional words for each object, it is possible to include a backpointer in each object to identify the object table entry directly and thus make lookups trivial. With this design, moving an object reduces to adjusting a single reference, which can be found via the backpointer starting from the object itself.

Drawback of Object Tables The use of an object table also has a number of disadvantages. For the standard case of accessing an object, the address-lookup in the table has a performance penalty. Furthermore, it also brings additional complexity to the system, since the move operation of an object needs to be atomic to prevent any other core to write to an object that is currently being moved. Moreover, storing the object table itself is an issue with such restricted caching schemes. Object table entries need to be reclaimed after an object got garbage collected and the implied compaction of the table

is a problematic operation, too. Another detail that needs to be considered is the approximately 10% space penalty imposed by the extra header word we use as a backpointer from the object to the object table entry. In addition to assertion-checking for debugging, this backpointer is required for the sweep phase of the garbage collector.

As discussed in Sec. 8.1.3, the object table is disabled for the performance evaluation in order to avoid the performance overhead.

4.4.3. Process-based Parallel VM

Ungar and Adams implemented the VM for the TILE64 and used libraries that relied on operating system processes instead of threads to utilize the 64 processor cores. When we ported the RoarVM to commodity multicore systems, we decided to use traditional threads instead of the process-based variant. The main driver for this decision was the absence of sufficient debugging tools and that the initially used libraries were only available for the TILE64.

However, the thread-based implementation is significantly slower than the process-based implementation. The performance cost for using thread-local variables instead of static globals is significant. Therefore, the decision was revisited and a version using processes instead of threads was also implemented on classic Linux and Mac OS X systems.[8]

For this dissertation, the ability to use multiple operating system processes is relevant in the context of future work (cf. Sec. 9.5.3). It enables the VM to use different memory protection settings as an implementation technique for MANAGED STATE (cf. Sec. 3.4). A similar approach has been used for instance by Hoffman et al. [2011].

4.4.4. Final Remarks

While the RoarVM is an interpreter and performance evaluation yields results that are not generalizable to high-performance VMs with JIT compilers,[9] the RoarVM as a research platform has a number of relevant features that facilitate experiments and opens opportunities for future research.

The support for arbitrary additional words in front of objects greatly simplifies experiments that need to adapt the notion of objects and extend it with

[8] https://github.com/smarr/RoarVM/tree/features/processes-on-x86

[9] An addition of a JIT compiler to the RoarVM would be possible and it would improve the generalizability of the results, but it is outside of the scope of this dissertation.

custom features. The process-based parallel execution model can be the foundation for optimizations that utilize techniques such as the operating systems memory protection of memory pages.

Furthermore, the RoarVM compared to the CogVM is significantly less complex, which reduce the implementation effort and enables a wider range of experiments. Its support for parallel execution also preserves the opportunity to experiment on multicore systems, while the CogVM would also restrict future experiments to sequential execution.

4.5. Summary

This chapter presented SOM (Simple Object Machine), a Smalltalk that is intended for teaching. Its minimalistic approach and concise implementation allows the definition of an executable semantics of the OMOP in terms of a bytecode interpreter that is similar to the actually used VMs.

Furthermore, this section presented Squeak and Pharo Smalltalk as a foundation for the implementation of the evaluation case studies. Both Smalltalks are mature platforms that have been used in other research projects before and provide a stable foundation for the experiments in this dissertation.

Finally, this section discusses the RoarVM, which is used for experiments with VM support. It describes the main implementation features, i. e., the execution model, primitives, and the bytecode set as a foundation for the explanation of the implementation in Chapter 7.

5

AN OWNERSHIP-BASED MOP FOR EXPRESSING CONCURRENCY ABSTRACTIONS

This chapter introduces the main contribution of this dissertation: the design of an ownership-based metaobject protocol (OMOP), which is meant to facilitate the implementation of concurrent programming concepts. To this end, it first introduces *open implementations* and *metaobject protocols* (MOP) [Kiczales et al., 1991] to motivate the choice of using such a mechanism as the foundation for the OMOP. Second, it details the design and properties of the ownership-based MOP. Third, its demonstrates how to apply the OMOP to enforce the notion of immutability and the semantics of Clojure agents. Fourth, this chapter describes the OMOP's semantics based on a bytecode-based interpreter that implements the OMOP. And finally, it situates the OMOP into the context of related work.

5.1. Open Implementations and Metaobject Protocols

Considering the requirements identified in Sec. 3.4 and that the contemporary multi-language VMs are object-oriented programming environments (cf. Sec. 3.1.1.2), reflective programming techniques provide the natural foundation for a program to interface with a VM, i. e., its runtime environment. Thus, this section briefly reviews the notion of *open implementations* and *metaobject protocols*, because they are commonly used to provide properties similar to the ones required for a unifying substrate for concurrent programming concepts.

The notion of *open implementations* described by Kiczales [1996] provides a general design strategy for increasing the control that client code has over a software component it is using. The general motivation is that client code often needs to adapt or work around the concrete software component it is using, to change semantics or performance characteristics to the context it is used in. Thus, *open implementations* are designed to facilitate the adaptation of implementation strategies [Kiczales et al., 1997]. This notion can also be used to enable adaptive language guarantees and semantics based on a meta interface. Meta interfaces for this purpose are commonly known as *metaobject protocols*:

> *Metaobject protocols are interfaces to the language that give users the ability to incrementally modify the language's behavior and implementation, as well as the ability to write programs within the language.*
> [Kiczales et al., 1991, p. 1]

Introduction Today, metaobject protocols (MOPs) can be found in a number of languages, for instance in the Common Lisp Object System (CLOS), Smalltalk-80 [Goldberg and Robson, 1983], Groovy,[1] and Perl's Moose object system.[2] While not all of them offer the full functionality of the MOP discussed by Kiczales et al. [1991], they provide capabilities to reflect on the executing program and adapt the language's behavior.

To start from the beginning, important foundations for MOPs are the notions of *reflection* and *reification* [Friedman and Wand, 1984]. Reflection builds on *introspection* and *intercession*. Introspection is the notion of having a program querying itself for information. This information is then *reified* in terms of program structures which can be processed. Based on these reified program structures, intercession enables the program to interfere with its own

[1] http://groovy.codehaus.org/
[2] http://search.cpan.org/~flora/Class-MOP/

execution. This means it can change state, adapt program structures, or trap certain operations to refine their behavior. The reified program structures are referred to as metaobjects. In order to explain MOPs, Kiczales et al. [1991] state that in general, a protocol is formed by *"a set of object types and operations on them, which can support not just a single behavior, but a space or region of behaviors"*. Therefore, they conclude that a MOP enables the encoding of individual decisions about language behavior via the operations of metaobjects.

Categories of MOPs Tanter [2009, p. 12] distinguishes metaobject protocols (MOPs) for object-oriented reflection by the correspondence of the meta relation to other aspects of the system. He identifies a number of common ideas: metaclass-based models, metaobject-based models, group-based models, and message-reification-based models.

Metaclass-based Metaclass-based models such as in CLOS, Perl's Moose, or Smalltalk enable for instance the customization of method dispatch or object fields semantics. The metaclass of a class therefore describes the semantics of this class, i. e., the metaclass' instance. The meta relation in this case is the instantiation relationship between a class and its metaclass. This technique has been used, e. g., to implement persistent objects, which are automatically mapped to a database [Paepcke, 1993], or even to parallelize programs [Rodriguez Jr., 1991].

Metaobject-based Metaobject-based models decouple meta interfaces from the class hierarchy. One example for a language using this model is the prototype-based language 3-KRS [Maes, 1987]. Since 3-KRS does not have the notion of classes, it is designed with a one-to-one relation between a base-level object and a metaobject. However, the model can be applied to class-based languages, for instance as is the case in Albedo, a Smalltalk system [Ressia et al., 2010]. The independence from the class hierarchy enables modifications that are orthogonal to the class hierarchy and results in a model with greater flexibility.

Instead of having a one-to-one mapping, other variations of the model are possible as well. One common example is proxy-based MOPs, for instance as the one proposed for the next version of ECMAScript [Van Cutsem and Miller, 2010].[3] Metaobjects are defined in the form of proxy objects that reify

[3] *Direct Proxies*, Tom Van Cutsem, access date: 4 July 2012
http://wiki.ecmascript.org/doku.php?id=harmony:direct_proxies

the operations on the target object and thus, allow developers to adapt the language's behavior as necessary. It is different from 3-KRS in that a target object can have multiple metaobjects associated with it. Furthermore, the semantics defined by the proxy are only applied if a client interacts with the target object through the proxy, which can be a significant drawback. On the other hand, this design gives a large degree of flexibility, since every target object can be adapted by arbitrarily many customizations as long as the client uses the proxy instead of the target object.

Group-based Tanter further discusses the variation of group-based MOPs. Instead of having a distinct metaobject for every base-level object, for instance Mitchell et al. [1997] and Vallejos [2011] argue that it can be an advantage to enable a metaobject to describe the semantics of a set of objects. Mitchell et al. [1997] use *meta-groups* to control scheduling decisions for base and metaobjects. Describing these semantics based on groups of objects instead of separate objects avoids the need for synchronization between multiple metaobjects, when scheduling decisions have to be made.

Message-based The message-reification-based models Tanter discusses provide a meta interface to specify the semantics of message sends, for instance by taking sender and receiver into account. For example, Ancona et al. [1998] propose a model they call *channel reification*, which is based on message-reification. However, since the model is based purely on communication and does not reify, e. g., object field access, it is too restrictive for our purposes.

Conclusions Overall, metaobject protocols seem to be a good fit with the requirements identified for a unifying substrate in multi-language VMs (cf. Sec. 3.4). However, some of the approaches do not provide sufficient support to satisfy all of the requirements.

The metaclass-based model is too restrictive, because it is not orthogonal to application and library code. Sec. 2.5 describes the vision of using the appropriate concurrent programming concepts for different challenges in a Mail application. It argues that event-loop-based actors are a good fit for implementing the user interface, while an STM is a good fit to interact with the database and to ensure data consistency. Considering that such an application would rely on a number of third-party libraries for email protocols and user interface components, these libraries need to be usable in the context of event-loop actors as well as in the context of the STM. When using a

metaclass-based model as foundation for the implementation of these concurrent programming concepts, the used library would only be usable for a single concurrent programming concept, because it typically has one fixed metaclass.

Message-reification-based models can be more powerful than metaclass-based models, but are best applied to languages designed with an *everything-is-a-message-send* approach. This is however an impractical constraint for multi-language VMs.

Hence, a better foundation for a meta interface is a metaobject-based model that fulfills all requirements identified in Sec. 3.4. For the problems considered in this dissertation, a proxy approach adds undesired complexity. Specifically, the constraint that every client needs to use the correct proxy imposes a high burden on the correct construction of the system, since the actual object reference can easily leak. Furthermore, the added flexibility is not necessary to satisfy the identified requirements, and the added overhead of an additional proxy per object might provoke an undesirable performance impact.

To conclude, a metaobject-based approach that allows one metaobject to describe the semantics for a set of objects similar to the group-based approaches provides an suitable foundation.

5.2. Design of the OMOP

Following the stated requirements (cf. Tab. 3.7) for the support of concurrent programming concepts, a unifying substrate needs to support the intercession of state access and execution, provide the notion of ownership at the level of objects,[4] and enable control over when the desired guarantees are enforced. As argued in the previous section, metaobject-based models for MOPs provide a promising foundation to design a minimal meta interface for the purpose of this dissertation. Note that the goal of this chapter is to design a unifying substrate for a multi-language VM. The main focus is to improve support for language semantics of concurrent programming concepts. Therefore, aspects such as security, reliability, distribution, and fault-tolerance are outside the scope for the design of this substrate.

To satisfy the requirement of representing ownership at the granularity of objects, the major element of the OMOP is the notion of a *concurrency domain*. This *domain* enables the definition of language behavior similar to metaobjects or metaclasses in other MOPs. The language behavior it defines

[4] Our notion of *ownership* does not refer to the concept of *ownership types* (cf. Sec. 5.6).

is applied to all objects the domain *owns*. A domain is similar to a meta-group in group-based MOPs (cf. Sec. 5.1) and the notion of object ownership realizes the meta relation in this MOP. However, compared to group-based MOPs, the OMOP requires every object to be *owned by* exactly one domain. Therefore the MOP proposed here is called an *ownership-based metaobject protocol* (OMOP). A visual representation of the OMOP is given in Fig. 5.1.

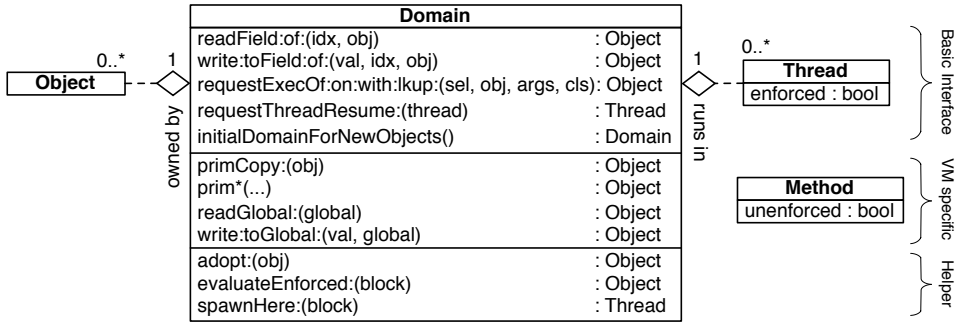

Figure 5.1.: Ownership-based Metaobject Protocol. The domain is the metaobject providing the intercession handlers that can be customized to adapt the language's behavior. Each object is owned by exactly one domain. Every thread executes in one domain. Execution is either enforced, i. e., operations on an object trigger intercession handlers, or it is unenforced and intercession handlers are not triggered. The handlers enable the customization of field reads and writes, method invocation, thread resumption, and initial owner of an object. If the VM offers primitives and globals, they are reified as well, but these handlers are VM-specific. Methods can be marked as unenforced to execute them always without triggering the intercession handlers.

Fig. 5.2 depicts a simple object configuration during the execution of an application that uses the OMOP. The example consists of two domains, represented by the dashed circles at the meta level, with interconnected object graphs on the base level. Note that the pointers between domains do not need special semantics. Instead, the concurrency properties of a base-level object are defined by the domain object that owns the base-level object. Thus, the *owned-by* relation is the meta relation of the OMOP.

The remainder of this section discusses in more detail the semantics associated with the OMOP and establishes the connection of its elements to the requirements.

Basic Interface The first compartment of the `Domain` class depicted in Fig. 5.1 contains the basic intercession handlers provided by the OMOP. This basic

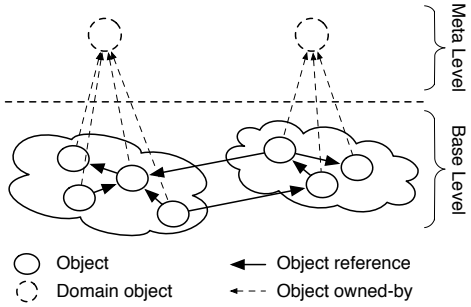

Figure 5.2.: Possible object configuration during runtime. Objects are owned by the domain, i. e., their corresponding metaobject. The owned-by relation is depicted with a dashed arrow. Object references remain unaffected.

part is universal, while the second compartment contains intercession handlers that depend on the target VM and can require variations for proper VM integration.

The basic intercession handlers constitute a meta interface to intercept *reading* of object fields, *writing* of object fields, and *invocation* of methods on objects. Also part of this basic interface is the notion of a *thread* that is executing in a domain. A thread indicates with an `enforced` state whether during its execution the semantics of the domain are to be realized to satisfy the requirement for controllable enforceability and to avoid infinite meta-recursion. Thus, in enforced execution mode, operations on an object are delegated to the intercession handlers of the owning domain. The enforced execution mode conforms to the program execution at the base level. The unenforced execution mode conforms to the execution at the meta level and does not trigger intercession handlers.

Since the unit of execution, i. e., a thread, is typically subject to restrictions of a concrete concurrent programming concept, the basic interface includes an additional intercession handler to respond to threads trying to resume execution inside a domain. Therefore, a domain can decide whether a thread is allowed to resume execution or whether any other actions have to be taken.

To determine the initial owner of a newly created object, the OMOP provides the `#initialDomainForNewObject` intercession handler, which is triggered on the domain the current thread is executing in when an object is created.

VM-specific Interface Depending on the VM, a domain also needs to manage globally accessible resources that may lie beyond its scope but that can

have an impact on the execution. This typically includes *lexical* globals and primitive operations of a VM. The corresponding meta interface is sketched in the second compartment of the `Domain` class. In addition to these operations, the VM-specific part can also include an `unenforced` bit for each method. If this `unenforced` bit is set, the corresponding method is always executed in the `unenforced` execution mode. If the bit is not set, the method will execute either enforced or unenforced depending on the current execution mode of the thread. In general, the bit is used to mark operations that should execute always at the meta level. This includes for instance the intercession handlers of a domain themselves to guarantee that their execution is performed unenforced. Other examples are methods that implement general language behavior, for instance in Sec. 5.3.2, it is used in the agent example to mark the `#read` method unenforced to represent the notion that the agent's state is readable without any restrictions.

Helper Methods The third compartment of the `Domain` class in Fig. 5.1 contains a set of helper methods. These methods sketch functionality that is merely convenient to have, but is not a fundamental part of the OMOP. Thus, the operations offered here are not strictly orthogonal to the other mechanisms offered by the OMOP. For instance, in the theoretical model depicted by Fig. 5.1, the owner of an object is represented directly by the *owned by* relation, and thus, can be modified directly. The helper methods offer the `#adopt:` method which requests a domain to take over ownership for the given object. This abstracts from the low-level details of how the *owned by* relation is realized. For instance, it could be just another object field, or it could be represented by arranging the objects of different domains in separate heaps. The `#evaluateEnforced:` method also provides a high-level interface to the execution mode by evaluating a block of code in the enforced execution mode. This is convenient for the implementation of concurrency policies, as is the spawning of new threads in a given domain via `#spawnHere:`.

Elements of the OMOP The remainder of this section gives a brief brief summary of the elements of the OMOP and relates them to the requirements.

Domains own objects, and every object has one owner. They define the concurrency semantics for owned objects by refining field read, field write, method execution, and thread resumption. Newly created objects belong to the domain specified by `#initialDomainForNewObjects`. With `#adopt:`, the owner of an object can be changed during execution. This

set of mechanisms satisfies the OWNERSHIP requirement and provides a way to adapt language behavior on an object level.

By requiring that all objects have an owner it becomes possible to adapt language behavior in a uniform way for all objects in the system. For instance, domain objects themselves are owned by a domain. In the normal case it is a standard domain that represents the language behavior of the language implemented by the VM.

Thread is the unit of execution. The `enforced` bit indicates whether it executes enforcing the semantics of domains, i. e., whether the intercession handlers are triggered or not. Each thread is said to *run in* the context of one specific domain. If a thread attempts to resume execution in a domain, the corresponding `#requestThreadResume:` intercession handler can be used to implement custom policies, for instance to prevent execution of more than one thread at a time. `#evaluateEnforced:` enables an existing thread to change the execution domain for the duration of the execution of the `block`. During the evaluation, guarantees are enforced, i. e., the corresponding flag in the thread is set. `#spawnHere:` creates a new thread in the given domain and starts its execution, iff `#requestThreadResume` has not specified a contradicting policy. These mechanisms are necessary to satisfy the MANAGED EXECUTION requirement.

Method representations can contain an additional bit to indicate that a particular method is always to be executed in the `unenforced` mode. This gives more flexibility to CONTROL ENFORCEMENT. If the bit is set, the thread executing the method will switch to execute in unenforced mode, i. e., at the meta level. If the bit is not set, the thread will maintain the current execution mode.

Read/Write operations of object fields are delegated to the `#readField:of:` and `#write:toField:of:` intercession handlers of the owning domain. The domain can then decide based on the given object and the field index, as well as other execution state, what action needs to be taken. This satisfies the MANAGED STATE requirement. Note, the intercession handlers are only triggered while the thread executes in the enforced execution mode.

Method Execution is represented by the `#requestExecOf:on:with:lkup:` intercession handler. It enables the domain to specify language behavior

for all method invocations based on the given object, the method to be executed, its arguments, and other execution state. This satisfies, together with the execution context of a thread, the MANAGED EXECUTION requirement.

External Resources i. e., globally shared variables and primitives need to be handled by the domain if otherwise they break semantics. To that end, the domain includes `#readGlobal/#write:toGlobal:`, which allows for instance to give globals a value local to the domain. Furthermore, it includes `#prim*` intercession handlers, such as `#primCopy:` to override the semantics of VM primitives. This is necessary, because unintercepted access to `#primCopy:` would enable copying of arbitrary objects without regard for domain semantics. Thus, depending on a specific VM, this extension to the basic meta interface is necessary to complete the support for MANAGED STATE and MANAGED EXECUTION. Furthermore, depending on a specific VM, all the primitives offered by the VM need to be supported by the domain. For the RoarVM, this extended meta interface includes in addition to `#primCopy:` for instance `#primNext:` and `#primNext:put:` to handle the support for stream data structures of the VM properly.

5.3. The OMOP By Example

This section informally illustrates the semantics of the OMOP by demonstrating how it can be used to enforce immutability as well as how it can be used to implement Clojure agents and enforce their intended semantics.

5.3.1. Enforcing Immutability

Immutability significantly simplifies reasoning over program behavior, not only in concurrent systems, and thus, it is a useful subject to study as an example. Using the OMOP, immutability can be realized by changing the owner of an object to a domain that forbids any kind of mutation. A domain such as the `ImmutableDomain` can be defined so that it throws an error on every attempt to mutate state. The definition of such a domain is discussed below.

Fig. 5.3 provides a sequence diagram showing how immutability can be enforced based on the OMOP. The example uses a simple mutable cell object and shows how it becomes owned by a domain guaranteeing immutability.

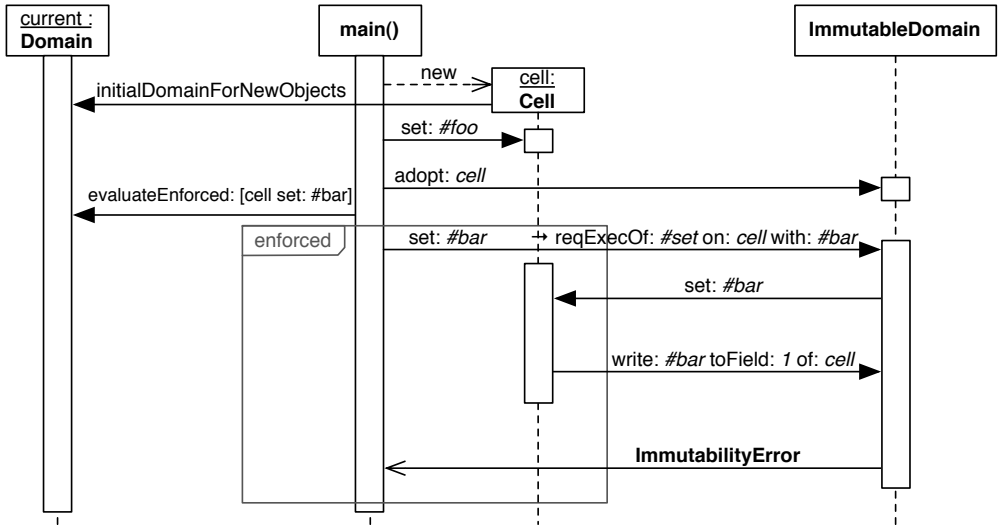

Figure 5.3.: Enforcing Immutability with the OMOP. This sequence diagram depicts a typical interaction between the application and a domain. The main program starts in unenforced execution mode and creates a cell object. Since it has not specified any domain as yet, it executes in the context of the standard, i. e., uncustomized domain. During the creation of the cell object the current domain is used to determine its initial owner. The main program continues to set an initial value for the cell (`#foo`) and then requests the immutable domain to adopt the cell object. This completes the initialization of the program and it enables the enforced execution mode. When the main program now attempts to execute the setter method on the cell, the immutable domain will reify the execution. However, immutability does not interfere with method execution and thus, the request is granted and the setter is executed. Eventually, the setter will attempt to write the object field of the cell, which the immutable domain specifies a custom policies for. Thus, the write attempt results in an invocation of the `#write:toField:of:` intercession handler, which signals a violation of immutability and does not perform the state change.

The `main()` procedure of the program executes in the context of the `current` domain, which can be assumed to be a default that does not enforce any specific semantics. Furthermore, note that `main()` starts executing without enforcement enabled.

The first action of the program is the creation of a new cell object. To determine the initial owner of the new object, the VM queries the `current` domain via a call to `#initialDomainForNewObject`. Thus, the owner of a newly created object is determined by the domain in which the current thread is executing. Afterwards, still in the unenforced execution mode, the cell is set to the value `#foo` and then adopted by the immutable domain. After adopting the cell object, the intercession handlers of the domain define the semantics for all operations on the cell. They will be triggered during the enforced execution mode, which is enabled by requesting the domain to evaluate a block: `current evaluateEnforced: [cell set: #bar]`.

When the VM reaches the point of executing `cell set: #bar` during enforced execution, it will not apply the method directly, but it will request the execution of the method by using the OMOP. The owner of the cell is identified and the intercession handler `#requestExecOf:on:with:` is triggered on it. The intercession handler itself executes in unenforced mode to avoid meta recursion. In this example, `#requestExecOf:on:with:` implements standard semantics of method execution, since immutability does not require any changes to it. Hence, the method `#set:` is executed with the enforcement enabled. At some point, the implementation of `#set:` tries to perform a write to the object field of the cell. Here, the VM triggers the OMOP's `#write:toField:of:` intercession handler of the owner of the cell, instead of performing the write directly. Thereby, the immutable domain is able to signal a violation of the requested immutability by throwing the corresponding exception.

To cover other reflective capabilities properly as well, an implementation of the OMOP requires that for instance reflective writes to fields obey the enforcement flag. For example, in case the setter would have used Java's `reflect.Field.set()` to write the value, the implementation of `Field.set()` would be required to determine the owner domain of the cell, and use the MOP to request the actual memory write.

After demonstrating how a program would execute in the presence of enforcement, Lst. 5.1 sketches of how such an immutability enforcing domain can be implemented. Note that the example uses the SOM syntax (cf. Sec. 4.2.1). The given `ImmutableDomain` overrides all intercession handlers that are related to state mutation. These are the intercession handlers for han-

dling writing to fields as well as all primitives of the VM that can cause state changes. This example only shows the primitive for array access and the primitive for reflective access to object fields. Note that the primitive `#priminstVarAt:put:on:` corresponds to Java's `reflect.Field.set()`.

```
ImmutableDomain = Domain (

  raiseImmutabilityError = (
    ImmutabilityError signal: 'Modification of object denied.' )

  write: val toField: idx of: obj = unenforced (
    self raiseImmutabilityError. )

  primat: idx put: aVal on: anObj = unenforced (
    self raiseImmutabilityError. )

  priminstVarAt: idx put: aVal on: anObj = unenforced (
    self raiseImmutabilityError. )

  "... and all other mutating operations"
)
```

Listing 5.1: Definition of a Domain for Immutable Objects

While the OMOP provides the capabilities to ensure immutability with such a domain definition, it is important to note that the semantics of immutability are only ensured during enforced execution, i. e., at the base level. As soon as execution moves to the unenforced mode, i. e., to the meta level, immutability is no longer guaranteed. Thus, a language implementer has to ensure that the language implementation, i. e., the meta-level code is correct. Furthermore, properties such as immutability or isolation that are enforced using the OMOP are not meant as a mechanism to enforce security policies of some sort. At this time, security has not been a design concern for the development of the OMOP.

5.3.2. Clojure Agents

Introduction Clojure agents, as introduced in Sec. 2.4.3, are the second example for demonstrating how the OMOP works. An agent represents a resource, i. e., a mutable cell, which can be read synchronously. However, state updates are only performed asynchronously by a single thread. Lst. 5.2 shows the implementation in SOM.

```
1  Agent = (
2    | state mailbox immutDomain |
3
4    read = unenforced ( ^ state )
5
6    "update blocks are of the form:
7       [:oldState | oldState produceNewState ]"
8    send: anUpdateBlock = unenforced (
9      mailbox nextPut: anUpdateBlock )
10
11   initialize = unenforced (
12     | domain |
13     mailbox     := SharedQueue new.
14     immutDomain := ImmutableDomain new.
15     domain      := AgentDomain new agent: self.
16     domain spawnHere: [
17       true whileTrue: [ self processUpdateBlock ]]. )
18
19   processUpdateBlock = (
20     | updateBlock newState |
21     updateBlock := mailbox waitForFirst.
22     newState    := domain evaluateEnforced: [
23                             updateBlock value: state].
24     state       := immutDomain adopt: newState.
25     mailbox removeFirst. ) )
```

Listing 5.2: Clojure agents implemented in SOM Smalltalk

For the purpose of this discussion, agents are represented by an object that has the field `state` to represent the state of the agent and the field `mailbox`, which holds incoming update requests. In addition to the object itself, an agent has an associated `Process`, i. e., thread, which evaluates the incoming update requests one by one. The `mailbox` is a `SharedQueue` object, i. e., a concurrent queue data structure. The queue implements `#waitForFirst` to block the current thread until the queue has at least one element and then return the element without removing it. This operation is complemented by `#removeFirst`, which removes the first element of the queue.

The example uses these operations to implement the basic methods of the agent. Thus, `#read` returns the current state and `#send:` enqueues an update request for the agent's state. The request is represented by `anUpdateBlock`, which is a lambda, i. e., a block, with a single argument. The argument is the old state, which is going to be replaced by the result the block returns.

The process that has been spawned during the initialization of the agent object tries indefinitely to process these update blocks in `#processUpdateBlock`. It waits for the next update block, evaluates it with the current state as argument, and sets the result value as new state.

This implementation is simplified but represents the essence of Clojure agents. The differences with the Clojure implementation sketched in Sec. 2.4.3 originate from language differences between SOM and Clojure. For instance, SOM does not have the notion of private methods, which leaves the implementation methods `#initialize` and `#processUpdateBlock` exposed to arbitrary use. Thus, these methods could be called from another thread and violate the assumption that only one update function is executed at a time. The remaining differences originate from the use of the OMOP. Note that the `#read`, `#initialize`, and `#send:` method are annotated with `unenforced`. They are by themselves not subject to the OMOP's enforcement, guaranteeing that it does not come to meta recursion during their execution.

Providing extended Guarantees based on the OMOP While Clojure guides developers to use agents in conjunction with immutable data structures, at the same time it makes the pragmatic decision not to enforce any such guarantees on top of the JVM. One reason for this is the anticipated performance impact. Another issue is the integration with Java itself, which makes it impossible to provide such guarantees consistently. This section uses this example to demonstrate how the OMOP can be used to express and in return enforce such guarantees concisely.

To this end, Lst. 5.3 defines the `AgentDomain`. The domain implements the intercession handler `#requestExecOf:on:with:` to ensure that only a single thread of execution modifies the agent and that the agent is not reinitialized. In this simple example, this is done by using a list of selectors that are allowed to be executed. In case any other selector is sent to an agent, an error is raised to report the violation.[5]

As the `ImmutableDomain` in Sec. 5.3.1 shows, adding the guarantee of immutability is also possible in a concise manner. Using the `ImmutableDomain` of Lst. 5.1 in this example, the agent ensures in `#processUpdateBlock` that the state is an immutable object. This is done by requiring the immutable domain to `#adopt:` the return value of an update block.[6]

[5] Our actual implementation of agents goes beyond what is presented here. For brevity, the example leaves out operations such as `#await` and `#send:with:`.

[6] In the current implementation `#adopt:` performs a shallow adopt only and thus, only the root object of the object graph is made immutable.

```
1  AgentDomain = Domain (
2    | agent |
3    "Use #agent: to set the agent that is protected by this domain"
4    agent: anAgent = ( agent := self adopt: anAgent )
5
6    requestExecOf: selector on: obj with: args = unenforced (
7      "Semantics are only enforced on the agent itself,
8       not on other objects created in its scope."
9      obj = agent ifFalse: [^ obj perform: selector with: args].
10
11     "Only allow execution of white-listed selectors"
12     (#read = selector or: [#send: = selector]) ifTrue: [
13       ^ agent perform: selector with: args ].
14
15     Error signal: 'Exec. of method ' + selector + ' is denied.' ))
```

Listing 5.3: Domain definition for an Agent, enforcing the expected guarantees.

5.4. Semantics of the MOP

Sec. 4.2 introduced the SOM language and the interpreter's execution model. Building on that foundation, this section describes the semantics of the OMOP by discussing its implementation in the SOM interpreter.

Derived from Fig. 5.1, Lst. 5.4 shows how the basic elements of the OMOP map onto the SOM interpreter. Every `SOMObject` has a field to refer to the owner domain. The domain field is initialized in the `#postAllocate` method after allocation is completed. The initial value, i. e., the owner domain is determined by the domain the thread is currently executing in. Since SOM is implemented as a stack-based interpreter, the current domain is derived from the current stack frame. The enforcement flag is realized as part of the interpreter's stack frames as well. Thus, every frame maintains an `enforced` flag and the current domain.

`SOMInvokable`, the superclass for methods and primitives, maintains the attribute that indicates to the interpreter that execution has to continue unenforced. Methods are annotated with `unenforced` instead of `enforced`, because applications and libraries need to execute in the enforced mode most of the time to benefit from the OMOP. Only the domain's intercession handlers and methods implementing language behavior require unenforced execution. Thus, it is a pragmatic decision to annotate the smaller set of methods that require unenforced execution.

```
SOMObject = ( | "..." domain | "..."
  postAllocate = (
    domain := interpreter frame domain
                initialDomainForNewObjects ) )

SOMFrame = SOMArray ( | "..." enforced | "..."
  enforced        = ( ^ enforced        )
  enforced: aBool = ( enforced := aBool ) )

SOMInvokable = SOMObject ( | "..." attribute | "..."
  unenforced = ( ^ attribute == #unenforced ) )

SOMUniverse = ( | globals symbolTable interpreter | "..."
  bootstrapFrameWithArguments: args = (
    (interpreter pushNewFrameWithMethod: self bootstrapMethod)
       push: (self globalAt: #system);
       push: args;
       enforced: false ) )
```

Listing 5.4: Structural Changes to support the OMOP in SOM

The execution of a program starts from a bootstrap frame created by the `#bootstrapFrameWithArguments:` method in `SOMUniverse`. The initialization of the bootstrap frame sets the execution mode to unenforced. This gives an application the freedom to set up the runtime environment properly before it continues with its normal execution in enforced mode. However, this also implies that a language implementation on top of the OMOP needs to *opt-in* to the enforced execution mode, which can make it prone to incorrect use by a language implementer.

Lst. 5.5 shows the adapted implementation of the `POP_FIELD` bytecode. During unenforced execution, the value that was on the top of the stack is stored directly into the receiver. When the OMOP's semantics are enforced however, it triggers the intercession handler by sending `#write:toField:of:` to the receiver's domain to reify the store operation. Note that the operations of the OMOP are executed with enforcement disabled. As shown in previous domain definitions such as Lst. 5.1 and Lst. 5.3, the OMOP operations are annotated with `unenforced`. The implementation of `#pushNewFrameWithMethod` will make sure that in these cases the `unenforced` flag overrides normal propagation of the enforcement flag of the frame.

Lst. 5.6 shows the corresponding read operation. It pushes the value of an object field onto the operand stack. The main difference between `POP_FIELD`

```
SOMInterpreter = (
  | frame universe currentBytecode |

  doPopField = (
    | omopWriteField args oldFrame value |
    value    := frame pop.
    frame enforced ifFalse: [
        ^ self currentObject
                fieldAtIndex: currentBytecode fieldIndex
                         put: value ].

    omopWriteField := self currentObject domain class
                          lookupInvokable: #write:toField:of:.
    oldFrame := frame.
    oldFrame pushAll: { self currentObject domain.
                        value.
                        currentBytecode fieldIndex.
                        self currentObject }.
    frame := interpreter pushNewFrameWithMethod: omopWriteField.
    frame copyArgumentsFrom: oldFrame )

  pushNewFrameWithMethod: method = (
    ^ frame := SOMFrame new
        method:             method;
        previousFrame:      frame;
        resetStackPointerAndBytecodeIndex;
        domain:             frame domain;
        enforced: (frame enforced and: [method unenforced not]);
        yourself )
"..." )
```

Listing 5.5: Reifying mutation of object fields

and `PUSH_FIELD` bytecode is that the latter uses its own OMOP intercession handler `#readField:of:`.

The bytecodes for sends and super sends themselves remain unchanged (cf. Sec. 4.2.2, Lst. 4.5). However, both bytecodes rely on adaptation of the actual send in the `#performSend:to:lookupCls:` method. Lst. 5.7 shows that `#performSend:to:lookupCls:` will send the message directly during unenforced execution, and it will rearrange the operand stack during enforced execution to be able to trigger the `#requestExecOf:with:on:lookup:` intercession handler on the receiver's domain instead.

While handling message sends implicitly covers the methods that represent the VM's primitives, they still need to be treated explicitly to redefine them

```
SOMInterpreter = ( doPushField = (
  | omopReadField args oldFrame |
  frame unenforced ifTrue: [
    ^ frame push: (self currentObject fieldAtIndex:
                        currentBytecode fieldIndex) ].

  omopReadField := self currentObject domain class
                     lookupInvokable: #readField:of:.
  oldFrame := frame.
  oldFrame pushAll: {self currentObject domain.
                     currentBytecode fieldIndex.
                     self currentObject }.
  frame := interpreter pushNewFrameWithMethod: omopReadField.
  frame copyArgumentsFrom: oldFrame )
"..." )
```

Listing 5.6: Reifying reading of object fields

```
SOMInterpreter = (
  performSend: selector to: receiver lookupCls: cls = (
    |  result args |
    frame unenforced ifTrue: [
      ^ self send: selector toClass: cls ].

    "Redirect to domain"
    args := frame popN: selector numArgs.
    frame pop;  "pops the old receiver"
      pushAll: {receiver domain. selector. args. receiver. cls}.
    result := self send: #requestExecOf:with:on:lookup:
                toClass: receiver domain class.
    ^ result )
"..." )
```

Listing 5.7: Perform reified message send

when necessary as part of the domain definitions. Hence, `#invokeInFrame:` is changed in Lst. 5.8 to trigger the intercession handler that corresponds to the primitive on the receiver's domain.

Note that globals need not be treated separately in SOM. The bytecode set does not include operations to change them, instead this functionality is provided via a primitive, and is thus already covered. Other VMs, such as the RoarVM require separate treatment of globals.

```
SOMPrimitive = SOMInvokable (
  invokeInFrame: frame = (
    | receiver omopPrim oldFrame |
    (self unenforced  or: [frame unenforced]) ifTrue: [
        ^ self invokePrimitiveInPlace: frame ].

    receiver := frame stackElementAtIndex: numArgs.
    omopPrim := receiver domain class
                    lookupInvokable: #prim, signature, #on:.
    frame stackElementAtIndex: numArgs
          put:                 receiver domain;
          push: receiver.
    oldFrame := frame.
    frame := interpreter pushNewFrameWithMethod: omopPrim.
    frame copyArgumentsFrom: oldFrame )
"..." )
```

Listing 5.8: Reifying primitive invocations

5.5. Customizations and VM-specific Design Choices

This section discusses a number of design choices that need to be considered when the OMOP is adapted for a concrete use case or a specific VM. Note that the design presented in Sec. 5.2 is a minimal representation of the key elements of the OMOP. Therefore, different interpretations of the OMOP are possible and can be desirable depending on the use case. This dissertation concentrates on the key elements with their minimal representation to evaluate the main idea and its applicability in the general case.

Representation of Ownership The proposed OMOP design uses a minimal set of intercession handlers, i. e., `#readField:of:`, `#write:toField:of:`, `#requestExecOf:on:with:lkup:`, as well as primitive handlers `#prim*`. An alternative design of the OMOP could for instance use the notion of ownership and project it onto the intercession handlers as well. Currently, the notion of ownership is only represented as a property of the object. By introducing it into the intercession handlers, an interpreter can perform the most common operations directly and potentially improve the performance for concurrency abstractions that define different rules based on ownership.

For example, a common use case is to guarantee properties such as isolation. To guarantee isolation, the OMOP needs to distinguish between an object being accessed by a thread executing inside the same domain and by a thread

executing in another domain. Consequently, an instantiation of the OMOP taking this use case into account could provide special intercession handlers for both cases. For reading fields, it could provide `#readFieldFromWithin:of:` and `#readFieldFromOutside:of:`. Similarly, the other intercession handlers could be provided in these two variants, as well. While the resulting OMOP would no longer be minimal, an implementation of isolation on top of it would not require testing for ownership but could rely on the VM, which might provide better performance.

In future work (cf. Sec. 9.5.6), the different properties of supported concurrent programming concepts can be studied and formalized, which could yield similar variation points and a declarative representation of possible language policies. Such a representation would again allow a different representation of key elements of the OMOP and allow more efficient implementation of common use cases.

Opt-In for Enforced Execution Mode Another design decision made in the presented OMOP is that execution starts out in unenforced mode. With the semantics discussed in Sec. 5.3.1 and Sec. 5.4 (cf. Lst. 5.4), a language implementer needs to *opt-in* explicitly to enforced execution. This design was chosen to provide predictable behavior and the ability to set up all relevant libraries and system parts before enforced execution mode is activated. This choice is partially motivated by the fact that the OMOP has been developed for an existing system with existing infrastructure. Thus, starting in unenforced execution mode provides more flexibility. However, in a system that is designed from the ground up as a multi-language VM with support for the OMOP, it might be desirable to execute code in the enforced execution mode from the beginning. The benefits of this choice would be that a language implementer does not need to *opt-in* to enforced execution and thus, the standard case will ensure that language semantics are ensured at all times.

Handling of Primitives Primitives, i. e., built-in functionality provided by the VM needs to be covered by the OMOP to guarantee that the semantics of a concurrent programming concept can be enforced in their entirety. Thus, if primitives are not covered correctly, they could be used, for instance to circumvent the isolation required between processes in CSP.

The presented OMOP design includes every single primitive as a separate intercession handler on the OMOP. This choice works well as part of the presentation in this dissertation and for VMs that have only a small number

of primitives that need to be covered. However, if the number of primitives grows too large, this design can become cumbersome.

An alternative solution to list all primitives in the form of `prim*` is to use a generic intercession handler similar to the one provided to handle all methods in a uniform way. Such a handler needs to encode the primitive as a parameter, for instance like `#requestPrim: prim on: obj with: arguments`.

Handling of Reified Execution State Since the presented implementations are based on Smalltalk, they have direct access to runtime state for instance to stack frames, i. e., context objects that are used for execution (cf. Sec. 4.2.2).

Context objects, and similar objects relevant for execution semantics, need to be treated carefully. For each of them, it has to be determined whether the notion of ownership should be provided or whether that would lead to unexpected consequences.

In the case of some context object it is conceivable that it has an arbitrary owner and that it is treated like any other object. As long as it is merely used for introspection, it is feasible to allow arbitrary ownership changes and the enforcement of arbitrary domain semantics. However, if a context object is used by the VM for its purpose as a stack frame, neither the notion of ownership nor the enforcement of arbitrary domain semantics are viable. The VM needs direct access to them, for instance to perform the necessary stack operations during execution.

Therefore, the implementations discussed in Chapter 7 regard context objects as metaobjects that are not subject to the OMOP.

5.6. Related Work

This section contrasts the proposed OMOP with other MOPs that cover closely related concerns or MOPs that have been used in the field of concurrent and parallel programming. As in Sec. 5.1, this section relies on the categorization of Tanter [2009] discussing metaclass-based, metaobject-based, as well as group-based hybrid approaches.

Metaclass-based Approaches The C++ extensions Open C++ [Chiba and Masuda, 1993] and PC++ [Stroud and Wu, 1995], which is built on top of Open C++, both use metaclass-based MOPs. Open C++ is used in the context of distributed programming and Chiba and Masuda [1993] give examples on

how to use it to implement synchronization semantics. Furthermore, they discuss its use for remote function calls. PC++ explores the implementation of transaction semantics on top of the same foundations.

They differ in two main points compared to the OMOP proposed here. On the one hand, Open C++ and PC++ exclusively rely on capturing method execution with the MOP. The OMOP exceeds this by including the notion of state access and the necessary extensions for VM-based implementations to cover global state and primitives. On the other hand, Open C++ and PC++ use a class-based meta relation. By relying on a meta relation that is built on ownership, the OMOP is more flexible and enables the use of classes in the context of different domains, i. e., with different meta semantics.

Another example of a metaclass-based MOP was proposed by Verwaest et al. [2011]. They enable the customization of memory layouts and object field accesses by reified object-layout elements. Using their approach the memory representation of instance fields can be adapted and for instance fields can be mapped to single bits in memory. The MOP's capabilities for refining the semantics of field accesses are similar to the OMOP. The main differences are that their MOP focuses on object layout and uses a different meta relation. While their meta relation is class-based, they introduce a new meta relation in addition to the existing metaclass. Each class has a specific layout metaobject, which describes the layout and field access semantics. However, while this approach decouples the metaclass from layout information, it remains coupled with a class and thus, it cannot be adapted based on the context an object is used in.

Metaobject-based Approaches One case of a metaobject-based approach that comes close to the OMOP is Albedo [Ressia et al., 2010]. Albedo enables structural and behavioral changes of objects via metaobjects. This approach provides the flexibility to change the structure of specific objects only, and also reifies events such as reading or writing of object fields and message sends to give a metaprogram the chance to respond to actions in the base program. Compared to the OMOP, Albedo does not provide conventional intercession, instead, it only provides events, to which a metaprogram can react. It does not provide the ability for instance to suppress the action that triggered the event. Changing Albedo to offer conventional intercession would enable it to provide a MOP that can satisfy the requirements for the OMOP. Albedo's approach of composed metaobjects could be used to implement the notion of ownership by providing a metaobject that applies to a set of objects. Thus,

while Albedo comes close, it would require changes to fulfill the requirements identified in Sec. 3.4.

The proxy-based metaobject protocol for ECMAScript [Van Cutsem and Miller, 2010] comes close to satisfying these requirements as well. It reifies all relevant intercession operations, however, since it is proxy-based, it does not provide the same guarantees the ownership-based approach provides. The use of proxies requires careful handling of references to avoid leaking the actual objects, which would then not be subject to an enforcement of language guarantees. On the other hand, the proxy approach provides a natural way to combine semantics of multiple metaobjects, since proxies can be chained. While the proxy-based approach has this advantage, the ownership-based approach simplifies the intended standard use cases. On the one hand, it does require only one metaobject, i. e., the domain object, to define the concurrency semantics for a group of objects, and on the other hand, it does not require distinguishing between proxies and target objects, which could be leaked, since the ownership property establishes the meta relation.

Grouping and Ownership-based Approaches The notion of *meta-groups* proposed by Mitchell et al. [1997] is close to the notion of ownership used by the OMOP. They use their meta-groups MOP to define real-time properties by providing the ability to reflect over timing-related information and to customize scheduling-related behavior. Their intention is to describe the semantics for groups of objects conveniently and to avoid the need for synchronization between multiple metaobjects, when scheduling decisions have to be made.

Similar notions have been used in ACT/R [Watanabe and Yonezawa, 1991] and ABCL/R2 [Masuhara et al., 1992]. The focus is on group communication and maintaining constraints among groups of objects. To this end, the MOP reifies message sends as *tasks*, the creation of new actors, the state change of existing actors via *become*, and the execution of a method body. In contrast to the OMOP, ACT/R supports infinite meta-regression in terms of an infinite reflective tower, while the OMOP uses the notion of *enforced* and *unenforced* execution to be able to terminate meta-regression explicitly. Another consequence of this design is the absence of ACT/R's inter-level communication, which requires special addressing modes when sending messages between the base and meta level. Instead, the `unenforced` flag and the `#evaluateEnforce:` method make the switch between the execution levels, i. e., enforcement mode explicit.

In ABCL/R2, meta-groups coordinate for the management of system resources, including scheduling. To overcome limitations of the metaobject and the meta-group approaches, both are combined in ABCL/R2. Similarly to the OMOP, ABCL/R2 also restricts each object to be member of one specific group, and thus has a strong similarity with the notion of domains. However, since the purpose of these metaobject protocols is different, they do not offer the same intercession handlers the OMOP provides. Hence, neither of the discussed group-based approaches fulfills the requirements identified in Sec. 3.4.

Terminology While the notion of meta-groups is very close to the notion of domains, this dissertation differentiates between both terms to make it more explicit that objects have a single owner, i. e., a domain. For groups, this is not necessarily the case. Other differences in the MOPs stem from the differences in the intended use case. The MOPs could be extended to include the intercession points provided by the OMOP, however the OMOP design minimizes the MOP to provide the necessary intercession points only.

Distinguishing Properties of the OMOP The main distinguishing characteristics of the OMOP are the combination of the specific set of intercession handlers, the notion of ownership, and the integration with the notion of concurrent execution, i. e., the inclusion of threads as a relevant concern for the metaobject protocol. This combination is to our knowledge novel and based on the previously identified requirements (cf. Sec. 3.4).

For example, it would be possible to achieve similar results by using the meta-group-based MOPs of ABCL/R2 or Mitchell et al. [1997] and combine them with the proposed intercession handlers for state access, execution, and primitives. Since Mitchell et al. [1997] also include the notion of threads in the MOP, all key elements of the OMOP are present and it could satisfy the requirements. However, on their own, without the proposed intercession handlers, neither of the two MOPs would cover all the required aspects for a multi-language environment, since for instance ABCL/R2 focuses on message sends between objects.

Unrelated Work: Ownership Types Clarke et al. [1998] proposed the notion of *ownership types* to tackle the problem of encapsulation in object-oriented programming languages by enabling a developer to describe rules on how and when pointers to objects can escape or be changed from an object that

is supposed to provide encapsulation. The proposed type system, and type systems extending this work, provide means to describe and restrict the structure of object graphs with types or annotations. These rules are typically defined with respect to an owning object, i. e., the encapsulating entity. The notion of *ownership domains* introduced by Aldrich and Chambers [2004] refines these mechanisms to decouple the notion of encapsulation and ownership for finer-grained control. Furthermore, a common topic of discussion is how immutability can be realized in such a setting [Leino et al., 2008; Östlund et al., 2008; Zibin et al., 2010].

However, while this dissertation also discusses the example of how to realize immutability, the solution proposed here is only related to ownership types and ownership domains by an overlap in terminology. Ownership types restrict the structure of object graphs, while the OMOP provides mechanisms for changing the semantics of state access and method execution in a concurrent context. The OMOP is a conventional MOP following the definition of Kiczales et al. [1991]. Thus, it provides *"the ability to incrementally modify the language's behavior and implementation"*. Ownership types on the other hand provide a static system to verify the correctness of aliasing operations.

Recent work proposed using a MOP as the foundation for a reflective implementation for an ownership type system [De Cooman, 2012]. The proposed MOP is different from the OMOP, because it captures aliasing related events instead of operations on objects. Thus, it captures the creation of an alias when parameters are passed, object references are returned from a function, or object fields are read and written. While there are similarities, the intended use case and the provided means are different and do not match the requirements identified in Sec. 3.4 for the OMOP.

5.7. Summary

This chapter introduced the design of an *ownership-based metaobject protocol* (OMOP) and situated it into the context of the related work.

First, it gave an overview of *open implementations* and different approaches to *metaobject protocols* to conclude that MOPs allow an incremental modification of language behavior and therefore are a good foundation to satisfy the requirements identified in Sec. 3.4.

Second, this chapter presented the design and properties of the OMOP. It uses the notion of ownership to determine how operations on objects and global resources are defined. The ownership notion is represented by a *domain*,

which owns a set of objects. The domain defines for its objects how field access and method execution are to be executed. This definition also includes thread resumption, primitives, and global state access to cover all aspects required to implement the semantics of concurrent programming concepts.

Third, it details the design of the OMOP by implementing two concurrent programming concepts: immutability and Clojure agents. It shows how the intercession handlers of the OMOP can be customized to provide the desired language behavior.

Fourth, it documents the semantics of the OMOP in the form of a bytecode-based interpreter. The interpreter is changed so as to check for every relevant operation whether it executes in the enforce mode. If this is the case, the interpreter triggers intercession handlers instead of performing the operation directly.

Finally, the differences between the OMOP and the related work are discussed to argue that the OMOP is novel but based on existing work.

6

EVALUATION: THE OMOP AS A UNIFYING SUBSTRATE

The goal of this chapter is to show that the proposed OMOP constitutes a unifying substrate for the implementation of concurrent programming concepts. To this end, it derives the evaluation criteria from the thesis statement and the research goal. These criteria assess the benefits of implementing concurrent programming concepts, the fulfillment of the requirements stated in Sec. 3.4, applicability, novelty, and the unifying properties of the OMOP. This chapter demonstrates the implementation benefits and fulfillment of the requirements based on three case studies and an evaluation of how the OMOP facilitates the implementation of programming concepts that require VM support to guarantee their semantics (cf. Sec. 3.2). The three case studies cover OMOP-based as well as ad hoc implementations of Clojure agents, software transactional memory, and event-loop actors. Based on this evaluation, the chapter demonstrates that the OMOP provides the necessary flexibility to host a wide range of concepts and enables a concise implementation of the case studies. Furthermore, it discusses the limitations of the OMOP. Overall, it concludes that the OMOP *enables the flexible definition of language semantics* as required by the thesis statement (cf. Sec. 1.3), i. e., it facilitates the implementation of concurrent programming concepts and thereby constitutes a unifying substrate for multi-language VMs.

6.1. Evaluation Criteria

In order to prepare the evaluation, this chapter first discusses the criteria used.

6.1.1. Evaluation Goal

The goal of this evaluation is to support the first two parts of the thesis statement (cf. Sec. 1.3). The last part of the thesis statement, i. e., the performance of the OMOP is evaluated in Chapter 8.

Consequently, this chapter needs to support the first part of the thesis statement, i. e., that *"there exists a relevant and significant subset of concurrent and parallel programming concepts that can be realized on top of a unifying substrate"*. To evaluate all parts of this statement, this chapter needs to argue that the subset of concurrent programming concepts supported by the OMOP is relevant, i. e., that it covers concepts that are used today, that these concepts are today not widely supported in VMs, and that these concepts benefit from VM support. Furthermore, it needs to argue that this set of concepts is significant, i. e., the supported set of concepts needs to be sufficiently large. Finally, it needs to argue that the OMOP is a unifying substrate, i. e., it makes an abstraction of the supported concrete programming concepts and provides a minimal set of elements to support them.

The second part of the thesis statement requires the evaluation to show that *"this substrate [i. e., the OMOP] enables the flexible definition of language semantics that build on the identified set of semantics"*. This means, this chapter needs to argue that the provided abstractions are powerful enough to allow variation over the supported concurrent programming concepts.

In the research goal, it is stated more concretely that the solution, i. e., the OMOP is to be designed based on a *to-be-determined set of requirements* (cf. Sec. 3.4). Therefore, this chapter needs to discuss how the OMOP fulfills these requirements. Furthermore, the research goal states that the evaluation needs to show applicability, i. e., it needs to demonstrate that the implementation of concurrent programming concepts on top of the OMOP does not have a negative impact on the implementation of these concepts.

6.1.2. Evaluation Criteria and Rationale

This section discusses the evaluation criteria for the identified goals. Furthermore, it discusses the rationale for each chosen criterion. Note that the order of this section reflects the overall order of the chapter.

Concurrent Programming Concepts benefit from the OMOP To demonstrate the benefits of the OMOP, this chapter argues that it simplifies the implementation of supported concurrent programming concepts by solving the problems implementers face when they use common ad hoc approaches. This discussion relies on the problems identified in Sec. 3.3. For each of the problems, it illustrates how it is solved using the OMOP and shows the improvements over existing ad hoc solutions. This comparison shows the benefits of the OMOP by showing that it solved the implementation challenges language implementers are facing when they target today's VMs.

The evaluation discusses how the OMOP addresses these problems as part of three case studies in Sec. 6.2. These case studies are the implementation of Clojure agents, an STM system, and event-loop actors. They are chosen to cover all of the identified problems and all mechanisms provided by the OMOP. Furthermore, the characteristics of the three implemented concurrent programming concepts are sufficiently different from each other to discuss the OMOP and the problems it addresses from different angles. Therefore, the case studies provide the necessary foundation to fully evaluate the OMOP and its benefits.

Fulfillment of Requirements Sec. 3.4 identified concrete requirements as a foundation for the design of the OMOP. This evaluation discusses these requirements as part of the three case studies in Sec. 6.2. For each case study the section evaluates which mechanisms the case study requires and how it maps onto the OMOP. Furthermore, to demonstrate the fulfillment of the requirements the discussion includes the evaluation criteria Sec. 3.4 stated for each of the requirements.

To complete the discussion of how the OMOP fulfills its requirements, Sec. 6.3 discusses the concepts from which the requirements were initially derived. For each of these concepts, it evaluates the degree to which it is supported by the OMOP. The evaluation differentiates between concepts where the main aspects are covered and concepts where only some aspects are supported by the OMOP to assess the extent to which the concepts benefit. Even if only some of the concept's aspects are supported, the benefits are substantial, since the OMOP addresses common implementation challenges.

Applicability Sec. 6.4 assesses the applicability of the OMOP by showing that its use does not increase the implementation size for the supported concurrent programming concepts.

Implementation size is often assumed to be an indication for implementation complexity, however, this assessment has an inherently social and subjective component. Therefore, the evaluation here considers only directly measurable aspects. It uses implementation size in terms of lines of code (LOC) as a surrogate measure for implementation complexity. As pointed out by a number of studies, size seems to correlate with other metrics to such a degree that it is not clear whether there is value in these additional metrics to assess complexity. Jay et al. [2009] found a strong correlation between cyclomatic complexity [McCabe, 1976] and lines of code. For their experiments, they go so far as to conclude that cyclomatic complexity has no explanatory power on its own. van der Meulen and Revilla [2007] found that Halstead Volume [Halstead, 1977] and cyclomatic complexity correlate directly with LOC on small programs. Emam et al. [2001] studied object oriented metrics and their relation to class size, i. e., LOC. They find that previous evaluations of metrics did not account for class size and that the effects predicted by these metrics can be explained based on size alone. They conclude that the value of these metrics needs to be reevaluated.

Since these indications cast doubt on the value of metrics other than size for assessing complexity, this evaluation uses only size-based metrics as a surrogate for measuring complexity to compare the ad hoc and OMOP-based implementations.

The second aspect of applicability is the ability to vary language guarantees. Sec. 6.5.1 discusses how the OMOP can be used to vary the semantics of concurrent programming concepts. The goal is to argue that it enables a wide range of variations covering a significant part of the design space spanned by the supported concurrent programming concepts. Based on the agents case study, the section argues that the additional semantic guarantees provided over Clojure agents are the kind of variations that are desirable and a good indication for the variability of the OMOP. The range of supported concepts is an other indication that supports the conclusion that the OMOP provides sufficient flexibility for the definition of language semantics for concurrent programming.

Relevance of supported Concepts To evaluate the relevance, Sec. 6.5.1 argues that the supported concepts are used today. It gives for each of the supported concepts an example of either recent research in the corresponding area, or a programming language that is respected and used in industry. Based on these indications for actual use, it argues that each of the supported

concepts has practical relevance. Consequently, the concepts supported by the OMOP are relevant.

Absence of wide Support in Today's VMs In order to evaluate the novelty of the OMOP, i. e., to assess whether these concepts are not yet widely supported in VMs, Sec. 6.5.1 relies on the VM survey in Sec. 3.1. For each concept, this analysis assesses whether the concept is supported. Furthermore, the analysis considers the whole set of concepts and its support by a single VM to validate the initial premise that today's VMs support is insufficient. By showing based on the VM survey that this is the case, it supports the claim that the OMOP can add relevant support to today's VMs.

Significance In order to evaluate the significance of the support the OMOP provides, Sec. 6.5.1 argues that the OMOP facilitates the implementation of all concepts that were identified as benefiting from VM support for semantic enforcement. Since the OMOP covers all concepts, there is an indication that the supported set is sufficiently large to warrant interest.

Unifying Substrate Finally, Sec. 6.5.1 discusses the unifying properties of the OMOP. The goal is to argue that the OMOP makes an abstraction of concrete programming concepts. The section shows this by relating the concepts to the mechanisms the OMOP exposes and by demonstrating that it is not a one-to-one relationship between mechanisms and concepts. Furthermore, it argues that the OMOP provides a minimal set of elements to fulfill the requirements. It demonstrates that the proposed design in its current form is minimal by arguing that none of the parts can be removed without reducing the number of concurrent programming concepts that can benefit from it, and that removing any part would also result in unsatisfied requirements.

6.2. Case Studies

This section discusses the case studies covering Clojure agents, software transactional memory, and event-loop actors.

In order to show that the OMOP simplifies the implementation of these concurrent programming concepts, it demonstrates how the OMOP is used to solve common implementation challenges. Tab. 6.1 recapitulates these challenges, which Sec. 3.3 discussed in more depth.

Furthermore, this section shows that the OMOP satisfies the requirements stated in Sec. 3.4. Tab. 6.2 recapitulates these requirements and the concrete properties an implementation needs to facilitate. Based on this table, this section demonstrates that the OMOP satisfies the requirements for MANAGED STATE, MANAGED EXECUTION, a notion of OWNERSHIP, and CONTROLLABLE ENFORCEMENT.

For each case study, this section gives a brief introduction to the implementation and then details the ad hoc as well as the OMOP-based implementation. In order to evaluate the OMOP, it summarizes for each case study how the OMOP supported the solution of common implementation challenges.

Table 6.1.: Common Challenges for the Implementation of Concurrent Programming Concepts on top of Multi-language VMs.

Enforcement of	
Isolation	• challenge to guarantee *state encapsulation* and *safe message passing* • *by-value* semantics problematic without proper ownership transfer
Scheduling Policies	• custom scheduler needs control over executed code • computational and primitive operations are problematic
Immutability	• used as as workaround to track mutation • reflection should obey it, when required
Execution Policies	• reflection should obey them, when required • implementation can be challenging without notion of ownership
State Access Policies	• reflection should obey them, when required • primitives need to be manageable to cover all state access • implementation can be challenging without notion of ownership

6.2.1. Clojure Agents

Introduction The main idea of Clojure agents (cf. Sec. 2.4.3) is to ensure that only a single process, the process executing the agent's event-loop, can write the agent's state and to enforce immutability of all data structures referenced by an agent. To that end, an agent asynchronously receives update functions and processes them one by one. Note that Sec. 5.3.2 used agents earlier as an example to introduce the OMOP.

In Clojure, the implementation is constructed in a way such that it is not possible to change the state by other means than by sending asynchronous

Table 6.2.: Requirements for a Unifying Substrate for Concurrent Programming, and the concrete concepts an implementation needs to facilitate.

Requirement	*Implementation needs to facilitate*
Managed State	isolation, immutability, reified access to object fields and globals
Managed Execution	asynchronous invocation, scheduling policies, interception of primitives
Ownership	definition of policies based on ownership
Controlled Enforcement	flexible switching between enforced and unenforced execution

update functions to the agent. However, if reflection is used the agent's state field can be changed, which violates the assumption of asynchronous updates. Furthermore, Clojure does not provide guarantees that the object graph that forms the agent's state is immutable. Thus, data races can occur even though it is generally suggested to rely on immutable data structures for the agent's state, in order to form an immutable object graph.

This implementation and the one in Sec. 5.3.2 use the OMOP to extend Clojure's notion of agents and provide the expected guarantees. First, this section briefly recapitulates the agent implementation, and then discusses the enforcement of asynchronous updates and immutability in the context of implementation challenges and the OMOP's requirements.

General Implementation Lst. 6.1 repeats the implementation of agents in SOM from Sec. 5.3.2. The main elements are the state field, the field for the mailbox, the `#read` method, and the `#send:` method for update blocks, i. e., anonymous functions. These methods are the interface to be used by a developer. In addition, the implementation has an `#initialize` method, which for instance creates the mailbox object for incoming update requests, and the `#processUpdateBlock` method, which processes the update requests one by one in its own thread, i. e., Smalltalk process.

Note that Lst. 6.1 already includes the code to use the OMOP domains to ensure the desired guarantees. If these operations are imagined to be without any effect, the implementation corresponds to an agent implementation without guarantees.

```
Agent = (
  | state mailbox immutDomain |

  read = unenforced ( ^ state )

  "update blocks are of the form:
     [:oldState | oldState produceNewState ]"
  send: anUpdateBlock = unenforced (
    mailbox nextPut: anUpdateBlock )

  initialize = unenforced (
    | domain |
    mailbox     := SharedQueue new.
    immutDomain := ImmutableDomain new.
    domain      := AgentDomain new agent: self.
    domain spawnHere: [
      true whileTrue: [ self processUpdateBlock ]]. )

  processUpdateBlock = (
    | updateBlock newState |
    updateBlock := mailbox waitForFirst.
    newState    := domain evaluateEnforced: [
                            updateBlock value: state].
    state       := immutDomain adopt: newState.
    mailbox removeFirst. ) )
```

Listing 6.1: Clojure agents implemented in SOM Smalltalk

Basic Guarantees The agent implementation of Clojure 1.4 uses an `Agent` class implemented in Java. This class has a private[1] field and setter for the agent's state. Thus, in normal circumstances the state of an agent is protected from race conditions. However, Java's reflection API enables modification of the Agent's state and thus, it could come to race conditions. Therefore, Java's agent implementation is reasonably protected as long as neither reflection is used, nor the object graph representing the state is modified arbitrarily.

The Smalltalk-based agent implementation has slightly different characteristics. While Smalltalk fields are only accessible from methods of the same object and thus protected, methods are public and modifiers to restrict access are not supported. It is therefore possible to invoke the `#processUpdateBlock` method directly, which could result in a corruption of the agent's state, because two threads can update it at the same time.

[1] In Clojure 1.4 the `state` field and the `setState(Object)` method of the `Agent` class have default visibility, but we will assume that they are private to simplify the argument.

Ad Hoc Solutions An ad hoc solution to prevent such issues would be to use a mutex inside `#processUpdateBlock`, which ensures that the body of the method, and thus the update is done only by one thread at a time. While this solution is straightforward and also covers reflective invocations of the method, it has an inherent performance overhead. Furthermore, reflective updates of the `state` field would still be possible. To make reflective updates safe, the `Agent` class could override the reflective methods `#instVarAt:` and `#instVarAt:put:` to use the mutex that also protects `#processUpdateBlock`.

A Java solution would require more elaborate approaches. One possible approach could utilize a custom `SecurityManager`,[2] however, because of its design with `setAccessible()` (cf. Sec. 3.3.5) a solution could not completely prevent race conditions, when reflection is used.

OMOP-based Solution Compared to ad hoc approaches, an OMOP-based solution can provide stronger guarantees. Lst. 6.2 repeats the implementation of the `AgentDomain` from Sec. 5.3.2. This domain is used in Lst. 6.1 on line 15. This leads to the agent object being adopted by the `AgentDomain` and thus, all operations on this object become subject to the adapted semantics of the domain. In this case it means that all method invocations on the object are reified and handled by the `#requestExecOf:on:with:` handler. For the agent object, this handler enforces that only `#read` and `#send:` can be performed directly. For a Smalltalk-based implementation, this approach handles all of the issues of the ad hoc solution. Thus, `#processUpdateBlock` can only be invoked by the agent's process and reflection is not possible on the agent either.

The agent's process executes in unenforced mode, and thus, is not subject to the restrictions of the domain. However, note that the domain evaluates the update block in enforced mode at line 22. Thus, the application-level update operations are performed with semantics enforced.

When applying the OMOP in a Java context, the primitives that realize the reflective capabilities would need to be handled as well. Since Java's reflection API is not part of the object, these operations are not covered by handling method invocations on the object.

To conclude, using the OMOP a domain such as the `AgentDomain` can provide the basic guarantees for Smalltalk as well as a Java-like languages.

[2] http://docs.oracle.com/javase/7/docs/api/java/lang/SecurityManager.html

```
AgentDomain = Domain (
  | agent |
  "Use #agent: to set the agent that is protected by this domain"
  agent: anAgent = ( agent := self adopt: anAgent )

  requestExecOf: selector on: obj with: args = unenforced (
    "Semantics are only enforced on the agent itself,
     not on other objects created in its scope."
    obj = agent ifFalse: [^ obj perform: selector with: args].

    "Only allow execution of white-listed selectors"
    (#read = selector or: [#send: = selector]) ifTrue: [
      ^ agent perform: selector with: args ].

    Error signal: 'Exec. of method ' + selector + ' is denied.' ))
```

Listing 6.2: Domain definition for an Agent, enforcing the expected guarantees.

Handled Challenges In order to provide the basic guarantees of Clojure agents, the implementation solves the challenge to define custom *execution policies*. It relies on the intercession handling of methods to restrict the set of methods that can be invoked on the agent object. This includes also reflective operations, as long as they are performed during enforced execution.

The intercession handler `#requestExecOf:on:with:lkup:` in conjunction with handlers for primitives satisfies the MANAGED EXECUTION requirement, because all execution of methods and primitives can be intercepted and customized. Note that by enabling inception of all operations, the OMOP avoids the restrictions in expressiveness that for instance Java's `SecurityManager` is subject to.

Furthermore, the flexibility provided by distinguishing between enforced and unenforced execution allows a language implementer to define precisely when semantics are enforced. One example is given in Lst. 6.1. The implementation of `#processUpdateBlock` uses the `#evaluateEnforced:` method of the domain to ensure that application-level update functions are evaluated with semantic enforcement enabled. In addition, with the `unenforced` attribute of methods, it is possible to restrict enforced execution flexibly. Thus, with the `AgentDomain` this case study demonstrates that the OMOP also satisfies the requirement for CONTROLLABLE ENFORCEMENT.

Immutability In an attempt to go beyond the guarantees provided by Clojure's agent implementation, Sec. 5.3.2 discussed how immutability can be guaranteed.

Ad Hoc Solution Enforcing immutability on top of a VM that neither supports it directly nor provides mechanisms for managing mutation is a challenge. While Java's `SecurityManager` can provide a partial solution, it cannot be used to enforce immutability for public object fields (cf. Sec. 3.3.5). Thus, an ad hoc solution would require implementing managed mutation including support for handling primitives. This could potentially be done using AOP for instance with AspectJ,[3] or program transformation as it is used for the implementation of the OMOP (cf. Sec. 7.1). Thus, it takes additional mechanisms to realize immutability on top of a JVM or Smalltalk for instance.

OMOP-based Solution With the OMOP's support for MANAGED STATE, the foundation is already available, and an `ImmutableDomain` such as given in Lst. 6.3 can be defined. The `ImmutableDomain` adapts field access operations by customizing `#write:toField:of:`. This prevents any mutation from succeeding for any object, i. e., any object that is owned by the immutable domain. Note that on line 24 of Lst. 6.1, the `ImmutableDomain` adopts the return value of an update function. Furthermore, note that it handles the issue of reflection by customizing not only the writing of fields but also all primitive operations, which includes the primitives used for reflection, e. g., `#priminstVarAt:put:on:`.

Handled Challenges With this case study, the evaluation showed solutions for two of the common implementation challenges (cf. Tab. 6.1). It uses the OMOP's mechanisms to intercept all mutating operations, i. e., writing to object fields and primitives performing mutation. This is an example of a solution to provide *immutability* also against reflective operations when desired, and it shows how the problems of defining *state access policies* can be solved.

Furthermore, this case study demonstrates that the OMOP satisfies the requirements for MANAGED STATE, because it reifies all state access and enables its customization.

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

```
ImmutableDomain = Domain (

  raiseImmutabilityError = (
    ImmutabilityError signal: 'Modification of object denied.' )

  write: val toField: idx of: obj = unenforced (
    self raiseImmutabilityError. )

  primat: idx put: aVal on: anObj = unenforced (
    self raiseImmutabilityError. )

  priminstVarAt: idx put: aVal on: anObj = unenforced (
    self raiseImmutabilityError. )

  "... and all other mutating operations"
)
```

Listing 6.3: Definition of a Domain for Immutable Objects

6.2.2. Software Transactional Memory

Introduction Software Transactional Memory (STM) (cf. Sec. 2.4.3) promises a solution to the engineering problems of threads and locks. This programming model enables a programmer to reason about all code as if it were executed in some sequential order. The runtime tries however to execute threads in parallel while giving the strong correctness guarantees implied by ensuring sequential semantics. While common consensus [Cascaval et al., 2008] seems to be that the performance overhead of STM systems is too high for many practical applications, the idea continues to be appealing.[4] Thus, it is used as one of the case studies.

This evaluation uses LRSTM (Lukas Renggli's STM), which is based on the implementation described by Renggli and Nierstrasz [2007] (cf. Sec. 7.1.3). The ad hoc version of LRSTM is a port of the original implementation to the current version of Squeak and Pharo. It uses the original approach, i. e., it uses abstract syntax tree (AST) transformation to add the tracking of state access and to adapt the use of primitives to enable full state access tracking. The implementation details are discussed below.

[4] The performance of an STM depends also on the programming language it is applied to. Languages such as Haskell and Clojure restrict mutation and thereby experience lower overhead from an STM than imperative languages with unrestricted mutation, where the STM needs to track all state access.

Important for this evaluation is that both, the ad hoc as well as the OMOP-based implementation provide the same guarantees.

General Implementation The general idea behind the implemented STM is the same in both cases. To be precise, both of our implementations share the main code parts, only the code that realizes the capturing of field accesses is different.

For the sake of brevity, the evaluation disregards LRSTM's support for nested transactions, error handling, and other advanced features. The implementation discussed here is a sketch of a minimal STM system that enables atomic execution of transactions.

The implementation sketch is given in Lst. 6.4. It builds on top of the ability to change all state access operations to work on a `workingCopy` of an object, instead of working on the object directly. To this end, in the context of a transaction each object can obtain a working copy for itself. A transaction maintains a set of changes, represented by `Change` objects. These change objects maintain the connection between the working copy and the original object, as well as a copy of the original object. Thus, all operations in the scope of a transaction result in read and write operations to the working copy of an object instead of changing the original.

When a transaction is to be committed, all threads are stopped and the `#commit` operation checks for conflicting operations. `#hasConflict` checks whether the original object has changed compared to when the current transaction accessed it the first time. If no conflicts are found, the changes made during the transaction are applied, and the commit succeeds.

Managing State Access The main mechanism both implementations need to provide is an interception of all state access operations. Thus, all field reads and writes need to be adapted to operate on a working copy instead of the original object. Furthermore, all primitives that read from or write to objects need to be adapted in a similar way.

Ad Hoc Solutions Renggli and Nierstrasz [2007] proposed to use program transformation to weave in the necessary operations. Instead of using an AST transformation as done here (cf. Sec. 7.1), such adaptions could be applied with higher-level approaches such as aspect-oriented programming (cf. Sec. 7.1.3). The main requirement is that all field accesses are adaptable.

```
Object = ( | transaction | "..."
  workingCopy = (
    "Answer a surrogate for use within current transaction."
    | transaction |
    transaction := Processor activeProcess currentTransaction
        ifNil: [ self error: 'No active transaction' ].
    ^ (transaction changeFor: self) working ) )

Change = (
  | original working previous |
  initializeOn: obj = (
    original := obj.
    previous := obj shallowCopy.
    working  := obj shallowCopy )

  apply = ( original copyFrom: working "copies all fields")

  "compares all field pointers"
  hasConflict = ( (original isIdentical: previous) not)

  ---- "class side initializer"
  on: obj = ( ^ self new initializeOn: obj ) )

Transaction = (
  | changes domain | "Transactions run in a give domain"
  "..."
  begin = ( changes := IdentityDictionary new )
  changeFor: obj = ( ^ changes at: obj
                              ifAbsentPut: [Change on: obj] )

  commit = unenforced (
    "Commit a transaction atomically"
    domain stopTheWorldExcept: Processor activeProcess.

    changes do: [ :each |
        each hasConflict ifTrue: [
            Error signal: 'Transaction aborted' ] ].

    "No conflicts, do commit"
    changes do: [ :each | each apply ].

    domain resumeTheWorldExcept: Processor activeProcess ) )
```

Listing 6.4: Sketch of the STM Implementation

The primitives require two different solutions for simple and for more complex functionality. For simple primitives, it is sufficient to ensure that the object they are applied to is the working copy. For more complex primitives that for instance modify collections and their elements, this solution is not sufficient, because they could obtain object references to the original objects and access those. Therefore, a solution needs to be able to provide substitutes for primitives. Renggli and Nierstrasz use Smalltalk reimplementations that emulate the primitives and thus were subject to the changes that adapt state access. Note that this approach works for primitives that are solely used for performance reasons. It does not work for functionality that can not be expressed in Smalltalk.

OMOP-base Solution The implementation of LRSTM on top of the OMOP defines the `STMDomain` in Lst. 6.5. It customizes the domain's intercession handlers `#readField:of:` and `#write:toField:of:` to redirect state access to the working copy. Furthermore, it intercepts all primitives that access state, including reflection primitives, to redirect them to the working copy as well.

As mentioned some primitives require a reimplementation because they perform complex operations on the given object. One example is the primitive for `#nextPut:`, which operates on writeable streams, i. e., collections that store sequences of elements and can grow if required. Lst. 6.5 implements the intercession handler for this primitive at line 20. Instead of merely obtaining the working copy of the stream, as is done by the other primitives, it uses a reimplementation of the primitive and executes it with enforcement enabled. Lst. 6.6 shows the reimplementation of the primitive in Smalltalk. This reimplementation enables the OMOP to track all state access as required for the STM.

A minor difference with the ad hoc version is that the OMOP-based implementation customizes `#requestThreadResume:` to register all threads that start executing inside the STM domain. Lst. 6.5 shows at line 25 how it is used to add threads to a set. The ad hoc implementation just assume that all threads are required to be stopped and iterates over all instances of the Smalltalk `Process` class.

Handled Challenges The main challenges handled by the OMOP are to provide custom *execution policies* for primitives, which allows them to be reimplemented, and to provide custom *state access policies*. Both need to be enforced against reflective operations to enable safe use of reflective field updates as

```
STMDomain = Domain ( | processes |
  readField: idx of: obj = unenforced (
    ^ obj workingCopy instVarAt: idx )

  write: val toField: idx of: obj = unenforced (
    ^ obj workingCopy instVarAt: idx put: val )

  primat: idx on: obj = unenforced (
    ^ obj workingCopy at: idx )

  primat: idx put: val on: obj = unenforced (
    ^ obj workingCopy at: idx put: val )

  priminstVarAt: idx on: obj = unenforced (
    ^ obj workingCopy instVarAt: idx )

  priminstVarAt: idx put: val on: obj = unenforced (
    ^ obj workingCopy instVarAt: idx put: val )

  primNext: stream put: val  = unenforced (
    ^ self evaluateEnforced: [stream noprimNextPut: val ] )

  "... and all other primitives that access state ..."

  requestThreadResume: process = unenforced (
    processes add: process.
    ^ process resume )

  initialize = ( processes := WeakIdentitySet new )

  stopTheWorldExcept:   proc = (
    (processes copyWithout: proc) do: [:each | proc suspend ] )

  resumeTheWorldExcept: proc = (
    (processes copyWithout: proc) do: [:each | proc resume  ] ) )
```

Listing 6.5: Definition of a Domain for an STM

```
WriteStream = Stream ( | position writeLimit collection |
  noprimNextPut: obj = (
    position >= writeLimit
        ifTrue:  [ ^ self pastEndPut: obj]
        ifFalse: [ position := position + 1.
                   ^ collection at: position put: anObject ] ) )
```

Listing 6.6: Primitive Reimplemented in Smalltalk to enable State Access Tracking for STM

part of a transaction. The OMOP provides the corresponding intercession handlers to enable the necessary customization without the need for using for instance AST transformations directly. The intercession handlers used for the STM satisfy the requirements for MANAGED STATE and MANAGED EXECUTION. In addition, the mechanisms to satisfy CONTROLLED ENFORCEMENT provide the necessary flexibility to reimplement primitives in Smalltalk for instance.

6.2.3. Event-Loop Actors: AmbientTalkST

Introduction The third case study implements event-loop actors[5] once with an ad hoc approach similar to the one used by JCoBox [Schäfer and Poetzsch-Heffter, 2010] and once based on the OMOP. The concurrency model is inspired by AmbientTalk [Van Cutsem et al., 2007], but this implementation provides only isolation properties and asynchronous message send semantics between actors (cf. Sec. 2.4.4). Other aspects such as complete libraries and support for non-blocking interactions are disregarded.

Basic Guarantees The case study implements the notion of event-loop actors that are containers for an object graph. An actor provides a number of guarantees for all objects it contains. Essential to the notion of an event-loop is that only the actor performs operations on the objects it owns. This eliminates low-level data races. All other actors in the system have to communicate with the actor via asynchronous messages to affect its objects.

Thus, an implementation needs to enforce that no direct operations are performed on objects owned by other actors. The operations have to be reified and added to the event queue of the actor. The actor processes events one

[5] The implementation is referred to as AmbientTalkST despite its lack of many of the essential features of AmbientTalk, because the name is a convenient metaphor to describe its main semantics of event-loop actors with proper isolation.

by one. Both of the presented implementations guarantee this basic aspect. Furthermore, both implementations guarantee the notion of *safe messaging* (cf. Sec. 3.3.2). Thus, messages exchanged between actors are guaranteed not to introduce shared state.

State encapsulation on the other hand is not supported by either of the implementation, even though it would be straightforward with the OMOP.

Ad Hoc Solution The ad hoc implementation strategy is similar to the one described by Schäfer and Poetzsch-Heffter [2010], but kept much simpler. Similar to their approach of wrapping objects, we use stratified [Bracha and Ungar, 2004] proxies [Van Cutsem and Miller, 2010] to represent *far references*, i. e., references between objects residing in different actors. The proxy wraps actual objects and reifies all operations on them to enforce *safe messaging* and asynchronous execution. This approach also provides coverage for Smalltalk's reflection, because it is based on message sends to the object.

Safe messaging is realized by wrapping all objects in parameters and return values. Thus, this implementation comes with the performance drawbacks described in Sec. 3.3.2. Each object needs to be wrapped with another object. However, it avoids the need for copying the entire object graph.

While not implemented, this approach can be extended to provide ownership transfer semantics upon message send. With Smalltalk's `#become:` it is possible to ensure that the original actor does not hold on to the object it sent via a message. However, the `#become:` operation is expensive. It scans the whole heap to exchange object references of the receiver with the reference given as an argument. Furthermore, the `#become:` operation is not supported by VMs such as the JVM and CLI, and, it cannot be used as a general implementation strategy.

Unfortunately, the approach of using proxies, i. e., wrapper objects does not account for global state. Thus, class variables are shared between actors and introduce undesired shared state. In Java, this could be avoided by using different class loaders for each actor. However, in Squeak/Pharo there are no facilities which could provide such functionality. Since the implementation with the described features is already larger than the OMOP-based implementation, we decided to accept the limitation of partial state encapsulation only.

OMOP-based Solution The OMOP-based implementation uses the `ATActor` domain as the actor itself. Thus, a domain instance corresponds to an execut-

ing actor. Upon creation, `#initialize` is invoked, which starts the actor's thread, i. e., Smalltalk process, to execute inside itself by using `#spawnHere`.

The implementation realizes *safe messaging* by adapting the semantics of method executions. All requests from outside the domain, i. e., from outside the actor, are served asynchronously (line 32). The result of such asynchronous sends is represented by a `Promise` (line 23). In this simple implementation, the sending actor blocks on the promise if it requires the result of the message, and continues once the promise is resolved, i. e., once the result is delivered when the event-loop processes the message (line 7).

All requests inside the actor are performed directly (line 29). Note that this listing does not include definitions for field accesses, because Smalltalk offers object-based encapsulation (cf. Sec. 4.2.1). This object-based encapsulation guarantees that object fields can be accessed synchronously from within an actor only. In languages like Java, we would also need to take care of fields, since Java's encapsulation is class-based. Thus, we would need to customize `#readField:of:` and `#write:toField:of` as well.

Wrapping of the objects sent via messages is unnecessary, because the ownership notion and the corresponding domain of an object guarantee the actor properties implicitly. The OMOP's `#adopt:` method can also be used to change the owner of an object as part of the message send, if desired. By changing the owner, it becomes unnecessary to scan the full heap to ensure that other references to the object are no longer valid, because isolation is ensured by the new owner domain, i. e., actor. For the moment, ownership transfer needs to be used manually by asking the receiver domain to `#adopt:` the received object, and therefore it is in the application code but not part of the definition in Lst. 6.7.

Note that Lst. 6.7 contains only a sketch and does not take care of *state encapsulation* in terms of global state, however, the `ATActor` domain can customize `#readGlobal:` and `#write:toGlobal:` to make sure that full isolation is guaranteed between actors. Thus, global state of classes could be replaced with domain local copies by customizing these two intercession handlers. This would be very similar to the working copies used in the STM implementation (cf. Sec. 6.2.2).

While Sec. 3.3.3 discussed the issue of scheduling guarantees, Lst. 6.7 does not include a corresponding solution. As summarized in Tab. 6.1, the main issue is the missing control over the execution of code. Specifically, custom schedulers built on top of a VM typically do not have the ability to reschedule an actor when it is executing computationally expensive code or primitives. With the OMOP, the reification of method invocations and primitives pro-

```
ATActor = Domain (
  | process mailbox terminate |

  processIncomingMessages = unenforced (
    | msg |
    msg := mailbox next.
    msg deliverPromise: msg send )

  initialize = (
    mailbox   := SharedQueue new.
    terminate := false.
    process   := self spawnHere: [
          [terminate] whileFalse: [
              self processIncomingMessages ]] )

  sendAsAsyncMessage: aSelector to: anObj
    with: anArgArray inLookupClass: cls = (
      | msg |
      msg := ATMessage new.
      msg setSelector: aSelector arguments: anArgArray.
      msg receiver: anObj lookupClass: cls.
      mailbox nextPut: msg.
      ^ msg promise )

  requestExecOf: aSelector with: args
    on: anObj lookup: cls = unenforced (
    | msg |
    self == Domain currentDomain ifTrue: [
      ^ anObj performEnforced: aSelector
              withArguments:   args inSuperclass: cls ].

      ^ self sendAsAsyncMessage: aSelector
             to: anObj with: args inLookupClass: cls )

  requestThreadResume: process = unenforced (
    ThreadResumptionDenied signal ) )
```

Listing 6.7: Definition of the ATActor Domain for Event-Loop Actors

vide additional points where a scheduler can hook in to enforce scheduling semantics. While it is beyond the scope of this dissertation to compare this approach with the approach of using a monitoring thread as proposed by Karmani et al. [2009] (cf. Sec. 3.3.3), additional intercession points provided by the OMOP provide additional control that can ease the enforcement of desired scheduling properties.

Handled Challenges The implementation of complete *isolation*, i. e., *state encapsulation* and *safe messaging* are commonly the most challenging aspects as Sec. 3.3 argues.

The OMOP-based implementation required all of the offered mechanisms. It uses OWNERSHIP to distinguish local method executions and state accesses from remote ones. In conjunction with the intercession handlers to MANAGED EXECUTION, the domain definition enforces that remote message sends are executed using the actor's event-loop only. This also covers reflective operations. The mechanisms provided to satisfy the requirements for MANAGED STATE enable a domain to ensure that in languages that do not provide object-based encapsulation only the owning actor can access state. An example where the domain uses the mechanisms for CONTROLLABLE ENFORCEMENT is the message queue of an actor. Since it is owned by the domain representing the actor, direct access to the message queue is not possible. Any kind of access while the actor semantics are enforced would lead to a recursion in the access handler. However, the domain can disable enforcement by executing the code at the meta level, and modify the message queue as desired.

As described, this case study relies on the notion of ownership to implement custom *execution policies* and *state access policies*. Arguably, the explicitly available notion simplified the definition of these policies. Furthermore, it enables ownership transfer, which is a challenge on other platforms.

6.2.4. Conclusion

This section shows that Clojure agents, software transactional memory, and AmbientTalk's actors can be implemented on top of the OMOP. Furthermore, it demonstrates how the case study implementations map onto the OMOP and how they relate to common problems in Tab. 6.1 and the requirements in Tab. 6.2.

The agent case study shows how problems related to *immutability* can be solved and how the OMOP helps to customize *execution policies*. The STM case study shows how *state access policies* are customized and how primitives can

be adapted. The actor case study covers the remaining challenges of *isolation*, *scheduling policies*, and ownership. For each of these, this section either provides an example implementation or discusses strategies on how the OMOP can be used to solve the issues.

Furthermore, the case studies demonstrate how the intercession handlers of the OMOP satisfies the requirements for MANAGED STATE and MANAGED EXECUTION. The evaluation shows how the OMOP's support for ownership and object adoption can be used to define policies and therefore satisfy the OWNERSHIP requirement. Finally, it demonstrates how `unenforced` methods and the `#evaluateEnforced:` of the OMOP can be used to control precisely when its semantics are enforced. Thus, the OMOP also satisfies the last of the requirements, i. e., it provides CONTROLLED ENFORCEMENT.

6.3. Supported Concepts

The goal of this section is to evaluate to which degree the various concepts for concurrent programming are supported. This evaluation focuses on the concepts for which Sec. 3.2 states that they benefit from VM support for their semantics. Thus, this section concentrates on the set of concepts that are the focus of this dissertation.

For each of the concepts, the evaluation distinguishes whether all major aspects of a concept are covered or whether only some aspects are supported by the OMOP. This differentiation allows the evaluation to assess the extent to which the concepts benefit. The benefits are the result of the OMOP's facilities that enable language implementers to address common implementation challenges. Therefore, the benefits the OMOP provides are substantial even if not always all of the concept's aspects are supported.

The survey in Sec. 3.2 identified 18 concepts that could benefit from VM support to enforce semantics (SEM). Tab. 6.3 shows these concepts (cf. Tab. 3.4) with the assessment of the degree to which the OMOP facilitates the implementation of a concept. An X indicates that the major semantics of a concept can be directly expressed using the OMOP, while a + indicates that it is only partially supported. The remainder of this section discusses the results in more detail.

6.3.1. Supported Concepts

The first nine concepts listed in Tab. 6.3 as *supported* are the concepts Sec. 3.4 identified as benefiting most from extended VM support. Previously, Sec. 6.2

Table 6.3.: Concepts Supported by the OMOP: Supported by OMOP (X), Partial Support by OMOP (+)

Supported	SEM	OMOP	*Supported*	SEM	OMOP
Active Objects	X	X	No-Intercession	X	X
Actors	X	X	Side-Effect Free	X	X
Asynchronous Invocation	X	X	Transactions	X	X
Axum-Domains	X	X	Vats	X	X
Isolation	X	X			

Partial Support	SEM	OMOP	*Partial Support*	SEM	OMOP
By-Value	X	+	Persistent Data Structures	X	+
Channels	X	+	Replication	X	+
Data Streams	X	+	Speculative Execution	X	+
Map/Reduce	X	+	Tuple Spaces	X	+
Message sends	X	+			

referred to most of them, and thus, this section discusses them only briefly. The structure of the discussion roughly mirrors the structure of Sec. 3.2.2.

Asynchronous Invocation, Active Objects Active objects and other concepts that rely on the guarantee that an object's methods are invoked asynchronously are directly supported by the OMOP. The provided intercession handler `#requestExecOf:on:with:lkup:` can be used to customize invocation constraints. As previously discussed in Sec. 6.2.1, intercession handlers can be used for instance to implement Clojure agents. Its flexibility enables the implementation of a wide range of different policies and asynchronous invocation is one of them.

Actors, Axum-Domains, Isolation, Vats The required support for proper state encapsulation, i. e., isolation, can be realized by combining support for ownership, intercession handlers for reading and writing of object fields and globals, as well as intercession handlers for primitives. By customizing these, as demonstrated in Sec. 6.2.3, it becomes possible to support concurrent programming concepts such as actors, axum-domains, and vats.

No-intercession, Side-effect Freedom The notion of disallowing intercession, i. e., reflective operations is often used to increase control over state changes. Similarly, the requirement of some concepts that operations be side-effect free stems from the desire to simplify reasoning over code. Both no-

tions can be enforced by the OMOP. Reflective operations can be completely forbidden by customizing intercession handlers for method and primitive invocation. Mutation can be forbidden by customizing the handing of writes to object fields, globals, and the semantics of primitives. Sec. 6.2.1 demonstrated this by defining a `ImmutableDomain`.

Transactions As demonstrated with the implementation of a domain for STM in Sec. 6.2.2, the semantics of transactions can easily be supported by customizing the intercession handlers for reading and writing of object fields and globals. Support for customizing the semantics of primitives completes this support and enables full tracking of all state access and mutation operations.

6.3.2. Partially Supported Concepts

The bottom part of Tab. 6.3 covers another set of nine concepts. Their implementation is partially facilitated by the OMOP. While they require additional mechanisms, their implementation is simplified when they can rely on the OMOP as a foundation.

Immutability The concepts of *data streams*, *map/reduce*, *persistent data structures*, and *tuple spaces* all come with the implicit assumption of being used with immutable entities. For instance, Google's MapReduce [Lämmel, 2008] is designed for side-effect free operators in a distributed setting. Thus, it assumes that the data entities it operates on are immutable. The OMOP can provide such a guarantee based on an appropriate domain definition. However, the OMOP itself does not facilitate the execution, data handling, and scheduling that needs to be performed by a MapReduce implementation. The other three concepts benefit from the ability to enforce immutability in a similar way, but a significant part of these concepts is orthogonal to the OMOP. Thus, the OMOP provides partial support for the concept by facilitating the implementation of a key aspect.

Ownership When it comes to the implementation of *message sends*, the related concept of *channels*, and the general notion of *by-value* semantics, the OMOP can be used to realize part of the desired guarantees. The notion of ownership enables the distinction between sender and receiver, and for instance enables a domain to restrict mutability, or perhaps even realize a copy-on-write notion for entities that are shared by-value. However, since message

sends and channels entail more than these semantics, the OMOP supports only one important aspect, and thus, these concepts are partially supported.

Managed State and Managed Execution The remaining three concepts of *replication*, and *speculative execution* could benefit from the available tracking of state mutation and method invocation. For instance, the implementation of replication with a certain consistency guarantee needs reliable tracking of all state changes, which can be realized similarly to how the OMOP is used for an STM implementation. Speculative execution has similar requirements and can be realized with some variation of STM semantics to enable a clean abort of a speculatively executed computational branch. However, the OMOP provides support for only one of the important aspects of these concepts and beyond that the concepts are largely orthogonal to the concerns handled by the OMOP.

6.3.3. Conclusion

This section discussed how the OMOP facilitates the implementation of the concepts identified in Sec. 3.2. For all 18 concepts that require VM support to enforce their semantics, the OMOP either provides full support or simplifies the implementation substantially. It does that by providing flexible mechanisms to realize MANAGED STATE, MANAGED EXECUTION, OWNERSHIP, and CONTROLLED ENFORCEMENT.

Language implementers benefit substantially from this support, because for all evaluated concepts it addresses challenges such as immutability, isolation, custom execution policies, and custom state access policies (cf. Tab. 6.1).

6.4. Comparing Implementation Size

This section argues that the OMOP provides an abstraction that has a positive impact on the implementation size for concurrent programming concepts. It demonstrates this by showing that the OMOP-based implementations of AmbientTalkST and LRSTM are smaller than their ad hoc counterparts. In addition, it argues that the OMOP-based implementations of active objects, and CSP are sufficiently concise, i. e., have small implementations as well. This section first details the used metrics and then discusses the three case studies, i. e., the implementation of Clojure agents, LRSTM, and AmbientTalkST.

6.4.1. Metrics

This evaluation restricts itself to implementation size as a directly measurable aspect. Thus, an evaluation of complexity, which has an inherently social and subjective component, as well as the assessment of debuggability and bug probability are outside the scope of this evaluation. Furthermore, based on the studies of Jay et al. [2009], van der Meulen and Revilla [2007], and Emam et al. [2001] Sec. 6.1.2 argues that it is not clear that other metrics than implementation size have significant value to assess overall implementation complexity. Therefore, this evaluation measures implementation size as a surrogate for complexity based on *number of classes*, *number of methods*, *lines of code*, and *number of bytecodes*.

Number of classes (#Classes) is the count of classes added to the system.

Number of methods (#Methods, #M) is the count of methods added or changed in the system. This includes methods added and changed in preexisting classes.

Lines of code (LOC) refers to the length of a method including comments but excluding blank lines.

Number of bytecodes (#Bytecodes, #BC) is the count of all bytecodes in all counted methods.

Since the LOC metric is not robust to trivial changes, i. e., it varies based on coding conventions and comments, the evaluation also assesses the number of bytecodes of all added or changed methods. The number of bytecodes is the metric which is most oblivious to coding styles, indentation, and comments. Since the size of the implementations is small, the evaluation relies on the number of bytecodes as the main metric to avoid a significant impact of superfluous differences in the implementations.

The missing support for variadic methods in Smalltalk leads to the duplication of methods to handle the cases of 0 to n parameters. Since the implementation uses code duplication for an improved debugging experience and better performance, the code itself does not add benefits on its own, and to avoid maintenance overhead it could be generated by a simple templating mechanism. In order to mitigate the significant effect of this code duplication, the evaluation collects the metrics with and without counting such code duplication.

6.4.2. Clojure Agents

This dissertation uses Clojure agents to introduce the OMOP and as a case study to demonstrate how the OMOP can be used to guarantee stronger properties than the original Clojure implementation. This section briefly examines the metrics for the three different variants with their increasingly stronger semantic guarantees. It reports the measurements for the implementation in Squeak/Pharo Smalltalk. It is marginally more elaborate than the simplified implementation given in Lst. 5.2 in SOM syntax. The results are given in Tab. 6.4.

Without Guarantees The first variant of the `Agent` class comes without any guarantees. It implements a feature set similar to the one of the Clojure implementation. However, since the used Smalltalk does not have the notion of private methods, it requires the programmer to be aware of the correct usage. For instance, the method that processes the message queue of the agent should be invoked only from the agent's process. However, since it is public, it can be invoked from any process, which could lead to data races. This implementation without any guarantees takes 8 methods with 36 lines of code and 85 bytecodes in total.

Basic Guarantees The second variant of the `Agent` uses the `AgentDomain` to enforce the basic guarantees private methods would provide. Thus, it enforces that only the public interface of the class be used, ensuring that the agent's process, i. e., its event-loop is the only activity updating its state. This implementation is approximately double the size of the simple implementation. It takes two classes with a total of 11 methods, 81 LOC and 193 bytecodes.

Table 6.4.: Agent Implementation Metrics

Class (w/o guarantees)	#M	LOC	#BC
Agent	8	36	85
Basic Guarantees			
Agent	8	48	98
AgentDomain	3	33	95
=	11	81	193

With Immutability	#M	LOC	#BC
Agent	10	55	115
AgentDomain	3	33	95
ImmutableDomain	8	25	33
=	21	113	243
Clojure 1.4			
Agent	22	243	280

With Immutability The third variant additionally guarantees the immutability for the object graph referenced by the agent. It uses the `ImmutableDomain`, which prevents the mutation of all objects owned by it. The agent uses it by changing the owner of the result of the update functions to the immutable domain, before setting the result as the new agent state. See Sec. 2.4.3 and Sec. 5.3.2 for details. This implementation adds to 21 methods, 113 LOC and 243 bytecodes. Overall, with 113 LOC it provides significantly stronger guarantees than the original Clojure agents. While this is three times the size of the implementation without any guarantees, these 113 LOC implement concepts as general as immutability and provide the guarantees in a concise and maintainable form.

Conclusion As a comparison, Clojure 1.4[6] implements agents entirely in Java. The implementation consists of 22 methods, 243 LOC, and the main class file for the agent class contains 280 bytecodes. The implementation strategy is very similar to the one presented here, thus, it is comparably straightforward without complex optimizations. Still, these measurements are not directly comparable because of language differences between Java and Smalltalk. However, the OMOP-based implementation has a similar if not smaller size, while providing stronger guarantees. This result is an indication that the OMOP's abstractions are appropriate to implement the desired guarantees for the agent concept.

6.4.3. LRSTM: Lukas Renggli's STM

As mentioned in Sec. 6.2.2, the LRSTM implementations are based on the work of Renggli and Nierstrasz [2007]. We ported the original implementation to recent versions of Squeak and Pharo and reimplemented the AST transformations in the process. The core parts of the STM system remain identical for the ad hoc and OMOP-based implementation. This section briefly restates differences in the two implementations, outlines supported features, and then discusses implementation sizes. Sec. 6.2.2 discusses the implementations in more detail.

Ad hoc Implementation Closely following the implementation of Renggli and Nierstrasz, this implementation uses AST transformations to reify all state access, redirect them to the active transaction object, and thereby pre-

[6] https://github.com/clojure/clojure/tree/clojure-1.4.0

cisely track all read and write operations of object fields and globals. Consequently, this implementation includes the necessary AST transformations and compiler adaptations.

OMOP-based Implementation The implementation of LRSTM based on the OMOP customizes the relevant intercession handlers on the domain object to track all state access. Lst. 6.5 in Sec. 6.2.2 shows the simplified STM domain definition. Even though the implementation approaches are different, both implementations have the same features and guarantees. Thus, the ad hoc and the OMOP-based implementation are functionally equivalent.

General Comparison Note that comparing the ad hoc LRSTM implementation and the one based on the OMOP is not necessarily fair, since this comparison does not consider the complexity of realizing the OMOP itself. Sec. 7.1 discusses an AST-transformation-based implementation of the OMOP. Since both AST-transformation-based implementations are very similar, this evaluation also compares them to give a better impression of how the implementation sizes differ between the ad hoc and the OMOP-based implementation.

Direct comparison of the ad hoc and OMOP-based LRSTM implementation demonstrates the expected benefit of relying on the underlying abstraction. The ad hoc implementation consists of 8 classes and 148 methods, which account for 990 LOC and 2688 bytecodes. The OMOP-based implementation consists of 7 classes and 71 methods, which accounts for 262 LOC and 619 bytecodes. Thus, it is roughly a quarter the size of the ad hoc implementation. However, these results are to be expected because the ad hoc implementation needs to replicate the same functionality the OMOP offers for MANAGED STATE and MANAGED EXECUTION of primitives.

Detailed Comparision Tab. 6.5 details the code size of the different parts of the LRSTM implementation to paint a more detailed picture of the size of the involved components. While these numbers indicate a significant benefit for the OMOP-based implementation, the detailed numbers point out a number of differences.

The *STM core routines* implement the basic STM functionality and data structures. For the OMOP-based implementation, this includes the STM domain class, which implements state access tracking. The STM domain class accounts for a significant part of the additional methods. Besides minor adaptations, the remaining core routines remain unchanged.

Table 6.5.: Detailed Comparison of the ad hoc and OMOP-based LRSTM Implementations

	#Classes	#Methods	LOC	#Bytecodes
LRSTM (ad hoc)	8	148	990	2688
LRSTM (OMOP)	7	71	262	619
STM Core Routines ad hoc	6	48	170	386
STM Core Routines OMOP-based	7	71	262	619
Compiler ad hoc	2	64	507	1698
Compiler AST-OMOP base system	2	80	650	2017
Runtime Support ad hoc	0	36	313	604
Runtime Support AST-OMOP base system	2	78	471	871
Runtime Support AST-OMOP base system*	2	112	639	1244
Complete ad hoc	8	148	990	2688
Complete incl. AST-OMOP base system	14	241	1448	3632
Complete incl. AST-OMOP base system*	14	313	1891	4471
Complete incl. RoarVM+OMOP base system	14	145	854	2049
Complete incl. RoarVM+OMOP base system*	14	175	1016	2467

* including duplicated code for variadic argument emulation

The different sizes of the *compiler*, i. e., bytecode transformation component originate with the additional support for managing method execution and ownership in the OMOP. Support for it was not necessary in the ad hoc LRSTM, since an STM only regards state access and requires only basic support for handling primitives.

The size increase of the *runtime support* in the OMOP runtime comes from additional features. The OMOP brings a minimal mirror-based reflection library, to facilitate the implementation of ownership transfer, ownership testing, and switching between enforced and unenforced execution, among others.

Taking the additional features of the OMOP into account, the *complete* OMOP-based implementation is about 35% larger in terms of the number of bytecodes than the ad hoc implementation. Comparing the ad hoc implementation to the one using VM support and the corresponding base system shows that the OMOP implementation is approximately 24% smaller.

Conclusion While direct comparison of the LRSTM implementations shows trivial benefits for the OMOP-based approach, the picture is less clear when the OMOP runtime support is considered. However, the OMOP implementation is generic and facilitates the implementation of a wide range of different concepts beyond STM. Therefore, an OMOP-based implementation can be more concise. On the other hand, even if the full OMOP-implementation is

taken into account, the additional implementation size remains reasonable and requires only a 35% larger implementation accounting for the added genericity.

6.4.4. Event-Loop Actors: AmbientTalkST

This section examines the ad hoc as well as the OMOP-based implementation of event-loop actors. It briefly restates the two implementation strategies (cf. Sec. 6.2.3 for details), outlines the supported features, and then discusses the implementation sizes.

Ad hoc Implementation The ad hoc implementation realizes the notion of safe messaging by implementing AmbientTalk's far references with stratified proxies. Thus, actors do not exchange object references directly when message are sent. Instead, all object references to remote objects are wrapped in a proxy, i. e., the far reference.

While this implementation achieves a good degree of isolation with low implementation effort, it does not ensure complete state encapsulation. For instance, it does neither handle global state, introduced by class variables, nor potential isolation breaches via primitives. This choice was made to keep the implementation minimal, accepting the restrictions discussed in Sec. 3.3.2.

OMOP-based Implementation As discussed in Sec. 6.2.3 and as sketched in Lst. 6.7, the OMOP-based implementation relies on the full feature set provided by the OMOP.

While it offers the same APIs and features as the ad hoc implementation, its semantic guarantees are stronger. By utilizing the OMOP, it can provide the full degree of isolation, i. e., state encapsulation and safe message passing. Thus, access to global state and use of primitives are properly handled and do not lead to breaches of isolation between actors. Note that this implementation does not use proxies to realize far references. Instead, it relies on the notion of ownership to enforce the desired semantics.

Results and Conclusion As listed in Tab. 6.6, the ad hoc implementation of AmbientTalkST uses 5 classes with 38 methods. These account for 183 LOC and have a total of 428 bytecodes. The OMOP-based implementation on the other hand has 2 classes, which have in total 18 methods with 83 LOC and 190 bytecodes.

Thus, the OMOP-based implementation is more concise than the ad hoc implementation and provides stronger guarantees. Furthermore, the implementation expresses its intent more directly. While Lst. 6.7 is simplistic and benefits largely from the object-based encapsulation of Smalltalk, such a domain definition uses the ownership notion to makes it explicit that only the owning actor is able to access the state of an object. The ad hoc implementation on the other hand needs to use stratified proxies to realize encapsulation by wrapping objects with far references to avoid shared state. Using the ownership notion expresses the original intent of the concurrency model directly, and has the practical benefit of avoiding the creation of far-reference objects.

6.4.5. Summary and Conclusion

In addition to the discussed case studies, active objects and communicating sequential processes with channels (CSP+π) were implemented based on the OMOP. These prototypes implement the basic functionality, including the enforcement of the desired semantics in about 50-70 LOC each. The OMOP significantly facilitates these implementations with the provided mechanisms. The OMOP enables the implementation to formulate the semantics in a concise and uniform way. While these two concepts have only been implemented using the OMOP, the effort for the ad hoc implementation of CSP+π would be comparable to the effort for the ad hoc implementation of AmbientTalkST. Furthermore, using the same implementation strategy would result in the same weaknesses. An implementation of active objects would be simpler. However, such an implementation would most likely have weaker guarantees than with the OMOP, because primitives and reflection would need to be handled properly.

Tab. 6.6 gives an overview over the metrics measured for all experiments.

Overall, the use of the OMOP does have a positive impact on the implementation size. There are indications that it can significantly reduce implementation size. Furthermore, the case studies showed that approaches such as Clojure agents, CSP+π, active objects, and event-loop actors can be implemented in 53 to 113 LOC each, which includes the enforcement of semantics and can exceed standard guarantees. Thus, the OMOP is applicable to the implementation of the discussed concurrent programming concepts.

Table 6.6.: Metrics for Ad hoc and OMOP-based Implementations

	#Classes	#Methods	LOC	#Bytecodes
Agents (ad hoc, without enforcement)	1	8	36	85
Agents (OMOP, with enforcement)	3	21	113	243
AmbientTalkST (ad hoc)	5	38	183	428
AmbientTalkST (OMOP)	2	18	83	190
LRSTM (ad hoc)	8	148	990	2688
LRSTM (OMOP)	7	71	262	619
Active Objects (OMOP)	3	15	68	130
CSP+π (OMOP)	5	16	53	110
AST-OMOP base system	7	176	1216	3079
RoarVM+OMOP base system	7	74	592	1433
AmbientTalkST (OMOP)*	2	28	152	451
Active Objects (OMOP)*	3	19	97	237
CSP+π (OMOP)*	5	19	71	149
AST-OMOP base system*	7	248	1659	3918
RoarVM+OMOP base system*	7	104	754	1851

* including duplicated code for variadic argument emulation

6.5. Discussion

This section discusses the OMOP with regard to the remaining evaluation criteria of Sec. 6.1.2 and limitations of the presented approach. It discusses the applicability of the OMOP, the relevance and significance of the concepts it supports, its novelty, and whether it is a unifying substrate. The discussed limitations are restrictions with respect to deadlock freedom, interaction semantics of domains, granularity of ownership, and enforcement of scheduling policies.

6.5.1. Remaining Evaluation Criteria

Applicability To assess the applicability of the OMOP, this section assesses its ability to provide different language guarantees in order to enable the implementation of a wide range of concurrent programming concepts.

The first indication is given in Sec. 6.3. It shows that the OMOP supports all the concepts that require VM support for their semantics. Thus, it is flexible enough to provide either full or partial support for these concepts by addressing the common implementation challenges.

The second indication is the flexibility Sec. 6.2 demonstrated with the case studies. It shows that agents can be implemented including guarantees that exceed what is typically provided. Furthermore, it demonstrates that the

OMOP is sufficiently flexible to express language guarantees for event-loop actors and a software transactional memory system. Sec. 6.4.5 briefly mentions the implementation of CSP+π and active objects, which are variations in the same design space as event-loop actors and agents.

Hence, there are significant indications that the OMOP provides the ability to implement a variety of language guarantees and enables a wide range of concurrent programming concepts.

Relevance of supported Concepts To evaluate the relevance of the OMOP, this section provides an example of either recent research in the corresponding area, or a programming language that is used in industry for each of the supported concepts (cf. Tab. 6.3). The list of examples is not intended to be complete. Instead, it gives a limited number of impressions to support the claim of relevance. The discussion of the concepts is grouped by the degree of support. The concepts are *highlighted* in the text below.

Full Support *Active objects*, which are an example for the use of *asynchronous invocation*, are a pattern discussed in the literature [Schmidt, 2000], but also an area of active research [Clarke et al., 2008; Nobakht et al., 2012].

Actors are supported in popular languages such as Erlang[7] or Scala,[8] and in libraries such as Akka.[9] Actors rely on *message sends* that have *by-value* semantics to ensure safe messaging. Thus, all three concepts are relevant today.

Sec. 3.3.5 discusses the notion of restricted or *no intercession*. While many languages rely on this notion to guarantee concurrency properties, JVM[10] and CLI[11] offer it to handle security issues only.

Languages such as Haskell[12] advocate *side-effect free* functional programming for its engineering benefits. However, knowledge of a function being side-effect free is also useful for compiler optimization or automatic parallelization. The OMOP enables a language implementer to guarantee that a function executed in an immutable domain does not suffer from side effects.

Transactions and STM are part of Clojure[13] and Haskell. Furthermore, recent versions of the GNU GCC[14] provide support for it.

[7] http://www.erlang.org/
[8] http://www.scala-lang.org/
[9] http://akka.io/
[10] http://docs.oracle.com/javase/7/docs/api/java/lang/SecurityManager.html
[11] http://msdn.microsoft.com/en-us/library/stfy7tfc.aspx
[12] http://www.haskell.org/
[13] http://clojure.org/
[14] http://gcc.gnu.org/wiki/TransactionalMemory

Vats, as prosed by the E language [Miller et al., 2005], are becoming popular for instance with JavaScript WebWorkers[15] or isolates in Dart.[16] Vats and their *isolation* property are also an important aspect of AmbientTalk's event-loop actors [Van Cutsem et al., 2007]. *Axum's domain*[17] concept is a variation of the vats concept. It enables sharing of data between actors. It is relevant because traditional pure actor languages do not provide means for data parallelism [De Koster et al., 2012; Scholliers et al., 2010].

Partial Support The notion of *channels* has been used, e. g., in CSP [Hoare, 1978] and is offered today in languages such as Go[18] and XC.[19]

Data streams are used for instance in stream-based programing languages such as StreamIt [Thies et al., 2002], but are also relevant for event processing [Renaux et al., 2012].

Map/Reduce as implemented by Google's MapReduce [Lämmel, 2008] or or Apache Hadoop[20] is relevant to process large amounts of data.

Persistent data structures are used for side-effect free programs in Clojure and Scala for instance. Supporting them by guaranteeing immutability simplifies interaction across languages and libraries.

Replication is used for instance in Mozart [Mehl, 1999]. More recently it is used in Orleans [Bykov et al., 2011], a framework for cloud computing, which exploits notions such as immutability to enable state replication. While both examples are intended for use in distributed systems, such techniques are relevant [Chen et al., 2008] for modern parallel processors such as the Cell B.E. processor [Johns and Brokenshire, 2007] as well.

Speculative execution is commonly used to improve the performance of sequential programs at the hardware level of processors [Hennessy and Patterson, 2007], but has also shown potential for parallelizing sequential programs on multiple cores [Herzeel and Costanza, 2010; Vajda and Stenstrom, 2010].

Conclusion This section gave one or more examples of support in programming languages or recent research for each of the supported concepts. These

[15] http://www.whatwg.org/specs/web-apps/current-work/multipage/workers.html
[16] http://www.dartlang.org/
[17] http://download.microsoft.com/download/B/D/5/BD51FFB2-C777-43B0-AC24-BDE3C88E231F/Axum%20Language%20Spec.pdf
[18] http://golang.org/
[19] http://www.xmos.com/technology/xc
[20] http://hadoop.apache.org/

Table 6.7.: Subset of concepts the OMOP supports, which are already supported in today's VMs.

Supported	VMs
Actors	Erlang
Transactions	GHC
Partial Support	
By-Value	Mozart
Channels	DisVM, ECMAScript+HTML5, Mozart, Perl, Python
Message sends	Erlang
Replication	Mozart

examples indicate that the supported concepts have practical relevance, and thus, their support is worthwhile.

Absence of wide Support in Today's VMs This section evaluates the novelty of the OMOP by assessing whether the concurrent programming concepts that are supported by the OMOP are widely supported in today's VMs. This assessment relies on the VM survey in Sec. 3.1.

Tab. 6.7 combines the survey and Tab. 6.3, which contains all concepts that benefit from an enforcement of their semantics by the VM. As a result, Tab. 6.7 shows the subset of supported concepts and the VMs that support the given concept. A concept is included in the table, if the VM supports it with *implicit semantics*, as part of the *instruction set architecture* (ISA), or *primitives* (cf. Sec. 3.1.1.1).[21]

The first observation based on Tab. 6.3 is that the surveyed VMs support 6 out of 18 concepts only.

Furthermore, only communication via channels has found widespread VM support. Most importantly, none of the surveyed VMs supports all of the concepts the OMOP supports.

In addition, none of the VMs provides a mechanism satisfying all the requirements of the OMOP. Lua and ECMAScript+HTML5 provide MOPs that provide the flexibility of metaobject-based or proxy-based MOPs, but neither offers the notion of ownership nor do their runtimes support parallel execution on shared memory, which significantly restricts the flexibility and the number of supported concurrent programming concepts.

[21] Appendix A.1 gives a full overview of the VM survey, including how these concepts are supported.

Based on these observations, the OMOP and its support for concurrent programming concepts can be considered novel.

Significance The support provided by the OMOP is significant, because it facilitates the implementation of all 18 concepts that require VM support to enforce their semantics. These 18 concepts are the full set of concepts identified by the survey in Sec. 3.2, which is designed to cover the whole field of concurrent and parallel programming concepts. 9 of these concepts are fully supported, while the OMOP address common implementation challenges for a key aspect of the other 9 concepts.

This means that if a VM such as the JVM or the CLI were to add support for the OMOP, languages built on top of these VMs could use any of the 97 identified concurrent and parallel programming concepts (cf. Sec. 3.2.2) without compromises on the enforcement of semantics. In addition to the concepts that are already supported by JVM and CLI, and in addition to the concepts that can be implemented as libraries without drawbacks, the OMOP completes VM support for concurrent programming.

Therefore, the set of concepts supported by the OMOP is significant.

Unifying Substrate Finally, this section discusses the unifying properties of the OMOP. The goal is to argue that the OMOP makes an abstraction of concrete programming concepts and that the set of provided mechanisms is minimal.

The claim that the OMOP makes an abstraction of concrete concepts is supported by the fact that the 18 supported concepts are not realized with a one-to-one relationship between the OMOP's mechanisms and the concepts. Tab. 6.3 lists the supported concepts: active objects, actors, asynchronous invocation, Axum-domains, isolation, no-intercession, side-effect free, transactions, and vats. The concepts supported by the OMOP are ownership, enforcement status, reading and writing of object fields and globals, method execution, thread resumption, and handling of primitives (cf. Sec. 5.2). Therefore, the OMOP's concepts need to be combined to build the higher-level concepts. Furthermore, to build different concepts multiple parts of the OMOP have to be combined. The OMOP does not possess the notion of actors or isolation and therefore, it makes an abstraction of these concepts.

The OMOP provides a minimal set of elements to satisfy the requirements of Sec. 3.4. The proposed design in its form is minimal because none of the parts can be removed without reducing the number of concurrent program-

ming concepts that can benefit from it, and removing any part would also result in unsatisfied requirements. The remainder of this sections supports this claim by discussing the *basic interface* of the OMOP. The VM-specific aspects and helper functionality are excluded, because they are not completely orthogonal to the basic interface (cf. Sec. 5.2).

Ownership Sec. 5.1 argues that a MOP needs a connection between base level, i. e., application and the meta level, i. e., the language implementation. The OMOP uses ownership as this meta relation. Other options could have been instantiation for metaclass-based approaches or a direct connection between object and metaobject in metaobject-based approaches. Without such a meta relation, it would not be possible to define more than one set of policies, i. e., a domain, because it would need to be applied implicitly by the runtime instead of being based on the meta relation. Thus, a meta relation is required and the choice of ownership combines with the requirements of concurrent programming concepts to represent the notion of owning entities.

Enforcement Status The OMOP needs a mechanism to distinguish between base-level and meta-level execution. The OMOP represents this difference by a simple bit. Other representations could allow for more flexibility, for instance to enable reflective towers [Smith, 1984]. However, in either case, it requires a representation of the execution levels to distinguish them. Otherwise every operation would trigger an intercession handler, which itself would again trigger an intercession handler, leading to infinite meta regress without base case. The OMOP uses the minimal required solution by only distinguishing base and meta level.

State Access Reification With `#readField:` and `#write:toField:to:` two intercession handlers are provided by the OMOP to capture reading and writing of object fields, enabling a customization of all state access. Concurrent state access is one of the fundamental issues that concurrent programming tackles by providing means to coordinate the use of state, i. e., resources. As this chapter demonstrates, customizing state access policies is required to implement an STM or immutability for instance. Without it, the OMOP would not be able to support these concepts and it could not provide the same guarantees for concepts such as actors, vats, and isolation it gives now. In conclusion, state access reification is a required mechanism and the OMOP's capabilities would be reduced significantly without it.

Method Invocation Reification The ability to manage execution is essential for active objects, actors, CSP, vats, and others. To this end, the OMOP provides the `#requestExecOf:on:with:lkup:` intercession handler. The case studies in Sec. 6.2 demonstrate how it can be used to customize method execution policies, to enforce asynchronous execution for instance. Removing this intercession handler from the OMOP would remove its ability to facilitate different policies for method invocation, and as a consequence significantly reduce the number of supported concepts for which it can enforce guarantees.

Thread Resumption Event-loop actors and CSP require that only a single thread executes the event loop or the program on shared mutable state. Based on the notion of ownership and domains, these concepts require control over the number of active threads. Since the thread itself is not necessarily owned by a domain when it attempts to execute in that domain, the resumption cannot be customized via custom method invocation policies. Thus, the OMOP needs the `#requestThreadResume:` intercession handler to capture this operation. Without it it would not be possible to guarantee that the number of threads executing in a domain be restricted.

Primitives Concrete primitives are specific to a VM. However, the notion of primitives is available in all VMs. Primitives either make functionality of the underlying platform available or they implement functionality that cannot be expressed in the language that the VM implements. Since such primitives are free to access object state, execute methods, or change internal state of the VM, language implementers need to adapt them in case they conflict with the semantics of a concurrent programming concept. For instance, reflection is implemented via primitives and needs to be subjected to concurrency semantics (cf. Sec. 3.3.5). To this end, the OMOP provides the `#prim*` intercession handlers. It is required because the primitives would otherwise void the desired semantics.

This section shows that the OMOP abstracts makes abstractions of concrete concepts by providing mechanisms that enable their implementation. Furthermore, for each mechanism of the OMOP it argues that it is required to satisfy the purpose of a substrate for concurrent programming concepts. Thus, the OMOP is minimal, because removing any of its parts would significantly decrease its functionality and it would not be able to satisfy its requirements (cf. Sec. 3.4).

Therefore, the OMOP can be considered to be a unifying substrate for the implementation of concurrent programming concepts.

6.5.2. Limitations

General deadlock freedom can not be guaranteed in the presence of blocking operations. *Deadlock freedom* is another guarantee that is given by a number of approaches for concurrent programming. However, it is a guarantee that cannot be given in a system which enables the arbitrary use of blocking operations. Since the OMOP is conceived to be used as an extension to existing VMs, it depends on the specific VM whether deadlock freedom can be guaranteed. Only if the VM provides access to a set of *disciplined* blocking mechanisms, i. e., it guarantees that any possible graph of *wait-for*-relations is cycle free, can the guarantee of deadlock freedom be given. However, since the OMOP targets multi-language VMs such a guarantee is impractical because it implies restrictions on what synchronization mechanisms are allowed.

While this is a conceptual limitation, using the ability to customize primitives, it is possible to manage the calls to blocking operations and adapt them as necessary. Thus, it is possible to work around this restriction.

Interaction semantics between domains need to be implemented explicitly. A second aspect that is currently not directly facilitated by the OMOP is the definition of *interaction semantics* for different concurrency abstractions. While an application can use multiple distinct concurrency domains for instance with STM or actor semantics, the implementations of these semantics need to be prepared to interact with each other to handle corner cases well. Thus, the STM implementation needs to have a specific strategy to handle interactions with the actor domain to ensure transactional semantics in the presence of asynchronous message sends. In the proposed OMOP, it is up to the language implementer to define these semantics and it requires them to anticipate which kind of concurrency domains might be used in combination.

Object-based ownership notion has tradeoffs. A third restriction is the use of *object-based ownership* as distinguishing criterion to enforce concurrency semantics. Arguably, the object-level is fine-grained enough and constitutes a good tradeoff in terms of engineering effort. However, since an object can only have a single owner, the design space is restricted and precludes abstractions such as partitioned objects. Partitioning is common in distributed systems, but outside the scope of this dissertation (cf. Sec. 1.3).

Limited support for guaranteeing scheduling policies. As part of the case study of event-loop actors Sec. 6.2.3 argues that the intercession handlers of the OMOP give additional opportunities to custom schedulers on top of the VM's threading facilities to enforce their policies. However, this support is still restricted, because as soon as a primitive operation is started, the VM yields complete control to the native code. Thus, the operating system's scheduler could enforce scheduling guarantees, but the VM typically cannot. It would therefore be beneficial if VM and operating system could interact more directly to enable the customization of scheduling policies for instance to ensure fairness or locality. Locality becomes a major performance consideration for non-uniform memory access (NUMA) architectures. While this question is outside of the scope of this dissertation, it as an issue that can be discussed independently from the presented OMOP to enable an additional set of concepts to benefit from performance improvements or semantic guarantees.

6.5.3. Conclusion

This section argues that the OMOP provides the required flexibility to enable the implementation of a wide range of concurrent programming concepts, and thereby shows that the OMOP is applicable to the problem. Furthermore, for each of the supported concepts it provides examples to support the claim that they are relevant. The discussion based on the VM survey supports the claim that none of the investigated VMs provide support for concurrent programming concepts to the same degree as the OMOP. The complete support for concepts that require semantic enforcement further supports the claim of significance for the set of supported concepts. Finally, the OMOP makes abstractions of concrete concurrent programming concepts and provides a minimal substrate for their implementation, and therefore has unifying properties. Thus, the OMOP fulfills all requirements for a unifying substrate for the support of concurrent programming in VMs.

However, the design of the OMOP currently has a number of limitations when it comes to guaranteeing deadlock freedom, defining interactions between domains, the granularity of ownership, and customizability of scheduling guarantees. While there are conceptual limitations, the OMOP already supports a wide range of concepts and facilitates their implementation.

6.6. Conclusion

The goal of this chapter was to assess to what extent the OMOP satisfies its requirements to support the thesis statement, and to discuss how the research goal is reached.

Case studies of Clojure agents, STM, and event-loop actors demonstrate how common implementation challenges are solved and how the OMOP fulfills its requirements. Sec. 6.2 demonstrates how Clojure agents, software transactional memory, and AmbientTalk's actors can be implemented on top of the OMOP. Furthermore, it showes how the parts of their implementation map to the OMOP and how they relate to the stated requirements of Tab. 3.7. The case studies also demonstrate how common implementation challenges such as isolation, scheduling guarantees, immutability, and reflection can be approached with the OMOP.

OMOP facilitates the implementation of all 18 concepts that require semantic enforcement from the VM. Sec. 6.3 shows that the OMOP facilitates the implementation of all 18 concepts. It provides full support for 9 concepts and solves implementation challenges for the remaining 9 concepts. It does so by providing flexible mechanisms to customize state access and method execution policies. Therefore, language implementers benefit substantially from this support provided by the OMOP.

The OMOP can lead to more concise implementations with full guarantee enforcement. Sec. 6.4 demonstrates that the use of the OMOP does not have a negative impact on implementation size. On the contrary, it indicates that the OMOP can facilitate the implementation of concurrency concepts in such a way that their implementation size is reduced. In conclusion, the abstractions provided by the OMOP are appropriate for the implementation of Clojure agents, active objects, actors, CSP, and STM.

The OMOP is applicable, supports relevant concepts, is novel, the support set of concepts is significant, and it is a unifying and minimal substrate. Sec. 6.5 discusses the remaining evaluation criteria. The case studies indicate the flexibility of the OMOP in solving common implementation challenges and show its applicability. To conclude, the OMOP provides the flexibility to reach beyond today's concurrent programming concepts, because guarantees

can be varied as demonstrated with the agents implementation. The relevance of the supported concepts is shown by examples of recent research or by its support for each of them in a programming language. The novelty of the OMOP is shown based on the VM survey, because the concepts the OMOP supports are not supported in VMs and none of the VMs provide a mechanism that satisfies all requirements for the OMOP. Derived from the set of supported concepts, it can be concluded that the OMOP's support is significant, because it facilitates the implementation of all concepts that require VM support for their semantics. Finally, the OMOP makes abstractions of concrete concepts and provides a minimal set of mechanisms to support them. Overall, the OMOP is a unifying substrate for the implementation of concurrent programming concepts on top of VMs.

Wide range of concepts supported despite current limitations. Sec. 6.5.2 identifies a number of limitations of the current OMOP design, such as missing support for general deadlock freedom, missing support for interaction semantics, the tradeoffs of the object-based and single ownership design, and limitations for enforcing scheduling guarantees. However, while there are conceptual limitations, the OMOP provides workarounds for these limitations, and thus, reduces their practical impact.

The OMOP can be used as a uniform substrate for the implementation of concurrent programming concepts. The conclusion of this chapter is that the OMOP enables the implementation of a wide range of concurrent programming concepts in a concise manner. With the abstractions it provides, it allows language implementers to implement language guarantees in a more direct way than with comparable ad hoc approaches. Furthermore, the OMOP enables new opportunities for experimenting with language guarantees and concepts. The implementation of Clojure agents demonstrates the ease of strengthening the provided guarantees. Beyond that, the OMOP provides a platform for experiments and variation of concepts based on its small number of abstractions.

7

IMPLEMENTATION APPROACHES

This chapter presents two implementation strategies for the OMOP. For each implementation strategy, it discusses how the OMOP's mechanisms are realized, how the requirements for the OMOP are satisfied, and what the limitations of the implementation are.

The first implementation uses program transformation at the level of the abstract syntax tree (AST). This approach avoids the need to change the VM and thus, facilitates experiments with the OMOP in new environments. The second implementation integrates the OMOP into a bytecode-based interpreter, going beyond the approach presented in Sec. 5.4, and adding an optimization to avoid triggering uncustomized intercession handlers.

7.1. AST Transformation

This implementation transforms the abstract syntax tree (AST) of Smalltalk methods to provide the OMOP's functionality. This dissertation refers to it as the AST-OMOP. As discussed in Sec. 4.3, it is itself implemented in Smalltalk. Since the implementation strategy is based on AST transformations, it has capabilities similar to common bytecode-transformation-based implementation strategies, e. g., for Java [Hilsdale and Hugunin, 2004]. The VM can remain unchanged with this approach and the OMOP can be applied in settings where modifying the VM is not an option.

Rationale The first OMOP implementation was the AST-OMOP. With its comparably low implementation effort, it facilitated the exploration of the notion of a MOP as unifying substrate for concurrent programming concepts. It was chosen as the first implementation strategy in order to be able to experiment in a productive environment and to focus on the implementation of the case studies. The results were the implementation of AmbientTalkST and LRSTM on top of the AST-OMOP [Marr and D'Hondt, 2012].

The remainder of this section details how the semantics of the OMOP are realized, and how the arising implementation problems have been solved. Since the implementation is performed in Smalltalk, it did not need to cope with the complexity that for instance a Java-like type system would introduce. This section discusses this and other issues to argue that the approach is transferable to other systems than Smalltalk.

7.1.1. Implementation Strategy

Since this implementation relies solely on AST transformation, it needs to take special measures to realize CONTROLLED ENFORCEMENT and MANAGED EXECUTION for primitives. Achieving MANAGED STATE and MANAGED EXECUTION for normal Smalltalk methods is comparatively simple. The implementation strategy used was first proposed by Renggli and Nierstrasz [2007] for the implementation of an STM system for Smalltalk. The AST-OMOP uses the same general ideas, but extends Renggli and Nierstrasz's approach to cover all aspects of the OMOP.

OMOP Representation The owner of an object is expressed by a new object field for the domain in all classes that support it. For some classes Smalltalk

VMs makes special assumptions and does not allow adding fields. One example is the `Array` class. Here a subclass that has the domain field should be used whenever possible. In cases where this approach is not feasible, the implementation uses a map to keep a record of these objects and to associate them with the owner domain. The map is a `WeakIdentityKeyDictionary` and thus allows objects to be garbage collected normally. The combination of these three strategies results in a acceptable tradeoff between performance overhead of maintaining the ownership relation and the complexity of changing existing code.

The `Domain` itself is provided as a class, implementing the methods depicted in Fig. 5.1. Threads of execution are the Smalltalk processes. Each process relies on a stack of `ContextPart` objects. However, we can not modify `ContextPart` to include a field representing the domain it executes in, because it is one of the classes that have a fixed layout that is expected by Smalltalk VMs. The AST-OMOP works around this restriction by using a field in the `Process` object itself to represent the notion of the currently executing domain. The previous domain, the process was executing in, is kept on the runtime stack. Lst. 7.1 shows the corresponding implementation of `#evaluateEnforced:`. This approach is sufficient for the AST-OMOP, since a process can only change the domain it is executing in via an explicit request to another domain to evaluate a block in its context. For the dynamic extent of the block, the process will execute in that domain.

```
1  Domain =  (
2    "..."
3    evaluateEnforced: aBlock = unenforced (
4      | oldDomain proc result |
5      proc      := Processor activeProcess.
6      oldDomain := proc currentDomain.
7
8      proc currentDomain: self.
9        result := aBlock value.
10     proc currentDomain: oldDomain.
11
12     ^ result ) )
```

Listing 7.1: Maintaining the domain a `Process` executes in.

Basic Transformations The AST-OMOP enforces the OMOP's semantics by transforming message sends, the accesses to object fields, and access to globals. For Squeak/Pharo Smalltalk, globals also include literals, because they are technically the same. Lst. 7.2 gives an example for these transformations.

The method #readSendWrite is transformed into its enforced variant with the prefix #__enforced__.

```
1  Object = (| "..."  domain |
2    requestExecOf: selector with: arg = unenforced (
3      domain requestExecOf: selector on: self with: arg)
4  "...")
5
6  Example = Object (
7    | field |                          "index of 'field' is 2"
8
9    readSendWrite = (
10     | valOfField result |
11     valOfField := field.             "for illustration only"
12     result     := self doSomethingWith: valOfField.
13     field      := result )
14
15   __enforced__readSendWrite = (   "transformed version"
16     | valOfField result |
17     valOfField := domain readField: 2 of: self.
18     result     := self requestExecOf: #doSomethingWith:
19                                      with: valOfField.
20     domain write: result toField:   2 of: self)
21
22   run: args = unenforced (
23     | domain |
24     domain := Domain new.
25     [Example new readSendWrite] enter: domain ) )
```

Listing 7.2: Applying transformation to #readSendWrite.

The read of an object field is transformed into a message send to the domain of the object. Line 17, corresponding to the original line 11, shows the transformation result of the field read, resulting in the send of #readField:of: with the corresponding parameters. During compilation it is known whether the class has a field for the domain, as in the given example. However, if this is not the case for instance for arrays, the transformation generates a message send to look up the domain object instead. The look up will then use the map introduced in the preceding section.

Message sends are treated differently. Instead of determining the domain to which the execution request needs to be redirected directly, the implementation uses an indirection defined on Object (cf. line 3). With this indirection, the implementation can handle Smalltalk's cascaded message sends uniformly with normal message sends. Cascaded message sends (cf. Sec. 4.2.1)

all have the same receiver object, which might be the result of an expression. However, the object may change its owner. This indirection, to determine the domain for every execution request directly, avoids a complete restructuring of the code as part of the transformation. With that, it also avoids the need of complex transformations and the use of additional temporary variables. The latter is a problem when a naive[1] transformation is used. The transformed method could require more temporary variables than what is supported by the bytecode set and VM. Furthermore, besides keeping the transformation simple, the approach has another benefit. It preserves the structure of the original code, which is beneficial while debugging the transformed code.

In Squeak/Pharo, globals such as classes and class variables are realized by mutable associations of a symbol with a value. These accesses are rewritten by sending either `#readGlobal:` or `#write:toGlobal:` to the current execution domain. Note that the current execution domain does not need to be the receiver's owner domain. Certain shared memory concurrency models might want to enable access from processes from multiple domains on the same object.

Strategy for Controlled Enforcement The AST-OMOP realizes the notion of CONTROLLED ENFORCEMENT by applying the approach Renggli and Nierstrasz utilized to differentiate between normal and transactional execution. As illustrated in Lst. 7.2, every method implementation needs to be available in two versions: the normal unchanged version, which corresponds to unenforced execution, and the transformed method. Enforced, i. e., transformed methods are made available with a name prefixed with `__enforced__`. The prefix is used to have the enforced methods side by side with the unenforced ones in the same Smalltalk method dictionary.

Switching to enforced execution is done by sending the `#enter:` message to a block. The parameter to `#enter:` is the domain which `#evaluateEnforced:` is going to be sent to with the block as argument. This double dispatch is used for idiomatic/esthetic reasons. However, it also simplifies the correct implementation of enforced execution. The example block given in line 25 needs to be transformed in the same way normal methods are transformed into their enforced equivalents. The current restriction of this implementation is that only blocks that directly receive the `#enter:` message statically in the

[1] Other approaches would require analysis of the transformed code to minimize the number of temporary variables required.

code can be executed in enforced mode. Other blocks will not be transformed as part of a the unenforced method version.

Switching back to unenforced execution is done based on an attribute, similarly to the approach discussed for SOM and used by Renggli and Nierstrasz. However, in Squeak and Pharo, the syntax for method definitions is slightly different from the SOM syntax used here. Therefore, instead of using an attribute, the method body of an unenforced method needs to contain the `<unenforced>` pragma. Methods marked with this pragma are not transformed, and thus, do not enforce the semantics of the OMOP. However, to be able to switch back to enforced execution, blocks that receive the `#enter:` message are still transformed correctly.

Handling of Primitives Since the AST-OMOP does not change the VM, it needs another approach to handle primitives. The approach of Renggli and Nierstrasz to solve the issue was again chosen. All Squeak/Pharo methods that rely on primitives are marked with a pragma: `<primitive: #id>`. The AST-OMOP adds the `<replacement: #selector>` pragma to each of these methods. The method identified by the selector is then used as the enforced version of the primitive method. This replacement method simply redirects the execution of the primitive to the corresponding intercession handler in the domain.

Integration with Smalltalk's Interactive Environment The image-based development tools of Squeak and Pharo require minor adaptations to accommodate the AST-OMOP. As a cosmetic change, the class browser does not show the enforced methods, because they are generated automatically and do not need to be changed manually. Suppressing them in the browser avoids problems and clutter in the IDE.

The generation of transformed methods is performed on demand. To avoid the memory overhead of keeping all methods in two versions, method transformation is only triggered when an existing method is changed, which had an enforced version, or when an enforced version could not be found and `#doesNotUnderstand:` got triggered.

To capture changes of existing methods, additions, or removals, the system offers corresponding events on which the AST-OMOP registers the appropriate handlers to generate transformed methods when necessary.

Problematic for this approach are the semantics of class hierarchies and method lookup. Thus, it is not sufficient to generate only the missing method,

because this could lead to a change in the semantics of the program. A send could accidentally chose an implementation in the wrong super class. To avoid inconsistencies and problems with inheritance and overridden methods in subclasses, the used mechanism compiles all required methods in the relevant part of the class tree with the same selector.

7.1.2. Discussion

Applicability The main benefit of the implementation approach for the AST-OMOP is its wide applicability to existing systems. By relying on AST transformation, it avoids the need for a customization of the underlying VM. While Smalltalk bytecodes are untyped, the approach can be applied to other VMs such as the JVM and CLI as well. The OMOP would need to be extended by appropriately typed variants for the intercession handlers. A JVM would require additional handlers for the reading and writing of fields of primitive types and arrays. Method invocation could be handled in the same way as `java.lang.reflect.Proxy` and `java.lang.reflect.InvocationHandler` provide intercession of method invocations. Another approach would be to use the INVOKEDYNAMIC infrastructure [Thalinger and Rose, 2010]. While the CLI does not provide the same notion of proxies, the DLR (Dynamic Language Runtime) adds infrastructure such as `DynamicObject`, which could be used as a foundation for an OMOP implementation on top of the CLI.

One drawback of this approach is consequently its dependency on the capabilities of the underlying platform. While for instance the CLI provides infrastructure in form of the DLR that can be used to achieve similar results, not all platforms provide these and an OMOP implementation can come with either high implementation complexity, or severe limitations.

Performance and Potential Optimizations Another drawback is the comparatively high performance impact of this solution. As Chapter 8 will show, a naive implementation results in a significant performance loss. While there is potential for optimization, it might come with a significant complexity penalty.

One potential optimization idea is to statically specialize code for certain concurrent programming concepts, i. e., domain definitions. While this dissertation envisions the OMOP as a foundation for applications that simultaneously use a number of concurrent programming concepts to address problems with appropriate abstractions, performance might require compromises. Therefore, the idea would be to statically generate code with for instance sup-

port for only the STM domain. Code that is specialized that way would only need to trigger the state access intercession handlers instead of triggering all handlers unconditionally. Thus, the generate code based on such a optimization could be similar to the code generated by the ad hoc implementation of the STM. The main drawback would be that it would not be possible to use the specialized code together with any other domain.

While the transformation-based approach might come with limitations imposed by the underlying platform, and restrictions in terms of the achievable performance, it is a viable option in scenarios where VM changes are not feasible.

7.1.3. Related Work and Implementation Approaches

As pointed out throughout this section, the proposed implementation strategy for the AST-OMOP is closely related to the STM implementation of Renggli and Nierstrasz [2007]. The main difference with their implementation is that the OMOP needs to cover a wider range of aspects such as message sends, which do not need to be considered for an STM. Furthermore, their implementation is based on Geppetto [Röthlisberger et al., 2008], a framework for sub-method reflection. Since Geppetto is not available for the current versions of Squeak and Pharo, their implementation is also only available[2] for an outdated version of Squeak.

To ease development, we ported their implementation to recent versions of Squeak and Pharo[3] and implemented the necessary code transformations without the use of Geppetto. Instead, the AST-OMOP relies on an AST representation of the Smalltalk code and applies transformations on the AST by using a custom visitor (cf. Sec. 4.3). The ad hoc implementation of the STM uses the same AST transformation techniques that are used in this implementation of the OMOP.

While sub-method reflection as offered by Geppetto and custom code transformation are viable options and are used for instance also by Ziarek et al. [2008] based on the Polyglot compiler framework [Nystrom et al., 2003], the required transformation have strong similarities with the capabilities offered by common aspect-oriented programming (AOP) languages. The pointcut languages as offered by AspectJ[4] are often powerful enough to satisfy the require-

[2] http://source.lukas-renggli.ch/transactional.html

[3] Our port of the STM, now name LRSTM for Squeak 4.3 and Pharo 1.4 is available at http://ss3.gemstone.com/ss/LRSTM.html

[4] http://www.eclipse.org/aspectj/

ments for the implementation of the OMOP. However, the AOP frameworks available for Smalltalk are typically designed around the notion of message sends between objects and do not provide the necessary capabilities.

Examples are AspectS [Hirschfeld, 2002] and PHANtom [Fabry and Galdames, 2012]. Both offer only pointcut definitions that expose join points for message sends. Phase [Marot, 2011], a third AOP framework for Smalltalk, goes beyond that by also offering pointcut definitions to capture field accesses, and thus, exposes the necessary join points to intercept the reading and writing of fields. However, it does not offer pointcut definitions that enable an interception of access to globals. Thus, while more powerful AOP languages for instance for Java would provide alternative implementation approaches similar to the ones used here, such frameworks currently do not exist for Smalltalk and a custom code transformation remains the simplest solution available.

7.2. Virtual Machine Support

Rationale The AST-OMOP was conceived to implement case studies to assess the value of the general idea. However, its use is limited to performance insensitive applications. In general, the AST-OMOP implementation is inappropriate for evaluating the last part of the thesis statement, i. e., whether the OMOP lends itself to an efficient implementation (cf. Sec. 1.3).

Therefore, this dissertation investigates direct VM support for the OMOP as well. In order to evaluate the overall performance potential, it was decided to start out with a prototype implementation that adapts a simple interpreter. The RoarVM is a sufficiently flexible platform for such experiments. Furthermore, as Sec. 4.4 details, the RoarVM also provides support for a number of techniques such as parallel execution that is implemented with operating system processes, which could facilitate future work on optimizations (cf. Sec. 4.4.3). Thus, this section discusses the OMOP implementation called *RoarVM+OMOP*.

While interpreters do not provide the same performance as highly optimized just-in-time compiling VMs, they remain widely used and are valued for their flexibility and portability. Therefore, the OMOP could be applied to interpreters to facilitate concurrent programming where a performance tradeoff compared to a highly optimized VM is acceptable. However, even in such a setting a significant slowdown from using the OMOP is to be avoided. A performance evaluation with an interpreter provides indications for the feasi-

bility of efficient implementation, without having to manage the significantly higher complexity of just-in-time compiling VMs.

This section discusses the design decisions and implementation challenges of the RoarVM+OMOP. The implementation changes the behavior of bytecodes and primitives to expose the OMOP's intercession points, reify reading and writing of fields and globals, as well as message sends and primitive invocations. Furthermore, it uses a simple optimization based on a bit mask in the domain objects. This bit mask, called the *customization constant*, encodes the intercession points that are customized and require reification.

7.2.1. Implementation Strategy

OMOP Representation For the implementation of object OWNERSHIP, the RoarVM+OMOP relies on the infrastructure in the RoarVM to add another header word to every object (cf. Sec. 4.4.2). This additional header word is a reference to the owner of the object. This solution corresponds to adding a field to every object in AST-OMOP. However, the header word is not part of the actual object, and thus, does not conflict with the Smalltalk image because it does not violate assumptions about object structure and field layout. Instead, the implementation needs to make sure that the garbage collector (GC) is aware of the extra header word and that it is to be treated as an object reference.

Domains are represented in the same way as in the AST-OMOP. A subclass of `Domain` can customize the intercession handlers with its own policies. In fact, both implementations are compatible and share implementation and unit tests as well as benchmarks and case study implementations. This code sharing greatly reduces the implementation effort, and it facilitates testing of both implementations ensuring that they behave identically.

The domain which a `Process` is executing in, is represented by the owner domain of the context object, i. e., the stack frame. This is different from the AST-OMOP, which maintains this information in the process object (cf. Sec. 7.1.1). Sec. 5.5 argues that the context objects can be considered to be metaobjects, and thus, are not subject to the OMOP. Their extra header field for the owner domain can therefore be used to indicate the domain in which the process with its context object executes.

The execution mode, i. e., whether the frame is executed in enforced or unenforced mode is encoded in a spare bit of the reference (`Oop`) referring

to the domain object.[5] This provides part of the support for CONTROLLED ENFORCEMENT.

The initial ownership of newly created objects is deduced from a special field in each domain object. During initialization of a domain, the domain can predetermine the owner domain for objects newly created during execution in the domain's context. The standard owner domain is the domain the `Process` is currently executing in. However, it can be useful to create objects with a domain different from their initial owner. For example, objects created by an update function in the context of a Clojure agent should be owned by the immutable domain (cf. Sec. 2.4.3). The RoarVM+OMOP represents the initial owner via a field in the domain because it avoids executing another Smalltalk method on every object creation, while giving some flexibility with regard to the initial owner. For more complex use cases, it is always possible to change the owner of an object during the course of the execution. Furthermore, setting the owner during object initialization avoids race conditions on the object owner as compared to when it is set afterwards.

Changed Bytecode Semantics and Customization Constant To expose the OMOP's intercession points, i. e., to support MANAGED STATE and MANAGED EXECUTION, the RoarVM+OMOP adapts most bytecodes and primitives. The general idea is to check first, whether currently an enforced execution is required, and if that is the case, the target object's domain is inspected to determine whether a relevant intercession handler has been customized based on the *customization constant*. If this is the case, the intercession handler is invoked instead of executing the bytecode's or primitive's operation.

During enforced execution, bytecodes reify their operations and delegate them to the intercession handlers. Most of the bytecodes have trivial semantics, in terms of the stack operations they perform. Thus, a Smalltalk method, such as the intercession handlers can stand in for the actual bytecode operation without loss of flexibility. During enforced execution, the bytecode implementations can therefore perform message sends, similarly to the standard behavior of the send bytecode, instead of performing their operations directly. This approach corresponds directly to the one describing the overall semantics of the OMOP in Sec. 5.4.

[5] On a 32-bit architecture with 4 byte alignment, the first bit of an Oop is used to indicate whether an Oop is an immediate integer, or a references. The second bit remains unused, and is used to encode the enforcement.

The most important aspect here is that the intercession handlers that stand in for the original bytecode operations preserve the required stack semantics. Bytecodes that for instance push the value of a receiver's field or a global literal will just result in a message send instead. The return value of the message send will then reflect the result of the read operation. Message sends and simple bytecodes that store into an object's field are handled very similarly.

The delegation of bytecodes with *store and pop* behavior requires an additional intercession handler. The *store and pop* bytecodes from the Smalltalk-80 bytecode set are problematic. These are more complex to handle because they require that the top element of the stack be removed after the operation. However, this demands care, because all Smalltalk methods return a value, which gets pushed onto the stack after the method returns. The RoarVM+OMOP solved this problem by adding a special intercession handler `#write:toField:of:return:` in addition to the normal `#write:toField:of:` handler. It takes an additional parameter, the predetermined return value for the handler.

The code implementing this adaptation is given in Lst. 7.3. It shows the C++ code of the RoarVM+OMOP for the *store and pop* bytecode. As line 6 shows, two values are popped from the stack, instead of only the value that is to be stored into the receiver. The second value, which corresponds to what the top value of the stack should be after the *store and pop* bytecode, is given to the intercession handler as a last parameter, and will be restored when the handler returns. The calling conventions and the current compiler in Squeak and Pharo guarantee that this approach is safe. Theoretically, it is problematic to assume that there are always two elements on the stack that can be removed by the `pop` operation. However, in this situation it is guaranteed that there is the original element to be popped and in addition, there is at least the receiver object at the bottom of the stack, which is never subject to a *store and pop* bytecode. Thus, it is safe to pop the second element, which might be the receiver, and restore it when the message send returns.

The *send special message* bytecodes and the *quick methods* require proper adaptation as well (cf. Sec. 4.4.1). For message bytecodes, the RoarVM+OMOP needs to force the intercession point for an actual message send when the domain customizes the *request execution* intercession points. While the quick methods need to force the intercession point for the reading of the receiver's fields to be able to intercept them as any other field read with the intercession handler.

```
1  void storeAndPopReceiverVariableBytecode() {
2    if (requires_delegation(roots.rcvr,
3                            Domain::WriteToFieldMask)) {
4      Oop value  = stackTop();
5      Oop newTop = stackValue(1);
6      pop(2);
7      redirect_write_field(roots.rcvr,
8                           currentBytecode & 7,
9                           value, newTop);
10   }
11   else {
12     fetchNextBytecode();
13     receiver_obj()->storePointer(prevBytecode & 7, stackTop());
14     pop(1);
15   }
16 }
17
18 bool requires_delegation(Oop rcvr, oop_int_t selector_mask) {
19   if (executes_unenforced())
20     return false;
21
22   Oop rcvr_domain   = rcvr.as_object()->domain_oop();
23   Oop customization = rcvr_domain.customization_encoding();
24   return The_Domain.customizes_selectors(customization,
25                                          selector_mask);
26 }
27
28 void redirect_write_field(Oop obj_oop, int idx,
29                           Oop   value, Oop newTop) {
30   Oop domain = obj_oop.as_object()->domain_oop();
31   set_argumentCount(4);
32   roots.lkupClass       = domain.fetchClass();
33   roots.messageSelector = The_Domain.write_field_with_return();
34   create_context_and_activate(domain, value,
35       Oop::from_int(idx + 1), obj_oop, newTop);
36 }
```

Listing 7.3: Implementation of *store and pop* bytecodes.

Customization Constant In addition to showing the implementation of the *store and pop* bytecode, Lst. 7.3 also contains the specifics of the *customization constant* implementation. On line 2, the bytecode implementation calls the method `requires_delegation(..)` with the receiver (`rcvr`) object and a bit mask, which indicates a write operation. Line 18 defines the called method. It first looks up the domain owning the given receiver. Then, it looks up the customization constant in the domain object and uses another helper method that checks whether the constant has the corresponding bits of the bit mask set. If that is not the case, the bytecode implementation will continue executing at line 11 and directly perform the operation, as if the code was executed in the unenforced execution mode. This avoids the overhead of reifying the operation and performing it later reflectively. In case a delegation is required, the bytecode implementation will call `redirect_write_field(..)` instead of performing the write directly.

Handling of Primitives Primitives are handled similarly to bytecodes. Once the primitive code is reached it is already clear that the message send that led to it succeeded and does not need to be considered anymore. However, the primitive itself might still need to be intercepted by the domain. Thus, it needs to check whether the domain customizes the corresponding intercession handler. Lst. 7.4 provides an example of the adapted primitive for shallow copying of objects. As described, the receiver is first checked whether it customizes the corresponding handler, and if this is the case, the handler is invoked instead. As `request_primitive_clone()` demonstrates, we rely for that operation on the standard method invocation, i. e., `commonSend()`.

Strategy for Controlled Enforcement As mentioned before, each context object keeps track of whether the VM is supposed to execute the corresponding code in enforced or unenforced mode.

The VM offers two ways to switch to enforced execution. The first one is the `#evaluateEnforced:` primitive, which takes a block that is going to be evaluated with enforcement enabled. The second way is enabled by the variants of the `#performEnforced:` primitives, which take a selector, arguments, and optionally a lookup class for super sends. Both approaches cause either blocks or methods to be executed in enforced mode.

To leave enforced execution, methods are marked similarly to the solution for SOM and the AST-OMOP. The RoarVM+OMOP implementation uses bit 30 (`UN`) in the method header (cf. Tab. 7.1) to indicate to the VM that the

```
1  void primitiveClone() {
2    Oop x = stackTop();
3
4    if (requires_delegation(x, Domain::PrimShallowCopy__Mask)) {
5      request_primitive_clone();
6      return;
7    }
8
9    if (x.is_int()) // is immediate value, no copying necessary
10     return;
11
12   Oop nc = x.as_object()->clone();
13   popThenPush(1, nc);
14 }
15
16 void request_primitive_clone() {
17   Oop value  = stackTop();
18   Oop domain = value.as_object()->domain_oop();
19
20   popThenPush(1, domain);
21   push(value);
22
23   set_argumentCount(1);
24
25   roots.messageSelector = The_Domain.prim_shallow_copy();
26   roots.lkupClass       = domain.fetchClass();
27
28   commonSend(roots.lkupClass);
29 }
```

Listing 7.4: Adapted primitive for shallow object copies.

method and its dynamic extent have to be executed in unenforced mode. If the bit is not set, the execution mode of enforced or unenforced execution remains unchanged on method activation.

Table 7.1.: RoarVM+OMOP Method Header: bit 30 is used to force unenforced method execution.

	UN	P.	#args	#temporaries	FS	#literals	Primitive	1
31	30	29	28 27 26 25	24 23 22 21 20 19	18	17 16 15 14 13 12 11 10	9 8 7 6 5 4 3 2 1	0

VM Interface Since part of the functionality is represented in the VM, and part is represented in the Smalltalk image, both have to agree on a common interface.

Intercession Handlers To enable the VM to invoke the intercession handlers, it requires knowledge about the corresponding selectors. A common strategy for such a case in Smalltalk VMs is the use of the `specialObjectsArray`. All objects or classes the VM needs to be aware of are registered in this array at a predetermined index. Similarly, the selectors for the intercession handlers are encoded as an array that is referenced from a fixed index of this `specialObjectsArray`.

The indices into the selector array are encoded in a header file of the VM. This header file is generated by the image together with the array, but the VM needs to be recompiled whenever the header and the array change.

Customization Constant The generated VM header file also encodes the bit masks corresponding to the intercession points, which enable a decoding of the customization constant of a domain to determine whether the domain requires the intercession point to be taken. The image itself uses a consistency check to verify that decoding of the customization is done correctly by both VM and image. The customization constant itself is encoded in each domain object in the field: `domainCustomizations`.

Initial Owner of Objects The domain objects have also a field to denote the initial owner for newly created objects (`initialDomainForNewObjects`). Thus, the VM can directly determine the new owner without the need for executing

Table 7.2.: VM Primitives for the OMOP

`#performEnforced:`	Invoke selector with the enforced execution mode. Arguments are passed on the stack.
`#performEnforced:args:`	Invoke selector with the enforced execution mode. Arguments are passed as an array.
`#performEnforced:args:lkupCls:`	Invoke selector with the enforced execution mode. Arguments are passed as an array and lookup starts in a given class.
`#evaluateEnforced:`	Evaluate the given block with the enforced execution mode.
`#executesEnforced`	Query whether current execution is enforced or unenforced.
`#getDomainOf:`	Get the owner domain of an object.
`#setDomainOf:to:`	Set the owner domain of an object to the given domain.

a Smalltalk method. The RoarVM+OMOP uses this approach to avoid the otherwise significant overhead for ever object creation.

Unenforced Methods Methods encode in their header the `unenforced` bit, which is set during method compilation when the `<unenforced>` pragma is present in the method body.

The primitives provided as part of the VM interface are listed in Tab. 7.2.

7.2.2. Discussions

Customization Constant for Debugging The use of a customization constant, which for every domain object encodes the set of intercession handlers that have been customized, greatly reduces the amount of Smalltalk code that is processed during enforced execution. Sec. 8.5.3 evaluates the performance benefits of using this customization constant. However, the main reason for

introducing it was not performance, but the system's debuggability needed to be improved.

Since most domains customize only a subset of all intercession handlers, it becomes a major problem to have all intercession handlers be invoked. While the standard intercession handlers only encode the semantics of unenforced Smalltalk execution, they significantly increase the amount of code that is executed for each operation with a corresponding intercession handler. For the purpose of debugging, the execution of this code only adds noise and results in additional code execution that one needs to *step through*, without relevance to the semantics.

During debugging at the VM bytecode level as well as during debugging at the Smalltalk source level, it was essential to avoid such noise in order to be able to focus on the problem at hand. The customization constant reduced unnecessarily executed code significantly and consequently improved the debugging experience for code executed in the enforced mode.

Representation of Ownership The RoarVM+OMOP represents object ownership by using an extra header word for every object. While this direct approach enables the VM to obtain the owner simply by object inspection, it comes with a significant memory overhead for small objects. In the field of garbage collection, different solutions have been proposed to track similar metadata for various purposes. One possible approach that is common for GCs is to partition the heap [Jones et al., 2011, chap. 8] based on certain criteria to avoid the direct encoding of related properties. For the OMOP, the ownership of an object could be encoded by it being located in a partition that belongs to a specific domain. Such an approach would avoid the space overhead of keeping track of ownership. While partitioning might add complexity to the memory subsystem of the VM, it might also open up opportunities for other optimizations. Depending on the use of the domains, and the concurrency concepts expressed with them, a GC could take advantage of additional partitioning. For instance, in an actor-like domain, additional partitioning could have benefits for generational collectors. In such a system, the nursery partition would be local to a single actor, which could have a positive impact on GC times.

Limitations of Primitives A final semantic issue is the handling of primitives. By reifying their execution and requiring the indirection to the domain object, the execution context changes when the primitive is applied. In

the RoarVM, primitives are executed on the current context object, which is different from normal method activations, which are done in their own context object. When primitive execution is intercepted by the domain, the RoarVM+OMOP cannot provide the same semantics. An intercession handler executes in its own context object, i. e., stack frame, and consequently if it invokes the primitive, it executes on the handler's stack frame instead of the stack frame it has originally been invoked in. One example where that matters is a primitive that implements loops in SOM. It resets the instruction pointer in the context object to achieve looping. In the RoarVM+OMOP, this is not possible. However, while this design restricts what primitives can implement, it is not a practical issue for the RoarVM codebase. The codebase does not contain primitives that use the context objects in problematic ways, even though the codebase includes the plugins reused from Squeak and the CogVM (cf. Sec. 4.4). The primitives merely work on the parameters provided to them and restrict access to the context object to the corresponding stack operations. Thus, the RoarVM+OMOP did not need to solve this limitation and it is assumed that it does not pose a problem in practice.

7.3. Summary

This chapter presented two implementation approaches for the OMOP. It details for each of the approaches how the mechanisms of the OMOP are realized and how the requirements are fulfilled.

The AST-OMOP is based on AST transformation and operates on top of the VM without requiring changes to it. The transformations add the necessary operations to trigger the OMOP's intercession handlers, e. g., on method execution, access to object fields or globals, or to intercept primitives. The AST-OMOP was the first prototype to investigate the ideas of using a metaobject protocol and to build the case studies on top, i. e., the AmbientTalkST and the LRSTM implementation. Thus, the AST-OMOP facilitated the evaluation of part of the thesis statement without requiring changes to the VM.

The RoarVM+OMOP is the implementation that is used to evaluate the performance part of the thesis statement. It adapts a bytecode interpreter to directly support the OMOP. The bytecode and primitive implementations are changed to trigger the OMOP's intercession handlers instead of performing their original operations if execution is performed enforced, and if the corresponding intercession handler has been customized. The check whether the intercession handler is customized is a simple optimization that facilitates debugging.

8

EVALUATION: PERFORMANCE

The goal of this chapter is to evaluate the performance of the AST-OMOP and the RoarVM+OMOP, by comparing the OMOP-based and ad hoc implementations of LRSTM and AmbientTalkST. This assessment is based on the assumption that application developers already use concurrent programming based on systems similar to LRSTM or AmbientTalkST. Therefore, this evaluation needs to show that applications can reach on par performance when LRSTM and AmbientTalkST are implemented on top of the OMOP compared to being implemented with ad hoc strategies.

This chapter first details the evaluation methodology and the generalizability of the results. These results allow conclusions for other bytecode interpreters. However, they preclude conclusions for high-performance VMs.

Subsequently, the performance comparison of the ad hoc and OMOP-based implementations of AmbientTalkST and LRSTM shows that the OMOP can be used to implement concurrent programming concepts efficiently. Currently, the OMOP provides a sweet spot for implementations that rely on the customization of state access policies as used in LRSTM. The evaluation further shows that the overhead of the AST-OMOP is high, while the overhead of the RoarVM+OMOP is significantly lower. The customization constant optimization introduces inherent overhead, but offers performance benefits when only a small number of intercession handlers are customized. Overall, the comparison of AST-OMOP and RoarVM+OMOP indicates the need of VM support for the OMOP to obtain performance on par with ad hoc implementations.

8.1. Evaluation Strategy

This section details the goals of the performance evaluation, outlines the experiments and their rationale, and the generalizability of the results.

8.1.1. Evaluation Goal

The goal of this chapter is to evaluate the last part of the thesis statement (cf. Sec. 1.3). Thus, it needs to evaluate whether the OMOP *"lends itself to an efficient implementation"*.

This evaluation is based on the assumption that application developers rely on concurrent programming concepts to trade off performance and engineering properties in order to build applications that satisfy their requirements. Thus, this evaluation is not required to assess the absolute performance of all of the concurrent programming concepts. Instead, it needs to show that OMOP-based implementations can perform on par with ad hoc implementations. Therefore, this evaluation compares the performance between individual ad hoc and OMOP-based implementations. Furthermore, it assesses the performance impact of the different implementation strategies for the OMOP.

8.1.2. Experiments and Rationale

Before detailing the experiments, this section briefly recapitulates the motivation for using the RoarVM as platform for these experiments. Furthermore, it argues the choice of LRSTM and AmbientTalkST as case studies. Finally, it outlines the experiments and describes which aspect they assess.

Motivation for AST-OMOP and RoarVM+OMOP The experiments with the AST-OMOP indicated that its performance is not on par with ad hoc solutions [Marr and D'Hondt, 2012]. Therefore, this dissertation also needs to evaluate whether VM support can improve on these results. The available VMs for this experiment are CogVM and RoarVM (cf. Sec. 4.4). The CogVM uses a just-in-time compiler that provides good performance (cf. Sec. 8.3). Compared to it, the RoarVM is a research interpreter, which is not optimized for performance. However, we used the RoarVM in earlier experiments and had ported it from the Tilera manycore processor to commodity multicore systems (cf. Sec. 4.4), and thus were familiar with its implementation.

The RoarVM was chosen over the CogVM for its significantly lower implementation complexity,[1] enabling the evaluation of the benefits of VM support with a significantly lower implementation effort.

Motivation for using LRSTM and AmbientTalkST The performance evaluation relies on the AmbientTalkST and LRSTM case studies to compare the performance of ad hoc implementations with OMOP-based implementations. The evaluation restricts itself to these two case studies, because the semantic differences between the ad hoc implementation of Clojure agents and the OMOP-based implementation are significant (cf. Sec. 6.2.1) rendering any performance result incomparable.

However, AmbientTalkST and LRSTM yield two relevant data points that represent two major concurrent programming concepts. As argued in Sec. 2.5, event-loop actors are a natural fit for the implementation of event-driven user-interfaces. JCoBox [Schäfer and Poetzsch-Heffter, 2010] and AmbientTalk [Van Cutsem et al., 2007] show that these ideas are explored in research. Furthermore, WebWorker[5] and Dart[6] demonstrate the adoption in industry. Software transactional memory found adoption in languages such as Clojure[7] or Haskell[8] and as a compiler extension for GNU GCC 4.7.[9]

Furthermore, these two case studies customize distinct intercession handlers of the OMOP, and thus are subject to different performance effects. AmbientTalkST primarily customizes method execution (cf. Sec. 6.2.3), while LRSTM customizes state access and primitives execution (cf. Sec. 6.2.2).

In order to have comparable results, both AmbientTalkST implementations observe the same semantics, i. e., neither implementation enforces state encapsulation for global state. While Sec. 6.2.3 discussed a solution for state encapsulation based on the OMOP, providing the same degree of encapsulation for the ad hoc implementation would have required significant addi-

[1] Our discussions with Eliot Miranda, author of the CogVM, at the *Deep into Smalltalk school*,[2] the VMIL'11 workshop,[3] and as a co-mentor for the 2012 *Google Summer of Code* project to port the CogVM to ARM,[4] made it clear that the CogVM would requires engineering and research effort that goes beyond the scope of this dissertation (cf. Sec. 9.5.2).

[2] http://www.inria.fr/en/centre/lille/calendar/smalltalk

[3] http://www.cs.iastate.edu/~design/vmil/2011/program.shtml

[4] http://gsoc2012.esug.org/projects/arm-jitter

[5] http://www.whatwg.org/specs/web-apps/current-work/multipage/workers.html

[6] http://api.dartlang.org/docs/continuous/dart_isolate.html

[7] http://clojure.org

[8] http://www.haskell.org

[9] http://gcc.gnu.org/wiki/TransactionalMemory

tional complexity. Instead, both implementations accept a common compromise [Karmani et al., 2009] and weaken state encapsulation. This compromises on the semantics, but makes both implementations semantically equivalent and provides performance results that can be meaningfully compared.

Choice of Experiments The performance comparison of the ad hoc and the OMOP-based implementations in Sec. 8.4 relies on a set of custom microbenchmarks that assess the performance of specific parts of the AmbientTalkST and LRSTM implementations. A characterization of these benchmarks is given in Appendix B.1.

Additionally, the evaluation uses kernel benchmarks chosen from the *Computer Language Benchmarks Game*[10] and other sources. These benchmarks have been chosen based on their characteristics as well as their availability. Kernel benchmarks exercise different parts of the runtime as part of small-sized programs. They cover aspects such as object graph traversal, a compilation, string manipulation, and different computational workloads. Appendix B.1 characterizes them and shows where they have been used in the literature to support their validity.

With the selected benchmarks, the evaluation can assess specific parts of the implementations and the performance for higher-level workloads. However, since these benchmarks are on a small scale, they can only give weak indications for the performance of concrete applications.

With these benchmarks Sec. 8.5 assesses the implementation characteristics of AST-OMOP and RoarVM+OMOP in more detail. In a first step, the evaluation measures the overall enforcement overhead of both OMOP implementations. Thus, it measures the cost of triggering intercession handlers instead of directly performing the corresponding operations in unenforced execution mode. The goal here is to assess and compare the absolute overhead of the OMOP while comparing implementations with equal semantic properties.

The RoarVM+OMOP implementation adapts the semantics of bytecodes and introduces additional operations on the critical path of the bytecode interpreter (cf. Sec. 7.2.1. Sec. 8.5.2 evaluates the inherent overhead of these changes. This overhead can have a significant impact on overall application performance, because it is independent from OMOP usage.

Sec. 7.2.1 discusses an optimization proposed for the RoarVM+OMOP implementation that avoids the cost of triggering intercession handlers for uncustomized operations. To determine the benefit of this optimization, the pro-

[10] http://shootout.alioth.debian.org/

posed evaluation measures its inherent overhead and its gains. It measures its overhead on unenforced execution, and uses LRSTM and AmbientTalkST to measure its performance benefit.

The evaluation is completed by measuring the absolute performance of AST-OMOP and RoarVM+OMOP. While these absolute numbers do not contribute to the evaluation goal, they are a relevant data points for the assessment of whether the OMOP can be applied in its current form.

8.1.3. Virtual Machines

The evaluation uses the CogVM and the RoarVM. This section briefly lists the basic information required to recreate the setup. Furthermore, it describes the characteristics of the RoarVM variants used in the experiments.

CogVM The used CogVM (cf. Sec. 4.3) is the official and unmodified VM for the Squeak and Pharo projects. It has been obtained from the *continuous integration* server of the Pharo project.[11] It was compiled on Aug. 13th, 2012, and identifies itself as a *6.0-pre* release.

RoarVM The RoarVM used in these experiments is based on the official sources,[12] but contains a number of customizations. In total, three variants of the RoarVM without OMOP support are used. *RoarVM (std)* is the variant that uses the standard settings of the VM and supports parallel execution. It is compiled with Clang 3.0,[13] since Clang is the compiler used during development. In order to assess whether the performance differences are significant between Clang 3.0 and the standard compiler of OS X 10.6, i. e., the GNU GCC 4.2,[14] the *RoarVM (GCC 4.2)* is included in the experiments. The only difference with RoarVM (std) is the use of the GNU GCC 4.2 compiler. Independent of the compiler, all RoarVM variants are compiled with maximum optimizations, i. e., with the `-O3` compiler flag.

Parallel Execution vs. Overhead Assessment For improved performance and to expose the introduced overhead of the OMOP more clearly, this evaluation uses *RoarVM (opt)*, which is configured with non-standard settings.

[11] https://ci.lille.inria.fr/pharo/view/Cog/job/Cog-VM/
[12] https://github.com/smarr/RoarVM
[13] http://clang.llvm.org/
[14] To be precise, the compiler identification is: (GCC) 4.2.1 (Based on Apple Inc. build 5658) (LLVM build 2336.1.00).

The goal of the applied optimizations is to ensure that the performance overhead introduced by the OMOP is not hidden by the interpretation overhead and other performance reducing factors. The main assumption here is that the OMOP introduces a constant amount of time as overhead for each of the adapted operations. Therefore, a reduction of the overall execution time more clearly exposes the introduced constant overhead per operation.

The *RoarVM (opt)* configuration does not support parallel execution, drops the use of the object table, comes without garbage collection, and uses direct access to important globals such as the interpreter object, and the memory system object. While these settings might sound like a strong restriction and an artificial setup, they are close to the settings for a experimental version of the RoarVM that supports parallel execution. One insight during the work on the RoarVM was that access to thread-local variables carries a significant performance penalty compared to direct access to global variables in a sequential version. Hence, we implemented a variant of the RoarVM that uses processes instead of threads to enable the utilization of multiple cores. This variant does not pay the penalty of thread-local variables and can benefit from compiler optimizations for statically known global variables. Thus, using direct access to global variables is one optimization applied to the *RoarVM (opt)*. By avoiding the overhead of the object table, it gains additional performance. While the current GC implementation is not adapted to work without an object table, supporting this setting is relatively straightforward and common in other VMs. Therefore, the configuration of *RoarVM (opt)* is not an artificial one, but is based on reasonable assumptions and can be made to support parallel execution. Furthermore, by improving overall performance any additional constant overhead introduced by the OMOP can be determined more clearly.

8.1.4. Generalizability and Restrictions of Results

Generalizability to Application Performance Benchmarks model only certain sets of behaviors and do not necessarily correspond to actual applications. Thus, their generalizability is typically restricted. For these reasons Blackburn et al. [2006] propose a suite of actual applications as benchmarks for JVMs. Since no such benchmark suite exists for Smalltalk, the evaluation relies on microbenchmarks and kernel benchmarks. Therefore, benchmarks can only give an indication of the order of magnitude of performance of applications, precluding predictions about the performance of concrete applications.

Generalizability beyond Interpreters The low performance of the RoarVM precludes any generalization beyond interpreters. Since its implementation is comparatively inefficient, it can hide performance effects that are caused by the OMOP implementation. In any highly optimized VM implementation, newly introduced overhead will have a higher impact than in an implementation that is already slow to begin with.

Consequently, it is not possible to generalize the results to highly optimized VMs that use JIT compilers. This is a serious limitation, but addressing it is outside the scope of this dissertation and part of future work (cf. Sec. 9.5.2). However, the results are transferable to common bytecode-based interpreters, e. g., Lua 5.2, Python 3.3, Ruby 1.9, because they have similar execution characteristics.

8.2. Methodology

This section gives an overview of the methodology used for the execution of the experiments as well as for the reporting in this dissertation.

8.2.1. Precautions for Reliable Results

The benchmarking methodology is derived from the advice of Georges et al. [2007] for a practical statistically rigorous performance evaluation. Consequently, it takes the following precautions for setting up the experiments:

- Measure steady-state performance: bytecode-interpreter VMs do not have a warmup phase, however the CogVM (cf. Sec. 4.3) with its baseline JIT compiler needs proper warmup to reach its steady-state. Currently, it is sufficient for the CogVM to execute the benchmark twice before the effectively measured run to ensure that the code is compiled and ready. Since the CogVM does not support adaptive compilation, steady state is reached and performance will remain unchanged for subsequent execution.

- Minimize nondeterminism incurred by concurrently executing threads: Since the performance evaluation focuses on the general performance impact (cf. Sec. 8.1.3) and since the OMOP itself does not affect the characteristics of parallel execution,[15] this evaluation focuses on sequential

[15] As shown with the case studies (cf. Sec. 6.2), the OMOP does not change execution semantics, but enables language implementers to enforce desired semantics more directly. It does

benchmarks, i. e., benchmarks that are executed on a single processor core. This removes a strong source of noise in the results and improves the quality of our measurements.

- Minimize nondeterminism incurred by Intel's TurboBoost: Modern processors often support dynamic frequency changes for processor cores to manage power consumption and heat dissipation. To avoid an influence of such techniques on measurements, they need to be disabled.[16]
- Minimize nondeterminism incurred by garbage collection: To avoid measurement noise coming from different GC strategies in the different VMs, the heap size are adjusted to a large enough value of 1024MB, and thus, GC operations are avoided.
- Problem sizes: The problem size of all benchmarks is adjusted to yield runtimes in the order of hundreds of milliseconds, ideally between 300ms to 1000ms. In some experiments, the runtimes go well beyond this range, because of the different performance characteristics of the VMs used. However, fixing the lower bound to an appropriate value avoids influence of potentially imprecise timer resolutions provided by the different VMs.

Execution With these precautions in place, systematic bias in the experiments is reduced. However, there remain other nondeterministic influences. For instance, operating system features such as virtual memory and scheduling of other processes cannot completely be disabled. Adjusting process priority helps partially, but other techniques such as caching in the processors remain. The remaining influences are handled by executing each benchmark 100 times to obtain statistically significant results.

Hardware And Software Setup The machine used for these experiments is a MacPro (version 4,1) with two quad-core Intel Xeon E5520 processors at 2.26 GHz with 8 GB of memory. Note that since the benchmarks are purely sequential, the number of cores is not relevant. The operating system is OS X 10.6.7 (10K549). The benchmarks are implemented using the SMark bench-

not provide mechanisms that interfere with parallel execution: it facilitates concurrent programming in contrast to parallel programming (cf. Sec. 2.3.2).

[16] On Mac OS X this is possible by using a kernel module: `https://github.com/nanoant/DisableTurboBoost.kext`

marking framework[17]. The benchmarks are executed based on a configuration that consists of the lists of benchmarks and their parameters, as well as the VMs and the necessary command-line arguments. ReBench[18] uses this configuration to execute the benchmarks accordingly. The configuration is given in Appendix B.2.

For the CogVM, an official 32-bit binary is used (cf. Sec. 8.1.3). The RoarVM is equally compiled as a 32-bit binary. The compiler used is either Clang 3.0 or GNU GCC 4.2 with full compiler optimization enabled, i. e., using the compiler switch `-O3`.

8.2.2. Presentation

Beanplots Most of the following diagrams use beanplots [Kampstra, 2008]. Beanplots are similar to violin and box plots. They show the distribution of measurements, and thus, enable an visual comparison of benchmark results, including an assessment of the significance of the observed differences (cf. Fig. 8.1). Note that the beanplots are directly based on the measured data points, i. e., they depict the distribution of the actual measurements and are not synthesized from aggregated values. Asymmetric beanplots such as in Fig. 8.3 are a variant that facilitates direct comparison of two data sets by depicting both data sets for each criterion side-by-side. The first example in the following section discusses the semantics of these diagrams in more detail.

Geometric Mean Note that all averages reported in this dissertation are based on the geometric mean. For most experiments normalized measurements are reported, and thus, the geometric mean needs to be used because the arithmetic mean would be meaningless (cf. Fleming and Wallace [1986]).

8.3. Baseline Assessment

This section discusses baseline performance of the VMs used for the experiments. The goal of this discussion is to provide an intuition of the performance characteristics of the individual VMs before discussing the impact of the modifications for the OMOP. This evaluation uses the kernel benchmarks described in Appendix B.1.2. As discussed in Sec. 8.2, every benchmark is executed 100 times for each of the VMs. All benchmarks have been executed with identical parameters.

[17] http://www.squeaksource.com/SMark.html
[18] http://github.com/smarr/ReBench

General Performance Ratios The beanplot in Fig. 8.1 is a visualization of the distribution of the kernel benchmark results for three RoarVM variants, as well as the results for the CogVM.

Note that Fig. 8.1 uses an unconventional approach to represent overall performance of the VMs. The presented data have not been aggregated to produce the plot. Instead, it is an overview of *all* measurements, which Fig. 8.2 presents in more detail. The goal of this plot is to combine all measurements and present them so as to compare performance effects of different VMs within a single representation.

The experiment uses eight kernel benchmarks for the measurements and the bean plot for the RoarVM (opt) shows these eight benchmarks as eight distinct accumulations of measurements. Each of the benchmarks has a distinct average around which the measurements are concentrated. This becomes visible in this representation as pearls on a necklace. For each of the VMs, the plot is generated from 800 distinct measurements, which are depicted as distributions.

Comparing the *RoarVM (opt)* and the *RoarVM (std)* shows that there are two benchmarks that are affected less by the differences in VMs than the other six benchmarks, i. e., the two pearls at the bottom move barely visible, while the other six pearls move up and also differ in their distance between each other. This representation visualizes the impact on the execution of the corresponding benchmarks, but only provides a rough overview without detail.

The only aggregation in this representation is the black average line, which averages over all measurements for a VM, and provides an additional intuition of the overall performance compared to each other.

Since the overall performance of the CogVM is a magnitude better, results are closer together and the individual benchmarks become indistinguishable for the scale chosen for the plot. A plot with a scale from 0 to 500 would reveal a similar pattern as is visible for the RoarVM variants.

The main insight to be derived from this plot is that the CogVM is significantly faster than any of the RoarVM variants. Furthermore, the differences between the RoarVM variants have a different impact on the benchmarks.

Consider all benchmarks separately as depicted in Fig. 8.2, the CogVM is on average 11.0x[19] faster than the RoarVM (opt).[20] Depending on the benchmark, the difference is between 7.1x and 14.4x. These performance variations for the different benchmarks are a result of their specific characteristics and their

[19] The reported averages in this chapter consistently refer to the geometric mean.

[20] A speedup or slowdown reported as nx refers to the ratio $VM_1/VM_{baseline}$. In this case it is: CogVM/RoarVM (opt).

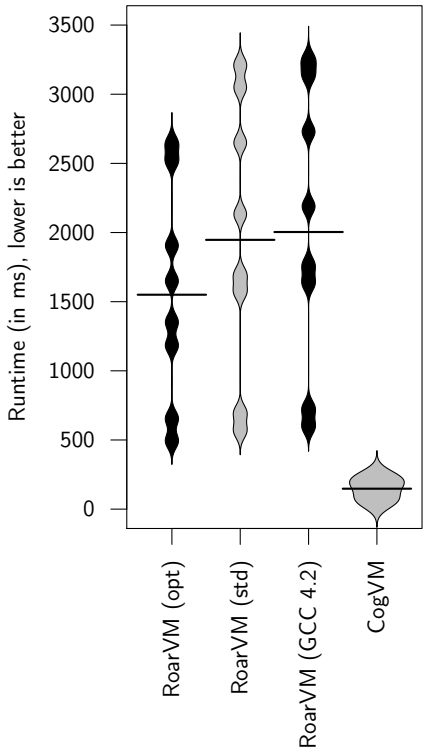

Figure 8.1.: Baseline Performance: Runtime measures of all kernel benchmarks for the different VMs. The beanplot (cf. Sec. 8.2) shows the direct, unaggregated distribution of the measured data points. The black horizontal line for each VM is the average over all data points for the VM. The *pearls* correspond to the different benchmarks, and show the accumulation of multiple measurements. The plot indicates the relative performance of the different VMs. The CogVM is on average 11.0x faster than the RoarVM (opt). The RoarVM (opt) is on average 23% faster than RoarVM (std).

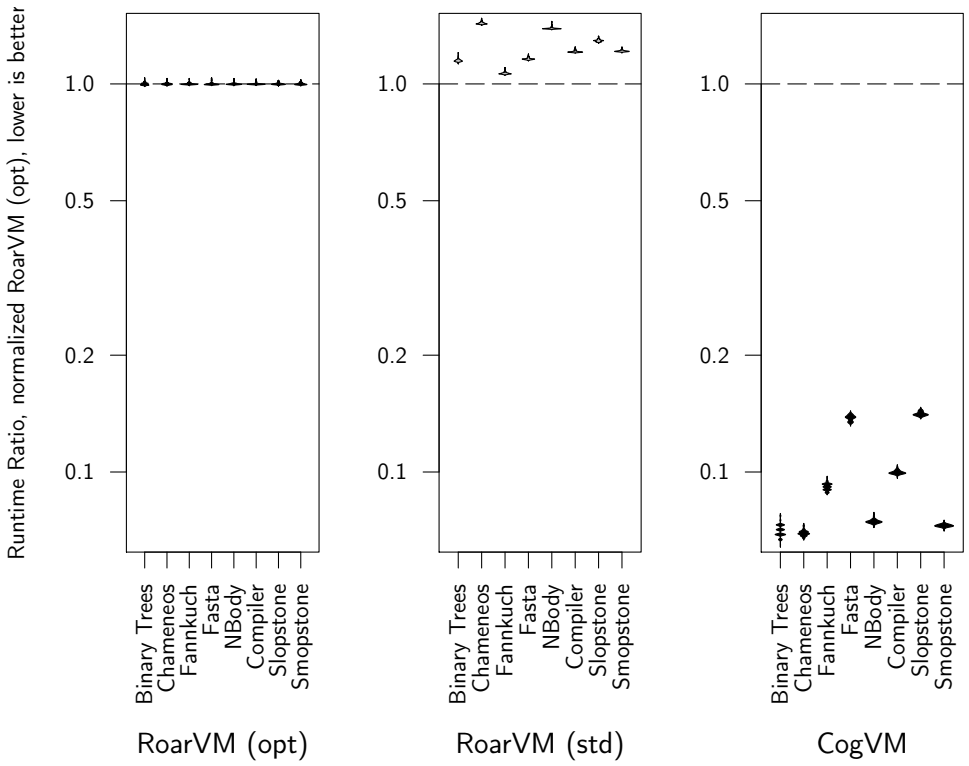

Figure 8.2.: Baseline Performance: Runtime ratio normalized to the RoarVM (opt) for the kernel benchmarks. This beanplot gives an overview of the relative performance of each of the VMs separately for each benchmark. The plot, as well as all other plots depicting ratios, uses a logarithmic scale for the y-axis.

different use of primitives, allocation, arithmetic, and stack operations. See Appendix B.1 for a characterization of the benchmarks.

RoarVM Variants Differences between the RoarVM variants are significant as well, but less pronounced. The RoarVM (opt) is about 23%[21] faster than the RoarVM (std). Here the performance difference ranges from a minimal speedup of 6% to a maximum speedup of 43% for the kernel benchmarks. In view of this significant difference, the RoarVM (opt) is used as baseline for the remaining benchmarks. As argued in Sec. 8.1.3, the optimizations applied to the RoarVM are realistic. Therefore, any constant overhead introduced by

[21] A speedup or slowdown reported as n% refers to $VM_1/VM_{baseline} * 100 - 100$. In this case it is: (RoarVM (opt)/RoarVM (std)$*100$) $- 100$

the OMOP is likely to show up more significantly in the RoarVM (opt) with its improved performance.

Since the Clang 3.0 compiler is used for development and experiments, its performance is compared to OS X's standard compiler GCC 4.2. Both compilers were instructed to apply full optimizations using the `-O3` compiler switch (cf. Sec. 8.2).

The results are depicted in Fig. 8.3. This beanplot uses asymmetric beans to allow an intuitive comparison between the results for Clang and GCC. The results are normalized with respect to the mean runtime of the RoarVM compiled with Clang 3.0. Thus, the black line indicating the mean is directly on the 1.0 line, i. e., the baseline. The resulting distributions for GCC are depicted in gray and indicate that the Clang compiler produces a slightly better result. The RoarVM compiled with GCC is on average 3% slower and the results vary over the different benchmarks between 1% and 5%. Since this difference is insignificant compared to the overall performance differences between RoarVM and CogVM, the evaluation proceeds with the Clang 3.0 compiler.

Another important aspect depicted in Fig. 8.3[22] is the relatively wide range of measurement errors. The measurements show outliers up to 10%, even though, the setup eliminates a range of common causes for nondeterminism (cf. Sec. 8.2), others such as optimizations in the operating system, and caching remain and need to be considered as potential measurement bias. For the given results however, the distribution of measurements is visible in the graph and indicates that the measured results are statistically significant.

Conclusion The measurements of the baseline performance of the RoarVM and the CogVM indicate that the CogVM is about 11.0x faster on the kernel benchmarks than the RoarVM (opt) (min 7.1x, max 14.4x). Consequently, the CogVM is used as execution platform for the AST-OMOP, so as to provide it with the best possible execution performance available.

The RoarVM (opt) is about 23% faster on the kernel benchmarks than the RoarVM (std) (min 6%, max 43%). This evaluation therefore relies on these optimizations to have an optimal baseline performance in order to assess the overhead that the changes for the OMOP introduce.

[22] The beanplot library of R enforced a linear scale for this graph.

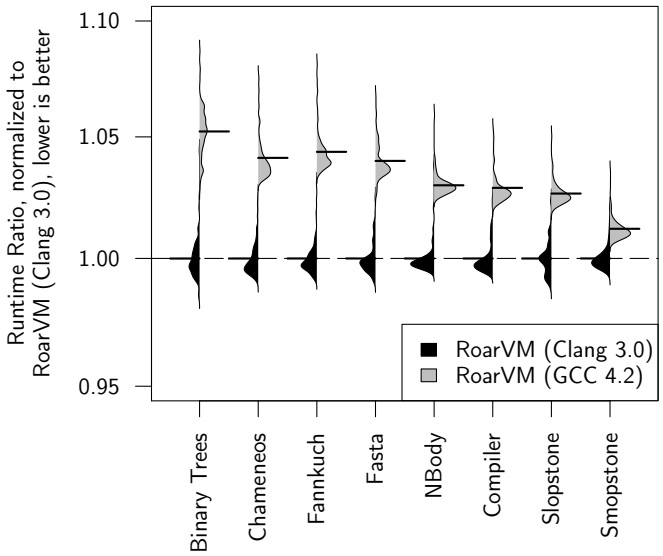

Figure 8.3.: Baseline Performance: Comparing RoarVM compiled with Clang 3.0 and GCC 4.2. All results are normalized to the mean of the measurement for Clang. The beanplot shows the distribution of measurements and indicates that GCC 4.2 produces a binary that is about 3% slower, which is not significant in view of the differences measured between the other VMs.

8.4. Ad hoc vs. OMOP Performance

When assessing the practicality of the OMOP compared to ad hoc approaches for implementing concurrent programming concepts, the main question is whether an implementation based on the OMOP will be required to accept performance penalties for using the extra level of abstraction. In short, the experiments discussed in this section indicate that this is not necessarily the case and OMOP-based implementations can be on par with ad hoc implementations.

Setup To compare the performance between OMOP-based and ad hoc implementations, the proposed experiments use the LRSTM and AmbientTalkST variants of the benchmarks and measure the runtime for the following eight configurations:

With this setup, on the one hand the performance of ad hoc implementations can be compared to OMOP-based implementations, and on the other hand, the efficiency of the AST-transformation-based (AST-OMOP) and the

Table 8.1.: Experiments for Evaluation of Ad Hoc vs. OMOP-based Performance

	Ad Hoc	vs	*OMOP-based*
AmbientTalkST	CogVM	vs	CogVM with AST-OMOP
	RoarVM (opt)	vs	RoarVM+OMOP (opt)
LRSTM	CogVM	vs	CogVM with AST-OMOP
	RoarVM (opt)	vs	RoarVM+OMOP (opt)

VM-based implementation (RoarVM+OMOP) can be compared. Ad hoc implementations are executed on top of the RoarVM (opt) without OMOP support to reflect the most plausible use case. Since VM-support for the OMOP introduces an inherent performance overhead (cf. Sec. 8.5.2), it would put the ad hoc implementations at a disadvantage if they were to execute on top of RoarVM+OMOP (opt).

The proposed evaluation only measures minimal variation and uses a simple bar chart with error bars indicating the 0.95% confidence interval. Since measurement errors are insignificant, the error bars are barely visible and the resulting Fig. 8.4 is more readable than a beanplot would have been. Fig. 8.4 only shows microbenchmarks. The runtime has been normalized to the mean of the ad hoc implementation's runtime measured for a specific benchmark. The y-axis shows this runtime ratio with a logarithmic scale, and thus, the ideal result is at the 1.0 line or below, which would indicate a speedup. All values above 1.0 indicate a slowdown, i. e., the benchmark running on top of the OMOP-based implementation took more time to complete than the benchmark on top of the ad hoc implementation.

Results for Microbenchmarks The first conclusion from Fig. 8.4 is that most LRSTM benchmarks on RoarVM+OMOP show on par or better performance (gray bars). Only *Array Access (STM)* (28%), *Class Var Binding (STM)* (14%), and *InstVar Access (STM)* (18%) show slowdowns. This slowdown can be explained by the general overhead of about 17% for OMOP support in the VM (cf. Sec. 8.5.2).

The AmbientTalkST benchmarks on the other hand experience slowdowns throughout. These benchmarks indicate that the differences in how message sends and primitives are handled in the ad hoc and the RoarVM+OMOP implementation, have a significant impact on performance.

The results for the AST-OMOP implementation show more significant slowdowns. The main reason for slowdown is that the implementation has to reify

all OMOP operations and does not benefit from an optimization similar to the customization check that is done by the RoarVM+OMOP implementation (cf. Sec. 7.2.1 and Sec. 8.5.3). In this case, the ad hoc implementations are more precisely tailored to the needs of the actor or STM system and do not show the same overhead.

Results for Kernel Benchmarks The kernel benchmarks depicted in Fig. 8.5 show a very similar result, even though the results are less extreme than for the microbenchmarks. The VM-based implementation comes very close to the ideal performance of the ad hoc implementations. The average overhead is about 28% on the given benchmarks. However, the overhead ranges from as low as 2% up to a slowdown of 2.6x. The overhead greatly depends on the requirements of AmbientTalkST and LRSTM with respect to which intercession handlers of the OMOP have to be customized. Considering only the LRSTM benchmarks, they have an average slowdown of 11% (min 2%, max 17%), which is lower than the inherent overhead of 17% caused by the VM changes that add the OMOP (cf. Sec. 8.5.2). Thus, the OMOP provides a suitable unifying substrate that can facilitate efficient implementation of abstractions for concurrent programming.

However, the AmbientTalkST implementation is about 48% (min 20%, max 2.6x) slower on the RoarVM+OMOP. Sec. 5.5 and Sec. 9.5.3 discuss potential optimizations that could reduce this overhead for all concurrency concepts that are similar to actors, Clojure agents, and CSP.

While the RoarVM+OMOP exhibits good performance characteristics compared to the ad hoc implementations, the AST-OMOP performs significantly worse. While the Fannkuch (AT) benchmark shows 5% speedup, the average runtime is 2.8x higher than the runtime of corresponding ad hoc implementations. For the NBody (AT) benchmark it is even 17.4x higher.

Conclusion The experiments show that OMOP-based implementations can reach performance that is on par with ad hoc implementations. Thus, the OMOP is a viable platform for the implementation of concurrent programming concepts. The results indicate that a VM-based implementation can lead to the ideal case of performance neutral behavior while providing a common abstraction. However, there is currently a sweet spot for abstractions similar to an STM that customize state accesses.

Furthermore, results show that an AST-transformation-based implementation can show a significant performance overhead. However, this may be ac-

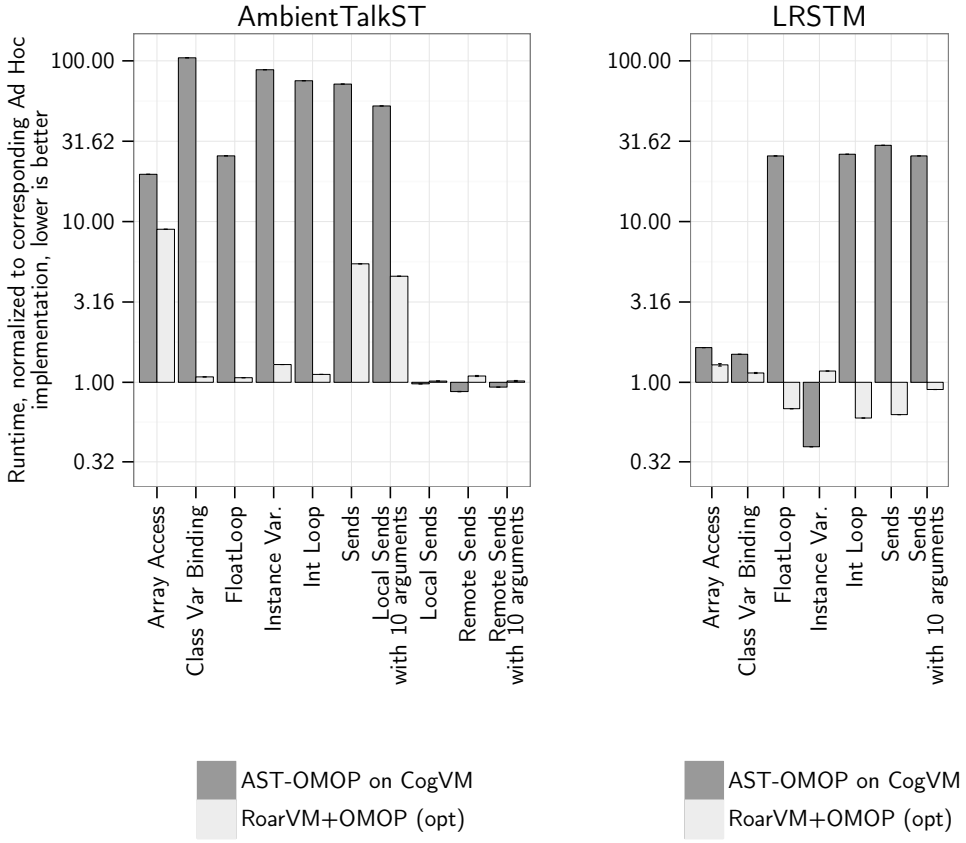

Figure 8.4.: Ad hoc vs. OMOP Microbenchmarks: Runtime normalized to Ad hoc implementations, logarithmic scale. The AST-OMOP implementation on top of the CogVM shows significant slowdowns especially for the AmbientTalkST microbenchmarks because more operations are reified. However, some benchmarks show speedups. The overhead for the VM-based implementation is generally lower and outperforms the ad hoc implementation of the STM on a number of benchmarks.

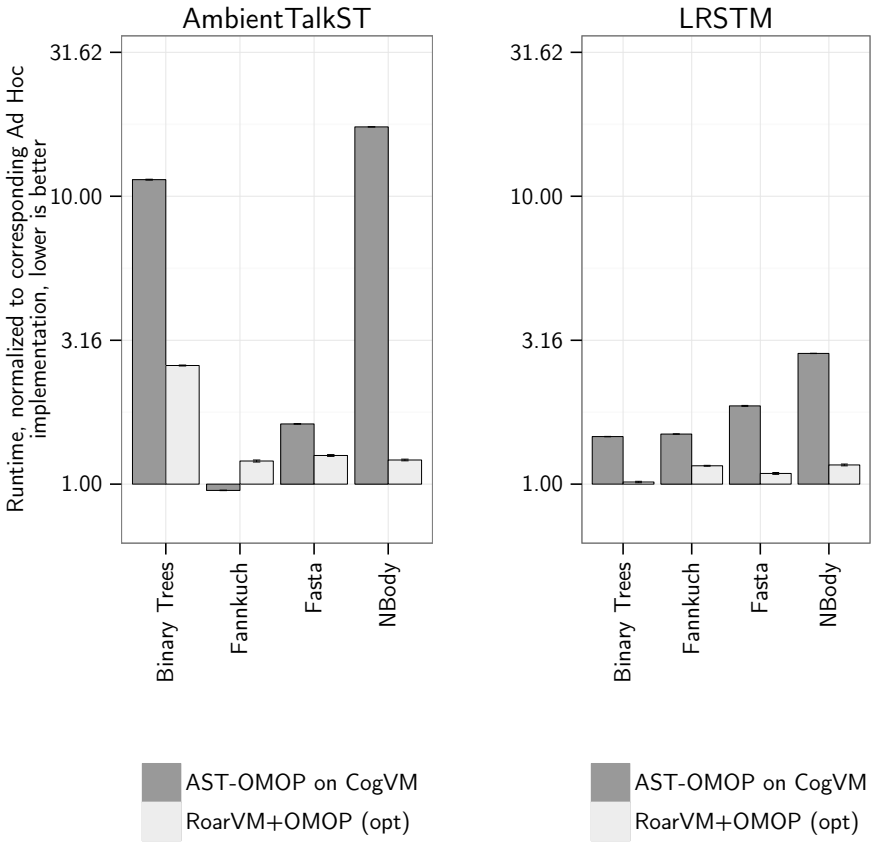

Figure 8.5.: Ad hoc vs. OMOP Kernel Benchmarks, logarithmic scale: For the kernel benchmarks, RoarVM+OMOP (opt) implementations show an average slowdown of 28% (min 2%, max 2.6x), which is close to the 17% general overhead for VM support. AST-OMOP on CogVM shows a significantly higher slowdown of 2.8x compared to ad hoc implementations. The RoarVM+OMOP (opt) implementation comes close to the ideal performance of the ad hoc implementations while providing a unifying abstraction. This shows that the OMOP can deliver on par performance when implemented efficiently.

ceptable in situations where the OMOP significantly simplifies the implementation of concurrent programming concepts or a VM-based solution is not yet available.

8.5. Assessment of Performance Characteristics

In order to understand the performance characteristics of the OMOP implementations, different aspects of the changed execution model need to be evaluated. Thus, the overhead of executing a program in the context of the OMOP is evaluated by measuring the cost of reifying operations, i. e., executing them reflectively. Furthermore, the experiments measure the inherent overhead that is introduced by the chosen implementation approach for the OMOP. Finally, the performance impact of the *customization constant* (cf. Sec. 8.5.3) for the RoarVM+OMOP is measured to determine the cost and benefit of this optimization.

8.5.1. OMOP Enforcement Overhead

Context and Rationale In the worst case customizing language behavior by using a MOP has the overhead of performing operations reflectively instead of directly. The OMOP in combination with Smalltalk represents this worst case, because the available metaprogramming facilities require that intercession handlers use reflective operations, which have an inherent overhead.

The evaluation needs to determine in which order of magnitude the overhead for these reflective operation lies to assess the practicality. Knowledge of this overhead can guide the implementation of concurrent programming concepts on top of the OMOP and helps explaining their performance.

Setup To measure the enforcement overhead, the kernel benchmarks are executed unchanged with enforcement enabled. The code executes within the standard `Domain` (cf. Fig. 5.1). Thus, the intercession handler only implements standard semantics of the operation they represent. For example, `#requestExecOf:on:with:lkup:` merely invokes the requested method reflectively, and `#readField:of:` as well as `#write:toField:of:` perform the corresponding read or write on the object field.

The overhead that is measured by this benchmark is the minimal overhead for a fully customized domain. Domains that customize only subsets of the intercession handler might have lower overhead, especially if the OMOP implementation checks the customization constant (cf. Sec. 7.2.1).

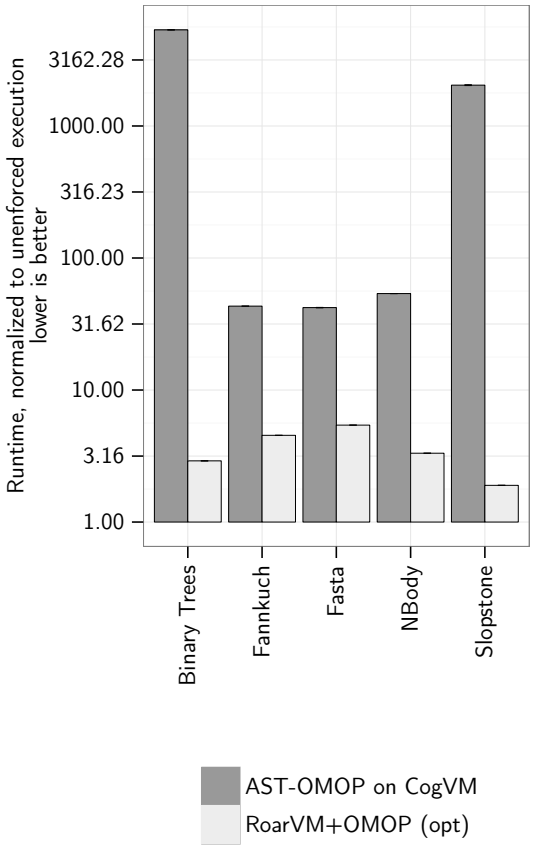

Figure 8.6.: Enforcement Overhead, logarithmic scale: Comparing unenforced execution of kernel benchmarks with enforced execution on RoarVM+OMOP (opt) and CogVM with AST-OMOP. The reified operations implement the standard semantics and on the RoarVM+OMOP (opt) lead to an average overhead of 3.4x (min 1.9x, max 5.4x). On the CogVM with the AST-OMOP, this overhead is on average 254.4x (min 42.1x, max 5346.0x).

Results Fig. 8.6 depicts the measured results. The performance overhead for enforced execution on CogVM with the AST-OMOP is very high. It reaches a slowdown of up to several thousand times (max 5346.0x). The minimal measured overhead on the given benchmarks is 42.1x while the average overhead is about 254.4x. For the RoarVM+OMOP (opt) the numbers are significantly better. The average overhead is 3.4x (min 1.9x, max 5.4x).

The high overhead on the CogVM can be explained by the differences in implementations. On the one hand, the AST transformation adds a significant number of bytecodes to each method that need to be executed, and on the other hand, the reflective operations preclude some of the optimizations of the CogVM. For instance the basic benefit of the JIT compiler to compiling bytecodes for field access to inline machine code cannot be realized, since these operations have to go through the domain object's intercession handler. Other optimizations, such as polymorphic inline caches [Hölzle et al., 1991] cannot yield any benefit either, because method invocations are performed reflectively inside the intercession handler, where polymorphic inline caches are not applied.

In the RoarVM+OMOP the overhead is comparatively lower, because it only performs a number of additional checks and then executes the actual reflective code (cf. Sec. 7.2.1). Thus, initial performance loss is significantly lower.

Speculating on how the CogVM could benefit from similar adaptations, there are indications that the JIT compiler could be adapted to deliver improved performance for code that uses the OMOP. However, because of the restricted generalizability of these results, it is not clear whether the CogVM could reach efficiency to the same degree as the RoarVM+OMOP implementation. However, this question is out of the scope of this dissertation and will be part of future work (cf. Sec. 9.5.2).

8.5.2. Inherent Overhead

Context and Rationale Sec. 7.2.1 outlined the implementation strategy for the RoarVM+OMOP. One performance relevant aspect of it is the modification of bytecodes and primitives to perform additional checks. If the check finds that execution is performed in enforced mode, the VM triggers the intercession handlers of the OMOP. However, since these checks are on the performance critical execution path, they prompted the question of how much they impact overall performance. This section assesses the inherent overhead on unenforced execution for both OMOP implementation strategies.

AST-OMOP The AST-OMOP has only an overhead in terms of memory usage but does not have a direct impact on execution performance. The memory overhead comes from the need of keeping methods for both execution modes, the enforced as well as the unenforced, in the image (cf. Sec. 7.1.1). Furthermore, since in most classes the AST-OMOP represents the owner of an object as an extra object field, there is also a *per-object* memory overhead. The per-

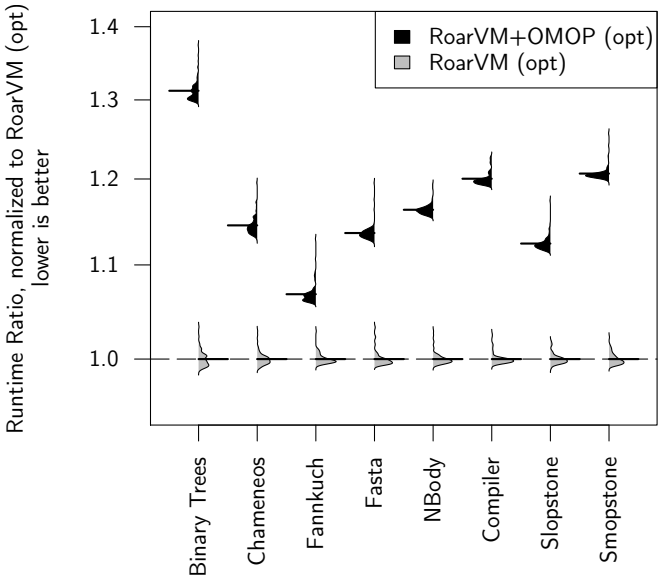

Figure 8.7.: Inherent Overhead: Comparing RoarVM (opt) with RoarVM+OMOP (opt) to assess the inherent overhead of OMOP support on the execution performance. The benchmarks do not utilize the OMOP, and thus, they expose that changes to the interpreter cause an average overhead of 17% on interpretation.

formance impact of the memory overhead has not been measured, but it is likely that the overhead significantly depends on average object size, and has an impact on GC times.

RoarVM+OMOP For the RoarVM implementation of the OMOP, the situation is different. While it also suffers from memory overhead per object for the additional object header that represents the owner (cf. Sec. 7.2.1), it also introduces changes that have a direct effect on the execution performance. The code path for unenforced execution through the interpreter now contains additional checks, which have an inherent impact on overall performance, also for unenforced execution.

Fig. 8.7[23] shows that the overhead ranges from minimal 7% up to 31%. The average runtime overhead of the OMOP support is about 17%. The overhead is explained by the additional code on the fast path of the code. All bytecodes that perform operations relevant for the OMOP apply an additional test to the interpreter object which checks whether execution is performed with enforce-

[23]The beanplot library of R forced a linear scale for this graph.

ment of the OMOP enabled. Only if that is the case, are additional checks and a potential invocation of the corresponding intercession handler performed. Thus, for the given benchmarks, the overhead consists of the additional check and the consequence of the additional code, which may cause cache misses.

8.5.3. Customization Constant Assessment

Context and Rationale As discussed in Sec. 7.2.1, the RoarVM+OMOP uses an optimization to avoid invoking a domain's intercession handler when it has not been customized. Originally, the optimization was added to ease debugging and reduce noise introduced by executing standard intercession handlers, which only implement standard language semantics. However, this optimization is not without drawbacks because it introduces a number of memory accesses and tests for all bytecodes that require modifications for the OMOP. The code that runs unenforced most likely incurs only a minor performance penalty. However, the effect on enforced execution is likely to be more pronounced. Specifically, code that runs in the context of domains that customize a majority of the intercession handlers is likely to experience overhead caused by additional checks. On the other hand, this optimization should yield performance benefits for code that runs in the context of domains that customize only a limited subset of the intercession handlers.

In order to assess the performance impact, a variant of RoarVM+OMOP (opt) is used that performs no additional tests but invokes the intercession handlers unconditionally. This variant with full intercession activation is referred to as *RoarVM+OMOP (full)*.

Overhead during Unenforced Execution Fig. 8.8[24] depicts the results measured for the kernel benchmarks for unenforced execution. Results vary from small performance benefits of ca. 3% to slowdowns of up to 4%. The average slowdown is 1%. Changes in performance are solely caused by the existence of additional code in the bytecode's implementation. Since the experiment used unenforced execution, this additional code was not executed. In conclusion, all measured effects are either caused by increased code size, triggering certain compiler heuristics, or other similar effects. Since the results are mixed and on average only show a difference of 1%, the overall effect on unenforced execution is negligible.

[24] The beanplot library of R forced a linear scale for this graph.

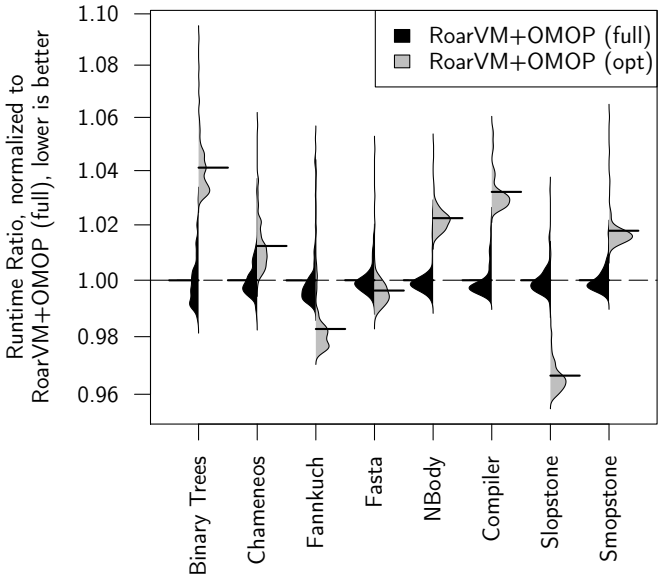

Figure 8.8.: Customization Constant, Unenforced Execution: Comparing RoarVM+OMOP (opt) with RoarVM+OMOP (full) to assess the inherent overhead of customization constant changes. The benchmarks do not utilize the OMOP, and thus, they expose that this check leads to an average runtime overhead of 1%. Since the impact is varying for the benchmarks and in some cases even reduces the runtime, the overall conclusion is that changes that introduce the check of the customization constant have a negligible impact on unenforced execution.

Overhead during Enforced Execution Fig. 8.9[25] shows the results for the enforced execution of the kernel benchmarks. Here the overhead comes from the additional check, which only confirms that all intercession handlers have to be used. The slowdown ranges from 2% to 4%, and average slowdown is 3%. Thus, there is a measurable, but minor overhead.

Speedup for AmbientTalkST and LRSTM Finally, the performance gains of using the customization constant for the AmbientTalkST and LRSTM benchmarks are measured. AmbientTalkST customizes all intercession handlers and should therefore show no gain or a minimal overhead for using the customization constant. LRSTM on the other hand only customizes state access while message sends are not customized. Consequently, it is expected to benefit significantly from avoiding overhead on every message send.

[25]The beanplot library of R forced a linear scale for this graph.

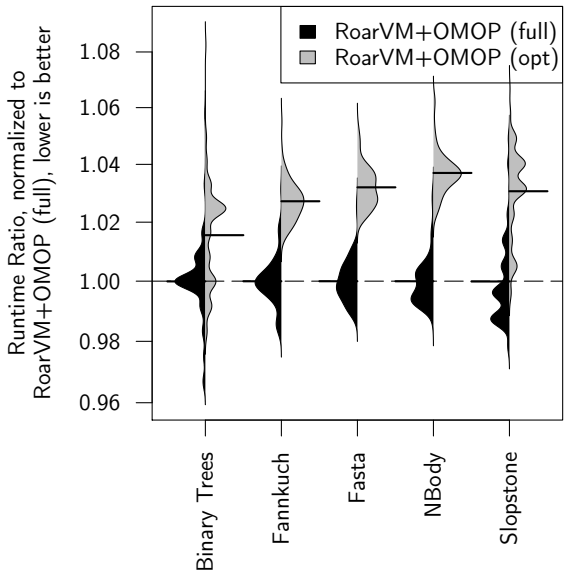

Figure 8.9.: Customization Constant, Enforced Execution: Comparing RoarVM+OMOP (opt) with RoarVM+OMOP (full) to assess the inherent overhead of checking the customization constant during fully enforced execution. The benchmarks utilize the OMOP but all intercession handlers implement standard semantics. The results show that the additional check leads to an average runtime overhead of 3%.

Fig. 8.10 depicts the results in the form of beanplots. The microbenchmarks show a significant benefit of avoiding reification for uncustomized intercession handlers. Especially the microbenchmarks for message sends for LRSTM show the benefit. Since the LRSTM system does not require message sends to be customized, they are between 2x and 2.5x faster, depending on the number of arguments.

The kernel benchmarks on the other hand show more realistically what the benefit of this optimization could be for application code. The average measured gain is 5%. For the Fannkuch benchmark on AmbientTalkST, the overhead of the additional check is higher and performance is not improved, because all intercession handlers have to be triggered. Thus, this benchmark shows a slowdown of 2%. The average 5% gain for the given benchmarks and concurrency models is rather modest. For other concurrency models, such as Clojure actors, higher benefits can be expected, because for instance the reading of object fields does not need to be reified.

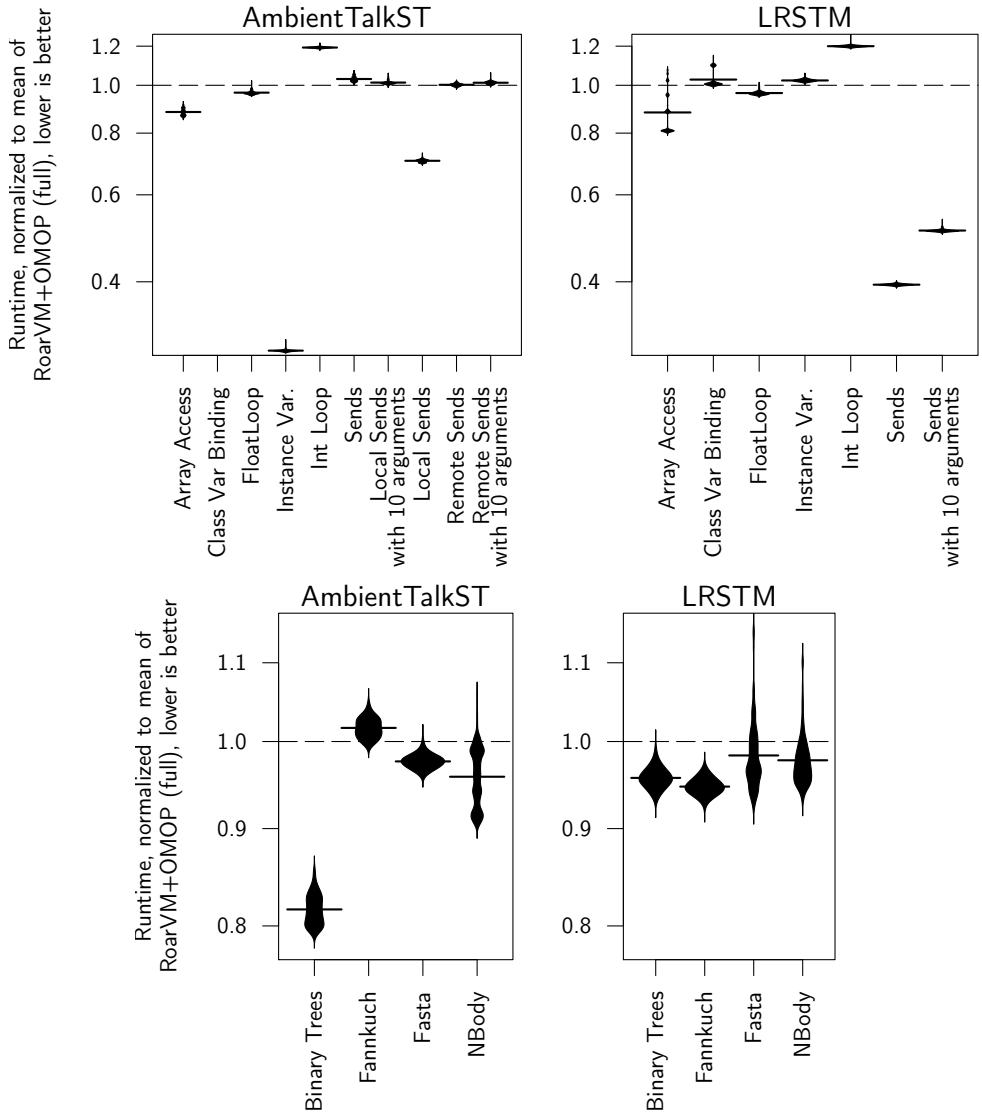

Figure 8.10.: Customization Constant, AmbientTalkST and LRSTM, logarithmic scale: Comparing RoarVM+OMOP (opt) with RoarVM+OMOP (full) on AmbientTalkST and LRSTM benchmarks. The microbenchmarks show the reduced runtime when checking the customization constant avoided reifying an uncustomized operation. However, they also show the overhead of the added check of the constant. The kernel benchmarks show an overall benefit from this optimization of about 5%. The AmbientTalkST Fannkuch benchmark ends up 2% slower because it does not benefit from the optimization but is exposed to the overhead of the additional check.

To conclude, direct performance gains depend to a large extent on the application and the concurrency model implemented on top of the OMOP. Thus, the modest overall performance gain might not justify the inherent overhead for other applications. However, the original motivation was to improve the debugging experience during the VM implementation by avoiding unnecessary reification. For this purpose, this optimization still provides the desired benefits so that debugging can focus on the relevant reified operations.

8.6. Absolute Performance

The final question for this performance evaluation is about which of the implementation strategies yields better performance in practice. Sec. 8.4 showed that direct VM-support brings the OMOP-based implementation of LRSTM on par with its ad hoc implementation on top of the RoarVM. However, Sec. 8.3 showed that the CogVM is about 11.0x faster than the RoarVM (opt), but the AST-transformation-based implementation (AST-OMOP) brings a performance penalty of 254.4x for enforced execution, which translates to a slowdown of 2.8x compared to the ad hoc implementations. This experiment compares the absolute performance of the benchmarks running on top of the CogVM and the AST-OMOP implementation with their performance on top of the RoarVM+OMOP (opt).

Fig. 8.11 shows the results of the AmbientTalkST and LRSTM being executed with identical parameters on CogVM with AST-OMOP and on the RoarVM+OMOP (opt). The benchmark results have been normalized to the mean of the corresponding result for the RoarVM+OMOP (opt). To improve the plot's readability, only the results for the CogVM are depicted.

They show that six of the microbenchmarks and one of the kernel benchmarks achieve higher absolute performance on the RoarVM+OMOP (opt). However, the majority maintains its higher absolute performance on top of the CogVM. For the kernel benchmarks, the benchmarks require on average a 5.7x higher runtime on the RoarVM+OMOP (opt). The AmbientTalkST NBody benchmark is about 21% slower, while the AmbientTalkST Fannkuch benchmark is about 14.3x faster.

To conclude, the AST-transformation-based implementation is currently the faster implementation because it can benefit from the overall higher performance of the CogVM. The results for RoarVM+OMOP (opt) are not optimal but an integration into the CogVM or other VMs with JIT compilation might result in better overall performance. This assumption is further supported by

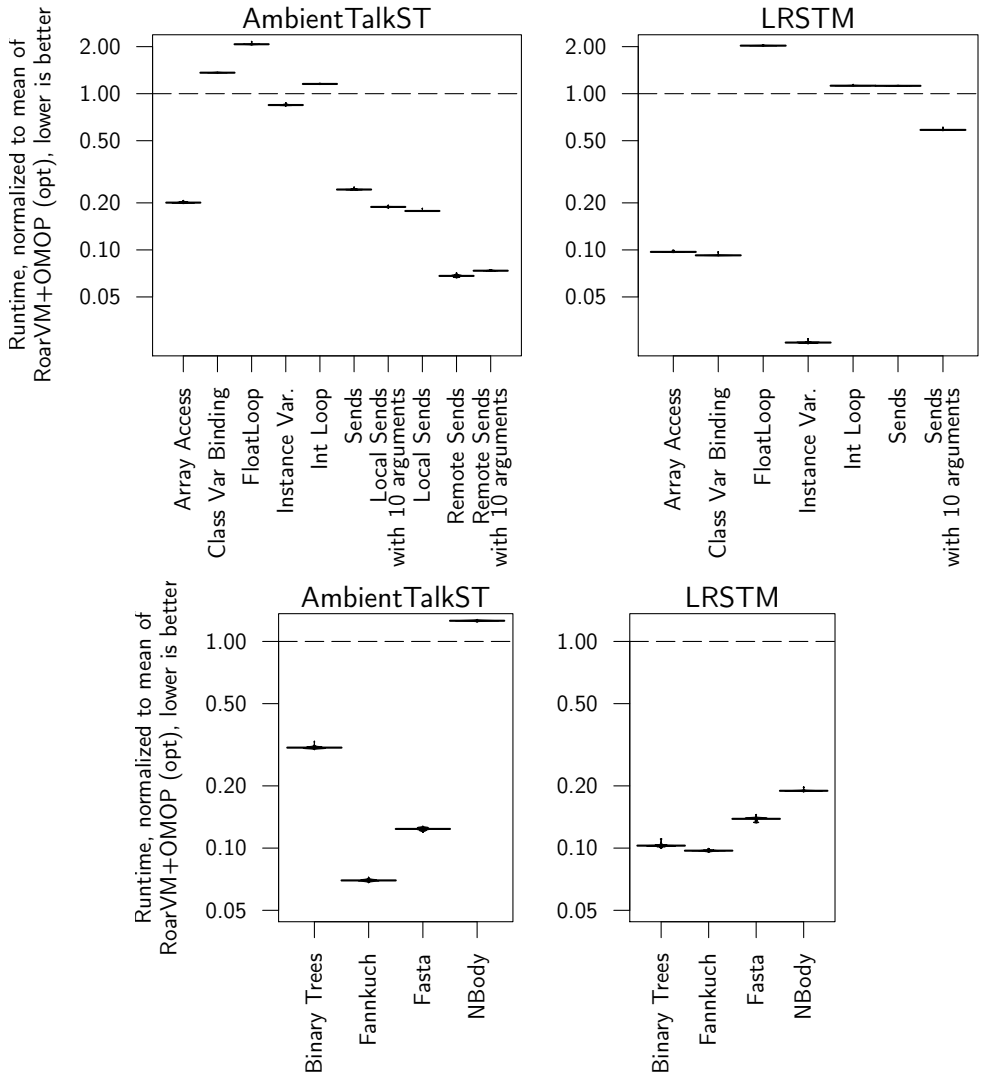

Figure 8.11.: Absolute Performance, CogVM and AST-OMOP vs. RoarVM+OMOP (opt), logarithmic scale: Although, the CogVM is 11.0x faster than the RoarVM (opt), the microbenchmarks indicate a significantly smaller difference. While most of them remain faster, the Int and Float Loop benchmarks show significant slowdowns. They are up to 2x slower on the CogVM with AST-OMOP than on the RoarVM+OMOP (opt). The kernel benchmarks maintain their higher absolute performance on CogVM with AST-OMOP and execute 5.7x faster than on the RoarVM+OMOP (opt). The AmbientTalkST NBody benchmarks shows a minor slowdown of 21%. The AmbientTalkST Fannkuch benchmark on the other hand is about 14.3x faster.

the observation that some of the microbenchmarks on the RoarVM+OMOP (opt) have higher absolute performance, which enforces the importance of VM level support.

8.7. Discussion and Threats to Validity

Threats to Validity Sec. 8.2 outlines approach to reduce structural bias as reported by Georges et al. [2007]. Thus, the experiments are set up to avoid sources of measurement errors and nondeterminism. However, today's computer systems have such a high complexity that measurement bias is a severe problem that cannot completely be excluded [Hennessy and Patterson, 2007; Mytkowicz et al., 2009]. However, while the chosen setup does not account for measurement variations caused by changes in order and size of the binaries, the used benchmark tools ensure that the environment, in which the benchmarks are executed, stays as constant as possible. This is achieved for instance by reducing influence from the operating system by increasing the process priority to the maximum. Thus, it is very probably that the measured performance differences between the VMs can be attributed to a significant degree to the changes in the VMs.

In order to ensure that the measured results are statistically significant, every benchmark executes 100 times. While no formal tests for statistic significance were applied, visual inspection of the results in form of beanplots and bar charts with error indications are sufficient to determine the significance of the results.

Furthermore, averages use the geometric mean to avoid bias in the reported results [Fleming and Wallace, 1986].

Thus, the confidence in the correctness of the reported results in this chapter is high. However, the chosen microbenchmarks and kernel benchmarks preclude generalization of the results to actual applications [Blackburn et al., 2006; Vitek and Kalibera, 2011]. The common point of view is that only an application itself can be used to obtain a reliable performance assessment. Thus, the results are merely an indication. They points out tendencies in the performance effects that result form the compared implementation approaches. However, the concrete numbers are unlikely to hold for any given concrete application.

Finally, the choice of using the RoarVM as experimentation platform for the implementation of VM support also restricts the generalizability of the benefits of this approach. Since the general overhead of interpretation could

hide inherent overhead of the OMOP implementation, the results cannot be generalize beyond interpreters with similar execution mechanisms. Thus, the indicated performance benefits will most likely transfer to other bytecode interpreters, but cannot be used to predict the performance of VMs that use JIT compilation such as HotSpot,[26] or the CogVM.

Discussion The thesis statement, i. e., whether the OMOP *"lends itself to an efficient implementation"*, needs to be evaluated with these restrictions in mind. The main criterion for assessing this part of the thesis statement is the question of whether OMOP-based implementations of concurrent programming concepts can have on par performance with ad hoc implementations that use other implementation strategies.

Thus, the main question is whether the results of Sec. 8.4 provide sufficiently reliable indications to support the claim that OMOP-based implementations can be on par with ad hoc implementations.

Measured Overhead AmbientTalkST and LRSTM are on average 28% slower than their ad hoc implementations. However, performance ranges from an overhead as low as 2% up to 2.6x on the kernel benchmarks. If only the results for LRSTM are considered, the average slowdown is 11%. Thus, the OMOP-based implementation is very close to the performance of the ad hoc implementation.

Overhead in Perspective To put these 11% into perspective, Mytkowicz et al. [2009] report on their experience with measuring the benefit of compiler optimizations and they find that simple changes such as link order or changes in environment size of their shell can cause variations that are as high as 30%. Furthermore, modern operating systems typically execute multiple applications at the same time, which can lead for instance to memory pressure and paging that has a significant negative effect on the application's performance [Silberschatz et al., 2002]. Thus, these 11% or even 28% slowdown are undesirable but might not have a significant consequence in practice, especially when the overall comparatively low performance of the interpreter is considered.

Potential Optimizations to Reduce Overhead Sec. 8.5.2 measured for the RoarVM+OMOP (opt) an inherent overhead of 17% (min 7%, max 31%). As-

[26] http://openjdk.java.net/groups/hotspot/

suming that the implementation of RoarVM+OMOP could be optimized to reduce this inherent overhead, it seems possible to achieve on par or better performance.

Sec. 9.5.7 discusses the possibility to use an alternative bytecode set for enforced execution. The interpreter would execute different bytecode routines based on whether it is executing in enforced or unenforced mode. Such an optimization has the potential to avoid the overhead of the additional checks at the cost of duplicating code in the interpreter. Assuming that the measured 17% overhead is solely attributed to the additional checks, such an optimization could improve performance in the best case to the point that the average slowdown would come down from about 28% to 11%. However, since experiments are not yet performed, the performance impact of these changes remains speculation.

Sec. 9.5.3 discusses another possible optimization. In order to avoid the need to check for accesses to object fields, the operating systems's memory protection support could be used [Silberschatz et al., 2002]. Thus, instead of performing active checks, the operating system could utilize the processor's hardware support for memory protection and trigger a software interrupt that can be handled if an actual memory access happens. However, while such an optimization avoids active checks, it is only applicable to memory accesses.

Performance Sweet Spot Overall, it remains to be noted that abstractions and optimizations typically have a certain performance *sweet spot*. To give just a single example, Self used *maps*, i. e., hidden classes to optimize performance, because in practice, the prototype-based object system was used preferably in a way that facilitated this particular optimization [Chambers et al., 1989]. However, when an application uses the object system in a different way, it will not benefit from this optimization. The situation is similar with other optimizations. Typically, certain usage patterns are preferred to gain optimal performance.

The performance evaluation of this dissertation shows that LRSTM uses the OMOP in a way that it has merely an overhead of 11%, while the overhead for AmbientTalkST is higher. The main difference between the two implementations is that LRSTM customizes the state access policies, which is an indication that concurrent programming concepts that rely on such policies could exhibit similar performance characteristics.

Conclusion To conclude, the average measured overhead of 28% for the kernel benchmarks for AmbientTalkST and LRSTM are undesirable. However, the 11% overhead measured for the LRSTM is a positive indication that the OMOP allows efficient implementation. Arguably, with merely 11% overhead, the OMOP-based implementation of LRSTM is on par with the ad hoc implementation. Furthermore, these results are based on a first RoarVM+OMOP implementation that does not include any extensive optimizations.

However, these 11% overhead are no prediction for actual application performance. Thus, only a performance evaluation with a concrete application can determine whether this slowdown is acceptable.

8.8. Conclusions

This chapter compared the performance of the two proposed implementation approaches for the OMOP. Furthermore, it analyzed how the OMOP-based implementations of AmbientTalkST and LRSTM compare to the corresponding ad hoc implementations.

CogVM can be 11.0x faster than RoarVM (opt). The experiments started by measuring the relative performance between CogVM and the RoarVM variants. Measurements indicate that the CogVM is about 11.0x faster (min 7.1x, max 14.4x) than the RoarVM (opt) on the kernel benchmarks. Therefore, it is used for assessing the performance of the AST-OMOP on top of a VM with JIT compiler, which gives it the best possible performance.

Measurements indicate that OMOP-based implementations can have on par performance with ad hoc implementations. The comparison of the ad hoc implementations with the OMOP-based implementations of AmbientTalkST and LRSTM used the CogVM with AST-OMOP as well as the RoarVM+OMOP (opt). While kernel benchmarks on top of the AST-OMOP show an average slowdown of 2.8x, the results for the RoarVM+OMOP (opt) are significantly better. VM support for the OMOP results in an average overhead of 28% (max 2.6x) over the ad hoc implementation. However, some benchmarks show overhead as low as 2%, being on par with the ad hoc implementations. Furthermore, the current implementation has an inherent performance overhead of 17% (min 7%, max 31%), which is part of these performance numbers.

Important to note is that LRSTM shows only an average slowdown of 11%, which is lower than the inherent overhead introduced by RoarVM+OMOP (opt). This seems to indicate that there is a performance sweet spot for approaches that use the OMOP in a way similar to LRSTM. Thus, there are confirmations for the claim that OMOP-based implementations can have on par performance with ad hoc implementations, even though it might be restricted to certain types of approaches. An important prerequisite for good overall performance however, is that the inherent overhead of the OMOP implementation can be reduced.

The overhead of the AST-OMOP on the CogVM is two magnitudes higher than the overhead on the RoarVM+OMOP. The comparison of the enforcement overhead of the two OMOP implementations shows that full enforcement on the CogVM with AST-OMOP causes an average overhead of 254.4x (min 42.1x, max 5346.0x) on the kernel benchmarks, while the overhead on the RoarVM+OMOP (opt) is significantly lower with 3.4x (min 1.9x, max 5.4x). However different factors, such as the initially lower performance of the RoarVM, have a significant impact on these results. Furthermore, the CogVM uses a comparably simple JIT compiler. Thus, more advanced VMs such as HotSpot that include techniques such as INVOKEDYNAMIC and related infrastructure [Thalinger and Rose, 2010] might yield significantly different results.

However, the extreme difference in relative performance indicates that VM support in a JIT compiled system such as the CogVM can yield significant performance improvements, bringing the resulting performance for OMOP-based implementations of LRSTM and AmbientTalkST closer to their ad hoc implementations.

The RoarVM+OMOP implementation comes with performance tradeoffs. The VM support built into the RoarVM+OMOP (opt) has a number of drawbacks. First, it has an inherent overhead of about 17% (min 7%, max 31%) on the kernel benchmarks because it requires an additional test on the fast-path of most bytecodes, which leads to a slowdown, also for the unenforced execution. The use of a customization constant to avoid unnecessary invocation of intercession handlers also comes with a number of tradeoffs. In cases where all intercession handlers need to be triggered, the additional check introduces 3% overhead (min 2%, max 4%) on the kernel benchmarks. On the mixed AmbientTalkST and LRSTM benchmarks it shows however that it can

improve performance. The average measured speedup is 5% for the kernel benchmarks.

The absolute performance of the AST-OMOP on CogVM is currently higher than the performance of the RoarVM+OMOP. Finally, assessing the absolute performance of the implementations, the higher performance of the CogVM leads to an absolute performance for the AST-OMOP that is about 5.7x better on the kernel benchmarks than the current RoarVM+OMOP (opt) implementation. Thus, when experimenting with the versatility of the proposed abstraction, the CogVM in combination with the AST-OMOP is a good option. For on par performance with ad hoc implementations, VM support remains indispensable however.

Overall Conclusions The goal of this chapter is to evaluate whether the OMOP *"lends itself to an efficient implementation"*. The main criterion for assessing this part of the thesis statement is the question of whether OMOP-based implementations of concurrent programming concepts can have on par performance with ad hoc implementations that use other implementation strategies.

As argued in this section, results indicate that OMOP-based implementations can indeed be on par with ad hoc implementations. The measured overhead on the interpreter-based VM is on average 28%. While the overhead is as high as a 2.6x slowdown, it can also be as low as 2%. Specifically for the LRSTM implementation, the measured overhead is merely 11%, which points to a performance sweet spot of the RoarVM+OMOP implementation.

Thus, there are concurrent programming abstractions, which can be implemented efficiently on top of an OMOP for interpreters. Currently, the experiments indicate that the performance sweet spot includes LRSTM-like usages of the OMOP that customize state access policies.

However, there are a number of potential optimizations to that could improve the current performance (cf. Sec. 5.5, Sec. 9.5.2, Sec. 9.5.3, Sec. 9.5.7). In case these optimizations prove to be successful, a wide range of concurrent programming concepts can benefit from them on top of the OMOP. The benefit of a unifying substrate in such a situation becomes relevant, because optimizations facilitate not only to a single programming concept.

To conclude, there are strong indications that the OMOP lends itself to an efficient implementation for a certain set of concurrent programming abstractions, for which it can enable performance that is on par with ad hoc implementations.

9

CONCLUSION AND FUTURE WORK

This dissertation presents a novel metaobject protocol that provides perviously missing abstractions to language and library implementers for building a wide range of concurrent programming concepts on top of multi-language virtual machines (VMs) such as Java Virtual Machine (JVM) and Common Language Infrastructure (CLI).

This chapter reexamines the problem statement, the research goals, and the thesis statement in the context of the proposed solution, i. e., the *ownership-based metaobject protocol* (OMOP). It restates the contributions, discusses the current limitations of the OMOP, and highlights avenues for future research.

9.1. Problem and Thesis Statement Revisited

This dissertation set out to address the problem of today's virtual machines not reconciling the trend for increased hardware parallelism, which is the result of the multicore revolution, with the trend of using VMs as general purpose platforms with a wide variety of languages. Today, language and library implementers struggle to build concurrent programming concepts on top of multi-language VMs, because these VMs do not provide the necessary abstractions. Hence, the objective of this dissertation is to find a solution for the *insufficient support for concurrent and parallel programming in VMs*.

Since today's VMs support only a limited range of concurrent and parallel programming concepts, first needed to be identified the concepts that would benefit from VM support for improved performance or enforcement of semantics. Initially the problem was that the *set of abstractions required from a VM was unknown*.

With these problems in mind, the following thesis statement was formulated:

> *There exists a relevant and significant subset of concurrent and parallel programming concepts that can be realized on top of a unifying substrate. This substrate enables the flexible definition of language semantics, which build on the identified set of concepts, and this substrate lends itself to an efficient implementation.*

Three concrete research goals were derived from the thesis statement:

Identify a Set of Requirements The first goal of this research was to understand how concurrent and parallel programming are supported in today's VMs and how the underlying programming concepts relate to each other. Chapter 3 approached this goal with a survey and concluded that concepts for parallel programming benefit mainly from support for optimization in a VM, and that concepts for concurrent programming require support for their semantics. Since these two concerns are largely orthogonal, this dissertation chose to focus on concepts for concurrent programming. Chapter 3 further discussed a number of problems language implementers face today and derived a set of requirements for their support in multi-language VMs.

Define a Unifying Substrate The second research goal was to identify a unifying substrate to support concurrent programming concepts. Based on

the identified requirements, this dissertation proposes an ownership-based metaobject protocol (OMOP) in Chapter 5.

Demonstrate Applicability The third and final research goal was to demonstrate the applicability of the proposed OMOP. On the one hand, Chapter 6 needed to evaluate the OMOP's capabilities as a unifying substrate. Thus, it needed to demonstrate that the OMOP facilitates the implementation of a wide range of concepts for concurrent programming. On the other hand, in Chapter 8 it was the goal to show that the OMOP gives rise to an efficient implementation. Thus, the evaluation needed to show that by using the OMOP it is possible to implement concepts for concurrent programming in a way that their performance is on par with conventional implementations.

Two prototypes of the OMOP were implemented to support the thesis statement and to achieve the research goals. The first implementation uses AST transformation on top of an unchanged VM. The second implementation adapts the bytecode semantics of an existing VM. Based on these implementations, Chapter 6 evaluated the OMOP and Chapter 8 demonstrated its applicability.

9.2. Contributions

This section recapitulates the main chapters of this dissertation so as to identify the individual contributions of each of them.

- Chapter 2 discusses the widespread use of VMs as general purpose platforms and identifies the need for a unifying abstractions to avoid complex feature interaction in VMs that hinder extensibility and maintainability.

  Furthermore, it revisits the reasons for the multicore revolution and concludes that software developers need to pursue parallelism to achieve the performance required for their applications. The vision of this dissertation is that applications are built using the appropriate concurrent and parallel programming techniques for, e. g., user interface and data processing.

  Moreover, the chapter defines the notions of *concurrent programming* and *parallel programming* to enable a categorization of the corresponding programming concepts based on their intent.

- Chapter 3 surveys the state of the art in VMs and determines the requirements for a unifying substrate to support concurrent programming. The survey shows that today's VMs relegate support for parallel programming to libraries, whereas they do provide support for concurrent programming at the VM level. The survey of the wide field of concurrent and parallel programming resulted in an understanding that parallel programming requires VM support mainly for optimized performance. Concurrent programming on the other hand requires VM support to guarantee its semantics. Sec. 3.2.4.1 concludes with a general set of requirements. Both sets of programming concepts require orthogonal mechanisms from a VM, and this dissertation chooses to focus on concurrent programming.

  With this focus in mind, common problems for the implementation of concurrent programming concepts on top of today's VMs are discussed. Based on the surveys and the identified problems, Chapter 3 concludes that a multi-language VM needs to support notions of *managed state*, *managed execution*, *ownership*, and *controlled enforcement* to provide a unifying abstraction for concurrent programming.

- Chapter 4 motivates the choice of Smalltalk and its different implementations as the foundation for the experiments. It introduces Smalltalk, SOM (Simple Object Machine), Squeak, Pharo, and the RoarVM.

- Chapter 5 introduces the ownership-based metaobject protocol (OMOP) as a unifying substrate for the support of concurrent programming in a multi-language VM. First, it motivates the choice of a metaobject protocol (MOP) by giving an overview of open implementations and MOPs and their capabilities to enable incremental modifications of language behavior. Second, it presents the design of the OMOP, details its based on examples, and describes the semantics based on a bytecode interpreter. Finally to emphasize its novelty, it discusses the OMOP in the context of related work.

- Chapter 6 evaluates the OMOP's applicability to the implementation of concurrent programming and it evaluates how the OMOP satisfies the stated requirements. The case studies demonstrate that Clojure agents, software transactional memory, and event-loop actors can directly be mapped onto the OMOP. Furthermore, the evaluation shows that the OMOP facilitates the implementation of concurrent programming concepts that require VM support for their semantics. Consequently, the

OMOP addresses common implementation challenges language implementers face today by providing abstractions for instance to customize state access and execution policies. Furthermore, the case studies required less code when implemented on top of the OMOP then when implemented with ad hoc approaches. Chapter 6 concludes with a discussion of the current limitations of the OMOP's design.

- Chapter 7 presents the two proposed implementation strategies for the OMOP. First, it details an implementation based on AST transformation on top of standard VMs. Second, it describes a VM-based implementation that adapts bytecode semantics for the OMOP. In addition to outlining the implementation, it proposes an optimization that avoids runtime overhead when parts of the OMOP remain uncustomized.

- Chapter 8 evaluates the performance of the OMOP implementations. The results that indicate a certain set of OMOP-based implementations can reach performance on par with ad hoc implementations. While the AST-transformation-based implementation has significant performance overhead, the VM-based implementation shows the potential of direct VM support. The evaluation further details the performance tradeoffs for instance by evaluating the impact of the proposed optimization.

9.3. Limitations

While the OMOP satisfies its requirements and the evaluation provides support for the thesis statement, its design has a number of limitations and gives rise to a number of research questions. This section briefly restates the limitations, which have been discussed in greater detail throughout the dissertation.

The limitations of the current implementation, as discussed in Sec. 7.2.2, are:

Restricted power of primitives. By reifying the execution of primitives their execution context is changed compared to when they are applied directly. Instead of being applied to the caller's context, they are applied in a dedicated context. Therefore, they cannot directly change the caller's context, and their power is limited. However, this restriction has no practical consequences: all primitives that were encountered in the RoarVM operate on the operand stack of the context only.

A solution to the problem could either be to change the primitives to be aware of the changed situation, or to ensure that the reflective invocation of primitives is performed with the correct context.

Conceptual limitations, as discussed in Sec. 6.5.2, are:

Deadlock freedom is a guarantee that cannot be given if a language or VM enables arbitrary and unconstrained use of blocking operations. However, restricting the use of blocking operations seems impractical in multi-language VMs where some languages might require blocking semantics.

However, the OMOP provides the ability to customize all primitives, and thus, enables a domain to manage calls to blocking operations and adapt them as necessary, which could be used to facilitate the guarantee of deadlock freedom.

Domain interaction requires explicit handling. A language or library implementer has to anticipate and adapt domain implementations to account for the semantics of other domains and ensure that interactions have reasonable semantics.

Currently, there is no mechanism that enables the specification of desired interactions in a general way, since the OMOP only takes the owner of an object into account for determining which intercession handler implements the desired semantics.

Object-based ownership notion has tradeoffs. The granularity of ownership per object might preclude relevant designs, especially when it comes to large arrays or objects that need to be shared. Together with the notion of having a single owner per object, the OMOP restricts the design space within which concurrent programming concepts can be expressed.

Limited support for guaranteeing scheduling policies. During the execution of primitives, complete control is yielded to the underlying platform by the VM, and thus, its ability to enforce scheduling policies is restricted. While intercession handlers for primitives enable customization, they do not provide a full solution.

To gain full control, the language implementer would need to be able to express policies which are communicated to the underlying platform as well. This is currently not concidered in the design of the OMOP.

The limitations of the performance evaluation, as discussed in Sec. 8.1.4, are:

Performance results are not generalizable to applications. The use of microbenchmarks and kernel benchmarks precludes a generalization of the performance results to applications. Thus, for fully generalizable results, a performance evaluation with application benchmarks would be required.

Performance results provide indications for interpreter-based VMs only. The low performance of the RoarVM precludes generalization beyond similar bytecode interpreters. Thus, the obtained results cannot be used to predict the performance for highly optimized VMs.

9.4. Overall Conclusions

This dissertation is the first to explore how a wide range of concurrent and parallel programming concepts can be supported by a multi-language VM. Until now, the research focus was often limited to one specific language or concept without considering the need for a unifying substrate. However, with the JVM and CLI being used as multi-language VMs, this is no longer sufficient.

Consequently, it surveys the field of VMs as well as the field of concurrent and parallel programming and concludes with a set of five general requirements for VM support. In order to host a wide range of different concepts efficiently and with correct semantics, a VM needs to provide a *flexible optimization infrastructure*, *flexible runtime monitoring facilities*, a *powerful VM interface*, *customizable semantics for execution and state access*, as well as *semantic enforcement against reflection*.

This dissertation focuses on concurrent programming concepts. They benefit foremost from guaranteeing their semantics in the presence of reflection, mutable state, and interaction with other languages and libraries. The main problem with today's VMs is that contemporary implementation approaches for concurrent programming concepts have to balance implementation simplicity, correctly implemented language semantics, and performance. The conclusion based on these observations is that VMs have to support the notions of MANAGED STATE, MANAGED EXECUTION, OWNERSHIP, and CONTROLLED ENFORCEMENT in order to facilitate the implementation of concurrent programming concepts.

The proposed solution is an ownership-based metaobject protocol. The OMOP offers intercession handlers to customize state access and method execution policies. Building on the notion of ownership, objects are grouped into concurrency *domains* for which the intercession handlers to customize the language's behavior can be specified. In addition to the OMOP itself, two implementation strategies are proposed. One is based on AST transformation and the another one on changes to the bytecode semantics of the RoarVM.

The evaluation of the OMOP shows that it directly facilitates the implementation of Clojure agents, event-loop actors, software transactional memory, active objects, and communicating sequential processes. Furthermore, it supports the concurrent programming concepts that require VM support to guarantee their semantics. The case studies demonstrate that the OMOP provides solutions to the common challenges language and library implementers are facing today. Thus, its abstractions facilitate the correct implementation of isolation, immutability, execution policies, and state access policies.

The performance is evaluated based on the two OMOP implementations. The measurements indicate that on par performance with ad hoc implementations can be reached when interpreters provide direct support. However, currently the results indicate that the performance sweet spot is restricted to concurrent programming concepts that feature custom state access policies. Furthermore, it remains open what the performance outlook is for VMs featuring optimizing just-in-time compilers. This question will be pursued together with others in future work, for instance to address limitations such as the required explicit handling of interactions between concurrency domains.

To conclude, the OMOP is the first metaobject protocol designed and evaluated with the goal to support a wide range of concurrent programming concepts in a multi-language VM. Furthermore, it is based on the analysis of the broader field of concurrent and parallel programming concepts, which has not been explored to this extent before. The OMOP provides a flexible mechanism to adapt language behavior with respect to concurrency issues and is the first promising unifying substrate that is meant to be integrated into multi-language VMs to solve common implementation challenges.

9.5. Future Work

This section an outlook on future research perspectives and point out open research questions that go beyond the discussed limitations of the OMOP.

9.5.1. Support for Parallel Programming

Sec. 3.2.4.1 identified requirements for the support of parallel programming in multi-language VMs. While this dissertation did not pursue the corresponding research questions, they are relevant for the future. The required mechanisms for adaptive optimization and monitoring are not yet widely supported in VMs and could be beneficial to parallel programming and beyond. Thus, future work will investigate how the optimization infrastructure, typically related to just-in-time compilation, can be exposed to language and library implementers. Related work in that direction is for instance the Graal project of Oracle for the JVM [Würthinger, 2011]. Furthermore, future work needs to investigate how runtime monitoring can be realized to enable flexible monitoring of execution and instrumentation of code by language and library implementers to gather the relevant information to perform adaptive optimizations for a wide range of parallel programming techniques.

There is strong potential for such techniques to facilitate language implementation on top of multi-language VMs to enable them to utilize the available optimization infrastructure to gain optimal performance.

9.5.2. Support for Just-in-Time Compilation

Future work needs to develop a strategy to support the OMOP in just-in-time (JIT) compiling VMs to enable its adoption in high-performance JVMs or CLI runtimes.

One challenge with the OMOP is the notion of ownership. One of the design goals was to enable the use of libraries in different domains. Thus, instead of using a metaclass-based approach where the meta relation is based on a predetermined relation with the metaclass, the OMOP uses a dynamic solution based on object ownership.

However, for highly efficient native code, this flexibility is a problem, because the code needs to account for changing owners of objects which could preclude operations such as inlining that are necessary to achieve the desired performance.

Nevertheless, it may be assumed that in common applications object graphs and computations will be less dynamic than they potentially could be. Thus, a JIT compiler could optimistically compile a method based on the currently known object and ownership configuration. Standard compiler optimizations could then be applied to inline the intercession handlers and remove the overhead of reflective operations to obtain optimal code. Instead of performing

the actual dispatch to the intercession handler, a simple guard could then protect the code path and ensure that execution either exits into the interpreter or performs the required invocation of the intercession handler in case the ownership condition does not hold. Such techniques are known in the context of optimistic optimizations and use deoptimization for cases where the optimistic assumptions do not hold or are not desired. Examples are deoptimization for debugging purposes [Hölzle et al., 1992], trace guards for tracing JIT compilers [Gal et al., 2006], handling of exceptional cases for instance for array bounds checks [Würthinger et al., 2009], or for techniques that only compile the fast path of code and rely on deoptimization for the less common slow path [Würthinger et al., 2012].

9.5.3. Relying on the CPU's Memory Management Unit

As the performance evaluation in Sec. 8.5 showed, it is costly to check actively so as to tell apart enforced from unenforced execution mode. While this overhead could be reduced and might to a large extent be eliminated by a JIT compiler, some concurrent programming concepts such as CSP and event-loop actors rely on the notion of ownership and require customization of memory access operations depending on whether the access was done from within or from outside an actor or process. While optimistic optimization might reduce the necessary overhead for such checks, a different potential solution to the problem exists.

Hoffman et al. [2011] use hardware memory protection support of processors to enable the isolation of program components in an otherwise shared memory model. This memory protection could be used in a similar way to delegate the necessary access checks to the hardware. The idea would be to protect memory from access by other entities, which would trigger a software interrupt that would then be used to invoke the intercession handler for reading or writing of fields. The entity, i. e., the actor itself, could freely operate on its own memory without interference, and without the need for active checks on every memory operation.

Such an approach would be beneficial to all concurrent programming concepts that utilize the notion of ownership to define different memory access policies for the owner and for other entities.

9.5.4. Representation of Ownership

The proposed implementations use an extra header word for every object to represent object ownership. While this direct approach avoids complex changes in the VM, it comes with a significant memory overhead for small objects. In the field of garbage collection, different solutions have been proposed to track similar metadata for various purposes. One possible approach that is common for GCs is to partition the heap [Jones et al., 2011, chap. 8] based on certain criteria to avoid the direct encoding of related properties. Future work needs to investigate these possibilities.

The ownership of an object could be encoded by it being located in a partition that belongs to a domain. Such an approach would avoid the space overhead for keeping track of ownership. While partitioning might add complexity to the memory subsystem of the VM, it might also open up opportunities for other optimizations. Depending on the use of the domains, and the concurrency concepts expressed with it, a GC could take advantage of additional partitioning. For instance, in an actor-like domain, additional partitioning could have benefits for generational collectors. In such a system, the nursery partition would be local to a single actor, which could have a positive impact on GC times. Furthermore, this representation would most likely be a prerequisite for using the memory management unit to avoid active state access checks (cf. Sec. 9.5.3).

9.5.5. Applying the OMOP to JVM or CLI

The experiments presented here are performed on top of Smalltalk. Major differences between the Smalltalk VMs used and JVM and CLI are the significantly larger feature sets of JVM and CLI. The main problems for applying the OMOP to either of these VMs are feature interactions between the OMOP and existing functionality.

Differences such as the typed instruction sets of those two VMs are not necessarily a problem. The JVM's INVOKEDYNAMIC infrastructure [Thalinger and Rose, 2010] and its dynamic proxies[1] demonstrate how metaobject protocols can be applied to such VMs. Thus, solutions for basic design questions of a MOP have already been solved for these VMs. Therefore, the OMOP mostly needs customizations to the VM specific parts. An integration with the memory model of these VMs might also require few adaptations when

[1] http://docs.oracle.com/javase/7/docs/api/java/lang/reflect/Proxy.html

all reflective operations are able to follow strictly the existing semantics of the memory model, it might integrate well with the OMOP.

A more complex challenge is the integration with the security mechanisms of the platforms (cf. Sec. 3.3.5). These and similarly involved features will require careful engineering to preserve the existing semantics and integrate them with the flexibility of an OMOP.

9.5.6. Formalization

A different research direction is opened up by the unifying characteristics of the OMOP. Using it as a target for a wide range of concurrent programming concepts enables their representation in one common framework.

Such a research effort could yield valuable understanding of the concrete design space of these concepts. On the one hand, this would open up opportunities to engineer libraries of these concepts that can then be used to build domain-specific abstractions. On the other hand, it would help understand the commonalities and variabilities of the concepts, yielding a common vocabulary. Currently, a major issue in the field of concurrent and parallel research is the lack of a sufficiently clear distinction between concepts and a missing common vocabulary, making it hard to relate research from different eras and research domains to each other. Building these concurrent programming concepts on top of the OMOP would yield a practically usable collection of artifacts and might result in the discovery of relevant points in the design space of concurrent programming concepts that have not yet been explored.

Furthermore, describing all concepts in a common framework could help in identifying fully declarative representations. Currently, the OMOP is a classic MOP and the customization of intercession handlers is prescriptive, while a declarative approach could yield a more descriptive representation. Descriptive representations could be valuable to synthesize interaction semantics, which is one of the current limitations. Furthermore, they could become the foundation for compiler optimizations to improve performance.

9.5.7. Additional Bytecode Set for Enforced Execution

Currently the inherent overhead of the RoarVM+OMOP implementation is significant enough to prompt the desire for optimization. One possible optimization would be a separate bytecode set for use during enforced execution. Thus, instead of checking for the enforcement mode during execution, the

bytecode dispatch could be adapted to separate the semantics of both execution modes.

Examples for such an approach can be found for instance in the CogVM[2] or LuaJIT2.[3] Dynamic patching of the interpreter's dispatch table as used by LuaJIT2 might remove the actual overhead during the bytecode dispatch completely and thus promises to eliminate the inherent overhead of the current approach.

9.6. Closing Statement

This dissertation proposes a novel approach to the support of concurrent programming concepts in VMs. It proposes an ownership-based metaobject protocol to facilitate the implementation of such concepts. It is designed to enable a language and library implementer to build domain-specific abstractions, and thus, to facilitate the exploration of concurrent programming in order to solve the problems of the multicore and manycore era.

The OMOP needs to be supported by a VM to provide the necessary performance. It is geared towards multi-language VMs where the complexity of the VM implementation is a concern. The OMOP's unifying properties help avoid the complexity that comes from direct support of an arbitrary number of concurrent programming concepts inside a VM, because it avoids complex feature interactions between them. Instead, it allows language implementers to build these concepts on top of the VM, and thus, avoids the need for direct VM support. This is beneficial, because concurrent programming concepts come with a wide range of variations which are useful for different purposes. Here the OMOP provides the necessary unifying substrate to keep VM complexity manageable and empower language implementers to provide these variations, and thus, to facilitate concurrent programming.

[2] Communication with Eliot Miranda:
http://forum.world.st/Multiple-Bytecode-Sets-tp4651555p4651569.html

[3] LuaJIT 2.0.0-beta11, in `lj_dispatch_update`, http://luajit.org/

A

APPENDIX: SURVEY MATERIAL

This appendix lists the templates used for the surveys in Chapter 3 and provides tables with additional material to complement the discussion.

The survey results have been recorded in a semi-structured textual format relying on YAML, a *"human-friendly data serialization language"*. In this semi-structured form, it was possible check certain consistency properties and process the structural and interconnection informations (cf. Sec. 3.2.3). The results have been processed with a Python script that is available online together with the other technical artifacts of this dissertation.[1]

A.1. VM Support for Concurrent and Parallel Programming

The first survey, presented in Sec. 3.1, is based on the survey template shown in Lst. A.1. For each subject, we recored a number of required information such as a brief description, a list of concepts supported, and our conclusion of the supported model. For each concept we state the way it is exposed as well. Concept exposure and the supported models have to be one from the given lists.

```
1 Survey:
2   description:
3       Which concepts are exposed by a VM,
4       and how are they exposed?
5   required information:
6     - description
```

[1] http://www.stefan-marr.de/research/omop/

```
7       - concepts
8       - spec:   bool
9       - source: bool
10      - version
11      - supported models
12      - references
13    concept exposure:
14      - implicit semantics
15      - ISA
16      - primitives
17      - library
18    supported models:
19      - threads and locks
20      - data parallelism
21      - communicating isolates
22      - communicating threads
23      - none
```

Listing A.1: Survey structure to assess support for concurrent and parallel programming in VMs

The resulting entry for each survey subject looks similar to the one for the DisVM given in Lst. A.2. Depending on the VM, the description highlights specific characteristics such as the documented design intent and details on the technical realization, for instance the relevant concepts provided as opcodes.

```
1  DisVM:
2    description:
3      It is inspired by CSP, but does not give the expected
4      guarantees. It is a VM for the Limbo language, which
5      supports threads and channels. Channels are the only
6      means for synchronization. Channels are use for
7      coordination, instead of being used to communicate all
8      data. Threads actually share their heap memory. The
9      Limbo manual shows how to build a monitor out of
10     channels. Something like a memory model is not
11     discussed in the specification.
12
13     == DisVM opcodes
14       alt, nbalt      # non-blocking, picks a ready channel
15                       # from a list of send/receive channels
16       spawn mspawn    # intra and inter module spawn
17
18       newcb, newcw,  # new channel with type
19       newcf, newcp, newcm, newcmp, newcl
20
21       send            # on channel, with rendezvous semantics
```

```
22        recv
23    concepts:
24      - shared memory:              implicit semantics
25      - channels:                   ISA
26      - threads:                    ISA
27    spec:    yes
28    source: no
29    references: http://doc.cat-v.org/inferno/
30                4th_edition/dis_VM_specification
31    version: 4th edition
32    supported models:
33      - communicating threads
```

Listing A.2: Example: Information recorded for the DisVM

Tab. A.1 complements the discussion of Sec. 3.1 by listing the full set of information gathered with regard to how VMs expose concepts for concurrent and parallel programming to their users. For reference, the table includes the basic data such as version, and supported models as well.

Table A.1.: Virtual Machines: Details and Concept Exposure

CLI		Concept	Exposure
Based on:	spec	asynchronous operations	library
Version:	5th edition	atomic operations	primitives
Supported		fences	primitives
Models:	threads and locks	locks	primitives
	communicating isolates	memory model	implicit semantics
		monitors	primitives
		parallel loops	library
		processes	primitives
		shared memory	implicit semantics
		synchronized methods	ISA
		threads	primitives
		volatile variables	ISA

Dalvik		Concept	Exposure
Based on:	source	atomic operations	primitives
Version:	Android 4.0	barriers	library
Supported		concurrent objects	library
Models:	threads and locks	condition variables	primitives
		fork/join	library
		futures	library
		locks	library
		memory model	implicit semantics
		monitors	ISA
		reflection	primitives
		semaphore	library
		shared memory	implicit semantics
		synchronized methods	ISA

		thread pools	library
		threads	primitives
		volatile fields	ISA

DisVM		Concept	Exposure
Based on:	spec	channels	ISA
Version:	4th edition	shared memory	implicit semantics
Supported		threads	ISA
Models:	communicating threads		

ECMAScript+HTML5		Concept	Exposure
Based on:	spec, source	channels	primitives
Version:	ECMAScript5.1, HTML5	processes	primitives
Supported			
Models:	communicating isolates		

Erlang		Concept	Exposure
Based on:	source	actors	ISA
Version:	Erlang/OTP R15B01	far-references	primitives
Supported		message queue	ISA
Models:	communicating isolates	message sends	ISA
		processes	primitives
		shared memory	primitives

GHC		Concept	Exposure
Based on:	source	MVars	primitives
Version:	GHC 7.5.20120411	channels	library
Supported		fork/join	primitives
Models:	communicating threads	map/reduce	library
		parallel bulk operations	library
		semaphore	library
		threads	primitives
		transactions	primitives

JVM		Concept	Exposure
Based on:	spec, source	atomic operations	primitives
Version:	Java SE 7 Edition	barriers	library
Supported		concurrent objects	library
Models:	threads and locks	condition variables	primitives
		fork/join	library
		futures	library
		locks	primitives
		memory model	implicit semantics
		monitors	ISA
		reflection	primitives
		semaphore	library
		shared memory	implicit semantics
		synchronized methods	ISA
		thread pools	library
		threads	primitives
		volatile fields	ISA

Mozart		Concept	Exposure
Based on:	source	active objects	library
Version:	1.4.0.20080704	atomic swap	primitives
Supported		by-value	primitives
Models:	threads and locks	channels	primitives
	communicating threads	data-flow variables	implicit semantics
		distribution	primitives
		far-references	primitives
		futures	primitives
		green threads	primitives
		immutability	primitives
		locks	ISA
		monitors	library
		replication	primitives
		shared memory	implicit semantics
		single assignment variables	primitives

Perl		Concept	Exposure
Based on:	source	channels	primitives
Version:	5.14.2	condition variables	primitives
Supported		locks	primitives
Models:	communicating isolates	processes	primitives
		semaphore	primitives
		shared memory	primitives

Python		Concept	Exposure
Based on:	source	barriers	library
Version:	3.2.3	channels	primitives
Supported		co-routines	library
Models:	threads and locks	condition variables	primitives
	communicating isolates	futures	library
		global interpreter lock	implicit semantics
		locks	primitives
		message queue	primitives
		processes	primitives
		semaphore	primitives
		shared memory	primitives
		threads	primitives

Ruby		Concept	Exposure
Based on:	source	co-routines	primitives
Version:	1.9.3	condition variables	library
Supported		global interpreter lock	implicit semantics
Models:	threads and locks	locks	primitives
		shared memory	implicit semantics
		threads	primitives

Self		Concept	Exposure
Based on:	source	green threads	library
Version:	4.4	semaphore	library
Supported		shared memory	implicit semantics
Models:	threads and locks		

Squeak		Concept	Exposure
Based on:	spec, source	green threads	primitives
Version:	4.3	semaphore	primitives
Supported		shared memory	implicit semantics
Models:	threads and locks		

A.2. Concurrent and Parallel Programming Concepts

To complement the survey of concurrent and parallel programming concepts of Sec. 3.2, we present here the used templates and a list of concepts recorded for each subject. Furthermore, we include the template used to assess the concepts themselves.

Lst. A.3 shows the template used to record the informations about all the survey subjects. Beside a description, we collected a list of concepts for each subject. Furthermore, we assessed how the various concepts' semantics are enforced. For later reference, we also recorded information on the paper the subject relates to, the online article, or specification describing a language in detail.

```
Survey:
  description:
      Which concepts does the language/paper provide,
      and how are guarantees enforced?
  required information:
    - description
    - concepts
    - enforcement approaches
  other information:
    - url
    - spec
    - bibkey
  enforcement approaches:
    - by-convention
    - by-compilation
    - by-construction
```

Listing A.3: Survey structure to record concepts provided by languages, or discussed in papers and surveys.

Lst. A.4 demonstrate how the survey template is applied to the Axum programming language. A brief description characterizes the main aspects of the language related to the survey for future reference. The list of concepts

includes all notable mechanisms and ideas exposed by the subject. Note however that we did not attempt to full completeness for every subject. Instead, we concentrated on overall completeness. Thus, we did not necessarily record common recurring concepts.

```
Axum:
  description:
      Axum was an experimental language at Microsoft. It
      provides agents, which are supposed to be proper actors
      like in Erlang. It makes the communication medium, the
      channel explicit. Messages are not send to actors, but to
      channels. Channels have potentially multiple input and
      output ports, which allows to associate semantics with
      raw-data values easily. Channels can define
      state-machines to express the protocol to be used.
      Further, it supports 'schemas' which are immutable value
      structs. Beyond agents and channels, they offer also the
      concept of a domain. That is basically a reader/writer
      lock and distinction of agents. An agent declared with
      the reader keyword is only allowed to read domain state,
      while writer agents can both read and write the state. An
      agent declared as neither reader nor writer can only read
      the immutable state of the domain. And they have an
      'unsafe' keyword which allows to break the abstractions.
      The 'isolated' keyword isolates an object from its class
      to avoid problems, i.e., race-conditions on shared state.
  concepts:
    - actors
    - channels
    - isolation
    - by-value
    - immutability
    - Axum-Domains
    - reader-writer-locks
    - synchronization
  enforcement approaches:
    - by-construction
    - by-convention
  spec: http://download.microsoft.com/download/B/D/5/
        BD51FFB2-C777-43B0-AC24-BDE3C88E231F/
        Axum%20Language%20Spec.pdf
  url:  http://download.microsoft.com/download/B/D/5/
        BD51FFB2-C777-43B0-AC24-BDE3C88E231F/
        Axum%20Programmers%20Guide.pdf
```

Listing A.4: Example: Information recorded for the Axum language.

Tab. A.2 gives the full list of concepts and the corresponding language or paper in which they have been identified.

Table A.2.: Concepts provided by languages and proposed in papers.

Language/Paper	Concept	Concept
Active Objects	asynchronous invocation	futures
	event-loop	
Ada	guards	synchronization
	monitors	threads
Aida	ownership	transactions
Alice	atomic primitives	futures
AmbientTalk	actors	futures
	asynchronous invocation	isolates
	event-loop	mirrors
	far-references	
Ateji PX	channels	reducers
	fork/join	speculative execution
	parallel loops	
Axum	Axum-Domains	immutability
	actors	isolation
	by-value	reader-writer-locks
	channels	synchronization
Briot et al.	active objects	locality
	actors	locks
	asynchronous invocation	message sends
	atomic operations	monitors
	concurrent objects	reflection
	data parallelism	replication
	event-loop	semaphore
	events	synchronization
	futures	threads
	green threads	transactions
	guards	
C#	concurrent objects	futures
	fork/join	parallel loops
C/C++11	asynchronous operations	locks
	atomic operations	memory model
	atomic primitives	thread-local variables
	condition variables	threads
	fences	volatiles

Table A.2 – continued from previous page

Language/Paper	Concept	Concept
	futures	
Chapel	PGAS	parallel loops
	channels	parallel prefix scans
	fork/join	reducers
	green threads	
Charm++	active objects	by-value
	asynchronous invocation	event-loop
Cilk	fork/join	speculative parallelism
	reducers	
Clojure	atoms	persistent data structures
	compare-and-swap	thread pools
	immutability	transactions
CoBoxes	actors	far-references
	by-value	immutability
Concurrent Haskell	channels	threads
	green threads	transactions
Concurrent ML	channels	threads
	synchronization	
Concurrent Objects	concurrent objects	
Concurrent Pascal	monitors	threads
Concurrent Smalltalk	asynchronous invocation	promises
	monitors	
Erlang	actors	single assignment variables
	immutability	
Fortran 2008	barriers	locks
	critical sections	parallel loops
Fortress		parallel loops
	immutability	reducers
	implicit parallelism	threads
	memory model	transactions
Go	channels	green threads
Io	actors	futures
	co-routines	

Table A.2 – continued from previous page

Language/Paper	Concept	Concept
JCSP	channels	processes
Java 7	atomic primitives	memory model
	concurrent objects	monitors
	condition variables	thread pools
	fork/join	thread-local variables
	futures	threads
	locks	volatiles
Java Views	locks	
Join Java	message sends	threads
	synchronization	
Linda	by-value	isolation
	coordination	processes
	immutability	
MPI	barriers	one-sided communication
	by-value	processes
	message sends	
MapReduce	immutability	side-effect free
	reducers	
MultiLisp	atomic primitives	locks
	futures	
Occam-pi	by-value	processes
	channels	
OpenCL	atomic primitives	vector operations
	data movement	
OpenMP		parallel blocks
	atomic primitives	parallel loops
	barriers	thread-local variables
	monitors	threads
Orleans	actors	state reconciliation
	futures	transactions
	replication	vats
Oz	active objects	locks
	channels	single assignment variables
	futures	threads
Parallel Actor Monitors	actors	reader-writer-locks
	message sends	

Table A.2 – continued from previous page

Language/Paper	Concept	Concept
Parallel Prolog	data-flow graphs	mvars
	fork/join	parallel bulk operations
	futures	
Reactive Objects	asynchronous invocation	encapsulation
SCOOP	condition variables	ownership
	locks	threads
STM	transactions	
Skillicorn and Talia	active objects	message sends
	actors	one-sided communication
	channels	parallel loops
	data parallelism	processes
	data streams	side-effect free
	implicit parallelism	synchronization
	isolation	threads
	join	tuple spaces
	locality	
Sly	barriers	map/reduce
	join	race-and-repair
StreamIt	data streams	message sends
Swing	event-loop	threads
	message sends	
UPC	PGAS	reducers
	barriers	
X10	APGAS	condition variables
	atomic primitives	far-references
	by-value	fork/join
	clocks	message sends
XC	channels	threads

Each identified concept was briefly characterized and then assessed with regard to the identified survey criteria (cf. Sec. 3.2.1.1). As discussed in Sec. 3.2.2 (cf. Tab. 3.5) certain concepts have not been regarded individually. Instead, closely related concepts have been surveyed together. This information, and general relationships between concepts have been recorded for each concept as well.

```
Survey:
  description:
    Characterize concepts to distinguish and assess them.
  required information:
    - description
    - survey: [PA, Lib, Sem, Perf]
  other information:
    - subsumed-by: concept
    - combines:    concepts
    - related concepts: concepts
```

Listing A.5: Survey structure for the identified concepts.

Lst. A.6 shows how the template was filled in for Clojure atoms concept. It gives a brief description of atoms and indicates that they are closely related to the atomic primitive *compare-and-swap*. Furthermore, the listing includes the results for the survey questions.

```
Atoms:
  description:
      An atom represents a mutable cell which is updated without
      coordination with other cells. Updates are expressed in
      terms of update functionsm, which take the old value as
      input and compute the new value. One requirement for these
      update functions is that they are idempotent. This is
      necessary since atoms are updated concurrently using the
      basic compare-and-swap concept. Thus, an update can fail
      if an other updated succeeded concurrently. In that case,
      the update is attempted again.
  related concepts:
    - compare-and-swap
  Survey:
    (PA)  Already available in VMs:                no
    (Lib) can be implemented as lib:               yes
    (Sem) requires runtime to enforce guarantees:  no
    (Pef) benefits from runtime support for perf.: no
```

Listing A.6: Example: Information recorded for Clojure atoms.

B

APPENDIX: PERFORMANCE EVALUATION

This appendix characterizes the used benchmarks and gives the configuration informations used for the benchmarks described in Chapter 8.

B.1. Benchmark Characterizations

This section gives an overview over the different benchmarks used in this evaluation. This characterization is meant to facilitate the discussion and interpretation of our benchmark results in Chapter 8.

We categorize the benchmarks in *microbenchmarks* and *kernel benchmarks* to distinguish their general characteristics and scope. None of these benchmarks qualifies as a *real world application* benchmark, as for instance defined by the DaCapo benchmarks [Blackburn et al., 2006].

The benchmarks used in this evaluation have either been implemented specifically for our experiments or are derived from commonly used benchmarks. While the microbenchmarks are designed to measure a single aspect of the implementation, the kernel benchmarks are meant to represent a number of typical performance sensitive work loads that can be found embedded in applications.

The remainder of this section discusses the used benchmarks, their characteristics, and our expectation of their behavior. Furthermore, we describe the adaptations to use them in the context of LRSTM and AmbientTalkST, as well as our expectation of how the benchmark behaves on top of ideal implementations of these systems.

B.1.1. Microbenchmarks

We implemented the microbenchmarks specifically for assessing the performance of the presented implementations. The changes to the LRSTM and AmbientTalkST variants of the benchmarks are restricted to the necessary setup operations to execute the benchmarks in the context of the actor or STM implementation.

Int Loop This benchmark is a tight `while` loop that subtracts `1` from an integer and executes until the integer is zero. This loop only executes basic stack operations and uses a special bytecode to do the subtraction. Thus, it does neither create context objects for message sends nor allocate other objects, because the integers are represented as immediate values.

LRSTM: Loop executed in context of the STM, but since no object field mutation is done, an ideal implementation will not exhibit any slowdown. An OMOP-based implementation might need to reify the send of the subtraction operation, which will lead to a significant reduction of performance.

AmbientTalkST: The loop is executed inside a single actor and thus, an ideal implementation would not exhibit any slowdown. OMOP-based implementations might have the same message reification penalty as for the LRSTM implementation.

Float Loop This benchmark is a tight `while` loop that subsequently does two additions of floating point numbers until the floating point value reaches a predefined limit. The loop results in corresponding stack operations for instance to store and load the float. Similarly to the *Int Loop*, the addition is done by a special bytecode that does not require a message send with context object creation. However, the floating point numbers are represented as heap objects and lead to high allocation rates. Note that we do not measure GC time but use a large enough heap instead, which avoids garbage collection completely.

LRSTM and AmbientTalkST: We expect the same results as for the *Int Loop*, because the implementation is nearly identical.

InstVar Access This benchmark is a tight `while` loop identical to the *Int Loop*, with additionally reads and writes an object field. All operations are encoded with bytecodes without resulting in message sends. Thus, neither allocation nor creation of context objects is done.

LRSTM: The STM system needs to track the mutation, which will result in a significant overhead. However, in the context of the OMOP, the same concerns apply as for the *Int Loop* with regard to the reification of the message send for the subtraction of the loop integer.

AmbientTalkST: The loop is executed local to an actor. Thus, an ideal implementation does not have any overhead. For the OMOP-based implementation, the reification of the message send for the subtraction might be an issue as well.

Array Access This benchmark is a tight `while` loop identical to the *Int Loop*, with additionally reads and writes of a fixed array index. All operations are encoded with bytecodes without resulting in message sends. Thus, neither allocation nor creation of context objects is done.

LRSTM and AmbientTalkST: We expect the same results as for the *InstVar access* benchmark, because the implementation is nearly identical.

ClassVar Access This benchmark is a tight `while` loop identical to the *Int Loop*, with additionally reads and writes to a class variable. Class variables are represented as literal *associations*. A reference to a pair of the name and the value of such global is encoded in the compiled method and the VM uses special bytecodes (`popIntoLiteral`, `pushLiteral`) to operate on them. Similar to the previous benchmarks, this one does neither message sends nor object allocation.

LRSTM: The STM system will track the mutation of the literal association.

AmbientTalkST: A faithful actor implementation needs to regard this kind of global state properly, and thus, should exhibit a slowdown.

Message Send The *Send* and *SendWithManyArguments* benchmarks are identical to the *Int Loop* but do a message send in addition. *Send* will send a message without any arguments, while *SendWithManyArguments* sends a message with ten arguments. Thus, for this benchmark context objects need to be allocated for the execution of the sends. The invoked methods return immediately without performing any work.

LRSTM: This benchmark does not perform any mutation and an ideal STM implementation will not exhibit any slowdown. However, OMOP-based implementations might take a penalty for the message sends.

AmbientTalkST: All message sends are local to a single actor and thus, should not exhibit any slowdowns. However, the OMOP might show

additional overhead and the penalty for the subtraction in the *Int Loop* loop.

Local vs. Remote Sends for AmbientTalkST The benchmarks for local and remote sends between actors are meant to measure cost of asynchronous message sends for the two AmbientTalkST implementations. The three benchmarks do local sends to a counter, remote sends to a counter in another actor, and remote sends with ten arguments. They each execute a tight loop which sends a message to increase the counter that is kept in a separate object. In the local version, the send ideally results only in the creation of a context object in which the counter's object field is read and modified, while in the remote version a message object has to be created that is enqueued in the actor's message inbox and then processed as soon as the receiving actor is scheduled.

B.1.2. Kernel Benchmarks

The used kernel benchmarks are collection from different sources, most notably the Computer Language Benchmark Game. We adapted them for our experiments with LRSTM and AmbientTalkST, but did not change the underlying algorithms or functionality.

Compiler We chose the compiler benchmark as a sufficiently complex processes, exercising a wide range of aspects of the system. The benchmark compiles a Smalltalk method of 11 lines of code, which include tests, loops, and a variety of message sends. The compiler will first construct an AST from the source code and then produce a `CompiledMethod` from this AST, which contains the final bytecode sequence. In the process, it creates and traverses the AST, uses a variety of collections, does string processing, and uses the stream facilities.

Slopstone and Smopstone These benchmarks[1] are collections of Smalltalk microbenchmarks and kernel benchmarks. The Slopstone microbenchmark collection tests integer addition, float addition, character access on strings, object creation, object cloning, reflective method invocation with `#perform:`, and block evaluation. We include this benchmark since the structure is different from our benchmarks. It does not use a tight loop but performs a larger number of operations directly in sequence.

[1] According to the source comment, they have been authored by Bruce Samuelson in 1993.

We treat the Slopstone benchmarks as a single kernel benchmark covering a wider range of aspects with a single result.

The Smopstone benchmarks collections contains kernel benchmarks. It calculates fractonaccis, which is a similar to a recursive fibonacci, but uses fractions instead of natural numbers. Furthermore, it generates a collection of prime numbers, generates and parses streams as well as strings, creates sets, sorts strings, and has a benchmark called *Sorcerer's Apprentice*, which exercises recursive blocks, integer arithmetic, and collections. Similar to the Slopstone benchmark, only the overall result is reported.

Binary Trees This benchmark is used from *The Computer Language Benchmark Game* (CLBG) and is an adaptation of Hans Boehm's GCBench.[2] It allocates many binary trees. The trees are created and afterwards traversed once as a consistency check.

LRSTM: The benchmark is executed unchanged in the context of the STM, which will track all mutations.

AmbientTalkST: The tree factory is represented by a separate actor. It will create the trees on the request of the main actor, and return a reference (a far reference in AmbientTalk terminology). Thus, the factory actor remains the owner and the main actor will send an asynchronous request for the consistency check, which will be processed by the factor actor. All overhead should be attributable to this constant number of asynchronous message sends per created tree.

Chameneos The Chameneos benchmark is used from the CLGB and has been used by Kaiser and Pradat-Peyre [2003] to study the differences of synchronization constructs in different languages. It repeatedly performs symmetrical thread rendezvous requests. It uses Smalltalk processes and semaphores for the synchronization. Thus, the main aspect measured is the overhead of scheduling, the synchronization, and notification operations.

Fannkuch The Fannkuch benchmark is used from the CLBG. It repeatedly accesses a tiny integer-sequence to generate permutations of it. It has been used by Anderson and Rettig [1994] to analyze the performance of Lisp implementations. The CLBG implementation uses a stateful object

[2] http://www.hpl.hp.com/personal/Hans_Boehm/gc/gc_bench/applet/GCBench.java

to generate the integer-sequence permutations. After the permutations are generated, they are post processed and checked for correctness.

LRSTM: The benchmark is used unchanged in the context of the STM and all mutations are tracked.

AmbientTalkST: The object generating the permutations is contained in a separate actor and the arrays containing the permutations remain owned by that actor as well. Thus, the message sends for creating the permutation as well as the ones for checking correctness are done asynchronously, and all overhead in AmbientTalkST implementations should be attributable to them.

Fasta The Fasta benchmark is used from the CLBG. It generates and writes random DNA sequences. In the process, it uses a variety of stream operations and a simple random number generator.

LRSTM: We use this benchmark unchanged in the context of LRSTM. Ideally, all measure overhead would come from the overhead of tracking mutations. The tracking of mutations is identical for the ad hoc and the OMOP-based implementations.

AmbientTalkST: We change the set up of the benchmark slightly, and treat the random number generator as a shared resource that is managed by a separate actor. Thus, the main actor needs to send asynchronous requests for randomization to the second actor. Ideally, all measured overhead can be directly attributed to these asynchronous message sends.

NBody The NBody benchmark is used from the CLBG. It performs an N-body simulation of the Jovian moons and uses a simple symplectic-integrator. Consequently, it is performing mostly floating point arithmetic on a fix set of stateful objects.

LRSTM: The benchmark is used unchanged in the context of the STM and all mutations are tracked.

AmbientTalkST: All objects representing the moons, i. e., bodies are contained in a single actor. In addition, we use the main actor to coordinate the simulation, which consequently sends a fixed number of asynchronous messages to the actor containing the moon objects. In an ideal implementation, all measured overhead can be attributed to these asynchronous messages.

B.2. Benchmark Configurations

The benchmark configuration is given in the YAML text format, which is interpreted by ReBench[3] to execute the benchmarks. The framework for writing and executing the benchmarks is SMark.[4]

```
statistics:       {min_runs: 100, max_runs: 100}
benchmark_suites:
  base-benchmarks:
    performance_reader: LogPerformance
    command: " %(benchmark)s "
    input_sizes: pharo-omni.image ReBenchHarness
    benchmarks:
      # Classic benchmarks
      - SMarkLoops.benchFloatLoop:             {extra_args: "1 30"}
      - SMarkLoops.benchIntLoop:               {extra_args: "1 200"}
      - SMarkCompiler:                         {extra_args: "1 300"}
      - SMarkSlopstone:                        {extra_args: "1 1 25000"}
      - SMarkSmopstone:                        {extra_args: "1 1 2"}
      # Computer Language Benchmarks Game
      - BenchmarkGameSuite.benchFasta:         {extra_args: "1 1 50000"}
      - BenchmarkGameSuite.benchBinaryTrees:   {extra_args: "1 1 10"}
      - BenchmarkGameSuite.benchFannkuchRedux: {extra_args: "1 1 8"}
      - BenchmarkGameSuite.benchNBody:         {extra_args: "1 1 20000"}
      - BenchmarkGameSuite.benchChameleons:    {extra_args: "1 1 70000"}

  LRSTM-adhoc-vs-omop:
    performance_reader: LogPerformance
    command: " %(benchmark)s "
    benchmarks:
      - LRSTMBenchmarkGameSuite.benchAtomicBinaryTrees:
          extra_args: "1 1 9"
      - LRSTMBenchmarkGameSuite.benchAtomicFannkuchRedux:
          extra_args: "1 1 7"
      - LRSTMBenchmarkGameSuite.benchAtomicFasta:
          extra_args: "1 1 2000"
      - LRSTMBenchmarkGameSuite.benchAtomicNBody:
          extra_args: "1 1 2000"
      - LRSTMBenchLoops.benchAtomicFloatLoop:{extra_args: "1 1 3000000"}
      - LRSTMBenchLoops.benchAtomicIntLoop:  {extra_args: "1 1 12000000"}
      - LRSTMBenchLoops.benchAtomicArrayAccess:
          extra_args: "1 1 400000"
      - LRSTMBenchLoops.benchAtomicClassVarBinding:
          extra_args: "1 1 500000"
      - LRSTMBenchLoops.benchAtomicInstVarAccess:
          extra_args: "1 1 500000"
      - LRSTMBenchLoops.benchAtomicSend:     {extra_args: "1 1 5000000"}
      - LRSTMBenchLoops.benchAtomicSendWithManyArguments:
          extra_args: "1 1 6000000"

  AmbientTalkST-adhoc-vs-omop:
```

[3] http://github.com/smarr/ReBench

[4] http://www.squeaksource.com/SMark.html

```
46      performance_reader: LogPerformance
47      command: " %(benchmark)s "
48      benchmarks:
49        - ATBenchLoops.benchATFloatLoop:      {extra_args: "1 1 3000000"}
50        - ATBenchLoops.benchATIntLoop:        {extra_args: "1 1 15000000"}
51        - ATBenchLoops.benchATArrayAccess:    {extra_args: "1 1 2000000"}
52        - ATBenchLoops.benchATClassVarBinding:{extra_args: "1 1 9000000"}
53        - ATBenchLoops.benchATInstVarAccess:  {extra_args: "1 1 5500000"}
54        - ATBenchLoops.benchATSend:           {extra_args: "1 1 5000000"}
55        - ATBenchLoops.benchATSendWithManyArguments:
56            extra_args: "1 1 6000000"
57        - ATBenchmarkGameSuite.benchATBinaryTrees:  {extra_args: "1 1 11"}
58        - ATBenchmarkGameSuite.benchATFannkuchRedux:{extra_args: "1 1 6"}
59        - ATBenchmarkGameSuite.benchATFasta:  {extra_args: "1 1 2000"}
60        - ATBenchmarkGameSuite.benchATNBody:  {extra_args: "1 1 12000"}
61        - ATMicroBenchmarks.benchLocalSend:   {extra_args: "1 1 5000000"}
62        - ATMicroBenchmarks.benchRemoteSend:  {extra_args: "1 1 50000"}
63        - ATMicroBenchmarks.benchRemoteSendWithManyArguments:
64            extra_args: "1 1 50000"
65
66    enforcement-overhead:
67      performance_reader: LogPerformance
68      command: " %(benchmark)s "
69      input_sizes:
70        - pharo-omni.image SMarkHarness
71        - omnist.image SMarkHarness
72      benchmarks:
73        # Computer Language Benchmarks Game
74        - BenchmarkGameSuite.benchBinaryTrees:  {extra_args: "1 1 7"}
75        - OstBenchGameSuite.benchBinaryTrees:   {extra_args: "1 1 7"}
76        - BenchmarkGameSuite.benchFannkuchRedux:{extra_args: "1 1 8"}
77        - OstBenchGameSuite.benchFannkuchRedux: {extra_args: "1 1 8"}
78        - BenchmarkGameSuite.benchFasta:        {extra_args: "1 1 20000"}
79        - OstBenchGameSuite.benchFasta:         {extra_args: "1 1 20000"}
80        - BenchmarkGameSuite.benchNBody:        {extra_args: "1 1 4000"}
81        - OstBenchGameSuite.benchNBody:         {extra_args: "1 1 4000"}
82        # Classic benchmarks
83        - SMarkLoops.benchFloatLoop:    {extra_args: "1 1 1000000"}
84        - OstBenchLoops.benchFloatLoop: {extra_args: "1 1 1000000"}
85        - SMarkLoops.benchIntLoop:      {extra_args: "1 1 10000000"}
86        - OstBenchLoops.benchIntLoop:   {extra_args: "1 1 10000000"}
87        - SMarkSlopstone:               {extra_args: "1 1 800"}
88        - OstBenchSlopstone:            {extra_args: "1 1 800"}
89
90  virtual_machines:
91    RoarVM+OMOP:
92      path: ../../bin/omnivm
93      args: "-min_heap_MB 1024 -num_cores 1 -headless %(input)s "
94    RoarVM+OMOP (full):
95      path: ../../bin/omnivm-always-trapping
96      args: "-min_heap_MB 1024 -num_cores 1 -headless %(input)s "
97    RoarVM+OMOP (opt):
98      path: ../../bin/omnivm-opt1core
99      args: "-min_heap_MB 1024 -num_cores 1 -headless %(input)s "
100   RoarVM:
```

```
    path: ../../bin/rvm
    args: "-min_heap_MB 1024 -num_cores 1 -headless %(input)s "
  RoarVM (opt):
    path: ../../bin/rvm-opt1core
    args: "-min_heap_MB 1024 -num_cores 1 -headless %(input)s "
  RoarVM (GCC 4.2):
    path: ../../bin/rvm-gcc42
    args: "-min_heap_MB 1024 -num_cores 1 -headless %(input)s "
  CogVM:
    path: ../../CogVM.app/Contents/MacOS/CogVM
    args: "-memory 1024m -headless %(input)s "

run_definitions:
  all-vms:
    benchmark: base-benchmarks
    executions: [CogVM, RoarVM, RoarVM (GCC 4.2), RoarVM (opt),
                 RoarVM+OMOP, RoarVM+OMOP (full), RoarVM+OMOP (opt)]
  enforcement-overhead:
    benchmark: enforcement-overhead
    executions: [RoarVM+OMOP (full), RoarVM+OMOP (opt), CogVM]
  adhoc-vs-omop:
    executions:
      - CogVM:
        benchmark:
          - LRSTM-adhoc-vs-omop:
              input_sizes:
                - omnist.image SMarkHarness
                - LRSTM.image SMarkHarness
          - AmbientTalkST-adhoc-vs-omop:
              input_sizes:
                - omnist.image SMarkHarness
                - AmbientTalkST.image SMarkHarness
      - RoarVM (opt):
        benchmark:   AmbientTalkST-adhoc-vs-omop
        input_sizes: "AmbientTalkST.image SMarkHarness "
      - RoarVM (opt):
        benchmark:   LRSTM-adhoc-vs-omop
        input_sizes: "LRSTM.image SMarkHarness "
      - RoarVM+OMOP (opt):
        benchmark:
          - AmbientTalkST-adhoc-vs-omop
          - LRSTM-adhoc-vs-omop
        input_sizes: "pharo-omni.image SMarkHarness "
  omnist-vs-omnivm:
   benchmark:
    - LRSTM-adhoc-vs-omop
    - AmbientTalkST-adhoc-vs-omop
   executions:
    - CogVM: {input_sizes: "omnist.image SMarkHarness "}
    - RoarVM+OMOP (full):{input_sizes: "pharo-omni.image SMarkHarness "}
    - RoarVM+OMOP (opt): {input_sizes: "pharo-omni.image SMarkHarness "}
    - RoarVM (opt):      {input_sizes: "omnist.image SMarkHarness "}
```

Listing B.1: Benchmark Configuration for Performance Evaluation

REFERENCES

Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer, New York, 1996. ISBN 0387947752 9780387947754. 91

Harold Abelson, Gerald Jay Sussman, and with Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press/McGraw-Hill, Cambridge, 2nd editon edition, 1996. ISBN 0-262-01153-0. 96

Jonathan Aldrich and Craig Chambers. Ownership domains: Separating aliasing policy from mechanism. In Martin Odersky, editor, *ECOOP 2004 – Object-Oriented Programming*, volume 3086 of *Lecture Notes in Computer Science*, pages 1–25. Springer, Berlin / Heidelberg, 2004. ISBN 978-3-540-22159-3. doi: 10.1007/978-3-540-24851-4_1. 134

George S. Almasi and Allan Gottlieb. *Highly Parallel Computing*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 2nd edition, 1994. ISBN 0805304436 9780805304435. 25, 35

M. Ancona, W. Cazzola, G. Dodero, and V. Gianuzzi. Channel reification: A reflective model for distributed computation. In *Performance, Computing and Communications, 1998. IPCCC '98., IEEE International*, pages 32 –36, February 1998. doi: 10.1109/PCCC.1998.659895. 112

Jakob R. Andersen, Lars Bak, Steffen Grarup, Kasper V. Lund, Toke Eskildsen, Klaus Marius Hansen, and Mads Torgersen. Design, implementation, and evaluation of the resilient smalltalk embedded platform. *Computer Languages, Systems & Structures*, 31(3–4):127 – 141, 2005. ISSN 1477-8424. doi: 10.1016/j.cl.2005.02.003. Smalltalk. 90

Kenneth R. Anderson and Duane Rettig. Performing lisp analysis of the fannkuch benchmark. *SIGPLAN Lisp Pointers*, VII(4):2–12, 1994. ISSN 1045-3563. doi: 10.1145/382109.382124. 265

Joe Armstrong. A history of erlang. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 6–1–6–26, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-766-X. doi: 10.1145/1238844.1238850. 42, 51, 59

Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 2006. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html`. 3

Edward A. Ashcroft and William W. Wadge. Lucid, a nonprocedural language with iteration. *Commun. ACM*, 20(7):519–526, July 1977. ISSN 0001-0782. doi: 10.1145/359636.359715. 35

Tom Axford. *Concurrent Programming - Fundamental Techniques for Real-Time and Parallel Software Design*. Series in Parallel Computing. Wiley, Chichester [u.a.], 1990. ISBN 0471923036. 18

John Aycock. A brief history of just-in-time. *ACM Computing Surveys*, 35(2): 97–113, 2003. ISSN 0360-0300. doi: 10.1145/857076.857077. 2, 14

Rajkishore Barik, Jisheng Zhao, David P. Grove, Igor Peshansky, Zoran Budimlic, and Vivek Sarkar. Communication optimizations for distributed-memory x10 programs. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 1101 –1113, May 2011. doi: 10.1109/IPDPS.2011.105. 61

Paul S. Barth, Rishiyur S. Nikhil, and Arvind. M-structures: Extending a parallel, non-strict, functional language with state. In *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture*, pages 538–568, London, UK, UK, 1991. Springer-Verlag. ISBN 3-540-54396-1. 50

Christian Bauer and Gavin King. *Hibernate in Action*. Manning, Greenwich, CT, 2005. ISBN 1932394370 9781932394375. 79

James R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, June 1973. ISSN 0001-0782. doi: 10.1145/362248.362270. 100

Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 169–190, New York, NY, USA, 2006. ACM. ISBN 1-59593-348-4. doi: 10.1145/1167473.1167488. 206, 229, 261

Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, 1995. ISSN 0362-1340. doi: 10.1145/209937.209958. 19, 22, 34, 59

Jayaram Bobba, Kevin E. Moore, Haris Volos, Luke Yen, Mark D. Hill, Michael M. Swift, and David A. Wood. Performance pathologies in hardware transactional memory. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 81–91, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-706-3. doi: 10.1145/1250662.1250674. 30

Gilad Bracha and David Ungar. Mirrors: design principles for meta-level facilities of object-oriented programming languages. In *Proc. of OOPSLA'04*, pages 331–344. ACM, 2004. ISBN 1-58113-831-9. doi: 10.1145/1028976.1029004. 63, 154

Jean-Pierre Briot. From objects to actors: Study of a limited symbiosis in smalltalk-80. In *Proceedings of the 1988 ACM SIGPLAN workshop on Object-based concurrent programming*, pages 69–72, New York, NY, USA, 1988. ACM. ISBN 0-89791-304-3. doi: 10.1145/67386.67403. 100

Jean-Pierre Briot. Actalk: A testbed for classifying and designing actor languages in the smalltalk-80 environment. In *ECOOP*, British Computer Society Workshop Series, pages 109–129. Cambridge University Press, 1989. 100

Jean-Pierre Briot, Rachid Guerraoui, and Klaus-Peter Lohr. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys*, 30 (3):291–329, September 1998. ISSN 0360-0300. doi: 10.1145/292469.292470. 4, 58, 59

Zoran Budimlic, Aparna Chandramowlishwaran, Kathleen Knobe, Geoff Lowney, Vivek Sarkar, and Leo Treggiari. Multi-core implementations of the concurrent collections programming model. In *he 14th Workshop on Compilers for Parallel Computing*, January 2009. 37

Sergey Bykov, Alan Geller, Gabriel Kliot, James R. Larus, Ravi Pandya, and Jorgen Thelin. Orleans: Cloud computing for everyone. In *Proc of SOCC'11*, pages 16:1–16:14. ACM, 2011. ISBN 978-1-4503-0976-9. doi: 10.1145/2038916.2038932. 59, 171

Bryan Cantrill and Jeff Bonwick. Real-world concurrency. *Commun. ACM*, 51 (11):34–39, 2008. ISSN 0001-0782. doi: 10.1145/1400214.1400227. 27, 36

Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5):46–58, 2008. ISSN 1542-7730. doi: 10.1145/1454456.1454466. 29, 148

Bryan Catanzaro, Armando Fox, Kurt Keutzer, David Patterson, Bor-Yiing Su, Marc Snir, Kunle Olukotun, Pat Hanrahan, and Hassan Chafi. Ubiquitous parallel computing from berkeley, illinois, and stanford. *IEEE Micro*, 30(2): 41–55, 2010. 2, 36, 37

Vincent Cavé, Jisheng Zhao, Zoran Budimlić, Vivek Sarkar, James Gunning, and Michael Glinsky. Habanero-java extensions for scientific computing. In *Proceedings of the 9th Workshop on Parallel/High-Performance Object-Oriented Scientific Computing*, POOSC '10, pages 1:1–1:6, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0546-4. doi: 10.1145/2039312.2039313. 33

Hassan Chafi, Zach DeVito, Adriaan Moors, Tiark Rompf, Arvind K. Sujeeth, Pat Hanrahan, Martin Odersky, and Kunle Olukotun. Language virtualization for heterogeneous parallel computing. In *Proc. of OOPSLA'10*, pages 835–847. ACM, 2010. 2, 36, 37

Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of self a dynamically-typed object-oriented language based on prototypes. *SIGPLAN Not.*, 24(10):49–70, 1989. ISSN 0362-1340. doi: 10.1145/74878.74884. 42, 48, 231

Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In

OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pages 519–538, New York, NY, USA, 2005. ACM. ISBN 1-59593-031-0. doi: 10.1145/1094811.1094852. 27, 33, 59, 75

Tong Chen, Haibo Lin, and Tao Zhang. Orchestrating data transfer for the cell/b.e. processor. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, pages 289–298, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-158-3. doi: 10.1145/1375527.1375570. 171

Shigeru Chiba and Takashi Masuda. Designing an extensible distributed language with a meta-level architecture. In *Proceedings of the 7th European Conference on Object-Oriented Programming*, ECOOP '93, pages 482–501, London, UK, UK, 1993. Springer-Verlag. ISBN 3-540-57120-5. 130

Cincom Systems, Inc. Visualworks 7 white paper. Technical report, 55 Merchant St., Cincinnati, OH 45246, United States, July 2002. URL http://www.cincomsmalltalk.com/CincomSmalltalkWiki/VisualWorks+7+White+Paper. 78

Dave Clarke, Tobias Wrigstad, Johan Östlund, and Einar Johnsen. Minimal ownership for active objects. In G. Ramalingam, editor, *Programming Languages and Systems*, volume 5356 of *Lecture Notes in Computer Science*, pages 139–154. Springer, Berlin / Heidelberg, 2008. ISBN 978-3-540-89329-5. doi: 10.1007/978-3-540-89330-1_11. 170

David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. *SIGPLAN Not.*, 33(10):48–64, October 1998. ISSN 0362-1340. doi: 10.1145/286942.286947. 133

John Clements and Matthias Felleisen. A tail-recursive machine with stack inspection. *ACM Trans. Program. Lang. Syst.*, 26(6):1029–1052, November 2004. ISSN 0164-0925. doi: 10.1145/1034774.1034778. 16

Cliff Click, Gil Tene, and Michael Wolf. The pauseless gc algorithm. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 46–56, New York, NY, USA, 2005. ACM. ISBN 1-59593-047-7. doi: 10.1145/1064979.1064988. 102

Iain D. Craig. *Virtual Machines*. Springer-Verlag, London, 2006. ISBN 978-1-85233-969-2. 14

Koen De Bosschere. Process-based parallel logic programming: A survey of the basic issues. *Journal of Systems and Software*, 39(1):71 – 82, 1997. ISSN 0164-1212. doi: 10.1016/S0164-1212(96)00164-1. 4, 58, 59

Nick De Cooman. A type system for organizing the object soup in ambient-oriented applications. Master thesis, Software Languages Lab, Vrije Universiteit Brussel, August 2012. 134

Joeri De Koster, Stefan Marr, and Theo D'Hondt. Synchronization views for event-loop actors. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 317–318, New York, NY, USA, February 2012. ACM. doi: 10.1145/2145816.2145873. (Poster). 10, 171

Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 137–149, Berkeley, CA, USA, 2004. USENIX Association. 35

Brian Demsky and Patrick Lam. Views: object-inspired concurrency control. In *Proc. of ICSE'10*, 2010. ISBN 978-1-60558-719-6. doi: 10.1145/1806799.1806858. 59

ECMA International. *Standard ECMA-334 - C# Language Specification*. 4 edition, June 2006. URL http://www.ecma-international.org/publications/standards/Ecma-334.htm. 27

ECMA International. *Standard ECMA-335 - Common Language Infrastructure (CLI)*. Geneva, Switzerland, 5 edition, December 2010. URL http://www.ecma-international.org/publications/standards/Ecma-335.htm. 42, 45

ECMA International. *Standard ECMA-262 - ECMAScript Language Specification*. 5.1 edition, June 2011. URL http://www.ecma-international.org/publications/standards/Ecma-262.htm. 42, 51

Tarek A. El-Ghazawi, François Cantonnet, Yiyi Yao, Smita Annareddy, and Ahmed S. Mohamed. Benchmarking parallel compilers: A upc case study. *Future Generation Computer Systems*, 22(7):764 – 775, 2006. ISSN 0167-739X. doi: 10.1016/j.future.2006.02.002. 61

Khaled El Emam, Saïda Benlarbi, Nishith Goel, and Shesh N. Rai. The confounding effect of class size on the validity of object-oriented metrics. *IEEE*

Transactions on Software Engineering, 27(7):630 –650, July 2001. ISSN 0098-5589. doi: 10.1109/32.935855. 140, 162

Johan Fabry and Daniel Galdames. Phantom: a modern aspect language for pharo smalltalk. *Software: Practice and Experience*, pages n/a–n/a, 2012. ISSN 1097-024X. doi: 10.1002/spe.2117. 189

Philip J. Fleming and John J. Wallace. How not to lie with statistics: The correct way to summarize benchmark results. *Commun. ACM*, 29:218–221, March 1986. ISSN 0001-0782. doi: 10.1145/5666.5673. 209, 229

Michael J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901 – 1909, December 1966. ISSN 0018-9219. doi: 10.1109/PROC.1966.5273. 24, 25, 35

Bertil Folliot, Ian Piumarta, and Fabio Riccardi. A dynamically configurable, multi-language execution platform. In *Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications*, EW 8, pages 175–181, New York, NY, USA, 1998. ACM. doi: 10.1145/319195.319222. 16

Cédric Fournet and Andrew D. Gordon. Stack inspection: Theory and variants. *ACM Trans. Program. Lang. Syst.*, 25(3):360–399, May 2003. ISSN 0164-0925. doi: 10.1145/641909.641912. 16

Daniel P. Friedman and Mitchell Wand. Reification: Reflection without metaphysics. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, LFP '84, pages 348–355, New York, NY, USA, 1984. ACM. ISBN 0-89791-142-3. doi: 10.1145/800055.802051. 110

Matteo Frigo, Pablo Halpern, Charles E. Leiserson, and Stephen Lewin-Berlin. Reducers and other cilk++ hyperobjects. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, SPAA '09, pages 79–90, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-606-9. doi: 10.1145/1583991.1584017. 34

Andreas Gal, Christian W. Probst, and Michael Franz. Hotpathvm: An effective jit compiler for resource-constrained devices. In *Proc. of VEE'06*, pages 144–153. ACM, 2006. ISBN 1-59593-332-6. doi: 10.1145/1134760.1134780. 244

Yaoqing Gao and Chung Kwong Yuen. A survey of implementations of concurrent, parallel and distributed smalltalk. *SIGPLAN Not.*, 28(9):29–35, 1993. ISSN 0362-1340. doi: 10.1145/165364.165375. 100

Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine—A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, Cambridge, Mass., 1994. ISBN 0-262-57108-0. 42

David Gelernter. Generative communication in linda. *ACM TOPLAS*, 7:80–112, January 1985. ISSN 0164-0925. doi: 10.1145/2363.2433. 59

Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous java performance evaluation. *SIGPLAN Not.*, 42(10):57–76, 2007. ISSN 0362-1340. doi: 10.1145/1297105.1297033. 207, 229

Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983. ISBN 0201113716. 27, 48, 92, 100, 105, 110

James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification*. California, USA, java se 7 edition edition, February 2012. 27, 46, 47, 59, 92

K John Gough. Stacking them up: a comparison of virtual machines. *Aust. Comput. Sci. Commun.*, 23(4):55–61, January 2001. doi: 10.1145/545615.545603. 45

John D. Mc Gregor. Ecosystems. *Journal of Object Technology*, 8(6):7–16, September-October 2009. 14

Dan Grossman. A sophomoric introduction to shared-memory parallelism and concurrency. Lecture notes, Department of Computer Science & Engineering, University of Washington, AC101 Paul G. Allen Center, Box 352350, 185 Stevens Way, Seattle, WA 98195-2350, December 2012. URL `http://homes.cs.washington.edu/~djg/teachingMaterials/spac/sophomoricParallelismAndConcurrency.pdf`. 22

Dan Grossman and Ruth E. Anderson. Introducing parallelism and concurrency in the data structures course. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, SIGCSE '12, pages 505–510, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1098-7. doi: 10.1145/2157136.2157285. 22

Gopal Gupta, Enrico Pontelli, Khayri A. M. Ali, Mats Carlsson, and Manuel V. Hermenegildo. Parallel execution of prolog programs: A survey. *ACM Trans. Program. Lang. Syst.*, 23(4):472–602, July 2001. ISSN 0164-0925. doi: 10.1145/504083.504085. 4, 58, 59

Rajiv Gupta and Charles R. Hill. A scalable implementation of barrier synchronization using an adaptive combining tree. *International Journal of Parallel Programming*, 18(3):161–180, June 1989. 27

Maurice H. Halstead. *Elements of Software Science*. Elsevier, New York, 1977. ISBN 0-4440-0205-7. 140

Robert H. Halstead, Jr. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7:501–538, October 1985. ISSN 0164-0925. doi: 10.1145/4472.4478. 59

Tim Harris. An extensible virtual machine architecture. In *Proceedings of the OOPSLA'99 Workshop on Simplicity, Performance and Portability in Virtual Machine Design*, November 1999. 16

Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, pages 48–60, New York, NY, USA, 2005. ACM. ISBN 1-59593-080-9. doi: 10.1145/1065944.1065952. 50

Michael Haupt, Bram Adams, Stijn Timbermont, Celina Gibbs, Yvonne Coady, and Robert Hirschfeld. Disentangling virtual machine architecture. *IET Software, Special Issue on Domain-Specific Aspect Languages*, 3(3):201–218, June 2009. ISSN 1751-8806. doi: 10.1049/iet-sen.2007.0121. 16

Michael Haupt, Robert Hirschfeld, Tobias Pape, Gregor Gabrysiak, Stefan Marr, Arne Bergmann, Arvid Heise, Matthias Kleine, and Robert Krahn. The som family: Virtual machines for teaching and research. In *Proceedings of the 15th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE)*, pages 18–22. ACM Press, June 2010. ISBN 978-1-60558-729-5. doi: 10.1145/1822090.1822098. 11

Michael Haupt, Stefan Marr, and Robert Hirschfeld. Csom/pl: A virtual machine product line. *Journal of Object Technology*, 10(12):1–30, 2011a. ISSN 1660-1769. doi: 10.5381/jot.2011.10.1.a12. 11

Michael Haupt, Stefan Marr, and Robert Hirschfeld. Csom/pl: A virtual machine product line. Technical Report 48, Hasso Plattner Institute, Am Neuen Palais 10, 14469 Potsdam, April 2011b. 11

John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, fourth edition, 2007. 17, 18, 66, 171, 229

Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, November 1993. ISSN 0164-0925. doi: 10.1145/161468.161469. 59

Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, April 2008. ISBN 0123705916. 30

Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3): 463–492, July 1990. ISSN 0164-0925. doi: 10.1145/78969.78972. 19

Charlotte Herzeel and Pascal Costanza. Dynamic parallelization of recursive code: part 1: managing control flow interactions with the continuator. *SIGPLAN Not.*, 45:377–396, October 2010. ISSN 0362-1340. doi: 10.1145/1932682.1869491. 66, 171

Charlotte Herzeel, Pascal Costanza, and Theo D'Hondt. An extensible interpreter framework for software transactional memory. *Journal of Universal Computer Science*, 16(2):221–245, January 2010. 29

Carl Hewitt. Actor model of computation: Scalable robust information systems. Technical Report v24, July 2012. URL `http://arxiv.org/abs/1008.1459v24`. 31

Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI'73: Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc. 31, 32

Mark D. Hill and Michael R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008. ISSN 0018-9162. doi: 10.1109/MC.2008.209. 17

Erik Hilsdale and Jim Hugunin. Advice weaving in aspectj. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, AOSD

'04, pages 26–35, New York, NY, USA, 2004. ACM. ISBN 1-58113-842-3. doi: 10.1145/976270.976276. 182

Robert Hirschfeld. Aspects - aspect-oriented programming with squeak. In Mehmet Aksit, Mira Mezini, and Rainer Unland, editors, *NetObjectDays*, volume 2591 of *Lecture Notes in Computer Science*, pages 216–232. Springer, 2002. ISBN 3-540-00737-7. doi: 10.1007/3-540-36557-5_17. 189

C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8): 666–677, 1978. ISSN 0001-0782. doi: 10.1145/359576.359585. 32, 49, 75, 171

C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International Series in Computer Science. Prentice Hall, 1985. ISBN 9780131532892. 32

Kevin J. Hoffman, Harrison Metzger, and Patrick Eugster. Ribbons: A partially shared memory programming model. *SIGPLAN Not.*, 46:289–306, October 2011. ISSN 0362-1340. doi: 10.1145/2076021.2048091. 107, 244

Urs Hölzle, Craig Chambers, and David Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP '91: Proceedings of the European Conference on Object-Oriented Programming*, pages 21–38, London, UK, 1991. Springer-Verlag. ISBN 3-540-54262-0. doi: 10.1007/BFb0057013. 221

Urs Hölzle, Craig Chambers, and David Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, PLDI '92, pages 32–43, New York, NY, USA, 1992. ACM. ISBN 0-89791-475-9. doi: 10.1145/143095.143114. 244

Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of squeak, a practical smalltalk written in itself. *SIGPLAN Not.*, 32(10):318–326, 1997. ISSN 0362-1340. doi: 10.1145/263700.263754. 42

Intel Corporation. Intel64 and IA-32 Architectures Software Developer Manuals. Manual, 2012. URL `http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html`. 20

ISO. *ISO/IEC 9899:2011 Information technology — Programming languages — C*. International Organization for Standardization, Geneva, Switzerland, December 2011. 27, 59

ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, February 2012. 27, 59

G. Itzstein and Mark Jasiunas. On implementing high level concurrency in java. In Amos Omondi and Stanislav Sedukhin, editors, *Advances in Computer Systems Architecture*, volume 2823 of *Lecture Notes in Computer Science*, pages 151–165. Springer Berlin / Heidelberg, 2003. ISBN 978-3-540-20122-9. doi: 10.1007/978-3-540-39864-6_13. 59, 80

Graylin Jay, Joanne E. Hale, Randy K. Smith, David P. Hale, Nicholas A. Kraft, and Charles Ward. Cyclomatic complexity and lines of code: Empirical evidence of a stable linear relationship. *JSEA*, 2(3):137–143, 2009. doi: 10.4236/jsea.2009.23020. 140, 162

Charles R. Johns and Daniel A. Brokenshire. Introduction to the cell broadband engine architecture. *IBM Journal of Research and Development*, 51(5): 503–519, 2007. ISSN 0018-8646. 171

Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 1st edition, 2011. ISBN 1420082795, 9781420082791. 3, 64, 198, 245

Claude Kaiser and Jean-François Pradat-Peyre. Chameneos, a concurrency game for java, ada and others. In *ACS/IEEE International Conference on Computer Systems and Applications, 2003. Book of Abstracts.*, page 62, January 2003. doi: 10.1109/AICCSA.2003.1227495. 265

Peter Kampstra. Beanplot: A boxplot alternative for visual comparison of distributions. *Journal of Statistical Software, Code Snippets*, 28(1):1–9, October 2008. ISSN 1548-7660. 209

Rajesh K. Karmani, Amin Shali, and Gul Agha. Actor frameworks for the jvm platform: A comparative analysis. In *Proc. of PPPJ'09*, pages 11–20. ACM, 2009. ISBN 978-1-60558-598-7. doi: 10.1145/1596655.1596658. 3, 37, 71, 72, 73, 74, 75, 76, 77, 157, 204

Gregor Kiczales. Beyond the black box: Open implementation. *IEEE Software*, 13(1):8–11, 1996. ISSN 0740-7459. doi: 10.1109/52.476280. 110

Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT, 1991. ISBN 978-0-26261-074-2. 109, 110, 111, 134

Gregor Kiczales, John Lamping, Christina Videira Lopes, Chris Maeda, Anurag Mendhekar, and Gail Murphy. Open implementation design guidelines. In *ICSE '97: Proceedings of the 19th international conference on Software engineering*, pages 481–490, New York, NY, USA, 1997. ACM. ISBN 0-89791-914-9. doi: 10.1145/253228.253431. 80, 110

Vivek Kumar, Daniel Frampton, Stephen M Blackburn, David Grove, and Olivier Tardieu. Work-stealing without the baggage. In *Proceedings of the 2012 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA 2012), Tucson, AZ, October 19-26, 2012*, volume 47, pages 297–314. ACM, October 2012. doi: 10.1145/2398857.2384639. 20, 63

Ralf Lämmel. Google's MapReduce Programming Model - Revisited. *SCP*, 70 (1):1 – 30, 2008. ISSN 0167-6423. doi: 10.1016/j.scico.2007.07.001. 34, 59, 160, 171

R. Greg Lavender and Douglas C. Schmidt. Active object: An object behavioral pattern for concurrent programming. In *Pattern Languages of Program Design 2*, pages 483–499. Addison-Wesley Longman Publishing Co., Inc., 1996. ISBN 0-201-895277. 27, 59

Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. The Java Series. Addison Wesley, Reading, MA, 1999. ISBN 9780201310092. 18

Doug Lea. A java fork/join framework. In *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*, pages 36–43. ACM, 2000. ISBN 1-58113-288-3. doi: 10.1145/337449.337465. 34, 37

Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006. ISSN 0018-9162. doi: 10.1109/MC.2006.180. 36

K. Leino, Peter Müller, and Angela Wallenburg. Flexible immutability with frozen objects. In Natarajan Shankar and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments*, volume 5295 of *Lecture Notes in Computer Science*, pages 192–208. Springer, Berlin / Heidelberg, 2008. ISBN 978-3-540-87872-8. doi: 10.1007/978-3-540-87873-5_17. 134

Yun Calvin Lin and Lawrence Snyder. *Principles of Parallel Programming*. Pearson/Addison Wesley, Boston, Mass, 2008. ISBN 978-0321487902. 18

Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java™ Virtual Machine Specification*. Oracle America, Inc., java se 7 edition edition, February 2012. 42, 46

Roberto Lublinerman, Jisheng Zhao, Zoran Budimlić, Swarat Chaudhuri, and Vivek Sarkar. Delegated isolation. *SIGPLAN Not.*, 46:885–902, October 2011. ISSN 0362-1340. doi: 10.1145/2076021.2048133. 59

Lucent Technologies Inc and Vita Nuova Limited. Dis virtual machine specification. Technical report, January 2003. URL `http://doc.cat-v.org/inferno/4th_edition/dis_VM_specification`. 32, 42, 49

Ewing L. Lusk and Katherine A. Yelick. Languages for high-productivity computing: the darpa hpcs language project. *Parallel Processing Letters*, 17 (1):89–102, 2007. doi: 10.1142/S0129626407002892. 33

Pattie Maes. Concepts and experiments in computational reflection. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPSLA '87, pages 147–155, New York, NY, USA, 1987. ACM. ISBN 0-89791-247-0. doi: 10.1145/38765.38821. 111

Antoine Marot. *Preserving the Separation of Concerns while Composing Aspects with Reflective AOP*. Phd thesis, Université Libre De Bruxelles, October 2011. 189

Stefan Marr. Encapsulation and locality: A foundation for concurrency support in multi-language virtual machines? In *SPLASH '10: Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, pages 221–222, New York, NY, USA, October 2010. ACM. ISBN 978-1-4503-0240-1. doi: 10.1145/1869542.1869583. 9

Stefan Marr and Theo D'Hondt. Many-core virtual machines: Decoupling abstract from concrete concurrency, December 2009. 9

Stefan Marr and Theo D'Hondt. Many-core virtual machines: Decoupling abstract from concrete concurrency. In *SPLASH '10: Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, pages 239–240, October 2010. ISBN 978-1-4503-0240-1. doi: 10.1145/1869542.1869593. 9

Stefan Marr and Theo D'Hondt. Identifying a unifying mechanism for the implementation of concurrency abstractions on multi-language virtual machines. In *Objects, Models, Components, Patterns, 50th International Conference, TOOLS 2012*, volume 7304 of *Lecture Notes in Computer Science*, pages 171–186, Berlin / Heidelberg, May 2012. Springer. ISBN 978-3-642-30560-3. doi: 10.1007/978-3-642-30561-0_13. 8, 9, 182, 202

Stefan Marr, Michael Haupt, and Theo D'Hondt. Intermediate language design of high-level language virtual machines: Towards comprehensive concurrency support. In *Proc. VMIL'09 Workshop*, pages 3:1–3:2. ACM, October 2009. ISBN 978-1-60558-874-2. doi: 10.1145/1711506.1711509. (extended abstract).

Stefan Marr, Michael Haupt, Stijn Timbermont, Bram Adams, Theo D'Hondt, Pascal Costanza, and Wolfgang De Meuter. Virtual machine support for many-core architectures: Decoupling abstract from concrete concurrency models. In *Second International Workshop on Programming Languages Approaches to Concurrency and Communication-cEntric Software*, volume 17 of *Electronic Proceedings in Theoretical Computer Science*, pages 63–77, York, UK, February 2010a. 9, 71

Stefan Marr, Stijn Verhaegen, Bruno De Fraine, Theo D'Hondt, and Wolfgang De Meuter. Insertion tree phasers: Efficient and scalable barrier synchronization for fine-grained parallelism. In *Proceedings of the 12th IEEE International Conference on High Performance Computing and Communications*, pages 130–137. IEEE Computer Society, September 2010b. ISBN 978-0-7695-4214-0. doi: 10.1109/HPCC.2010.30. Best Student Paper Award. 9, 22, 62

Stefan Marr, Mattias De Wael, Michael Haupt, and Theo D'Hondt. Which problems does a multi-language virtual machine need to solve in the multicore/manycore era? In *Proceedings of the 5th Workshop on Virtual Machines and Intermediate Languages*, VMIL '11, pages 341–348. ACM, October 2011a. ISBN 978-1-4503-1183-0. doi: 10.1145/2095050.2095104. 71, 72

Stefan Marr, David Ungar, and Theo D'Hondt. Evolving a virtual machine to execute applications fully in parallel: Approaching the multi- and manycore challenge with a simplified design. (unpublished), May 2011b. 11, 71

Stefan Marr, Jens Nicolay, Tom Van Cutsem, and Theo D'Hondt. Modularity and conventions for maintainable concurrent language implementations: A review of our experiences and practices. In *Proceedings of the 2nd Workshop*

on Modularity In Systems Software (MISS'2012), MISS'12. ACM, March 2012. doi: 10.1145/2162024.2162031. 10, 71

Hidehiko Masuhara, Satoshi Matsuoka, Takuo Watanabe, and Akinori Yonezawa. Object-oriented concurrent reflective languages can be implemented efficiently. In *conference proceedings on Object-oriented programming systems, languages, and applications*, OOPSLA '92, pages 127–144, New York, NY, USA, 1992. ACM. ISBN 0-201-53372-3. doi: 10.1145/141936.141948. 132

David May. Occam. *SIGPLAN Not.*, 18(4):69–79, April 1983. ISSN 0362-1340. doi: 10.1145/948176.948183. 32

Thomas J. McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, SE-2(4):308 – 320, dec. 1976. ISSN 0098-5589. doi: 10.1109/TSE.1976.233837. 140

Michael Mehl. *The Oz Virtual Machine - Records, Transients, and Deep Guards*. PhD thesis, Technische Fakultät der Universität des Saarlandes, 1999. 59, 171

Message Passing Interface Forum. Mpi: A message-passing interface standard, version 2.2. Specification, September 2009. URL `http://www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf`. 22, 30, 32, 33, 59

Pierre Michaud, André Seznec, and Stéphan Jourdan. An exploration of instruction fetch requirement in out-of-order superscalar processors. *International Journal of Parallel Programming*, 29(1):35–58, 2001. ISSN 0885-7458. doi: 10.1023/A:1026431920605. 17

Mark S. Miller, E. Dean Tribble, and Jonathan Shapiro. Concurrency among strangers: Programming in e as plan coordination. In R. De Nicola and D. Sangiorgi, editors, *Symposium on Trustworthy Global Computing*, volume 3705 of *LNCS*, pages 195–229. Springer, April 2005. doi: 10.1007/11580850_12. 31, 171

Eliot Miranda. Context management in visualworks 5i. In *OOPSLA'99 Workshop on Simplicity, Performance and Portability in Virtual Machine Design*, Denver, CO, November 1999. 100

Marcus Mitch and Akera Atsushi. Eniac's recessive gene. *Penn Printout*, 12 (4), March 1996. 2

S. E. Mitchell, A. Burns, and A. J. Wellings. Developing a real-time metaobject protocol. *Object-Oriented Real-Time Dependable Systems, IEEE International Workshop on*, 0:323, 1997. doi: 10.1109/WORDS.1997.609974. 112, 132, 133

Benjamin Morandi, Sebastian S. Bauer, and Bertrand Meyer. Scoop - a contract-based concurrent object-oriented programming model. In Peter Müller, editor, *Advanced Lectures on Software Engineering, LASER Summer School 2007/2008*, volume 6029 of *LNCS*, pages 41–90. Springer, 2008. ISBN 978-3-642-13009-0. doi: 10.1007/978-3-642-13010-6_3. 59

Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, Ben Liblit, Michael M. Swift, and David A. Wood. Supporting nested transactional memory in logtm. *SIGPLAN Not.*, 41(11):359–370, October 2006. ISSN 0362-1340. doi: 10.1145/1168918.1168902. 30

Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! *SIGPLAN Not.*, 44(3):265–276, March 2009. ISSN 0362-1340. doi: 10.1145/1508284.1508275. 229, 230

Behrooz Nobakht, Frank S. de Boer, Mohammad Mahdi Jaghoori, and Rudolf Schlatte. Programming and deployment of active objects with application-level scheduling. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1883–1888, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-0857-1. doi: 10.1145/2231936.2232086. 170

Johan Nordlander, Mark P. Jones, Magnus Carlsson, Richard B. Kieburtz, and Andrew P. Black. Reactive objects. In *Symposium on Object-Oriented Real-Time Distributed Computing*, pages 155–158, 2002. 59

Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, August 1998. ISSN 1061-7264. doi: 10.1145/289918.289920. 30

Nathaniel Nystrom, Michael Clarkson, and Andrew Myers. Polyglot: An extensible compiler framework for java. In Görel Hedin, editor, *Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 138–152. Springer, Berlin / Heidelberg, 2003. ISBN 978-3-540-00904-7. doi: 10.1007/3-540-36579-6_11. 188

OpenMP Architecture Review Board. Openmp application program interface. version 3.1. Specification, 2011. URL `http://www.openmp.org/mp-documents/OpenMP3.1.pdf`. 22, 59

Johan Östlund, Tobias Wrigstad, Dave Clarke, and Beatrice Åkerblom. Ownership, uniqueness, and immutability. In Richard F. Paige, Bertrand Meyer, Wil Aalst, John Mylopoulos, Michael Rosemann, Michael J. Shaw, and Clemens Szyperski, editors, *Objects, Components, Models and Patterns*, volume 11 of *Lecture Notes in Business Information Processing*, pages 178–197. Springer, Berlin Heidelberg, 2008. ISBN 978-3-540-69824-1. doi: 10.1007/978-3-540-69824-1_11. 134

Andreas Paepcke. User-level language crafting: Introducing the clos metaobject protocol. In *Object-Oriented Programming*, chapter User-level language crafting: introducing the CLOS metaobject protocol, pages 65–99. MIT Press, Cambridge, MA, USA, 1993. ISBN 0-262-16136-2. 111

Joseph Pallas and David Ungar. Multiprocessor smalltalk: A case study of a multiprocessor-based programming environment. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 268–277, New York, NY, USA, 1988. ACM. ISBN 0-89791-269-1. doi: 10.1145/53990.54017. 100

Jonathan Parri, Daniel Shapiro, Miodrag Bolic, and Voicu Groza. Returning control to the programmer: Simd intrinsics for virtual machines. *Queue*, 9(2): 30:30–30:37, February 2011. ISSN 1542-7730. doi: 10.1145/1943176.1945954. 64

Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent haskell. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 295–308, New York, NY, USA, 1996. ACM. ISBN 0-89791-769-3. doi: 10.1145/237721.237794. 50

Ravi Rajwar and James R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 34, pages 294–305, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1369-7. 58

Thierry Renaux, Lode Hoste, Stefan Marr, and Wolfgang De Meuter. Parallel gesture recognition with soft real-time guarantees. In *Proceedings of the 2nd edition on Programming Systems, Languages and Applications based on Actors, Agents, and Decentralized Control Abstractions*, SPLASH '12 Workshops, pages 35–46, October 2012. ISBN 978-1-4503-1630-9. doi: 10.1145/2414639.2414646. 10, 171

Lukas Renggli and Oscar Nierstrasz. Transactional memory for smalltalk. In *ICDL '07: Proceedings of the 2007 international conference on Dynamic languages*, pages 207–221. ACM, 2007. ISBN 978-1-60558-084-5. doi: 10.1145/1352678.1352692. 100, 148, 149, 151, 164, 182, 185, 186, 188

John H. Reppy, Claudio V. Russo, and Yingqi Xiao. Parallel concurrent ml. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 257–268, New York, NY, USA, August 2009. ACM. ISBN 978-1-60558-332-7. doi: 10.1145/1631687.1596588. 59

Jorge Ressia, Lukas Renggli, Tudor Gîrba, and Oscar Nierstrasz. Run-time evolution through explicit meta-objects. In *Proceedings of the 5th Workshop on Models@run.time at the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MODELS 2010)*, pages 37–48, Oslo, Norway, October 2010. 111, 131

Luis Humberto Rodriguez Jr. Coarse-grained parallelism using metaobject protocols. Master thesis, MIT, September 1991. 111

Raúl Rojas. Konrad zuse's legacy: The architecture of the z1 and z3. *Annals of the History of Computing, IEEE*, 19(2):5 –16, April-June 1997. ISSN 1058-6180. doi: 10.1109/85.586067. 2

John R. Rose. Bytecodes meet combinators: Invokedynamic on the jvm. In *VMIL '09: Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages*, pages 1–11. ACM, 2009. ISBN 978-1-60558-874-2. doi: 10.1145/1711506.1711508. 3, 15, 42

David Röthlisberger, Marcus Denker, and Éric Tanter. Unanticipated partial behavioral reflection: Adapting applications at runtime. *Computer Languages, Systems & Structures*, 34(2–3):46 – 65, 2008. ISSN 1477-8424. doi: 10.1016/j.cl.2007.05.001. Best Papers 2006 International Smalltalk Conference. 188

Israel J. Mojica Ruiz, Meiyappan Nagappan, Bram Adams, and Ahmed E. Hassan. Understanding reuse in the android market. In *Proceedings of the 20th IEEE International Conference on Program Comprehension (ICPC)*, pages 113–122, Passau, Germany, June 2012. doi: 10.1109/ICPC.2012.6240477. 14

Vijay Saraswat, George Almasi, Ganesh Bikshandi, Calin Cascaval, David Cunningham, David Grove, Sreedhar Kodali, Igor Peshansky, and Olivier

Tardieu. The asynchronous partitioned global address space model. Technical report, Toronto, Canada, June 2010. 75

Vijay Saraswat, Bard Bloom, Igor Peshansky, Olivier Tardieu, and David Grove. X10 language specification. Technical report, IBM, January 2012. URL http://x10.sourceforge.net/documentation/languagespec/x10-222.pdf. 33, 75

Jan Schäfer and Arnd Poetzsch-Heffter. Jcobox: Generalizing active objects to concurrent components. In Theo D'Hondt, editor, *Proc. of ECOOP'10*, volume 6183 of *LNCS*, pages 275–299. Springer, 2010. ISBN 978-3-642-14106-5. doi: 10.1007/978-3-642-14107-2_13. 59, 73, 153, 154, 203

Michel Schinz and Martin Odersky. Tail call elimination on the java virtual machine. In *Proc. ACM SIGPLAN BABEL'01 Workshop on Multi-Language Infrastructure and Interoperability.*, volume 59 of *Electronic Notes in Theoretical Computer Science*, pages 158 – 171, 2001. doi: 10.1016/S1571-0661(05)80459-1. BABEL'01, First International Workshop on Multi-Language Infrastructure and Interoperability (Satellite Event of PLI 2001). 16

Hans Schippers, Tom Van Cutsem, Stefan Marr, Michael Haupt, and Robert Hirschfeld. Towards an actor-based concurrent machine model. In *Proceedings of the Fourth Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, pages 4–9, New York, NY, USA, July 2009. ACM. ISBN 978-1-60558-541-3. doi: 10.1145/1565824.1565825. 10

Douglas C. Schmidt. Pattern-oriented software architecture volume 2: Patterns for concurrent and networked objects, 2000. 170

Christophe Scholliers, Éric Tanter, and Wolfgang De Meuter. Parallel actor monitors. In *14th Brazilian Symposium on Programming Languages*, 2010. 59, 171

Arnold Schwaighofer. Tail call optimization in the java hotspot(tm) vm. Master thesis, Johannes Kepler Universität Linz, March 2009. URL http://www.ssw.uni-linz.ac.at/Research/Papers/Schwaighofer09Master/schwaighofer09master.pdf. 16

Olivier Serres, Ahmad Anbar, Saumil G. Merchant, Abdullah Kayi, and Tarek A. El-Ghazawi. Address translation optimization for unified parallel c multi-dimensional arrays. In *Parallel and Distributed Processing Workshops*

and Phd Forum (IPDPSW), 2011 IEEE International Symposium on, pages 1191–1198, May 2011. doi: 10.1109/IPDPS.2011.279. 62

Amin Shali and William R. Cook. Hybrid partial evaluation. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 375–390, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0940-0. doi: 10.1145/2048066.2048098. 64

Nir Shavit. Data structures in the multicore age. *Commun. ACM*, 54(3):76–84, March 2011. ISSN 0001-0782. doi: 10.1145/1897852.1897873. 20

Nir Shavit and Dan Touitou. Software transactional memory. In *Proc. of PODC'95*. ACM, 1995a. ISBN 0-89791-710-3. doi: 10.1145/224964.224987. 29, 59

Nir Shavit and Dan Touitou. Elimination trees and the construction of pools and stacks: Preliminary version. In *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, SPAA '95, pages 54–63, New York, NY, USA, 1995b. ACM. ISBN 0-89791-717-0. doi: 10.1145/215399.215419. 19

Jun Shirako, David M. Peixotto, Vivek Sarkar, and William N. Scherer. Phasers: A unified deadlock-free construct for collective and point-to-point synchronization. In *ICS'08: Proceedings of the 22nd annual international conference on Supercomputing*, page 277—288, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-158-3. doi: 10.1145/1375527.1375568. 22, 27, 62

Abraham Silberschatz, Greg Gagne, and Peter Baer Galvin. *Operating System Concepts*. Wiley, 6 edition, 2002. ISBN 0471250600. 230, 231

David B. Skillicorn and Domenico Talia. Models and languages for parallel computation. *ACM CSUR*, 30:123–169, June 1998. ISSN 0360-0300. doi: 10.1145/280277.280278. 4, 58, 59

Brian Cantwell Smith. Reflection and semantics in lisp. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '84, pages 23–35, New York, NY, USA, 1984. ACM. ISBN 0-89791-125-3. doi: 10.1145/800017.800513. 174

James E. Smith and Ravi Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005. ISBN 1558609105. 14

Matthew J. Sottile, Timothy G. Mattson, and Craig E. Rasmussen. *Introduction to Concurrency in Programming Languages*. Chapman & Hall/CRC Press, Boca Raton, 2010. ISBN 9781420072136. 19

Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-typed actors for java. In *ECOOP 2008 – Object-Oriented Programming*, pages 104–128, 2008. doi: 10.1007/978-3-540-70592-5_6. 73

Robert J. Stroud and Z. Wu. Using metaobject protocols to implement atomic data types. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP '95, pages 168–189, London, UK, UK, 1995. Springer-Verlag. ISBN 3-540-60160-0. 130

Venkat Subramaniam. *Programming Concurrency on the JVM: Mastering Synchronization, STM, and Actors*. Pragmatic Bookshelf, Dallas, Tex., 2011. ISBN 9781934356760 193435676X. 18

Sun Microsystems, Inc. Multithreaded programming guide. Technical report, 4150 Network Circle, Santa Clara, CA 95054 U.S.A., September 2008. URL http://docs.oracle.com/cd/E19253-01/816-5137/816-5137.pdf. 19

Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3):202–210, 2005. 36

Éric Tanter. Reflection and open implementations. Technical report, DCC, University of Chile, Avenida Blanco Encalada 2120, Santiago, Chile, 2009. URL http://www.dcc.uchile.cl/TR/2009/TR_DCC-20091123-013.pdf. 80, 111, 112, 130

Christian Thalinger and John Rose. Optimizing invokedynamic. In *Proc. of PPPJ'10*, pages 1–9. ACM, 2010. ISBN 978-1-4503-0269-2. doi: 10.1145/1852761.1852763. 3, 15, 42, 64, 187, 233, 245

William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 179–196, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-43369-4. 35, 59, 171

David A. Thomas, Wilf R. LaLonde, John Duimovich, Michael Wilson, Jeff McAffer, and Brian Barry. Actra - a multitasking/multiprocessing smalltalk. In *OOPSLA/ECOOP '88: Proceedings of the 1988 ACM SIGPLAN workshop on Object-based concurrent programming*, pages 87–90, New York, NY, USA, 1988. ACM. ISBN 0-89791-304-3. doi: 10.1145/67386.67409. 100

David Ungar. Everything you know (about parallel programming) is wrong!: A wild screed about the future, October 2011. URL http://www.dynamic-languages-symposium.org/dls-11/program/media/Ungar_2011_EverythingYouKnowAboutParallelProgrammingIsWrongAWildScreedAboutTheFuture_Dls.pdf. 20

David Ungar and Sam S. Adams. Hosting an object heap on manycore hardware: An exploration. In *DLS '09: Proceedings of the 5th symposium on Dynamic languages*, pages 99–110, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-769-1. doi: 10.1145/1640134.1640149. 11, 100, 101, 104, 106, 107

David Ungar and Sam S. Adams. Harnessing emergence for manycore programming: Early experience integrating ensembles, adverbs, and object-based inheritance. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '10, pages 19–26, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0240-1. doi: 10.1145/1869542.1869546. 59, 104

UPC Consortium. Upc language specifications, v1.2. Tech Report LBNL-59208, Lawrence Berkeley National Lab, 2005. URL http://www.gwu.edu/~upc/publications/LBNL-59208.pdf. 30, 59

András Vajda and Per Stenstrom. Semantic information based speculative parallel execution. In Wei Liu, Scott Mahlke, and Tin fook Ngai, editors, *Pespma 2010 - Workshop on Parallel Execution of Sequential Programs on Multicore Architecture*, Saint Malo, France, 2010. 171

Jorge Vallejos. *Modularising Context Dependency and Group Behaviour in Ambient-oriented Programming*. Phd thesis, Vrije Universiteit Brussel, July 2011. 112

Tom Van Cutsem. *Ambient References: Object Designation in Mobile Ad Hoc Networks*. PhD thesis, Vrije Universiteit Brussel, Faculty of Sciences, Programming Technology Lab, Pleinlaan 2, 1050 Elsene, Beligum, May 2008. 31

Tom Van Cutsem and Mark S. Miller. Proxies: Design principles for robust object-oriented intercession apis. In *Proc. of DLS'10*, pages 59–72. ACM, October 2010. doi: 10.1145/1899661.1869638. 111, 132, 154

Tom Van Cutsem, Stijn Mostinckx, Elisa Gonzalez Boix, Jessie Dedecker, and Wolfgang De Meuter. Ambienttalk: Object-oriented event-driven program-

ming in mobile ad hoc networks. In *Proc. of SCCC'07*, pages 3–12. IEEE CS, 2007. doi: 10.1109/SCCC.2007.4. 31, 59, 73, 153, 171, 203

Tom Van Cutsem, Stefan Marr, and Wolfgang De Meuter. A language-oriented approach to teaching concurrency. Presentation at the workshop on curricula for concurrency and parallelism, SPLASH'10, Reno, Nevada, USA, 2010. URL http://soft.vub.ac.be/Publications/2010/vub-tr-soft-10-12.pdf. 10

Meine J.P. van der Meulen and Miguel A. Revilla. Correlations between internal software metrics and software dependability in a large population of small c/c++ programs. *IEEE 18th International Symposium on Software Reliability Engineering (ISSRE'07)*, pages 203–208, November 2007. ISSN 1071-9458. doi: 10.1109/ISSRE.2007.12. 140, 162

Nalini Vasudevan, Olivier Tardieu, Julian Dolby, and Stephen Edwards. Compile-time analysis and specialization of clocks in concurrent programs. volume 5501 of *Lecture Notes in Computer Science*, pages 48–62. Springer, March 2009. doi: 10.1007/978-3-642-00722-4_5. 62

Toon Verwaest, Camillo Bruni, Mircea Lungu, and Oscar Nierstrasz. Flexible object layouts: Enabling lightweight language extensions by intercepting slot access. In *Proc. of OOPSLA'11*, pages 959–972, 2011. ISBN 978-1-4503-0940-0. doi: 10.1145/2048066.2048138. 131

Jan Vitek and Tomas Kalibera. Repeatability, reproducibility, and rigor in systems research. In *Proceedings of the ninth ACM international conference on Embedded software*, EMSOFT '11, pages 33–38, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0714-7. doi: 10.1145/2038642.2038650. 229

Takuo Watanabe and Akinori Yonezawa. An actor-based metalevel architecture for group-wide reflection. In J. de Bakker, W. de Roever, and G. Rozenberg, editors, *Foundations of Object-Oriented Languages*, volume 489 of *Lecture Notes in Computer Science*, pages 405–425. Springer, Berlin / Heidelberg, 1991. ISBN 978-3-540-53931-5. doi: 10.1007/BFb0019450. 132

Peter H. Welch and Frederick R. M. Barnes. Communicating mobile processes: Introducing occam-pi. In Ali Abdallah, Cliff Jones, and Jeff Sanders, editors, *Communicating Sequential Processes. The First 25 Years*, volume 3525 of *Lecture Notes in Computer Science*, pages 712–713. Springer Berlin / Heidelberg, April 2005. ISBN 978-3-540-25813-1. doi: 10.1007/11423348_10. 32, 59

Peter H. Welch, Neil Brown, James Moores, Kevin Chalmers, and Bernhard H. C. Sputh. Integrating and extending jcsp. In Alistair A. McEwan, Steve A. Schneider, Wilson Ifill, and Peter H. Welch, editors, *The 30th Communicating Process Architectures Conference, CPA 2007*, volume 65 of *Concurrent Systems Engineering Series*, pages 349–370. IOS Press, 2007. ISBN 978-1-58603-767-3. 32, 59, 75

David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III, and Anant Agarwal. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27(5):15–31, 2007. ISSN 0272-1732. doi: 10.1109/MM.2007.89. 101, 104

Thomas Würthinger. Extending the graal compiler to optimize libraries. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, SPLASH '11, pages 41–42, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0942-4. doi: 10.1145/2048147.2048168. 64, 243

Thomas Würthinger, Christian Wimmer, and Hanspeter Mössenböck. Array bounds check elimination in the context of deoptimization. *Science of Computer Programming*, 74(5–6):279 – 295, 2009. ISSN 0167-6423. doi: 10.1016/j.scico.2009.01.002. Special Issue on Principles and Practices of Programming in Java (PPPJ 2007). 244

Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-optimizing ast interpreters. In *Proceedings of the 8th Dynamic Languages Symposium*, DLS'12, pages 73–82, October 2012. ISBN 978-1-4503-1564-7. doi: 10.1145/2384577.2384587. 244

Yasuhiko Yokote. *The Design and Implementation of ConcurrentSmalltalk*. PhD thesis, Keio University Japan, Singapore; Teaneck, N.J., 1990. 59, 100

Weiming Zhao and Zhenlin Wang. Scaleupc: A upc compiler for multi-core systems. In *Proceedings of the Third Conference on Partitioned Global Address Space Programing Models*, PGAS '09, pages 11:1–11:8, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-836-0. doi: 10.1145/1809961.1809976. 62

Jin Zhou and Brian Demsky. Memory management for many-core processors with software configurable locality policies. In *Proceedings of the 2012 international symposium on Memory Management*, ISMM '12, pages

3–14, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1350-6. doi: 10.1145/2258996.2259000. 62

Lukasz Ziarek, Adam Welc, Ali-Reza Adl-Tabatabai, Vijay Menon, Tatiana Shpeisman, and Suresh Jagannathan. A uniform transactional execution environment for java. In *Proc. of ECOOP'08*, pages 129–154, 2008. doi: 10.1007/978-3-540-70592-5_7. 188

Yoav Zibin, Alex Potanin, Paley Li, Mahmood Ali, and Michael D. Ernst. Ownership and immutability in generic java. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 598–617, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0203-6. doi: 10.1145/1869459.1869509. 134