SIAAM: Simple Isolation for an Abstract Actor Machine

Quentin Sabah

▶ To cite this version:

Quentin Sabah. SIAAM: Simple Isolation for an Abstract Actor Machine. Distributed, Parallel, and Cluster Computing [cs.DC]. Université de Grenoble, 2013. English. NNT : . tel-00933072

HAL Id: tel-00933072
https://theses.hal.science/tel-00933072

Submitted on 19 Jan 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministérial :

Présentée par

Quentin Sabah

Thèse dirigée par **Jean-Bernard Stefani**

préparée au sein **INRIA**
et de **EDMSTII**

Siaam: Simple Isolation for an Abstract Actor Machine

Thèse soutenue publiquement le **4 Décembre 2013**,
devant le jury composé de :

Monsieur, Thomas Jensen
Directeur de Recherche, INRIA Rennes, Rapporteur
Monsieur, Gaël Thomas
Maître de Conférences, LIP6, Rapporteur
Monsieur, Stéphane Ducasse
Directeur de Recherche, INRIA Lille, Examinateur
Monsieur, Germain Haugou
Ingénieur R&D, STMicroelectronics, Examinateur
Monsieur, Jean-Bernard Stefani
Directeur de Recherche, INRIA Grenoble, Directeur de thèse

Remerciements

Je remercie les membre du jury, Stéphane Ducasse, Germain Haugou, Thomas Jensen et Gaël Thomas, qui m'ont fait l'honneur d'évaluer les travaux rapportés dans le présent document. Cette thèse n'aurait pas vu le jour sans l'implication de Philippe Guillaume qui a impulsé le financement CIFRE auprès de STMicroelectronics. Enfin je suis reconnaissant à Jean-Bernard Stefani de m'avoir accueilli depuis 2009 et durant près de cinq ans au sein des équipes Sardes puis Spades à (l')INRIA Rhône-Alpes.

Merci à mes nombreux collègues à ST et à l'INRIA grâce auxquels l'aventure scientifique et technique a aussi pris une tournure bon enfant, chaleureuse et humaine.

Enfin mes relations amicales de plus longue date ont compté chaque jour et il en sera ainsi encore longtemps.

Contents

Remerciements . . . 3

1 Introduction **9**
1.1 Informal overview . . . 10
1.1.1 The actor model . . . 10
1.1.2 Siaam's actor model . . . 11
1.2 Contributions . . . 17
1.2.1 Contributions to the Open-Source communities . . . 17
1.3 Organization of this document . . . 17

2 State of the Art **19**
2.1 Isolation in scalable concurrent systems . . . 19
2.1.1 Ownership-based isolation . . . 20
2.1.2 Software-based processes. . . . 27
2.1.3 Classloader isolation. . . . 28
2.2 Static analyses for efficient concurrency . . . 30
2.3 Motivations . . . 32

3 Formal specification of Siaam **35**
3.1 Introduction. . . . 35
3.2 HOL Language . . . 36
3.3 Abstract Syntax. . . . 38
3.3.1 Java's features not offered by Jinja . . . 42
3.4 Siaam Global Semantics . . . 42
3.4.1 Siaam Native API. . . . 43
3.4.2 Global State. . . . 43
3.4.3 Global Semantics. . . . 44
3.4.4 Transitive ownership verification and transfer. . . . 45
3.4.5 Actor actions. . . . 48
3.4.6 Native Calls Semantics. . . . 51
3.5 Single-Actor small-steps semantics . . . 52
3.5.1 Single-Actor state. . . . 53
3.5.2 Jinja's source language semantics. . . . 53
3.5.3 Single-Actor semantics . . . 55
3.5.4 Actors interleaving. . . . 57
3.6 The Siaam Virtual Machine . . . 57
3.6.1 Single-Actor Machine Model. . . . 57
3.6.2 Operational Semantics. . . . 58
3.6.3 Instructions execution . . . 59

3.6.4 Virtual Machine Interleaving. 63
3.7 SIAAM isolation and formal proof 64
3.7.1 Abstract SIAAM: Types 64
3.7.2 Abstract SIAAM : Transition rules 66
3.7.3 Isolation property 71

4 Ownership-based isolation for the JVM 77
4.1 Introduction 77
4.1.1 JikesRVM 77
4.1.2 Outline 78
4.2 Virtual Machine's core 78
4.2.1 Static contexts 79
4.2.2 Object's owners. 80
4.2.3 Thread's owner-ID. 82
4.2.4 Opaque objects. 82
4.2.5 Ownership transfer 83
4.3 Ownership-based isolation 92
4.3.1 Owner-checking barriers. 92
4.3.2 Constructors and instance fields. 93
4.3.3 Native methods. 94
4.3.4 Threads 94
4.3.5 Object finalizers 94
4.4 Support for immutability and static variables 95
4.4.1 Final instance fields. 95
4.4.2 Immutable objects 96
4.4.3 Arbitrary frozen objects 98
4.4.4 Static fields and Enum types 99
4.4.5 Conclusion 100
4.5 Trusted programming models 101
4.5.1 Siaam Actors API. 101
4.5.2 ActorFoundry. 104
4.5.3 M:N cooperatively scheduled actors 105

5 Static analysis and its usages 107
5.1 Introduction 107
5.1.1 Outline 108
5.2 Intermediate Representation 109
5.2.1 Program representation. 109
5.2.2 IR language. 109
5.2.3 Body's control-flow graph. 109
5.3 Standard analyses framework. 110
5.3.1 Program's call graph. 111
5.3.2 Points-To Analysis. 112
5.3.3 Must-Alias Analysis. 113
5.3.4 Liveness Analysis. 114
5.4 The Siaam analysis framework. 115
5.4.1 Ownership-oriented programming models semantics. 115
5.4.2 Programming models API abstraction. 117

5.5 Some programming models and their ownership-based semantics. 117
5.5.1 Implicit owner-checks 118
5.5.2 Siaam actor programming model. 118
5.5.3 Reflection-based message dispatch APIs 120
5.6 The Siaam Analysis 120
5.6.1 Most conservative analysis. 123
5.6.2 Intra-procedural *SafeEff* side-effects propagation 124
5.6.3 Using the call graph information. 127
5.6.4 Using accurate points-to information. 130
5.6.5 Inter-procedural Transferred Abstract Objets analysis 132
5.6.6 Intra-procedural *UnsafeEff* side-effects propagation 134
5.6.7 Propagation of the safe abstract objects towards the callees. 135
5.6.8 Escapement analysis for *TAO* 136
5.7 Extension to frozen objects. 140
5.7.1 The *Read* ownership state 140
5.7.2 A programming model with frozen arrays 141
5.7.3 A programming model with frozen scalars 142
5.8 Usages of the Siaam analysis results 142
5.8.1 Owner-check elimination 142
5.8.2 Programming assistant 143
5.9 Implementation 145

6 Evaluation **147**
6.1 Overall performances 147
6.1.1 Dacapo Benchmark 147
6.2 Micro benchmarks 149
6.2.1 Zero-Copy vs. Deep-Copy. 150
6.2.2 Graph-traversal optimizations 151
6.2.3 Owner Check 152
6.2.4 Memory consumption 152
6.3 Owner-checking elimination 153
6.3.1 Actory Foundry Benchmarks 153

7 Conclusion **155**
7.1 Synthesis 155
7.2 Future Work 156
7.2.1 Just-in-time whole-program analysis. 156
7.2.2 Cohabitation with ownership types annotations. 156
7.2.3 Modular actor analysis. 157
7.2.4 Demand-driven Siaam analysis 157

A Jinja **159**

B Siaam **163**

C Abstracts **167**
C.1 Résumé en français 167
C.2 English abstract 167

Chapter 1

Introduction

It is now reasonable to state that scalability of concurrent systems is much more achievable, both theoretically and technically, in the absence of shared state. First it allows one to reason about the sequential behavior of each node independently of the others and the global architecture. Second it guarantees race-free execution, thus avoiding need for synchronization. In these systems, the state isolation property must be enforced by the inter-node communication mechanism so that no piece of data may be simultaneously accessed by more than one node, which would otherwise result in inconsistent observations and updates of the same data by several nodes.

Message-passing has been widely adopted as the main communication scheme in concurrent systems. The exchanged data are copied from one node's memory region to another's. Since messages are replicated in the receiver's private state, each node of the system reads and updates its own copy of the data, thus isolation is trivially preserved. Ideally, copy operations are transparent whether memories are physically disjoint or not. To obtain the best performance, it is preferable to pass a message in a zero-copy fashion when the communication takes place in the same shared memory. However, in order to preserve the isolation property and avoid data race with the receiver, the sender node must never access the content of that message again.

In this thesis we study isolation and message-passing in the context of concurrent object-oriented programming. Messages are aggregates of objects living in a shared heap and passed by reference for the hereinbefore reasons. Object-oriented languages allows arbitrary objects aliasing, a beneficial feature for developers, raising serious difficulties to avoid data races when mixed with reference passing. The challenge accepted by all the previous works on that aspect consists in making sure a node cannot find or use any access path to the aggregate of objects after passing it to another node. Symmetrically when a node receives a message it must be the sole allowed accessor so that no other node may spy or alter the aggregate concurrently.

Several variations of *ownership* and *reference uniqueness* have been extensively employed in the past to control aliasing in object-oriented programming languages, and more recently to address concurrency. Strikingly the vast majority rely on a set of statically checkable typing rules, either requiring an annotation overhead or introducing object aliasing restrictions. Our contribution with Siaam is the demonstration of a *purely run-time*, *actor-based*, *annotation-free*, *aliasing-proof* approach to concurrent state isolation allowing reference passing of arbitrary objects graphs. Even though we claim its extreme simplicity, Siaam is not a paradigm shifter, its *modus operandi* may also be understood in traditional terms of ownership and uniqueness, but in a rather dynamic way. Unlike statically-checked approaches that

prevent isolation violation mostly or completely ahead of time in a conservative way, Siaam thwarts violation attempts just-in-time as the nodes access or communicate objects. By being optimistic and simple, Siaam trades annotation burden and aliasing constraints with runtime overhead. Nevertheless, we hope this tradeoff will facilitate a progressive adoption of all the available forms of isolation.

Our dynamic approach can cooperate with traditional static aliasing control for mutual benefits. Regarding runtime performance, Siaam would benefit from certain aliasing invariants enforced by static annotations. On the other hand, annotated portions of programs would be able to safely interoperate with unannotated or untrusted portions by letting Siaam dynamically enforce or verify some expected invariants at the sensible interfaces.

Furthermore, since our formal model of isolation accommodates an abstract concurrent object-oriented machine, it is possible to extend available virtual machines such as a JVM (for Java), a CLR (for .NET) or the VMKit[47] with only minimal modifications to obtain the isolation guarantees offered by Siaam. We also see potential applications to sandboxing, allowing for instance to run critical and untrusted programs in a unique Android virtual machine. Finally, several languages and frameworks have recently emerged and provides concurrency exclusively through message passing; some provide race-freedom (Dart) and some other allow zero-copy (Go, Akka), but none is able to guarantee both. The ideas developed in Siaam fits well in these projects and could be used to transparently enforce the missing property.

1.1 Informal overview

Siaam is the acronym for *simple isolation for an actor-based abstract machine*, to emphasis the simplicity of our approach. Siaam encompass three components. First it is a formal specification of an actor-based concurrent programming model for an abstract object-oriented system and a definition of the isolation property that comes with it. Second, Siaam is an implementation of this specification for the Java language in the Jikes Research Virtual Machine. Third, it is a static analysis capturing the dynamic nature of Siaam with two objectives: limit the runtime overhead of the modified virtual machine and assist the application developers by pinpointing and explaining potential violations of the isolation property.

1.1.1 The actor model

The actor model [56, 38, 2] is a model of concurrent programming that emphasize on execution efficiency and expressiveness. Both objectives relates to scalability, the former in term of performance and the latter in term of program understanding and behavior correctness. Unlike an operating system where concurrent processes are very independent and carries out computation without interacting with each others, an actor system is meant to allow large number of actors to cooperate.

In the vocabulary used by Gul A. Agha[2], actors sends *communications* to each others through a buffered *mail system*. Actors are sequential units of computation with a dynamic *behavior* which for each incoming communication may undertake the following actions *(i)* send communications to other actors, *(ii)* create new actors, *(iii)* compute a replacement behavior to respond to the next communications received. A behavior may be *history sensitive* meaning it contains stateful information that will influence the next response to an incoming communication. The information an actor maintains and updates through consecutive behaviors is called its *local state*. Communications have asynchronous message-passing semantics,

in [2] the mail system have a guarantee of delivery so that all sent messages are eventually received, but no particular reception ordering is enforced. Other systems may chose different semantics.

The actor model banishes any form of mutable shared state, data communicated through message passing is usually copied from the sender to the receiver. This is the strategy adopted by the Erlang[7] language. As a result, the local state of an actor cannot be read nor updated by another actor, we may call that the *strong isolation* property.

1.1.2 Siaam's actor model

In this thesis, we integrate the actor-based concurrency into an object-oriented environment with a single shared heap. The provided message passing communications have a zero-copy semantics, meaning that messages are not copied, only their references are communicated. As a result, after a communication, both the sender and the receiver possess references to the objects contained in the message. In order to make sure that actors cannot read or modify the same objects concurrently, each actor is attributed a unique colour and may only access objects with a matching color. An object is initially coloured with the colour of the actor that created it. Then, objects are recoloured through communications so that they match the receiver's colour.

Ownership. In Siaam's terminology, we say an object *belongs* to an actor if the former matches the unique colour attributed to the actor. Conversely the actor *owns* every object with a matching colour. Objects with a common owner form an *ownership domain* or *ownership context*. Ownership domains are never nested, therefore Siaam's ownership topology is very flat. The action of recoloring an object is called *ownership transfer*, and the verification that an object colour matches an actor's unique colour is called *owner checking*. There is absolutely no restriction on the references between objects, in particular objects with different owners may reference each-others. An actor may read the field of an object with a matching colour and retrieve the reference to another object that it may or may not own. But accessing a field of the retrieved object is forbidden, it raises a runtime exception.

We illustrate these notions in Figure 1.1. On the left side (a) is a configuration of the heap and the ownership relation where each actor presented in gray owns the objects that are part of the same dotted convex hull representing the ownership domain. Directed edges are heap references. On the right side (b), the objects $1, 2, 3$ have been transferred from the ownership domain of a to b's context, and object 1 has been attached to the data structure maintained in b's local state. The reference from a to 1 has been preserved, but actor a is not allowed to access the fields of $1, 2, 3$: owner-checking one of these objects against a would fail.

Strong isolation property. Although many works make reference to the concept of memory isolation, very few have actually given a clear and formal definition of it. We now express how we consider *strong isolation* in Siaam.

In Siaam the only means of information transfer between actors is message exchange. A strongly isolated actor that makes all kind of actions but message receptions cannot observe unexpected updates of the objects' fields it is allowed to reach and access. By unexpected we mean field updates that are not immediate side-effects of the considered actor.

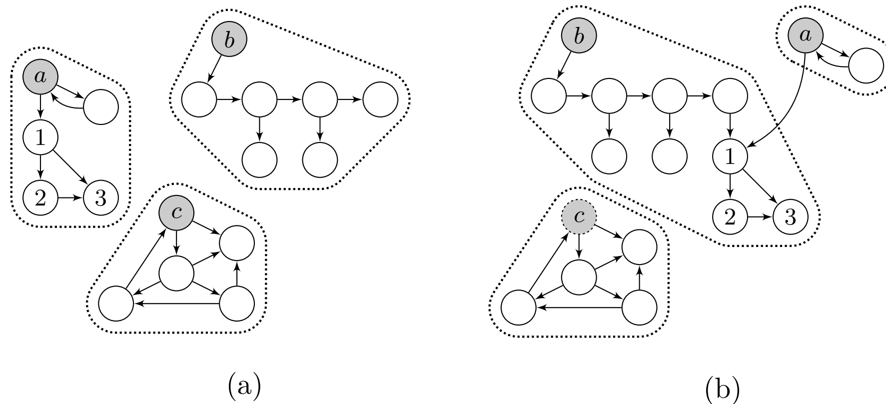


Figure 1.1: (a) Three ownership domains with their respective actor in gray. (b) The configuration after the ownership of objects $1, 2, 3$ was transferred from actor a to actor b.

Mail system. Each actor may have zero, one, or several *mailboxes* from which it can retrieve messages at will. Mailboxes are created dynamically and may be communicated without restriction. Any actor of the system may send messages through a mailbox. However each mailbox is associated with a receiver actor, such that only the receiver may retrieve messages from a mailbox. More detailed information on the mailboxes are deferred to the *mailboxes* paragraph.

Actors. The local state of each actor is represented by an object, and the associated behaviour is a method of that object. The behaviour method is free to implement any algorithm, the actor terminates when that method returns. Here we clearly deviate from the definition of an actor given in 1.1.1, where actors "react" to received communications. Siaam's actor are more active in the sense that they can arbitrarily chose when to receive a message, and from what mailbox.
Although it is possible to replicate Agha's actor model with the Siaam actor model (and conversely) by simply fixing a unique mailbox for each Siaam's actor with an infinite loop behaviour processing incoming messages one by one.

Ownership transfer. The ownership transfer procedure, previously described as a colouring operation, happens in two situations; first when a message is communicated from one actor to another, and second when a new actor is started. A message is a graph of objects where all the objects have the same colour — or owner. An actor is not allowed to send a message containing objects it does not own. Moreover an actor is not allowed to send itself. The content of a message is defined as the set of objects in the transitive closure of a *starting reference* which may be any arbitrary object. Given the reference to an object, a message from that starting reference comprises all the objects reachable by traversing the heap through objects' fields.

Figures 1.2 to 1.4 features some examples of valid and invalid message starting objects. In configuration (a) all the objects but the actor 0 may be employed as the starting object for a valid message. The transitive closure of 1 contains the objects $\{1, \dots, 6\}$. Objects 2 to 5 have the same closure made of $\{2, \dots, 6\}$. And sending object 6 would only transfer the ownership of that object.

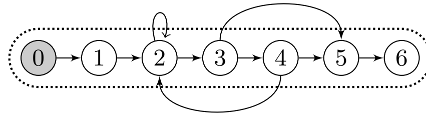


Figure 1.2: (a) Valid message starting objects are $1, 2, 3, 4, 5, 6$.

In configuration (b) no object may be used as a starting reference for a valid message since the actor 0 is transitively reachable from every object.

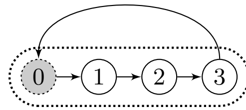


Figure 1.3: (b) No valid message starting objects.

In configuration (c) object 5 reaches the *foreign* object 3 which is not in the same ownership domain as 5. Therefore actor 4 cannot transfer the ownership of the message starting from 5. All the others non-actor objects are valid message starting objects.

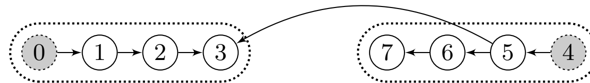


Figure 1.4: (c) Valid message starting objects are $1, 2, 3, 6, 7$.

Message passing. To pass a message, an actor simply enqueues a valid message's starting reference into a mailbox. The enqueue operation has the side-effect of erasing the ownership of every object that is part of the message. This erasing can be seen as colouring all the objects in black and ensuring no actor is ever assigned the black colour. This way objects in pending messages are protected against any attempt to read or update their fields, from any actor. Note that the message passing procedure preserves all the references between objects, whether they are inside or outside of the message. However, since a valid message is by definition the transitive closure of its starting reference, there can be no reference from the inside to the outside of a message. In the colour metaphor, black objects only refer to other black objects of the same message.
Figure 1.5 features examples of allowed and impossible references. Objects $1, 2, 3, 4$ are "black coloured", they are outside any actor's domain, we represent this with the black border surrounding these nodes. Let 1 and 3 be the starting objects of two different messages enqueued in potentially different mailboxes. We represented some impossible heap edges with crossed gray arrows. Object 3 cannot hold a reference to 1 since they are not part of the same message, indeed the automatic message validation process does not allow references between different messages. Object 2 cannot refer to actor b nor any object outside of its own message. The message passing mechanisms of Siaam simply prevents these situations from happening by applying the message validation rules.

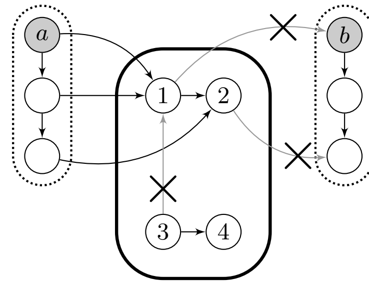
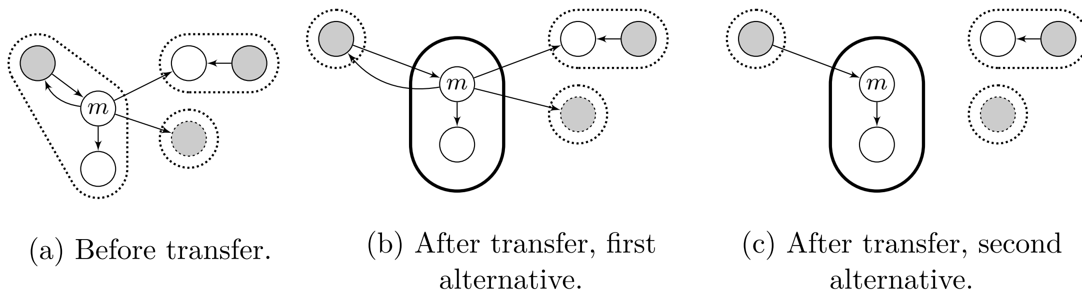


Figure 1.5: Allowed and impossible heap edges. Graphs $\{1, 2\}$ and $\{3, 4\}$ are two different messages.



(a) Before transfer.

(b) After transfer, first alternative.

(c) After transfer, second alternative.

Figure 1.6: Illustration of the ownership transfer with two alternative definitions of "a valid message".

When a message is dequeued from a mailbox, its entire graph of objects is recoloured with the receiver's colour, so that the objects that are part of the message are integrated into the actor's ownership domain. It is worth noticing that both the emission and the reception are explicit operations undertaken by the actors. No object may enter nor exit an actor's ownership domain without the direct involvement of that actor.

We draw the reader's attention to the fact that the definition of a valid message is an arbitrary choice which could be exchanged. An alternative would, for instance, transfers the ownership of the transitive closure of objects belonging to the sender actor and simply cut the graph exploration when reaching a foreign object or even the sender. This way objects inside a message would possibly refer to objects outside the message.
This alternative is illustrated in figure 1.6. Object m is a valid message starting in configuration (a), as a matter of fact any non-actor object would be valid. In configuration (b), m has been used as a message starting object and enqueued in a mailbox. As a side-effect all the non-actor objects reachable from m without going through a foreign object have been transferred out of their initial ownership domain.
Another alternative would, in addition, nullify the references from objects inside the message to objects outside of the message, as illustrated in (c).

Mailboxes. We now introduce the mailboxes in the big picture. Conceptually, a mailbox is a name associated with a message queue and a receiver actor. In our object-oriented environment, that name is the reference to a mailbox object, but the message queue and the receiver information associated with it are part of the mail system, outside the object representation.

In other words, there is no heap edge going out of a mailbox object. Furthermore, mailbox objects are outside every ownership domain.

An example of mail system is illustrated in Figure 1.7, it is a snapshot of a producer-consumers system took at an arbitrary moment. We represent mailbox objects with double circles nodes. In the example, p is a producer and c_i actors are consumers. Actor p receives orders through the mailbox M from which it can retrieve messages. The producer maintains a linked list of consumers's mailboxes, to which it distributes data in a round-robin fashion. The mail system, described in the right part of the figure, contains 3-tuples $\langle M, q, a\rangle$ of a mailbox object reference M, a message queue q and an actor object reference a — the receiver. M's queue contains the two messages starting respectively with objets 1 and 2, M's receiver is set to p. Each consumer has a mailbox from which it can receive data to process, and also holds a reference to the producer's mailbox in order to issue special requests. A special request is a message containing a reference to the mailbox of the consumer; messages 1 and 2 are special requests issued respectively by c_0 and c_1. Messages $3, 4$ are pending in c_0's mailbox, c_1 already consumed all the messages sent to it.

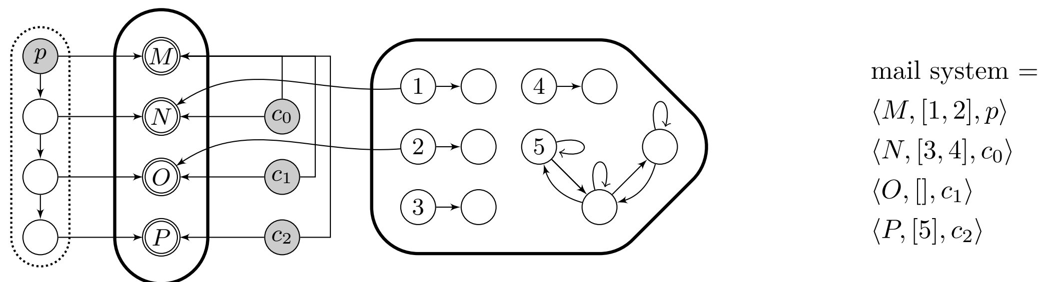


Figure 1.7: Snapshot of a producer-consumers system.

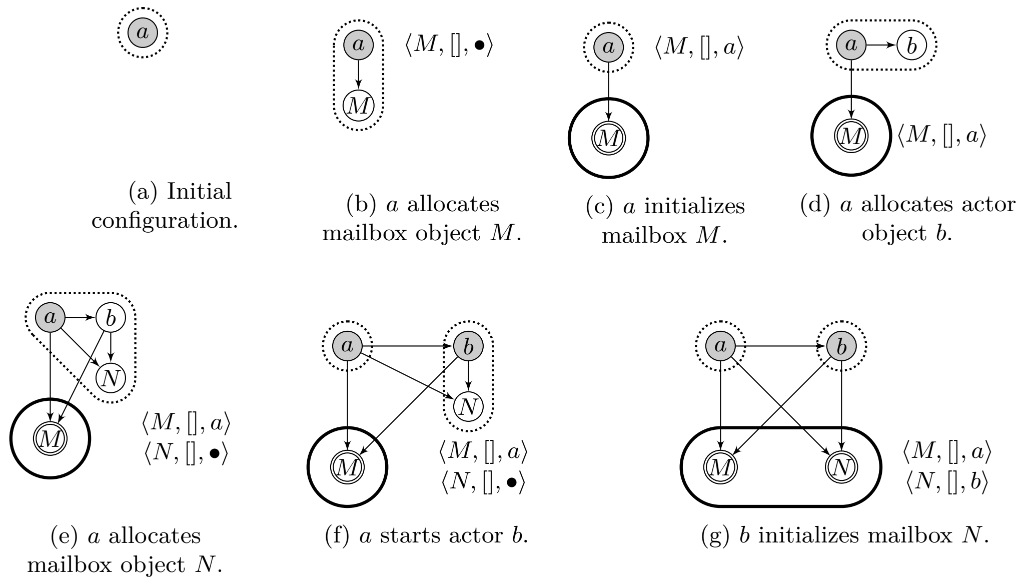
In order to create a new mailbox, an actor must first allocate a fresh mailbox object. The allocation is immediately taken into account in the mail system, by creating a new entry for the mailbox, where the message queue is empty and more importantly, the receiver's name is initially undefined: $\langle M, [], \bullet\rangle$, but messages may already be enqueued. The fresh mailbox object initially belongs to the actor that created it. Therefore it is subject to the same ownership transfer rules as other regular objects. We say the mailbox is *not initialized* until its receiver is set. The only actor that may initialize a mailbox is its current owner, the initialization operation fixes the receiver (which may be different from the actor initializing the mailbox) and erases the ownership of the mailbox object. We give a complete example of mailbox initialization at the end of the next paragraph.

We conclude this paragraph with an updated definition of a valid message: if an object o is a valid starting message object without taking references to initialized mailboxes into account, then o is also a valid starting message object taking these references into account.

Actor creation and activation. The actor creation process is similar to the mailbox creation process, initially an actor is a single object owned by its creator. The creator is in charge of crafting the initial state of the new actor out of the actor object. For instance, a mailbox can be attached to the new actor object so that it may communicate with the rest of

the system. Once the actor local state is ready, the ownership of all the objects transitively reachable from the actor object is transferred to the fresh actor, and the behaviour method of the fresh actor object is called in a new unit of sequential execution. Structurally, Actor objects only differ from regular objects in the behaviour method they must implement.

The example unrolled in Figure 1.8 shows the consecutive steps from the initial configuration (a) where a single actor populates the system, to the configuration (g) where a second actor has been created and is ready to communicate with the first one. Configurations (a) to (c) demonstrate how actor a allocates and initializes a mailbox M from which it may receive messages. In configurations (d) and (e), a allocates a fresh actor object b, a mailbox N, and prepares b's the local state by giving b some references to M and N. b becomes active from step (f), where n is transferred into b's ownership domain. Finally in step (g) b sets itself as the receiver for mailbox N.



(a) Initial configuration.

(b) a allocates mailbox object M.

(c) a initializes mailbox M.

(d) a allocates actor object b.

(e) a allocates mailbox object N.

(f) a starts actor b.

(g) b initializes mailbox N.

Figure 1.8: Actors and mailbox creation.

1.2 Contributions

A formal model of dynamic ownership-based isolation

— Specify a simple actor programming model.

— Extends Jinja, a Java-like machine-checked semantics.

— A proof of the isolation property verified with Coq.

A modified Java Virtual Machine

— Based on the JikesRVM.

— Provides low-level ownership machinery.

— Provides trusted APIs guaranteeing isolation.

A static analysis

— Significantly decreases the runtime overhead.

— Can provide programmer assistance.

— Implemented in Java using the Soot framework.

1.2.1 Contributions to the Open-Source communities

In this thesis, I used and modified the JikesRVM and the Soot framework. Those software benefit from an excellent code material and some extremely qualified and devoted community members. From time to time, I had a question that I couldn't answer and there were always someone to give me good advices. Less often, I found a bug that was important enough for me to dig in and fix. Thus I invited myself in these two communities, most importantly in the Soot's one though. Both software are still evolving and I've been happy to add my modest contributions.

1.3 Organization of this document

This document is organized around the threefold contribution. The current state of the art is presented in Chapter 2. In Chapter 3 we develop the formal specification and give an outline of the machine-checked proof of isolation. The fourth chapter details the actual implementation of the virtual machine. Chapter 5 documents the static analysis, we show two usages of the results and discuss the implementation in JikesRVM and Soot. The evaluation of the virtual machine and the static analysis is deferred to Chapter 6. In the last chapter we expose some future works perspectives along with our conclusions.

Chapter 2

State of the Art

Contents

2.1 Isolation in scalable concurrent systems 19
2.2 Static analyses for efficient concurrency 30
2.3 Motivations 32

2.1 Isolation in scalable concurrent systems

A concurrent system is an environment where several units perform sequential computations in parallel. Whether units work on disjoint data sets or not is a design choice with far-reaching implications. A pertinent metric is the potential *scalability* of a concurrent system. Scalability measures the propensity of a system to maintain a given property as it grows of an arbitrary order of magnitude. The computational performance of a system scales if it is able to deliver near-linear acceleration as the number of processing units increase. Another concern is the complexity to reason about large concurrent systems. If one need to prove an invariant, then the required proof should be reasonably tractable regardless of the architectural complexity of the system.

Sharing the data has been the dominant model since threads and fine-grained locks were introduced with the advent of shared-memory multiprocessor computers, but it is now considered too error-prone and unscalable[67]. In this thesis we study the case when every units can only access its own private state in a share-nothing fashion. The *isolation* property guarantees that a unit cannot access or corrupt the state of another unit. That approach has the following benefits. First, since units don't interfere with each others, each can be designed and reasoned about independently. Second, no costly and error-prone synchronization is required to access data, which avoid an important bottleneck. Thus it offers a greater potential of architectural design and scalability.

But share-nothing concurrency have very limited interest unless isolated units can communicate. *Message-passing* is a very well adapted communication pattern that is widely implemented in modern concurrent systems. The concurrent units can only exchange data through explicit messages that are copied from a unit's memory region to another's. Message-passing is practical because it accommodates any memory hierarchy, once the data is marshaled at the emitter's location, it can be transmitted over arbitrary distances and supports to the receiver's location. Since messages are replicated in the receiver's private state, each unit of the system reads and updates its own copy of the data, thus isolation is trivially preserved.

On a theoretical point-of-view, that scheme is entirely satisfiable.

Competing against traditional threaded programs for performances demands a less naive solution. Indeed, if we focus on concurrent units accessing a shared memory, the message-passing duplication cost is clearly disadvantaging the share-nothing approach. To overcome this cost, communication mechanisms must leverage the availability of a shared memory to pass messages without copying them. This is easily realizable at first sight, it suffices that concurrent units communicate by exchanging memory pointers, very much like a threaded program would proceed. However it becomes challenging to enforce the isolation property once arbitrary memory references can be exchanged because one must ensure that after a communication between a sender and a receiver, the sender won't access the piece of data reachable from the transferred pointer.

We identify three streams of runtime systems with respect to the isolation property. The first, *ownership-based isolation* is mostly based on external uniqueness, a mix of reference uniqueness and ownership. The second stream is specific to the Java language, it has a notion of *classloader isolation* which guarantees isolation between objects of classes loaded from different classloaders[34]. The third stream guarantees strong isolation between *software-based processes* at the cost of deep-copy communications.

2.1.1 Ownership-based isolation

Uniqueness and ownership

Techniques of safe parallelism have been developed to prevent unintended side-effects between threads sharing data. They are directly inspired by techniques that were initially addressing side-effects control and encapsulation in sequential programs. In this section we first describe the first works on uniqueness and ownership. Then we show how these two concepts are reused for concurrent programming and isolation.

Minsky[79] balances the blessings of sharing objects with its hazards. The multiplication of references to a given object makes it hard to reason about this object since it might be modified by any method of the system that is given such alias. He points out the contradiction with the fundamental principles of object-oriented programming, namely *encapsulation* and *hiding*. The techniques of reference uniqueness and object ownership we present now were introduced to cope with the difficulty to enforce information hiding in object oriented programming languages.

Encapsulation. Encapsulation, one of the fundamental principles of object-oriented programming, is the expression of a software design choice, an architectural dogma, where data and procedures serving a common functionality are gathered inside a logical unit that supplies services through a normalized interface without directly exposing its internal components. Booch[21] gives the following definition that has been widely accepted:

"The process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation."

Such structure allows to share, reuse and update the implementation of a given service with no update needed for the clients of that service. This modularity is a central concern in software engineering because it allows the development of large programs as a set of separate building blocks that can be assembled through pre-determined interfaces, regardless of their

respective implementation.

Modern programming languages offer various forms of encapsulation, most imperative languages provide at least a notion of module or package. In object-oriented programming, the notion of classes and instances of classes allows encapsulation at a finer level of abstraction.

Information hiding. Although the concept of encapsulation creates a logical compartmentalization of the services in a program, it doesn't prevent a module from exposing some of its internal components to its clients. Information hiding aims at actually protecting, hiding, the parts of a module that are specific to its implementation so that clients won't interfere with it. Again, Booch summaries the concept of information hiding as following:

"The process of hiding all the secrets of an object that do not contribute to its essential characteristics; typically, the structure of an object is hidden, as well as the implementation of its methods."

As we can see, an arbitrary degree of porosity is left to the appraisal of the software designer which is the sole judge of the "essential characteristics" of its module. The similar definition given by Parnas[85] also includes a form of tolerance in the hiding:

"Every module [...] is characterized by its knowledge of a design decision which it hides from all others. Its interface or definition was chosen to reveal as little as possible about its inner workings."

Hiding is critical in the presence of shared memory and pointer aliasing because encapsulation would no longer hold if a reference to internal structures of a module is passed to a client. Furthermore, if a "secret" escapes the encapsulation boundaries, it might be modified in an uncontrollable way, leading to an inconsistent state of the module that would probably cascade software faults to other clients. Conversely, if a module receives a reference from a client and integrates it in its own encapsulation boundaries, it is required that the client won't attempt to modify the data by accessing its own copy of the reference.

Unfortunately, imperative languages offers very limited support for hiding. They usually associate type declarations with visibility modifiers such as `public` and `private` that respectively shows or hides a member of a module to the external world. But it's not sufficient to avoid uncontrolled aliasing of the internal state of a module. Consider a Java class with an instance field of a reference type, nothing can prevent a public method of the class from returning the value of the reference field, even if it is declared `private`. If the reference escapes to a caller outside of the designed logical encapsulation boundaries of the class, then invariants that are supposed to be guaranteed by the encapsulation might be violated from external manipulations of the referenced data.

Reference Uniqueness

Minsky argues that the uncontrollable dispersal of pointers is mainly due to the "almost universal practice in programming to transfer information by copy" – the copy of pointers. He proposes the concept of unshareable objects through the *unique pointer* mechanism. Unique pointers employ the *destructive-read* operation initially introduced by Hogg in Islands[58] in order to maintain unaliasing of unique references, and later reused in the Baker's "use-once" variables[12]. Minsky identifies the inconvenience of references volatility induced by the destructive-read operations and proposes a notion of non-consumable unique references.

Reference uniqueness[58, 79, 4, 24, 23] eliminates the sharing of one object at a time by making sure there is only one reference to an object.

Static and dynamic references. In object-oriented programming, stack variables and instance fields either hold primitive value or reference to an object allocated in the heap. References and objects form an evolving directed graph rooted in the growing and reducing stack. The convention is that *dynamic* references are found on the stack and *static* references in object fields. Dynamic references are short-living, they only exist temporarily and disappear when the stack frame they are contained in is released. Static references on the other hand have an indefinite lifetime.

Aliasing and uniqueness. As soon as there exists more than one reference to a given object, we say these references *alias* each others. Aliases are either dynamic or static depending on the nature of the reference holding the alias, which are respectively stack variables and object fields. An unaliased reference is therefore *unique*, meaning all the access paths to the uniquely pointed object goes through that reference. Trivially, the reference to a freshly allocated object is unique. However preserving that property requires some precautions since copying such reference would immediately create an alias.

How uniqueness relates to information hiding. We saw previously that encapsulation can be violated if the internal parts of a module can be accessed and therefore modified from the outside. We gave the brief example of a Java object that can leak a private reference simply by returning it from one of its public methods. Hogg's proposal is to employ reference uniqueness as a way to prevent static aliasing from crossing encapsulation boundaries. Indeed, if a static reference to an object is unique, then it is either held outside of any encapsulation boundaries, or by a single unit of encapsulation at a time. If a unique reference is to be received by an encapsulation unit, then it cannot be held outside of it anymore. And conversely if a module returns the unique reference to one of its internal component, then the static reference to that component must vanish from the encapsulation boundaries to prevent aliasing. Therefore reference uniqueness is a technique to enforce information hiding in object oriented systems.

Destructive-read. The solution proposed by Hogg to enforce the uniqueness of a reference is based on a destructive-read operation in which a unique reference variable or object field is immediately nullified after being read so that only a single reference may subsist.

However destructive read is cumbersome as soon as one need to pass a unique field reference to a function since the field must be nullified at the call site to be available as a parameter for the receiver. Restoring the value of the field requires that the function explicitly returns the unaliased reference.

Borrowing. More flexible solutions have been proposed, in which a unique reference can be *borrowed* instead of being destructed if the created aliases are temporary. A borrowed variable is created from a unique reference field without consuming it, but its value cannot be used to create a static alias. Borrowed variables can be safely passed to methods expecting borrowed parameters, when they return all the aliases on borrowed values are framed-out.

The borrowing techniques must address the callbacks problem: the unique field for which borrowed values exist must not be accessed until all its dynamic aliases die because otherwise

some unexpected data dependencies may appear under the false assumption that the field is unique. One solution is to scope each borrowing in a block during which the field is nullified and restored at the end[32]. The *alias-burying* technique is based on a static analysis removing the need for temporary nullification [23]. Another technique employs statically verified *capabilities*[24, 53], a type system of permissions that can be consumed only once.

Object ownership. Ownership[34, 83, 35] aims at protecting aggregates of objects. The ownership relation superimposes a topology over the objects of the heap so that every object has an owner, and may owns other objects. This relation limits the visibility and the updates of objects inside an aggregate from the outside. Ownership types are statically verified type systems enforcing the encapsulation. They rely on types annotations at the source code level, these annotations determine the ownership context in which the object is evolving. To be correct, the annotations of a program are statically verified by the ownership type system, if an inconsistency is detected the program must be rejected.

There exists two disciplines of ownership. The *owner-as-dominator* (aka. *deep-ownership*) guarantees that every chain of references from an object o in a context to an object o' in another context goes through the owner of o'. In this discipline an owner has the control over the access ot the objects it transitively owns, and objects in the same context may reference each-others freely. The second discipline, *owner-as-modifier* is less restrictive, it allows all kind of references and no constraint on read operations, but objects may only modify other objects they own or that are part of the same context.

External uniqueness. External uniqueness[32] is a restricted form of deep-ownership allowing uniqueness-like operations on aggregates of objects. An owner is externally unique if it is uniquely aliased from the outside of itself (it and the objects it owns) but freely aliased from the inside. If an aggregate is owned by an externally unique object, then the whole aggregate can be moved from one unique variable or field into another, the former being nullified. Externally unique aggregates may also be borrowed. And most importantly the ownership of an externally unique aggregate can be transferred. *Ownership transfer* is the action of moving an aggregate in the ownership hierarchy by simply moving its reference from one context to another.

The following projects use a notion of external uniqueness and ownership transfer in ways that Siaam can relate to. Ownership transfer is particularly interesting to build actor-based programming models because it allows actors to communicates externally unique messages in a zero-copy fashion through their mailboxes.

PRJF. PRJF [22] is a static type system guaranteeing that well-typed programs are free of data race. It is based on the observation that one can obtain the race-free property trivially if every thread acquires a lock on every object it accesses. The type system of PRJF, which mixes ownership types and uniqueness, reduces the amount of necessary locks in many typical situations. First it introduces a special “thisThread” owner indicating thread-local objects for which no synchronization is required. Second, PRJF employs ownership to embed owned objects access permissions into the owner’s lock. Meaning it is sufficient to acquire a lock on the owner of an aggregate in order to gain exclusive access to the enclosed objects. Finally, reference uniqueness avoid locking an unaliased object since at most one thread can refer to the object. That latest case composed with the previous one allows unsynchronized race-free access to every objects owned by an unaliased object. For all the other shared objects

where no protection invariant can be inferred, the compiler fallbacks to the traditional locking mechanism.

Reference uniqueness permits ownership transfer of unaliased objects. To achieve this, the thread holding the unique reference stores it in a shared container, by doing so it loses the reference. Later, another thread may enter the shared container (protected by a traditional lock) and retrieve the unique reference.

Minimal Ownership for Active Objects. In MOAO[33], the authors differentiate *active* objects from *passive* objects. Active objects handle the concurrency control and encapsulate unshared state made of passive ordinary objects. A minimal ownership type system guarantees that active objects can access the passive objects they embed without the need of any locking mechanism (race freedom invariant). Unlike most existing ownership systems, the ownership in MOAO is very flat, passive objects belong to an active object, and active objects belong to the global owner. There is a *unique* type to declare unshared references; uniqueness is preserved through destructive reads; the static semantics provides borrowing blocks that temporarily nullify the borrowed reference.

Active objects expose a public interface, a set of methods expecting arguments pointing to —by default— disjoint object graphs. When an asynchronous method of an active object is called, the objects reachable from the parameters are either passed by reference (ownership transfer) or copied depending on their owner. Active objects are always passed by reference. Unique stack variables can be passed by reference (destructive read) but the references inside an unshared object still need to be examined. For the other references, cloning is required in order to obtain graph disjointness. From our understanding of the cloning process, it is safe to pass an object by reference when there exists a unique path from the stack to that object and that path is solely made of unique references. Consequently, asynchronous arguments should be tree-shaped graphs of uniquely referenced objects in order to avoid any data copying.

Siaam has a similar notion of flat ownership, all the objects can be considered passive excepting the mailboxes and the actors once they are activated. Actors in Siaam play the same role as active objects in MOAO, they encompass a thread of execution and an unshared state made of regular objects.

Kilim[91]. Kilim is a framework for developing isolated actor-based concurrent systems on top of the Java Virtual Machine. It offers a zero-copy message-passing communication scheme and employs light-weight cooperatively-scheduled threads.

The cooperative scheduler requires a bytecode post-processor that "weaves" (a *kilim* is a Persian carpet) the application bytecode in a continuation passing style.

The isolation property is statically verified by an intra-procedural heap analysis performed during a post-processing phase. To simplify this analysis, messages must be made of uniquely aliased objects, meaning messages are tree-shaped. The Java source code requires some ownership type qualifiers in order to guide the static analysis and render explicitly which references are unique.

Kilim is used in the ActorFoundry framework and inspired the O-Kilim framework, both described later.

The choice in Kilim is to post-process the application bytecode so that it may be executed by an unmodified Java virtual machine. The reverse of the medal is the need for source annotations and strong aliasing constraints on zero-copy transferred objects. With Siaam we follow the opposite path, we introduce enough modifications in the Java virtual machine to

offer an annotation-free and aliasing-proof actor-based programming model without changing the application bytecode. Furthermore Siaam does not enforce a particular underlying scheduling policy, we implemented both a 1:1 actor-thread mapping and a N:M policy able to schedule run-to-completion tasks for M actors over a pool of N Java threads.

Capabilities for Uniqueness and Borrowing. Haller & Odersky [53] propose a statically checkable type system based on a "separate" uniqueness invariant to enforce race safety in Scala's actors. Two *separate* variables points to disjoint object graphs in the heap. A variable is *separately-unique* if it is separate from all the remaining variables accessible in the stack, accessible meaning potentially used before being redefined in the current execution. Using this strong invariant, they prove it is safe to pass the reference of a separately-unique variable among concurrent processes because 1) the sender won't reuse the reference and 2) the receiver becomes the sole holder of the unique reference.

The authors achieve efficient message passing; according to their experiment feedbacks they managed to integrate their type system in the Scala compiler and the standard library within less than 5000 lines of code and a limited number of annotations since the default types where often adapted.

The message passing semantics in Siaam is very similar to the proposition of Haller & Odersky: passed references point to disjoint graphs of objects. In Siaam the disjointness is enforced when a message is about to be passed. The ownership of every object contained in a message is dynamically switched to a fresh owner. Thus every existing external reference into the message is virtually invalidated: if an actor tries to access an object of the fresh owner, a special exception is raised. These run-time errors are statically avoided with capability-based uniqueness.

O-Kilim[51]. Ownership-Kilim resurrects the event-driven actor model – without CSP weaving – described in Kilim and brings the ability to transfer object graphs based on previous works by Claudel et al.[36, 37] on "first-reachability". First-reachability ownership fixes the owner of a *free* object the first time it is pointed-to by an *owned* object. This runtime mechanism is handled by write-barriers inserted whenever the field of an object is updated, a very similar usage of write-barriers is employed in [43]. Unlike in most ownership systems, here ownership isn't designed to restrict the usage of references, instead it is employed to delimit boundaries of aggregates of objects that can be safely passed by reference.

Initially allocated objects are free, meaning they do not belong to any ownership domain (aka. context). The only exceptions are the instances of the `Task` and `Message` classes that always own themselves. The desired invariant is that an object may at most belong to one owner and there cannot be a reference from a domain to another or from a domain to a free object. As soon as a free object becomes transitively reachable from an actor or a message, it enters the ownership domain of that owner. The rule is applied transitively to every object referenced by a freshly owned object. To preserve the invariant, references between different ownership domains are illegal and raise a run-time exception during the ownership propagation. All in all, the first-reachability mechanism guarantees full-encapsulation with internal aliasing of messages and actor state.

Actors implement run-to-completion tasks called *reactions*, which is the equivalent of the *communications* received by Agha's actors[2]. Each actor is bound to one or several `Mailboxes` and registers reactions to these mailboxes. Upon message reception, a mailbox calls back the reaction registered by the actor it is bounded to, if any. Reactions are processed in a serial way for a given actor. Message-passing is achieved by storing a `Message` reference in a mailbox.

The reaction-based approach of O-Kilim was carefully chosen to allow temporary dynamic (in stack) aliases of objects inside a message, the actual communication being deferred to the termination of the current reaction. The *tail-migration* consists in waiting the completion of a reaction before actually sending the messages it emitted. Once the reaction is completed, all the dynamic aliases are costlessly framed-out and it is finally safe to pass message references to other actors. That choice can be seen as a form of reference borrowing, encompassing all the stack variables.

Although the ownership of an object is fixed without the explicit intervention of the developer, an extraction primitive is provided to free the graph of objects transitively reachable from an arbitrary reference. The `extract` primitive can be seen as the matching pair of the `cut` operation in Kilim. The extraction nullifies every remaining reference in the original domain, leaving a “hole” of illegal pointers. Once freed, the objects can be absorbed by another owner or will simply become garbage at the end of the current reaction. The extraction mechanism is essential to create a durable heap reference from an actor’s state to a received object that can be used in successive reactions. Indeed, it is illegal to transitively reference an object of the `Message` class from an instance of `Actor`, which are both owners.

This work is, to our knowledge, one of the only actor-based system before Siaam where the ownership is purely managed at run-time without any annotation being required. However it is not clear how the extraction function proceeds to find and nullify references internal to a message when a subgraph is extracted and what are its complexity and run-time cost. Furthermore we could not determine how O-Kilim prevents an actor from passing the same message twice during a reaction and how (should?) it prevents an actor from extracting or modifying a part of a message it has virtually already passed during the reaction. It would also be interesting to understand the consequences of the communication latency induced by the tail-migration.

In Siaam, the ownership relation is used to control data access whereas in O-Kilim it is employed to control static aliasing. O-Kilim also get rid of the dynamic aliasing problems through the choice of a reactive execution model. The invariant guarantees that every reachable object can be accessed safely during a reaction. Siaam doesn’t have any notion of message instances, in particular our model doesn’t require extracting received objects to create durable references because every received objects is automatically transferred in the ownership domain of the receiver. Both approaches generate illegal references when a graph of objects is transferred from one context to another (extraction in O-Kilim, message-passing in Siaam). In both cases, accessing an illegal reference raises an exception that may be handled by the program. However in O-Kilim, an illegal pointer is equivalent to a `null` pointer, whereas in Siaam it is still referencing the original object and automatically recover to a legal pointer when the object is received back, which allows new programming idioms.

Comparing the run-time costs of both approaches isn’t trivial. Siaam pays every object access (not taking optimization into account) and O-Kilim pays every static reference created. Siaam pays a transitive ownership-transfer of each passed graph and O-Kilim pays an equivalent sum of transitive ownership-transfers to create a message. Forwarding an unmodified message is free with O-Kilim if it has been received during the same reaction, but passing the content of a message in a different reaction demands two extractions and two absorptions. Our conclusion is that in average Siaam and O-Kilim probably have a very close run-time overhead.

2.1.2 Software-based processes.

Several projects aims to host multiple applications on the same runtime system. They provide a form of full isolation inherited from operating systems where each process has its own memory and cannot access to another process' memory. Most runtime systems hosting isolated software-based processes only support communication by copy[8, 54, 84, 61, 40]. A few others rely on communication through shared heaps [11, 3, 59].

Erlang. Erlang[8] is a declarative programming language focusing on fault-tolerance and large-scale parallel computing. The concurrency is modeled by lightweight processes that can interact only by exchanging messages. Erlang has no shared memory and all the data structures are immutable, therefore there are no locks and the processes are trivially isolated. The messaging mechanism is asynchronous, the sender doesn't wait for its message to be processed, and the receiver is not interrupted in its current task. Although communications use a costly deep-copying semantic, Erlang became the de-facto language to develop massively concurrent systems thanks to its industrial-grade fault-tolerance. The main reasons of its wide adoption are certainly its simplicity and its robustness.

More recently, Carlson et al. described a "message analysis" for a subset of Erlang that enables zero-copy message passing whenever possible. This work is discussed in Section 2.2.

ActorFoundry. ActorFoundry[61] is an actor-based framework for the Java Virtual Machine. It provides a simple API including asynchronous messaging (`send`) with the possibility to wait for an asynchronous response (`call`), and actor creation (`create`). In the early versions of the framework each actor maps to a Java thread and messages are sent by deep-copying their content with the Java's serialization API. Therefore objects that are part of a message must implement the `java.lang.Serializable` interface.

Later ActorFoundry was updated to integrate Kilim's continuation passing style scheduler based on the same bytecode weaver, so the number of concurrent actors is not limited by the number of Java threads that may coexist in the virtual machine and the thread context switching cost is limited. However the messages must still be copied, this constraint is partially solved by SOTER which is discussed in the next section.

Java Application Isolation API. The Java community has fulfilled the Java Specification Requirement JSR-121[74], the Java Application Isolation Specification ("Isolate API"). It defines a fundamental set of classes and interfaces for supporting multiple strongly isolated applications — *Isolates* — in a single JVM.

Each isolated application has a dynamic set of threads and everything happens like it fully occupies the virtual machine. Even the system classes are replicated for each application so they cannot communicate through static variables. Isolates cannot share objects except communication channels called *Links* which have a single sender and a single receiver Isolates. A communication through a Link consists in a rendezvous where a *LinkMessage* is transferred from the sender to the receiver. A LinkMessage may only contain a limited range of datatypes: a single string, a Link reference, an Isolate reference, a set of other LinkMessage references or an array of bytes. Arrays of bytes are copied when the rendezvous eventually happens, so that the communicating Isolates have their own private copy. Any other kind of data must be serialized into a byte array before being sent, and deserialized by the receiver Isolate. In particular objects graphs must be deep-copied.

Isolates are created by specifying the name of a class and an string array. The given class must declare a static `main` method that will be invoked with the string array parameter in a new thread. Optionally the Isolate object can be given an initial set of Links before it is started, so it may communicate with other applications.

The Isolated Application API enforces strong isolation between Isolates at the price of deep-copy communications. The principles in the Isolated API where initially experimented by Sun in an early prototype named The Multitasking Virtual Machine or MVM[41, 40].

Capability based isolates. In the Object Space Model[26] and the J-Kernel[54], *bridges* and *capabilities* respectively, can be shared between isolates. These objects act as isolation proxies for cross-domain calls.

Methods of a capability are invoked with a special calling convention: parameters and returned value are passed by reference if they are capabilities, or by deep-copy if they are primitive values or ordinary references. Methods of a bridge have a different calling convention: parameters and returned references are wrapped into bridges unless they are already bridges.

The Object Space Model guarantees strong isolation because a context cannot obtain an ordinary reference to an object of another context. But bridges must be created for every object accessed from a foreign context, moreover bridges are cached in a map associating a unique bridge to each object.

The J-Kernel modifies the application bytecode at runtime and runs over unmodified JVMs. To reduce thread-switching costs on cross-domain calls, the threads actually perform the calls directly like in the I-JVM.

Process isolation with exchange heaps. Both the KaffeOS[11] and the Singularity OS[3, 44, 59] provide isolated processes that communicates through shared heaps. In order to communicate, a process must create an exchange heap and populate it with data.

The KaffeOS uses write barriers on every pointer write to check for illegal references. After a shared heap is populated, it is frozen so that the object topology in the heap is fixed until reclaimed by the garbage collector. Once frozen the heap may be communicated to other processes, the specificity of KaffeOS is to allow concurrent modifications of the primitive-typed fields in the frozen heaps. The authors admits the non-negligible overhead of the write barrier, and conclude: “a good JIT compiler could perform several kind of optimizations to remove write barriers. A compiler should be able to remove redundant write barriers . . . ”.

Singularity enforces strong isolation between software processes. Processes communicate by passing messages through channels. Messages contain pointers to data allocated in the exchange heap, however the exchange heap does not contain ordinary objects but only data blocks. Furthermore pointers to the data blocks are unique, meaning that the sender of a block loses the reference which is passed through a message to the receiver.

2.1.3 Classloader isolation.

Classloader isolation is based on the type-safety rules governing the Java language. It distinguishes types not only by class name but also by the classloader from which the class originates. This distinction make objects originating from different classloaders unusable for each-others, meaning a method created by one classloader cannot read nor write the fields of an object originating from a different classloader.

This form of isolation does not impose a notion of software component, whether we call it actors, processes etc. Although objects from different classloaders are isolated, objects from common classloaders may be freely exchanged and accessed from different software components of the system. Therefore encapsulation and information hiding is not trivial since developers must carefully remember what classes are isolated and what objects are shared and may mutate without control. Building complex dynamic systems based on classloader isolation demands some global conventions. The OSGi[78] platform provides a software component based environment, it specifies how components get isolated using private classloaders, and how they can exchange shared data using common classloaders.

OSGi. OSGi is an open specification of dynamic component systems for the Java language. Components in an OSGi system are called *bundles*, they can be installed, started, stopped and updated at will without requiring the whole system to be halted. Bundles implement and publish services which can be consumed by other bundles. A complete description of the architecture of OSGi is out of the scope of this thesis, however we are particularly interested in the isolation features offered by the specification.

Although OSGi integrates a weak notion of isolation: bundles can selectively share objects. Moreover a number of vulnerabilities have been identified[86] in the most common implementations of OSGi, some of them showing flaws in the expected degree of isolation.

Bundles do not encompass threads, they only provide operations through object interface methods and there are only application threads. When an application thread needs to obtain a service from a bundle, it calls the appropriate method and enters the bundle's implementation. Such calls are *inter-bundle* calls by opposition to intra-bundle calls when an application thread has already entered a bundle and invokes an internal method without crossing a bundle interface.

In order to maintain state encapsulation and information hiding within a bundle, the OSGi specification makes a strong difference between each bundle's private and shared classes. The isolation property is that objects of private classes of a bundle cannot be accessed by other bundles. It uses the classloader isolation strategy where each bundle is given a private classloader from which it loads its private classes. Bundles can also export public classes or import the public classes of another bundle by loading them from the same classloader, these classes instantiate shareable objects which may be used as parameters and return values of inter-bundle calls.

I-JVM. I-JVM is a Java virtual machine supporting OSGi and solving some of its vulnerabilities, in particular isolation related. Wide-spread implementations of OSGi let bundles interfere through system classes static variables, either by modifying them or by locking shared content referenced by static variables.

The I-JVM employs a technique originating from the Java Isolates to solve this problems. The static variables, the strings and the class objects are duplicated so that each OSGi bundle manages its own copies. However in the Java Isolates API, threads are confined to a single process to ensure strong isolation. The confinement obliges processes to communicate through a costly object serialization protocol. In OSGI this mode of communication is not realistic to maintain good performances since bundles communicates by direct calls with shared parameters.

The contribution of I-JVM with respect to this problem is to allow thread migration between bundles represented by different isolates. It introduces an indirection in the access

to static variables. When a thread crosses an inter-bundle border, the indirection is updated to point the memory region containing static variables managed by the entered bundle. I-JVM pays a performance penalty less than 20% when running standard Java programs, the cost for resource accounting and intra-bundle calls. The cost for inter-bundle calls, compared with classical isolates communication, is extremely low since it does not introduce parameters copy.

2.2 Static analyses for efficient concurrency

Message Analysis [28]. Asynchronous communications in the default configuration of Erlang requires messages to be copied from the sender process to the receiver, with a cost growing linearly with the message size. It is worth remembering that data is immutable in Erlang but process-local heaps provide fast, synchronization-free allocation and garbage collection.

In this article the authors investigates an hybrid architecture where each process has its local heap and a shared heap is available to store inter-process messages. In comparison with process-local heaps, the shared heap has less efficient synchronized allocation and garbage collection but allows zero-copy pointer communications. The heap segregation comes with the runtime invariants that “there are no pointers from the shared heap to the local heaps, nor from one process-local heap to another”.

The allocation strategy at runtime is to speculatively allocate data that may be part of a message in the shared heap, and the other data in the process-local heaps. Since it may happen that process-local data is part of a message, the communication operations implement a copy-on-demand mechanism, which verifies that transferred data resides in the shared heap and copies the locally-allocated parts to the shared heap. Note that checking if an object was allocated on the shared heap is a $O(1)$ pointer comparison in the hybrid architecture designed in this work. The verification must explore the graph of data reachable from a communication operand, which can be a costly operation. Thanks to the pointer directionality invariant, the graph exploration can be cut whenever a piece of data already allocated on the shared heap is encountered, therefore if the top communication operand has been allocated on the shared heap the verification may complete immediately.

The *message analysis* tries to identify allocations that should be performed on the shared heap to avoid the cost of verifying and copying the data when it is eventually communicated. It rewrites the analyzed program with shared-heap allocations at places where data is likely to be part of a message. The copy-on-demand mechanism is still active in the runtime, so that the static analysis may safely misses some shared allocations.

The *message analysis* benefits from the data immutability in Erlang. In Siaam objects are mutable, meaning that it is not sufficient to pass messages by reference to enforce isolation, we also have to make sure that only the receiver of the message may modify its content and conversely that the other actors cannot access that content. Other than that, we can find several common points between the *message analysis* and Siaam.

First, our virtual machine handles the communication of immutable objects in a very similar way because the graph exploration can be cut when such object is encountered: an immutable object may only points-to other immutable objects, which can be seen as having a special heap for immutable objects with the invariant that there are no pointers from that special heap to the heap of mutable objects.

Second, our static analysis is able to identify an under-approximation of the message-passing places where every object that is part of the message is owned by the current thread. For these communications, it is not necessary to dynamically check the ownership of the message graph. However it is still mandatory to explore the graph in order to switch the objects' owner and prevent the sender from accessing them further.

SOTER, inferring ownership transfer. SOTER[81] (for Safe Ownership Transfer enableER) is a static analysis aiming to discover message-passing sites in an actor-based program where the deep-copy semantics can be safely replaced by the zero-copy reference passing semantics with respect to the isolation property. SOTER is applied to ActorFoundry applications where it replaces `send` and `call` methods of the API by their zero-copy alternative when possible. The analysis is versatile enough to be adapted to handle other frameworks, the authors mention they extended SOTER to support the Scala actor API in a couple of weeks.

The analysis itself is inter-procedural but employs two intra-procedural sub-analyses. The first one is a custom live variable analysis which computes the set of objects reachable from live variables after each program point that potentially communicates objects (either directly or through transitive calls). The second one is a forward data-flow problem over the call graph nodes. It propagates the object liveness information along the call graph edges. At the end of the propagation process, the analysis can tell which objects are live after a given communication site. If any of the objects transferred by the communication site is live after that site, then it is not safe to replace the deep-copy semantics with the zero-copy semantics for that site.

The static analysis in Siaam comprises a phase that shares some common points with SOTER but proceeds the other way around. In our analysis, it is the transferred objects information that flows along the call graph edges. Our local object liveness analysis reduces the amount of objects flowing through each edge. Although the Siaam analysis was not designed to infer safe ownership transfer, it is well adapted for that task. The programming assistant detects unsafe statements and identifies the possible communication sites where the unsafe data was transferred. Using the result of the assistant we could replace the incriminated zero-copy communications with a deep-copy message-passing semantics.

The weakness to SOTER's pessimistic approach is that among the live objects, a significant part won't actually be accessed in the control-flow after a communication site. On the other hand, Siaam do care about objects being actually accessed, which is a stronger evidence criterion to incriminate message passing sites. It incriminates a message-passing site only when it detects the actual usage of a transferred object. We provide a comparative evaluation of SOTER's and Siaam's analyses accuracy in Chapter 6.

Other related works. Static analyses have been developed to optimize-out redundant runtime checks to detect concurrency-related problems. They often rely on a pointer and escape analysis[16, 30, 93, 80]. Runtime checks have been used to check dangling references to region allocated objects[88], and an associated static analysis eliminates redundant checking. User defined ownership policies in concurrent C++ programs are dynamically checked[72], and a static analysis removes unnecessary verifications. Typical dynamic race detectors must check memory accesses for concurrent races, in [45] the authors propose a static analysis to remove redundant checking within a "release-free span", it reduces the number of runtime

checks by 40% in their experiments. In [31] the authors presents an analysis based on escapement that allows stack allocation and synchronization removal for Java. Similarly a custom thread escape analysis is developed in [19] in order to remove unnecessary synchronization in Java.

Thread-Local heaps for Java[43]. This work is based on the observation that a single shared heap is not the most scalable memory management scheme in a multithreaded system. Authors propose to introduce thread-local heaps in the Java virtual machine, their approach starts from the opposite side of the *message analysis*[28] but arrives to the same conclusion that local heaps offer fast synchronization-free allocation and garbage collection.

Every object is flagged with a *global bit*. Initially objects have their global bit cleared so they are considered as thread-local objects. Then the runtime system dynamically monitors the locality of the objects using a write-barrier. As soon as a local object becomes reachable from a global object, its global bit is set so it is also considered as global afterwards. Note that the process is repeated recursively for every object pointed-to by a local object becoming global. Global objects never return to local. The second contribution covers the specification and the implementation of the local and global mark&sweep garbage collectors. Performance measurements shows that the overhead introduced by the write barriers is balanced by the speed gain of thread-local allocation and collection.

The heap management described in this work, the allocation scheme and the garbage collection algorithm for both local and global heaps are certainly applicable in the Siaam virtual machine. It would be straightforward to add a *global bit* to every object and set that global bit the first time an object is transferred between two actors. Then following the ideas of[28], a static analysis could infer allocation sites where the data is likely to be part of a message and perform direct global allocation as suggested in[43].

A specificity of Siaam is to allow references between ownership contexts. It may sometime happen that an object with the global mark is only reachable from actors in a different context. In this case, if the object is not part of any pending message, there is no way it may reenter the ownership context of an actor. Therefore it is theoretically possible to reclaim the object's space since no actor may be allowed to access its fields and simply replace the remaining references with `null` or any dummy reference that would still trigger an owner mismatch exception when accessed. But on the other hand it might not be correct to update these references with a unique replacement object since the actors holding them may actually rely on the inequality of foreign references. To circumvent this eventuality, it should be feasible to shift the remaining addresses outside of virtual machine memory range, where a system segmentation fault would be raised and caught in case of access and translated to an owner mismatch exception in the application (a similar method is already in use in the JikesRVM to "detect" null-pointer exceptions). Unfortunately the whole solution seems to bring us back to the starting point since all the objects allocated at the same address would have the same shifted address.

2.3 Motivations

The Erlang language has been well established in the industrial world, delivering fault-tolerant, massively concurrent and actor-based software for decades. The recent apparition of the Erlang-based trendy CouchDB[73] scalable database for web applications is revealing

of the modernity of the actor approach.

With the explosion of multi-core processors in every consumer devices, we see a renewed growing interest in the actor-based programming models. In the Java and Scala communities, new frameworks such as Akka and Netty[87] emphasis on modularity, separation of concerns and safe parallelism. It seems like bringing concurrency to the masses — of users, developers and processor cores — requires a higher level of abstraction than the threading model which has always been a matter of experts. In the .NET ecosystem, the Axum[52] project aimed to bring agent-based isolation through deep-copy message passing.

Now even web developers are considering actors or structured concurrent programming models because it captures parallelism in a much more natural and instinctive manner. Several Javascript frameworks address concurrency, among them is the event-driven Node.js[75] project. Not so long ago, Google initiated the Dart[60] language that compiles to Javascript and provides in-language Isolates.

This extremely short go around the table shows the need for simple programming models to address safe concurrency without actually thinking about applications being parallel. The current propositions either provide strong isolation through deep-copy message passing or unsafe but efficient zero-copy communications. Therefore we identify the need for a solution that stands in the middle, Siaam aims to blend strong isolation and efficient zero-copy message passing. The variety of platforms where safe concurrency may be used motivates us to propose a specification that abstracts as much as possible the underlying environment. The fact that more and more developers, with very different backgrounds, are investigating new solutions to build scalable applications is our main argument to aim for simplicity and dynamicity. Furthermore we see that many industrial projects are emerging, with their own priorities and constraints in mind, and Siaam should fit them. The fact that many of these projects have chosen isolation over efficiency (deep-copy over zero-copy) reinforce our approach evicting any form of statically verified source code annotations even if it means paying some runtime overhead.

We think that shifting the intelligence from the source code (the developer) to the runtime is a promising approach, especially when the runtime can optimize the induced overhead. Related works corroborates in that direction. The authors of the KaffeOS evaluates the total cost of the write barriers and conclude that "a goot JIT compiler could perform several kinds of optimizations to remove write barriers". In Thread-Local Heaps for Java, a similar write barrier monitors global reachability at runtime, although the authors advocates for a fully dynamic approach, they agree that "it would be worthwhile to check how well static determination of object locality works with our memory manager".

Chapter 3

Formal specification of Siaam

Contents

3.1 Introduction. **35**
3.2 HOL Language **36**
3.3 Abstract Syntax. **38**
3.4 Siaam Global Semantics **42**
3.5 Single-Actor small-steps semantics **52**
3.6 The Siaam Virtual Machine **57**
3.7 SIAAM isolation and formal proof **64**

3.1 Introduction.

We extend Jinja [63, 62], a sequential object-oriented programming language with the core features of the Java language and a formal semantics. The extension presented in this chapter builds actor-based concurrency and isolation above Jinja's sequential semantics.

In Siaam, every concurrent unit of sequential computation is identified as an *actor*. Actors mostly behave as independent sequential Jinja programs. However, occasionnaly they have to interract with the global state of the system to communicate, initiate new actors and access the heap which is shared by all the actors.

We follow the elegant methodology employed in JinjaThread[71] to extend Jinja with threads. Thanks to an ingenious parametric interleaving framework, JinjaThreads builds a multithreaded version of Jinja with only minimal modifications to the original source language. Furthermore, the framework is instantiated a second time to create a multithreaded virtual machine supporting the Jinja bytecode. For readers familiar with Jinja and its threaded extension, we describe Siaam in similar terms and notations. However Siaam is much simpler than JinjaThread because it doesn't have to struggle with the concurrency of the Java memory model. Since actors can only observe and mutate the objects they own, sequential consistency is satisfied "for free" in our model.

The formal specification is divided in several logical components, as illustrated in Figure 3.1. This chapter is organized accordingly. The program representation (Section 3.3) fixes how classes and methods of a program are encoded in Jinja, and provides predicates to query

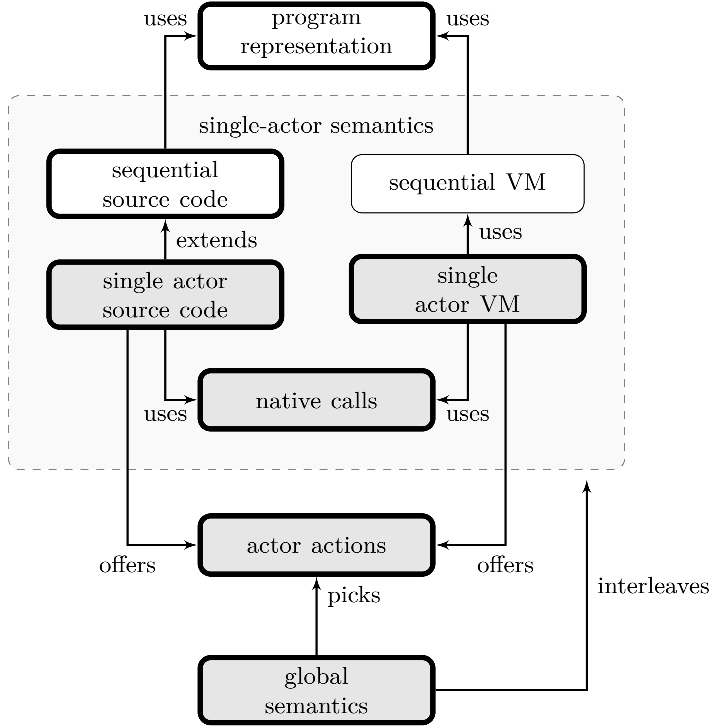


Figure 3.1: An overview of the components used in the formal specification of Siaam.

its structure. It also define an abstract syntax for the source language. Jinja defines small-steps semantics for the sequential source code and a sequential virtual machine. We use them as a basis to specify the small-steps semantics of a single-actor source code (Section 3.5) and a single-actor virtual machine (Section 3.6).

A global semantics governs the reductions of the global state and interleaves the actors, it is presented in Section 3.4.

The original work on Jinja proves the type safety of the two sequential small-steps semantics, it also formalises a bytecode verifier and its correctness proof, as well as a compiler and a proof of semantics and well-typeness preservation. We do not replicate this work here, however the additions we bring to the Jinja sequential semantics are orthogonal to the concern of these proofs. Therefore we are confident that the original properties proved in Jinja are preserved in Siaam. We are conforted in this orthogonality by the fact that Siaam's isolation property have be expressed and proved with a complete abstraction of the reduction rules involved at the single-actor level. The formalization and the proof of the isolation property, conducted with the Coq proof assistant[76], are described in Section 3.7.

3.2 HOL Language

For the sake of homogeneity, we respect as much as possible all the notations introduced in the Jinja's litterature. Thus we adopt the HOL language for the formal descriptions.
HOL includes notations for types, pairs and tuples, sets and lists, functions and inference rules. The most frequent notations are described in this section, further specific writtings are introduced over the course of the paper when they are required. Variable names are

typed in italic (a, V, fn) and names of defined constants are slanted (a, V, fn). The relation of implication is noted $\longrightarrow$ and the equivalence is $\longleftrightarrow$; inference rules are displayed in the following form:

$$\frac{P \longrightarrow Q \quad P}{Q} \text{ is equivalent to } ((P \longrightarrow Q) \wedge P) \longrightarrow Q$$

HOL types. Basic HOL types are boolean (*bool*, naming convention b), natural numbers (*nat*, naming convention n), integers (*int*, naming convention i) and strings (*string*). Type variables are prefixed with a quote ($'a$, $'b$). The notation $t :: \tau$ means term t is of hol type τ. The space of total functions is written $\Rightarrow$.

$$\begin{aligned} \textit{hol-type} = {} & 'a \mid \textit{bool} \mid \textit{int} \mid \textit{nat} \mid \textit{string} \mid \\ & \tau \times \tau' \mid \tau\ \textit{set} \mid \tau\ \textit{list} \mid \tau\ \textit{option} \mid \\ & \tau \Rightarrow \tau' \mid \tau + \tau' \end{aligned} \tag{3.1}$$

Pairs. A pair (a, b) of elements $a :: \tau$ and $b :: \tau'$ is of type $\tau \times \tau'$. Components of a pair are accessed with the functions $\textit{fst} :: ('a \times 'b) \Rightarrow 'a$ and $\textit{snd} :: ('a \times 'b) \Rightarrow 'b$. Tuples identify with pairs nested to the right, meaning (a, b, c) is identical to $(a, (b, c))$ and $('a \times 'b \times 'c)$ is identical to $('a \times ('b \times 'c))$. The disjoint sum of $'a$ and $'b$ is written $'a + 'b$, with the injections $\textit{Inl} :: 'a \Rightarrow 'a + 'b$ and $\textit{Inr} :: 'b \Rightarrow 'a + 'b$.

Sets. Sets (type $'a\ set$) are isomorphic to predicates (type $'a \Rightarrow bool$), such that $y \in \{x.\, P\, x\} \longleftrightarrow P\, y$. The image operator $f\text{`}A$ applies $f$ to all elements of $A$, $f\text{`}A = \{f\ a \mid a.a \in A\}$. The empty set is noted $\emptyset$.

Lists. Lists have the HOL type $'a\ list$. The empty list is $[\,]$, the infix list constructor is $\cdot$ and the infix @ operator appends two lists, just like the *append* function. Variables of a list type usually ends with an "s" like in xs, and $|xs|$ denotes the length of the list. If $n < |xs|$, $xs_{[n]}$ is the n-th element of xs. The function *hd* returns the head element of a list, on the opposite *tl* removes the head and returns the rest of a list. More generally, *take* n xs returns the list of the first n elements of xs, and *drop* n xs removes the first n elements of xs and returns the rest of the list. *rev* xs is the reverse of xs, and *replicate* n x is the list where the element x is replicated n times. The usual functions *map* f xs and *filter* P xs respectively applies the function f to each element of xs, and retains the elements satisfying predicate P. The ordered list of integers in the half-opened interval $[i, j[$ is noted $[i.. < j]$, similarly $[i.. \leq j]$ is the list of integers in the closed interval.

Optionals. The $'a$ *option* data type have the *None* element and all the elements from $'a$ prefixed by *Some*. The notation $\lfloor a \rfloor$ is identical to *Some* a, and *the* is a partial function realizing the inverse of *Some*, such that *the* $\lfloor a \rfloor = a$.

$$\textbf{datatype} \quad 'a\ \textit{option} = \textit{None} \mid \textit{Some}\ \ 'a$$

Functions. Function update is written $f\ (x := y)$ where $f :: 'a \Rightarrow 'b$, $x :: 'a$ and $y :: 'b$, which is equivalent to $\lambda\ a\,.\ \mathsf{if}\ a = x\ \mathsf{then}\ y\ \mathsf{else}\ f\ a$.

Maps. Partial functions (*maps*) are modeled out of total functions returning an *option* type: $'a \Rightarrow 'b\, option$ also written $'a \rightharpoonup 'b$; $f\ x = Some\ y$ means the function f is defined in x and *maps to* y, whereas $f\ z = None$ denotes undefinedness in z. Partial function update is written $f(x \mapsto y)$ which is equivalent to $f\ (x := Some\, y)$. For convenience, several consecutive updates would be written $f(x_1 \mapsto y_1, \ldots, x_n \mapsto y_n)$; alternatively $[\mapsto]$ performs bulk updates using lists $f([x_1, \ldots, x_n][\mapsto][y_1, \ldots, y_n])$. The empty map $\lambda a.None$ is noted *empty*; it is updated by $[x \mapsto y]$.

3.3 Abstract Syntax.

Types. Jinja uses the following variable conventions: V is a variable name, F is a field name, M is a method name, C is a class name, e is an expression, v is a value, T is a type and P is a program. Class names, method names and variable names have respectively the type *cname*, *mname* and *vname*, which are all synonyms for the HOL type *string*. We typeset strings either with double quotes "*a string*" or underlined *AnotherString*.

Values. Values in Jinja (HOL type *val*) use type constructors associated with a HOL value. Types range over booleans (*Bool* b), integers (*Intg* i), references (*Addr* a and the null reference *Null*) and the dummy value *Unit*, where $b :: bool$, $i :: int$, $a :: addr$. The address type *addr* is synonym for *nat*. Jinja types, of type *ty*, are the type *Void* for *Unit*, the primitive types *Boolean* and *Integer* for *Bool* and *Intg*, the null reference type *NT*, and *Class C* for classes references.

datatype $val = Unit \mid Bool\ bool \mid Intg\ int \mid Null \mid Addr\ addr$
datatype $ty = Void \mid Boolean \mid Integer \mid NT \mid Class\ cname$

Expressions. The Jinja source language is expression-based. Expressions, of type *expr* (notation e) are presented in Table 3.2. Every expression in Jinja reduces to a final expression that is either a value `Val` v or an exception `throw` (`Val` (*Addr* a)). Statement expressions evaluate to `Val` *Unit*. User defined exceptions may throw any arbitrary object. There are also system exceptions throwing an instance of one of the system exception classes: $sys\text{-}xcpts \equiv \{NullPointer, ClassCast, OutOfMemory\}$. Pre-allocated instances of the system exception classes are available through the function $addr\text{-}of\text{-}sys\text{-}xcpt :: cname \Rightarrow addr$. To shorten the notation of literal values and exceptions, Jinja introduces the following short-hands:

$true \equiv \texttt{Val}\ (Bool\ True) \qquad false \equiv \texttt{Val}\ (Bool\ False)$
$addr\ a \equiv \texttt{Val}\ (Addr\ a) \qquad null \equiv \texttt{Val}\ Null$
$unit \equiv \texttt{Val}\ Unit$
$\texttt{Throw}\ a \equiv \texttt{throw}\ (addr\ a) \qquad \texttt{THROW}\ C \equiv \texttt{Throw}\ (addr\text{-}of\text{-}sys\text{-}xcpt\ C)$

Program representation. A Jinja program (of type *'m prog*, Figure 3.3) is a list of class declarations. Each class declaration (type *'m cdecl*) is made of a class name and a class definition (type *'m class*). A class is defined with the name of its super-class, a list of field declarations (each of type *fdecl*) and the list of method declarations. A field declares a name

expression	description
`new` C	object allocation
`Cast` $C\ e$	casting
`Val` v	literal value
$e_1 \ll bop \gg e_2$	binary operation
`Var` V	variable access
$V := e$	variable assignment
$e.F\{D\}$	field access
$e_1.F\{D\} := e_2$	field assignment
$e.M(es)$	method call
$\{V : T;\ e\}$	block with locally declared variable
$e_1 ; e_2$	sequential composition
`if` $(e)\ e_1$ `else` e_2	conditional
`while` $(e)\ e'$	while loop
`throw` e	exception throwing
`try` e_1 `catch` $(C\ V)\ e_2$	exception catching

Table 3.2: Jinja source language expressions.

and a Jinja type. A method declaration (type *'m mdecl*) comprises the method name, the parameter types, the result type and the method body (type *'m*). In Siaam we make the method body optional in a method declaration. Native methods, introduced in [71, §2.1.3], only declare a signature along with a *None* body, written *Native*. The parametrization of the method body type is a convenience that allows us to specialize the program type for the Jinja source language and later for the virtual machine language. The method body for the source language (type *J-mb*) is made of the list of formal parameters names and the top-level expression. The corresponding program type *J-prog* is the instantiated type *J-mb prog*.

$$
\begin{array}{lrcl}
\textbf{type_synonym} & 'm\ prog & = & 'm\ cdecl\ list \\
 & 'm\ cdecl & = & cname \times 'm\ class \\
 & 'm\ class & = & cname \times fdecl\ list \times 'm\ mdecl\ list \\
 & fdecl & = & vname \times ty \\
 & 'm\ mdecl & = & mname \times ty\ list \times ty \times 'm\ option \\
 & & & \\
 & J\text{-}mb & = & vname\ list \times expr \\
 & J\text{-}prog & = & J\text{-}mb\ prog
\end{array}
$$

Figure 3.3: Program declaration.

In Figures 3.4 and 3.5 are demonstrated a small program in Java and its equivalent in the Jinja source code. The top-level definition is *program* :: *J-prog*, which comprize the declaration for the three classes Object, A and B. Since the Object class is the at the top of the hierarchy, it has an empty superclass name. Assuming the *hashCode* and *toString* methods of the Object class are implemented natively, both declarations have the *Native* method body.

Focusing on the Jinja declaration for A, appear in order: the name of the class, the name of its superclass (Object), the field declarations list containing *"fi"* and *"fo"*, and finally the list of methods declared in A. The initializer method name is arbitrarily translated to

```
1 class Object {
2   Object() {}
3   int hashCode() {...}
4   String toString() {...}
5 }

1 class B extends A {
2   int fi;
3   B(int i) {
4     super(i+1, null);
5     this.fi = i;
6   }
7   int getI() { return this.fi; }
8   int swapI(int i) {
9     int tmp = this.fi;
10    this.fi = i;
11  return tmp; }
12 }
```

```
1 class A extends Object {
2   int fi; Object fo;
3   A(int i, Object o) {
4     super();
5     this.fi = i;
6     this.fo = o
7   }
8   int getI() { return this.fi; }
9   Object getO() { return this.fo; }
10  int swapI(int i) {
11    int tmp = this.fi;
12    this.fi = i;
13    return tmp;
14  }
15  void main(String arg) {
16    B b = new B(0);
17  }
18 }
```

Figure 3.4: Small program.

"<init>", though it is the name used internaly in Java.

A and B both declare an integer field named *"fi"*. Since B extends A, instances of the former actually have two *"fi"* fields; the one inherited from A is accessible through the notation $e.$*"fi"*$\{$*"A"*$\}$, wherease the one declared in B is accessible through $e.$*"fi"*$\{$*"B"*$\}$, where e is an expression evaluating to the address of an object of class B.

Lookups predicates. Jinja defines a set of predicates for accessing program declarations. In a program P, the subclass relation $P \vdash D \preceq^\star C$ means D is a subclass of class C. The subtype relation $P \vdash _ \leq^\star _ :: {}'x\ prog \Rightarrow ty \Rightarrow ty \Rightarrow bool$ is defined as:

$$P \vdash D \preceq^\star C \longrightarrow P \vdash Class\ D \leq^\star Class\ C$$

The type of the *null* reference is a subclass of all classes:

$$P \vdash NT \leq^\star Class\ C$$

The method lookup taking overriding into account is written $P \vdash C\ \mathsf{sees}\ M : Ts \rightarrow T = mb\ \mathsf{in}\ D$ and means that starting from class C and scanning the class hierarchy upward, a method named M is visible in class D. The visible method declares the arguments types Ts, the return type T and the optional method body $mb :: {}'m\ option$. The predicate $P \vdash C\ \mathsf{has\text{-}fields}\ FDTs$ verifies that class C and its superclasses declares the fields $FDTs :: ((vname \times cname) \times ty)\ list$. For a given class, the function $fields :: prog \Rightarrow cname \Rightarrow ((vname \times cname) \times ty)\ list$ retrieves the corresponding $FDTs$.

As an exercise, we apply these lookup predicates to the example Figure 3.5. In the class hierarchy, Object is at the top, A directly sublasses Object, and B is a direct subclass of A and by transitivity a subclass of Object. Also note that every class is a subclass of itself.

$$\begin{array}{ll} P \vdash \underline{A} \preceq^\star \underline{Object} & P \vdash \underline{B} \preceq^\star \underline{A} \\ P \vdash \underline{B} \preceq^\star \underline{Object} & \forall C \in \{\underline{Object}, \underline{A}, \underline{B}\}, P \vdash C \preceq^\star C \end{array} \tag{3.3}$$

$$
\begin{aligned}
&program = [ObjectC,\ AC,\ BC]\\
\\
&ObjectC = (\underline{Object},\ \text{“”},\ [\,],\\
&\qquad [(\underline{<init>},\ [\,],\ Void,\ \lfloor([\,],\ e_1)\rfloor),\\
&\qquad\ (\underline{hashCode},\ [\,],\ Integer,\ Native),\\
&\qquad\ (\underline{toString},\ [\,],\ Class\ \underline{String},\ Native)\\
&\qquad])\\
\\
&AC = (\underline{A},\ \underline{Object},\ [(\underline{fi},\ Integer),\ (\underline{fo},\ Class\ \underline{Object})\ ],\\
&\qquad [(\underline{<init>},\ [Integer,\ Class\ \underline{Object}],\ Void,\ \lfloor([\underline{i},\ \underline{o}],\ e_4)\rfloor),\\
&\qquad\ (\underline{getI},\ [\,],\ Integer,\ \lfloor([\,],\ e_5)\rfloor),\\
&\qquad\ (\underline{getO},\ [\,],\ Class\ \underline{Object},\ \lfloor([\,],\ e_6)\rfloor),\\
&\qquad\ (\underline{swapI},\ [Integer],\ Integer,\ \lfloor([\underline{i}],\ e_7)\rfloor),\\
&\qquad\ (\underline{main},\ [Class\ \underline{String}],\ Void,\ \lfloor([\underline{arg}],\ e_8)\rfloor)\\
&\qquad])\\
\\
&BC = (\underline{B},\ \underline{Object},\ [(\underline{fi},\ Integer)\ ],\\
&\qquad [(\underline{<init>},\ [Integer],\ Void,\ \lfloor([\underline{i}],\ e_9)\rfloor),\\
&\qquad\ (\underline{getI},\ [\,],\ Integer,\ \lfloor([\,],\ e_{10})\rfloor),\\
&\qquad\ (\underline{swapI},\ [Integer],\ Integer,\ \lfloor([\underline{i}],\ e_{11})\rfloor)\\
&\qquad])
\end{aligned}
\tag{3.2}
$$

Figure 3.5: Jinja representation of the example Java program of Figure 3.4. Not showing the declaration for the String class.

The *hashCode* and *toString* methods declared by Object are visible from A and B since none of them override the methods. The *getI* and *swapI* method declared by B take precedence over those declared by A when looking from B. The *main* method declared in A is not visible from the Object class.

$$
\begin{aligned}
&\forall C \in \{\underline{Object}, \underline{A}, \underline{B}\},\\
&P \vdash C \text{ sees } \underline{hashCode} : [\,] \rightarrow Integer = Native \text{ in } \underline{Object}\\
&P \vdash C \text{ sees } \underline{toString} : [\,] \rightarrow Class\ \underline{String} = Native \text{ in } \underline{Object}\\
\\
&P \vdash \underline{B} \text{ sees } \underline{swapI} : [Integer] \rightarrow Integer = \lfloor([\underline{i}], e_{11})\rfloor \text{ in } \underline{B}\\
&\neg P \vdash \underline{B} \text{ sees } \underline{swapI} : [Integer] \rightarrow Integer = _ \text{ in } \underline{A}\\
&P \vdash \underline{B} \text{ sees } \underline{getO} : [\,] \rightarrow Class\ \underline{Object} = \lfloor([\,], e_6)\rfloor \text{ in } \underline{A}\\
\\
&P \vdash \underline{A} \text{ sees } \underline{main} : [Class\ \underline{String}] \rightarrow Void = \lfloor([\underline{arg}], e_8)\rfloor \text{ in } \underline{A}\\
&\neg P \vdash \underline{Object} \text{ sees } \underline{main} : [Class\ \underline{String}] \rightarrow Void = _ \text{ in } \underline{A}
\end{aligned}
\tag{3.4}
$$

The predicate for fields lookup collects all fields declared from a specified class upward to the top of the hierarchy. Therefore the Object class has no fields, A has the two fields it declares and B has the fields it declares plus those inherited from A.

$$
\begin{aligned}
&P \vdash C \text{ has-fields } FDTs \longrightarrow FDTs = fields\ P\ C\\
&fields\ P\ \underline{Object} = [\,]\\
&fields\ P\ \underline{A} = [((\underline{fi}, \underline{A}), Integer), ((\underline{fo}, \underline{A}), Class\ \underline{Object})]\\
&fields\ P\ \underline{B} = ((\underline{fi}, \underline{B}), Integer) \cdot fields\ P\ \underline{A}
\end{aligned}
\tag{3.5}
$$

3.3.1 Java's features not offered by Jinja

Jinja only models a subset of the Java language. It lacks of class member qualifiers such as `static`, `final`, `private`, `protected`, and `public`. Classes in Jinja cannot declare or implement interfaces as in Java using the `interface` and `implements` keywords. The original Jinja semantics does not encompass any concurrent construction, the Java Memory Model and the multithreading support are described in [71]. The latter work also extends Jinja with arrays. Finally, heap garbage collection is out of the scope of Jinja, the maximum heap size is undefined, but the dynamic allocation expression `new` C may reduce to the *OutOfMemory* exception.

3.4 Siaam Global Semantics

We divide the Siaam semantics between the *single-actor* semantics, and the *global* semantics providing isolation and communication mechanisms for the actors. The single-actor semantics reduces the *actor-local state* and the global semantics maintains a *global state* not directly accessible from the single-actor semantics. In particular, actors cannot read nor update objects' fields without consent of the global semantics, and communications are handled by the global semantics as well. The single-actor semantics can emit *actor actions* in order to request intervention of the global semantics when it has to perform an operation involving the global state. Actor actions express global preconditions that are not always immediately satisfiable or may even never be, therefore the single-actor semantics should offer more than one action if it doesn't want to block indefinitely. Most available actions come by pairs with dual preconditions so there is always an alternative with satisfiable preconditions. For instance, an actor willing to read an object's field should expect the cases when it owns the objects and when it doesn't. To do so, the actor must offer two actions, each satisfied by a different case. The first action would reduce to the state where the value is retrieved, and the second one would reduce to an exception.

Hence, out of global context, the single-actor semantics is non-deterministic since for a fixed actor-local state it may offer several actions, each one reducing to a different state. However the semantics and the actions are designed such that once the global semantics is introduced, there is at most one action satisfiable for a given actor at each step. A step of the global semantics picks an actor offering a satisfiable action, applies the corresponding single-actor reduction and updates the global state accordingly. Globally, picking an actor is non-deterministic, in practice there would be some kind of scheduler enforcing an interleaving policy. Nevertheless, once an actor has been chosen, there is only one reduction carrying a satisfiable action, so the actor-local evaluation is completely deterministic after-all.

Section 3.4.1 presents the API of Siaam; in section 3.4.2 we define the global state and section 3.4.3 gives the associated semantics. The available actor actions are described in section 3.4.5 after some major functions related to transitive ownership verifications and transfers are detailed in section 3.4.4. The semantics for native calls to the Siaam API are detailed in section 3.4.6. We defer the addressing of the single-actor semantics to sections 3.5 for the source language and 3.6 for the virtual machine.

3.4.1 Siaam Native API.

Actor-oriented operations accessing and updating the global state are not implementable with the Java syntax. Instead of extending the Jinja language with specific expressions, Siaam's actor features are available through native methods of certain system classes and hard-wired in the dedicated *native semantics* addressed subsequently in section 3.4.6. The signatures for native methods defined by Siaam are shown in Figure 3.6. Actors are reified by objects sub-classing the *Actor* class. The behaviour of an actor of class $A \preceq^\star Actor$ is implemented in its *run* method of signature: $A.run([\,]) :: Void$. When the *start* method is called on an actor object for the first time, a new instance of the single-actor semantics starts executing the body of *run*, isolated from the other concurrent actors of the system. *Object.currentActor* returns a reference to the object representing the actor currently executing. Since Jinja lacks of static methods, we make *currentActor* an instance method of *Object* that can be invoked with any receiver. Objects of the special *Mailbox* class represent one-way communication mailboxes. User classes are not allowed to declare subclasses of *Mailbox* and the class itself doesn't declare any field. The *put* and *get* methods respectively sends and receives a message. There is a unique recipient actor associated to each mailbox that is set with *setReceiver* and cannot be changed afterward. Finally, it is an error to send an active actor as part of a message.
We also define a few exception classes and add them to the set of system exception classes; the *OwnerMismatch* exception is thrown when an actor attempts to access, update or send an object it doesn't own, or when it isn't allowed to configure a mailbox; *ReceiverMismatch* is thrown when an actor isn't allowed to retrieve a message from a mailbox.

$$
\begin{array}{c}
Actor.start([\,]) :: Void \\
Object.currentActor([\,]) :: Class\ Actor \\
Mailbox.setReceiver([Class\ Actor]) :: Void \\
Mailbox.put([Class\ Object]) :: Void \\
Mailbox.get([\,]) :: Class\ Object \\
sys\text{-}xcpts \equiv \{NullPointer, ClassCast, OutOfMemory, \\
\quad OwnerMismatch, ReceiverMismatch\}
\end{array}
$$

Figure 3.6: Signatures of native methods for Siaam actor-based concurrency

3.4.2 Global State.

The global state (xs, ws, ms, h), defined in equation (3.6), comprises four components: a store of actors xs, an ownership relation ws, a store of mailboxes ms, and a shared heap h. The projection functions *acs*, *ows*, *mbs*, *shp* return respectively the actors store, the ownerships relation, the mailboxes store, and the shared heap component of the global state.

$$
\begin{aligned}
\textbf{type_synonym} \quad {}'x\ state &= {}'x\ actors \times ownerships \times mailboxes \times heap \\
{}'x\ actors &= owner \rightharpoonup {}'x \\
ownerships &= addr \rightharpoonup owner \\
mailboxes &= addr \rightharpoonup (addr\ list \times owner\ option) \\
heap &= addr \rightharpoonup obj \\
obj &= cname \times fields \\
fields &= (vname \times cname) \rightharpoonup val
\end{aligned}
\tag{3.6}
$$

Shared heap. The shared heap is a map from addresses to objects. Jinja's objects (type *obj*) are pairs (C, fs) of the object's class name C and the fields table fs. A field table is a map holding value (type *val*) for each field of an object, where fields are identified by pairs $(F,\ D)$ of the field's name F and the name D of the declaring class.

Ownership relation. Objects belong to actors, which are represented by heap objects as well. The ownership relation (type *ownerships*) is a map from addresses of objects to the identifier of their respective owner. Although the identifier of an owner may simply be the address of an actor object, we prefer to introduce the opaque *owner* type, which will be used to denote owner-IDs. Owner identifiers (notation convention w) come with the operations $w2a :: owner \Rightarrow addr$ and $a2w :: addr \Rightarrow owner$ associating exactly one unique owner identifier to each address. We say an object of address a currently *belongs to* an owner w if the ownership relation maps a to w, and symmetrically we say w *owns* the object at a.

Actors store. The actors store (type $'x$ *actors*) maps from actor owner-IDs to local-states of parametric type $'x$. Basically, if w is the owner-ID of an object reifying an actor, then $xs\ w$ is the actor-local state of this actor in the current single-actor semantics. The global state type is parametrized by $'x$ so it may be instantiated for various single-actor semantics with custom actor-local state definitions.

Mailboxes store. The store of mailboxes (type *mailboxes*) is a map from *Mailbox* objects addresses to pairs of a messages queue and a receiver owner-identifier. The queue is a list of addresses corresponding to starting references of pending messages.
The predicate *is-initialized-mailbox* $ms\ a$ verifies that the object a is a mailbox associated with a receiver actor in the store ms.

$$
\begin{aligned}
&\textit{is-initialized-mailbox} :: mailboxes \Rightarrow addr \Rightarrow bool \\
&\textit{is-initialized-mailbox}\ ms\ a = \mathsf{case}\ ms\ a\ \mathsf{of}\ None \Rightarrow False \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mid \lfloor(_,\ None)\rfloor \Rightarrow False \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mid \lfloor(_,\ \lfloor_\rfloor)\rfloor \Rightarrow True
\end{aligned}
\tag{3.7}
$$

3.4.3 Global Semantics.

We give a definition of the global semantics parametrized by a single-actor semantics function $r :: ('m, 'x)$ *single-semantics* where $'m$ is the type for representing method bodies, and $'x$ is the type for representing actor-local state. A reduction of the single-actor semantics computed by $r\ P\ w\ (x, h)\ wa\ (x', h')$ is written $P, w \vdash (x, h) -wa\rightarrow (x', h')$, meaning in program P, the actor w reduces its local state x and the shared heap h to the local state x' and the

$$\textsc{Global}\ \frac{\begin{array}{cc} acs\ s\ w = \lfloor x \rfloor & P, w \vdash (x, shp\ s) -wa\rightarrow (x', h') \\ ok\text{-}act\ P\ s\ w\ wa & upd\text{-}act\ P\ s\ w\ wa = (xs', ws', ms', _) \\ \multicolumn{2}{c}{s' = (xs'(w \mapsto x'), ws', ms', h')} \end{array}}{P \vdash s \rightarrow s'}$$

Figure 3.7: Global semantics reduction rule.

shared heap h', performing action $wa :: {}'x\ actor\text{-}action$. The definition of the *actor-action* type is delayed to Section 3.4.5.

$$\textbf{type_synonym} \quad ({}'m, {}'x)\ single\text{-}semantics = {}'m\ prog \Rightarrow owner \Rightarrow {}'x \times heap \Rightarrow {}'x\ actor\text{-}action \Rightarrow {}'x \times heap \Rightarrow bool \tag{3.8}$$

$$\textbf{locale}\ global\text{-}semantics = \ \text{fixes}\ r :: ({}'m, {}'x)\ single\text{-}semantics\ (_, _ \vdash (_, _) -_\rightarrow (_, _)) \tag{3.9}$$

The **locale**[13] notation is borrowed from HOL/Isabelle where it is used to define parametric modules, which is exactly the purpose here. We instantiate a locale with an interpretation of its free variables using the keyword **interpretation**. The *global-semantics* is interpreted twice in this document, first with the definitions of $'x$, $'m$ and r from the single-actor small steps semantics and then with the definitions from the virtual machine semantics.

The reduction rule for the global semantics is shown in Figure 3.7. The judgment, written $P \vdash s \rightarrow s'$, means in program P, global state s reduces to global state s'. Global reduces the global state by applying a single step of the single-actor semantics for the actor of owner-ID w. The shared heap h and the current local state x for w are retrieved from the global state. Then if the actor can reduce to x' with new shared heap h' and perform the action wa, *ok-act* tests the action precondition against s. If the actor action is satisfiable, *upd-act* applies the effects of wa to the global state, yielding the new tuple of state components $(xs', ws', ms', _)$ where the heap is actually left unchanged. The new state is assembled from the new mailbox store, the new ownership relation, the new heap from the single actor reduction and the new actor store where the state for the actor is updated with its new local state. Later we will instantiate the global semantics with a single-actor semantics r corresponding either to the source language or the virtual machine.

3.4.4 Transitive ownership verification and transfer.

Siaam's message-passing scheme relies on dynamic ownership verification and transfer of arbitrary graph of objects. In this section, we present the two primitives handling the verification and the transfer of ownership in Siaam.

Setup. In a program P, the function *addr-fields* $P\ C$ returns the list of reference-typed fields (F, D) that an object of type *Class* C carries, where F is the name of a field declared in a superclass $C \preceq^\star D$.

$$\begin{array}{l} addr\text{-}fields :: {}'m\ prog \Rightarrow cname \Rightarrow (vname \times cname)\ list \\ addr\text{-}fields\ P\ C = \text{let}\ FDTs = fields\ P\ C \\ \qquad\qquad \text{in}\ map\ fst\ (filter\ (\lambda fdt.\ snd\ fdt = (Class\ _))\ FDTs) \end{array} \tag{3.10}$$

Given the heap h and the address a of an object, *non-null-addrs* $P\ h\ a$ (3.11) produces the list of possibly duplicated addresses contained in a's fields that are not *Null*. The class name C and the field table *fs* of a are retrieved from h. Then *FDs* is assigned with the result of calling *addr-fields* with C. Each field declaration in *FDs* is mapped in the list *vs* with the value (either *Addr* a' or *Null*) found in a's field table at the corresponding slot. Elements of *vs* are filtered so that remain only the non-*Null* address values in *As*. Finally the list of addresses (type *addr list*) is obtained by destructing each element of *As* with *the-Addr* (*Addr* a') $= a'$. It is worth noting that *non-null-addrs* always returns the empty list when the passed address refers to a *Mailbox* object since they don't declare any field.

$$
\begin{array}{l}
\mathit{non\text{-}null\text{-}addrs} :: {}'m\ \mathit{prog} \Rightarrow \mathit{heap} \Rightarrow \mathit{addr} \Rightarrow \mathit{addr\ list}\\
\mathit{non\text{-}null\text{-}addrs}\ P\ h\ a =\\
\quad \mathsf{let}\ \lfloor (C,\ \mathit{fs}) \rfloor = h\ a;\\
\qquad \mathit{FDs} = \mathit{addr\text{-}fields}\ P\ C;\\
\qquad \mathit{vs} = \mathit{map}\ (\lambda \mathit{fd}.\ \mathit{the}\ (\mathit{fs}\ \mathit{fd}))\ \mathit{FDs};\\
\qquad \mathit{As} = \mathit{filter}\ (\lambda v.\ v \neq \mathit{Null})\ \mathit{vs}\\
\quad \mathsf{in}\ \mathit{map}\ \mathit{the\text{-}Addr}\ \mathit{As}
\end{array}
\tag{3.11}
$$

In the following HOL definitions, the predicate *exists* $P\ xs$ verifies the existence of an element of the list xs satisfying the predicate P; *concat* concatenates a list of lists, and *distincts* reduces duplicated elements of a list.

Transitive ownership checking. The transitive ownership checking (3.12) is achieved by *transitive-owner-check* $P\ s\ w\ a$, verifying that every object transitively reachable from the root address a belongs to w according to the ownership relation in the current global state s, provided there is a path in the shared heap from a to the object, that doesn't comprise an initialized mailbox.

$$
\begin{array}{l}
\mathit{transitive\text{-}owner\text{-}check} :: {}'m\ \mathit{prog} \Rightarrow {}'x\ \mathit{state} \Rightarrow \mathit{owner} \Rightarrow \mathit{addr} \Rightarrow \mathit{bool}\\
\mathit{transitive\text{-}owner\text{-}check}\ P\ s\ w\ a = \mathit{twc}\ P\ s\ w\ [a]\ (\mathit{dom}\ (\mathit{shp}\ s) \setminus \{a\})\\
\\
\mathit{twc} :: {}'m\ \mathit{prog} \Rightarrow {}'x\ \mathit{state} \Rightarrow \mathit{owner} \Rightarrow \mathit{addr\ list} \Rightarrow \mathit{addr\ set} \Rightarrow \mathit{bool}\\
\mathit{twc}\ P\ (\mathit{xs},\ \mathit{ws},\ \mathit{ms},\ h)\ w\ \mathit{roots}\ \mathit{unreached} =\\
\quad \mathsf{let}\ \mathit{roots}' = \mathit{filter}\ (\lambda\ a\,.\,\neg\ \mathit{is\text{-}initialized\text{-}mailbox}\ \mathit{ms}\ a)\ \mathit{roots}\\
\quad \mathsf{in\ if}\ \mathit{exists}\ (\lambda a.\ \mathit{ws}\ a \neq \lfloor w \rfloor \vee \mathit{a2w}\ a = w)\ \mathit{roots}'\ \mathsf{then}\ \mathit{False}\\
\qquad \mathsf{else\ let}\ \mathit{reached} = \mathit{distincts}\ \mathit{concat}\ (\mathit{map}\ \mathit{non\text{-}null\text{-}addrs}\ \mathit{roots}')\ ;\\
\qquad\qquad\quad \mathit{newroots} = \mathit{filter}\ \mathit{unreached}\ \mathit{reached}\\
\qquad\quad \mathsf{in\ if}\ |\mathit{newroots}| = 0\ \mathsf{then}\ \mathit{True}\\
\qquad\qquad \mathsf{else}\ \mathit{twc}\ P\ (\mathit{xs},\ \mathit{ws},\ \mathit{ms},\ h)\ w\ \mathit{newroots}\ (\mathit{unreached} \setminus \mathit{set}\ \mathit{newroots})
\end{array}
\tag{3.12}
$$

The auxiliary, tail-recursing function *twc* explores the object graph from a and performs the verification for each node. It takes the same arguments as *transitive-owner-check*, plus the list of reached-but-not-yet-verified object addresses (*roots* :: *addr list*) and the set of not-yet-reached addresses *unreached* :: *addr set*. Initially, the roots list comprises the root address a and the unreached set contains every addresses in the domain of the heap but a; the verification succeeds when the roots list is empty. *twc* maintains the invariant $(\mathit{set}\ \mathit{roots}) \cap \mathit{unreached} = \emptyset$.

Each iteration verifies the roots either belong to w or are initialized mailboxes, and then looks for new reachable objects to verify. First the function filters-out initialized mailboxes roots, and verifies the ownership of the retained objects. If at least one of them doesn't belong

to w the function immediately returns *False*. If the object with the owner-ID matching the current owner w is reached, the function returns *False* as well. Indeed, the checking must fail if it can reach the current actor to prevent an active actor from transferring itself.

When all the roots have passed the checking, the addresses of objects directly reachable from the roots are collected. Among these, it retains addresses of not-yet-reached objects (appearing in *unreached*). The retained addresses form a new list of roots, yet unverified. An empty list means the complete closure of the initial a root has been explored and successfully verified and the function returns *True*. Otherwise *twc* recurses with the list of new unverified roots and the set of unreached objects addresses where the new roots have been removed.

Termination of *twc* is ensured since at each iteration, either it returns *False*, or no new root can be found in *unreached* and it returns *True* or else *unreached* strictly decreases.

Transitive ownership transfer. Assuming the graph of objects to be transferred has already been checked by *transitive-owner-check*, transferring the ownership of the graph is straightforward. The initial ownership relation is progressively updated as objects with an ownership mismatching the target are encountered while exploring the graph. Since the graph has been checked previously, we know that initially all the reachable objects belong to the same owner, except the initialized mailboxes.
The application *transitive-owner-transfer* P s *target* a (3.13) transfers the ownership of every object reachable from address a to the *target* owner, discarding initialized mailboxes. The target owner may be *None* or some owner-ID.

The auxiliary function *twt* takes a non-empty list *roots* of addresses of objects to transfer. Initially the list only contains a, unless it corresponds to an initialized mailbox or an object already owned by the target. However the latter case never happens because we always transfer ownership between different owner-IDs.

First the ownership relation *ws* is updated so that all the root objects maps to *target* in ws'. Then the function collects the addresses of every objects directly reachable from these roots and filters-out those already transferred to *target* and those corresponding to initialized mailboxes. If no addresses remains, all the transferable objects have been transferred so the function returns ws'. Otherwise *twt* recurses with the updated ownership relation and the addresses list of newly reached and not-yet-transferred objects.

The *roots* list is never empty and all its elements refer to objects not belonging to *target*, so at each iteration of *twt* at least one object of the heap is transferred to the target. Therefore the termination of *twt* is ensured since the heap contains a finite set of objects.

$$
\begin{array}{l}
\textit{transitive-owner-transfer} :: \textit{'m prog} \Rightarrow \textit{'x state} \Rightarrow \textit{owner option} \Rightarrow \textit{addr} \Rightarrow \textit{ownerships} \\
\textit{transitive-owner-transfer}\ P\ s\ \textit{target}\ a = \\
\quad \textsf{if}\ (\textit{is-initialized-mailbox}\ ms\ a \vee ws\ a = \textit{target})\ \textsf{then}\ \textit{ows}\ s\ \textsf{else}\ \textit{twt}\ P\ s\ \textit{target}\ [a] \\
\\
\textit{twt} :: \textit{'m prog} \Rightarrow \textit{state} \Rightarrow \textit{owner option} \Rightarrow \textit{addr list} \Rightarrow \textit{ownerships} \\
\textit{twt}\ P\ (xs,\ ws,\ ms,\ h)\ \textit{target}\ \textit{roots} = \\
\quad \textsf{let}\ ws' = ws(\textit{roots}\,[\mapsto]\,\textit{replicate}\ |\textit{roots}|\ \textit{target}) \\
\qquad \textit{reached} = \textit{distincts concat}\ (\textit{map non-null-addrs roots})\ ; \\
\qquad \textit{newroots} = \textit{filter}\ (\lambda\ a\,.\ ws'\ a \neq \textit{target} \wedge \neg\ \textit{is-initialized-mailbox}\ ms\ a)\ \textit{reached} \\
\quad \textsf{in if}\ |\textit{newroots}| = 0\ \textsf{then}\ ws' \\
\qquad \textsf{else}\ \textit{twt}\ P\ (xs,\ ws',\ ms,\ h)\ \textit{target}\ \textit{newroots}
\end{array}
\tag{3.13}
$$

$$\textbf{datatype}\quad {}'x\ \textit{actor-action} = \textit{Silent} \mid \textit{NewObj cname addr} \mid \textit{OwnerCheck addr bool} \mid \textit{Start }'x\textit{ addr} \mid \textit{StartFail addr} \mid \textit{SetReceiver addr owner} \mid \textit{SetReceiverFail addr} \mid \textit{Send addr addr} \mid \textit{SendFail addr} \mid \textit{Receive addr addr} \mid \textit{ReceiveFail addr}$$

Figure 3.8: Actor actions syntax

3.4.5 Actor actions.

Siaam defines ten actor actions (Figure 3.8) plus the always-satisfiable action identified by the *Silent* element. Actor actions allow expressing conditions and updates on the global state from the single-actor semantics. When the global semantics make a step, it picks an actor reduction with a satisfiable condition and updates the global state accordingly.

Informal overview. We first give an informal description of each actor action. The *NewObj* C a updates the ownership relation to record the fact that the object of class C, allocated at address a belongs to the current actor. The owner checking action *OwnerCheck* a *True* requires that the current actor is the owner of the object at address a. Symmetrically, *OwnerCheck* a *False* is the predicate requiring that the object at address a doesn't belong to the current actor. The *Start* x a action starts the actor object of address a by calling it's *run* method within a new isolated local-state. x is employed to pass the actual initial local-state of the actor, its concrete type differs in the source language and the bytecode, thus it is parametrized by the $'x$ type variable. The dual exceptional action *StartFail* a expresses a failure in the process of starting the actor at a. The *SetReceiver* a w action sets the receiver of the mailbox at address a with the actor of owner identifier w. The dual exceptional action *SetReceiverFail* a is satisfiable when the current actor would fail at configuring a receiver for the mailbox a. The *Send* a a' action sends the message starting with the object of address a' through the mailbox a. The dual exceptional action *SendFail* a' is available when the current actor would fail at sending the message starting with a'. Finally, the *Receive* a a' action receive the underspecified message starting with the object of address a' from the mailbox at a. When receiving from a would fail, the exceptional action *ReceiveFail* a is satisfiable.

Each Siaam's actor action is associated with a precondition evaluation function *ok-act* and a global state update function *upd-act* (3.14). Both functions takes a global state s, the owner-identifier w of the current actor, and the actor action issued. *ok-act* evaluates to a boolean indicating whether the condition associated with the actor action is met. *upd-act* computes an updated global state accordingly to the specified action.

$$\begin{aligned}&\textit{ok-act} :: {}'m\ \textit{prog} \Rightarrow {}'x\ \textit{state} \Rightarrow \textit{owner} \Rightarrow {}'x\ \textit{actor-action} \Rightarrow \textit{bool}\\&\textit{upd-act} :: {}'m\ \textit{prog} \Rightarrow {}'x\ \textit{state} \Rightarrow \textit{owner} \Rightarrow {}'x\ \textit{actor-action} \Rightarrow {}'x\ \textit{state}\end{aligned} \tag{3.14}$$

Silent action. As expected, the precondition for the *Silent* action always evaluates to *True* and the update function returns the global state unmodified.

$$\begin{array}{l} \textit{ok-act}\ P\ s\ w\ \textit{Silent} = \textit{True} \\ \textit{upd-act}\ P\ s\ w\ \textit{Silent} = s \end{array} \tag{3.15}$$

Object allocation. The *NewObj C a* action is always satisfiable, it has the effect of associating address a with the current owner-ID in the ownership relation. When the new object is a mailbox, the mailboxes store is updated to associate the couple of an empty message queue and the undefined receiver to the new mailbox.

$$\begin{array}{l} \textit{ok-act}\ P\ s\ w\ (\textit{NewObj}\ C\ a) = \textit{True} \\ \textit{upd-act}\ P\ (xs,ws,ms,h)\ w\ (\textit{NewObj}\ C\ a) = \\ \quad \mathsf{let}\ ws' = ws(a \mapsto w)\ \mathsf{in} \\ \quad\quad \mathsf{case}\ C\ \mathsf{of}\ \textit{Mailbox} \Rightarrow (xs,\ ws',\ ms(a \mapsto ([\,],\ \textit{None})),\ h) \\ \quad\quad \mid\ _ \Rightarrow (xs,\ ws',\ ms,\ h) \end{array} \tag{3.16}$$

Owner checking. The precondition for *OwnerCheck a b* starts by matching the current owner-ID w with the actual owner of a recorded in the ownership relation *ows s* of the global state. If both the result of the matching and the boolean b equal, then the precondition evaluates to *true*, otherwise it evaluates to *false*.

$$\begin{array}{l} \textit{ok-act}\ P\ s\ w\ (\textit{OwnerCheck}\ a\ b) = ((\textit{ows}\ s\ a = \lfloor w \rfloor) = b) \\ \textit{upd-act}\ P\ s\ w\ (\textit{OwnerCheck}\ a\ b) = s \end{array} \tag{3.17}$$

Actor start. The *Start x a* action is a little more complex: the current actor must withdraw the ownership over the new actor's private state starting from the object at address a. The precondition is that the current actor owns every object reachable from a and is not itself aliased in the message. The latter condition, verified by *transitive-owner-check* also avoid attempts to start the current actor in case *a2w* $a = w$. The native calls semantics, presented in the next section, has the responsibility to make sure the object pointed-to by a is indeed an instance of the Actor class.

If the conditions are met, a new global state is computed by *upd-act* as following. The ownership relation is updated so that the current actor withdraw its ownership over the message starting at a, and transfers the ownership to the starting actor. A new entry is created in the actor store that maps the owner-ID *a2w* a of the starting actor to the initial state x.

The *StartFail a* action has the exact opposite preconditions as *Start x a*, and has no effect on the global state.

$$\begin{array}{l} \textit{ok-act}\ P\ s\ w\ (\textit{Start}\ x\ a) = \textit{transitive-owner-check}\ P\ s\ w\ a \\ \textit{upd-act}\ P\ (xs,ws,ms,h)\ w\ (\textit{Start}\ x\ a) = \\ \quad \mathsf{let}\ w' = \textit{a2w}\ a;\ ws' = \textit{transitive-owner-transfer}\ P\ (xs,\ ws,\ ms,\ h)\ \lfloor w' \rfloor\ a\ \mathsf{in} \\ \quad\quad (xs(w' \mapsto x),\ ws',\ ms,\ h) \end{array} \tag{3.18}$$

$$\begin{aligned}&\textit{ok-act}\ P\ s\ w\ (\textit{StartFail}\ a) = \neg\ \textit{transitive-owner-check}\ P\ s\ w\ a\\&\textit{upd-act}\ P\ s\ w\ (\textit{StartFail}\ a) = s\end{aligned} \tag{3.19}$$

Mailbox' receiver. If the mailbox a belongs to the current owner w, the *SetReceiver* $a\ w'$ action is satisfiable. The update function sets the mailbox's receiver with w'. a's ownership is switched to *None* so that any consecutive attempt to set the receiver for a will be unsatisfiable. Note that the mailbox's queue may already contains some messages, therefore the queue must be preserved in the update of the mailboxes store. The dual action *SetReceiverFail* a has indeed no effect on the state. Again, the native calls semantics verifies that a addresses an instance of the Mailbox class.

$$\begin{aligned}&\textit{ok-act}\ P\ s\ w\ (\textit{SetReceiver}\ a\ w') = (\textit{ows}\ s\ a = \lfloor w \rfloor)\\&\textit{upd-act}\ P\ (xs,ws,ms,h)\ w\ (\textit{SetReceiver}\ w'\ a) =\\&\quad \mathsf{let}\ ws' = ws\ (a := \textit{None});\\&\qquad ms' = ms(a \mapsto (\textit{fst}\ (\textit{the}\ (ms\ a)),\ \lfloor w' \rfloor))\\&\quad \mathsf{in}\ (xs,\ ws',\ ms',\ h)\end{aligned} \tag{3.20}$$

$$\begin{aligned}&\textit{ok-act}\ P\ s\ w\ (\textit{SetReceiverFail}\ a) = \neg\ (\textit{ows}\ s\ a = \lfloor w \rfloor)\\&\textit{upd-act}\ P\ s\ w\ (\textit{SetReceiverFail}\ a) = s\end{aligned} \tag{3.21}$$

Message emission. The *Send* $a_{mb}\ a_{msg}$ action requires that the current owner owns every object reachable from the message starting address a_{msg}, discarding initialized mailboxes. If that prerequisite is satisfied, the ownership relation is updated so that every transferable object of the message belongs to *None*. The starting address of the message is enqueued in the message list of a_{mb} by simply updating the appropriate entry in the mailboxes store. The dual action is satisfiable when at least one of the non-shared object reachable from a_{msg} doesn't belong to the current owner. *SendFail* a_{msg} has no side-effect on the global state.

$$\begin{aligned}&\textit{ok-act}\ P\ s\ w\ (\textit{Send}\ a_{mb}\ a_{msg}) = \textit{transitive-owner-check}\ P\ s\ w\ a_{msg}\\&\textit{upd-act}\ P\ (xs,ws,ms,h)\ w\ (\textit{Send}\ a_{mb}\ a_{msg}) =\\&\quad \mathsf{let}\ ws' = \textit{transitive-owner-transfer}\ P\ (xs,\ ws,\ ms,\ h)\ \textit{None}\ a_{msg};\\&\qquad \lfloor(\textit{msgs},\ \textit{rec})\rfloor = ms\ a_{mb};\\&\qquad ms' = ms(a_{mb} \mapsto (\textit{msgs}\ @\ [a_{msg}]),\ \textit{rec})\\&\quad \mathsf{in}\ (xs,\ ws',\ ms',\ h)\end{aligned} \tag{3.22}$$

$$\begin{aligned}&\textit{ok-act}\ P\ s\ w\ (\textit{SendFail}\ a_{msg}) = \neg\ \textit{transitive-owner-check}\ P\ s\ w\ a_{msg}\\&\textit{upd-act}\ P\ s\ w\ (\textit{SendFail}\ a_{msg}) = s\end{aligned} \tag{3.23}$$

Message reception. There are two prerequisites to message reception through the *Receive* $a_{mb}\ a_{msg}$ action. First the current owner-ID must match with a_{mb}'s receiver, and second, the message queue associated with the mailbox shouldn't be empty. The reception failure action *ReceiveFail* only requires that the current owner and the mailbox's receiver differ. There is intentionally no behaviour defined when the current owner is the receiver but

the mailbox is empty. In that case, none of the actions is satisfiable and the current actor is simply blocked until a message is available in the mailbox's queue. When the reception action is satisfiable, the address at the head of the mailbox's message queue is retrieved and the queue is replaced with the rest of the messages. In the same time, ownership of the objects reachable from the retrieved address is switched to match the current actor's owner-ID.

$$\begin{array}{l} \textit{ok-act}\ P\ s\ w\ (\textit{Receive}\ a_{mb}\ a_{msg}) = \mathsf{let}\ \lfloor(\textit{msgs},\ \textit{rec})\rfloor = (\textit{mbs}\ s)\ a_{mb} \\ \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathsf{in}\ \textit{rec} = \lfloor w\rfloor \wedge \textit{msgs} \neq [\,] \\ \textit{upd-act}\ P\ (\textit{xs},\textit{ws},\textit{ms},h)\ w\ (\textit{Receive}\ a_{mb}\ a_{msg}) = \\ \quad \mathsf{let}\ \lfloor(a_{msg}\cdot \textit{msgs},\ \lfloor w\rfloor)\rfloor = \textit{ms}\ a_{mb}; \\ \qquad \textit{ws}' = \textit{transitive-owner-transfer}\ P\ (\textit{xs},\ \textit{ws},\ \textit{ms},\ h)\ \lfloor w\rfloor\ a_{msg} \\ \quad \mathsf{in}\ (\textit{xs},\ \textit{ws}',\ \textit{ms}(a_{mb} \mapsto (\textit{msgs},\ \lfloor w\rfloor)),\ h) \end{array} \tag{3.24}$$

$$\begin{array}{l} \textit{ok-act}\ P\ s\ w\ (\textit{ReceiveFail}\ a_{mb}) = \neg\ (\textit{snd}\ (\textit{the}\ \ (\textit{mbs}\ s\ a_{mb})) = \lfloor w\rfloor) \\ \textit{upd-act}\ P\ s\ w\ (\textit{ReceiveFail}\ a_{mb}) = s \end{array} \tag{3.25}$$

3.4.6 Native Calls Semantics.

We introduce the native calls semantics for the Siaam API methods presented in 3.4.1. Each native method call executes in one atomic step of the form $P, w \vdash \langle a.M(vs), h\rangle\ {-wa}{\rightarrow_{\mathsf{nc}}}\ \langle vx, h'\rangle$, meaning in program P the actor w can call the native method M on receiver object a with parameters vs and heap h while the action wa is globally satisfiable. The call returns the value or exception address vx :: *native-ret*, and updates the heap to h'. Native calls either return a value v encoded *Ret-Val* v or throw an exception encoded *Ret-Xcp* a where a is the address of the exception object. The type for the actor action wa is *cname actor-action*, thus the actor start action is of the form *Start* C a where C is the class name of the started actor object at address a.

$$\begin{array}{ll} \textbf{type_synonym} & \textit{native-call} = {}'x\ \textit{prog} \Rightarrow \textit{owner} \Rightarrow \textit{addr} \Rightarrow \textit{mname} \Rightarrow \textit{val list} \\ & \qquad\qquad \Rightarrow \textit{heap} \Rightarrow \textit{cname actor-action} \Rightarrow \textit{native-ret} \Rightarrow \textit{heap} \Rightarrow \textit{bool} \\ \textbf{datatype} & \textit{native-ret} = \textit{Ret-Val val} \mid \textit{Ret-Xcp addr} \\ _, _ \vdash \langle _._(_), _\rangle - _\rightarrow_{\mathsf{nc}} \langle _, _\rangle :: \textit{native-call} & \end{array} \tag{3.26}$$

Each of the next native rules carry one of the actor action introduced in section 3.4.5. Most rules form couples of a normal execution reduction and an exceptional reduction for the same premise. Obviously the native semantics is not deterministic *per se*, but it is carefully designed such that there is always at most one rule with a satisfiable actor action when embedded in the context of the global semantics.

(CurrAct) The simplest rule implements the *Object.currentActor*() method, it returns the address of the current actor obtained from the current owner-ID w. Notice that the rule can always reduce since it carries the silent action.

$$\textsc{CurrAct}\ \ P, w \vdash \langle a.\underline{\textit{currentActor}}([\,]), h\rangle\ {-\textit{Silent}}{\rightarrow_{\mathsf{nc}}}\ \langle \textit{Ret-Val}\ (\textit{Addr}\ (\textit{w2a}\ w), h\rangle$$

(Start/StartFail) The normal execution for the actor initiation returns *unit* and performs the *Start* C a action with a being an object of class C sub-classing *Actor*. Both the source

language and the bytecode must translate C into the appropriate representation of the initial actor-state. The associated exceptional reduction carries the *StartFail a* action and throws the *OwnerMismatch* exception.

$$\textsc{Start}\frac{h\ a = \lfloor(C, _)\rfloor \quad P \vdash C \preceq^{\star} \ Actor}{P, w \vdash \langle a.\underline{start}([\,]), h\rangle \ {-Start\ C\ a}{\to_{\mathsf{nc}}} \langle \mathit{Ret\text{-}Val\ unit}, h\rangle}$$

$$\textsc{StartFail}\frac{h\ a = \lfloor(C, _)\rfloor \quad P \vdash C \preceq^{\star} \ Actor}{P, w \vdash \langle a.\underline{start}([\,]), h\rangle \ {-StartFail\ a}{\to_{\mathsf{nc}}} \langle \mathit{Ret\text{-}Xcp\ OwnerMismatch}, h\rangle}$$

(SetRecv/SetRecevFail) The *Mailbox.setReceiver*(*Class Actor*) method normally returns *unit*. Exceptionnaly it throws the *OwnerMismatch* exception warning that the current actor is not allowed to setup the mailbox either because a receiver has already been chosen or the ownership mismatches. In both rules the expected parameter for the method call is an *Actor* object, otherwise the semantic blocks. This typing rule would typically be checked by the compiler and the bytecode verifier.

$$\textsc{SetRecv}\frac{\begin{array}{c} h\ a = \lfloor(\mathit{Mailbox}, _)\rfloor \\ h\ a' = \lfloor(C, _)\rfloor \quad P \vdash C \preceq^{\star} \ Actor \end{array}}{P, w \vdash \langle a.\underline{setReceiver}([\mathit{Addr}\ a']), h\rangle \ {-SetReceiver\ a\ (a2w\ a')}{\to_{\mathsf{nc}}} \langle \mathit{Ret\text{-}Val\ unit}, h\rangle}$$

$$\textsc{SetRecvFail}\frac{\begin{array}{c} h\ a = \lfloor(\mathit{Mailbox}, _)\rfloor \\ h\ a' = \lfloor(C, _)\rfloor \quad P \vdash C \preceq^{\star} \ Actor \end{array}}{P, w \vdash \langle a.\underline{setReceiver}([\mathit{Addr}\ a']), h\rangle \ {-SetReceiverFail\ a}{\to_{\mathsf{nc}}} \langle \mathit{Ret\text{-}Xcp\ OwnerMismatch}, h\rangle}$$

(Send/SendFail) Message emission (*Mailbox.put*) returns *unit* when the *Send* action is satisfiable. Otherwise, it throws the *OwnerMismatch* exception when the *SendFail* action is satisfied, meaning at least one object of the message does not belong to the current owner.

$$\textsc{Send}\frac{h\ a = \lfloor(\mathit{Mailbox}, _)\rfloor}{P, w \vdash \langle a.\underline{put}([\mathit{Addr}\ a']), h\rangle \ {-Send\ a\ a'}{\to_{\mathsf{nc}}} \langle \mathit{Ret\text{-}Val\ unit}, h\rangle}$$

$$\textsc{SendFail}\frac{h\ a = \lfloor(\mathit{Mailbox}, _)\rfloor}{P, w \vdash \langle a.\underline{put}([\mathit{Addr}\ a']), h\rangle \ {-SendFail\ a'}{\to_{\mathsf{nc}}} \langle \mathit{Ret\text{-}Xcp\ OwnerMismatch}, h\rangle}$$

(Recv/RecvFail) Successful message reception (*Mailbox.get*) is labeled with *Receive* a a' where a is the address of a mailbox for which the current actor is the receiver, and a' receives the address of the retrieved message. In this rule, a' corresponds to the address found at the head of the a mailbox' queue in the global state. The exceptional reduction carries *ReceiveFail* a and throws a *ReceiverMismatch* exception, meaning the current actor is not the receiver specified for the mailbox.

$$\textsc{Recv}\frac{\exists a' :: \mathit{addr} \in \mathit{dom}\ h \quad h\ a = \lfloor(\mathit{Mailbox}, _)\rfloor}{P, w \vdash \langle a.\underline{get}([\,]), h\rangle \ {-Receive\ a\ a'}{\to_{\mathsf{nc}}} \langle \mathit{Ret\text{-}Val}\ (\mathit{Addr}\ a'), h\rangle}$$

$$\textsc{RecvFail}\frac{h\ a = \lfloor(\mathit{Mailbox}, _)\rfloor}{P, w \vdash \langle a.\underline{get}([\,]), h\rangle \ {-ReceiveFail\ a}{\to_{\mathsf{nc}}} \langle \mathit{Ret\text{-}Xcp\ ReceiverMismatch}, h\rangle}$$

3.5 Single-Actor small-steps semantics

Siaam's single-actor small steps semantics reuses entirely the Jinja's small step semantics, and adds the ability to express actor actions within reductions. We already presented the

abstract syntax of the source language and the type *J-prog* for the program representation in section 3.3. First we overview the original state and semantics for the source language in sections 3.5.1 and 3.5.2. Then in section 3.5.3 we show how this semantics is slightly transformed to include actions and fit into the generic single-actor semantics as previously defined in section 3.4.3.

3.5.1 Single-Actor state.

We keep the static semantics of the Jinja source language unmodified, in the literature it is referred as *state* [63, §2.2.1], however we prefer the name *J-state* to avoid any confusion with other state types we introduce in this document and because it also respects the naming convention established in JinjaThreads. A single-actor state is a pair of a heap and a store of local variables (type *locals*). The heap has exactly the same type as in the global semantics, in fact the single-actor semantics doesn't maintain a private heap, instead it receives the shared heap and updates it at each step. A store is a map from variable names to values. The naming conventions are h for a heap, l for a locals store, and s for an actor state – since we don't mix notations for global and local states there is no possible confusion about s. The projection functions *hp s* and *lcl s* respectively return the heap and the store component of the single-actor state.

$$\textsf{type_synonym} \qquad \begin{aligned} \textit{J-state} &= \textit{heap} \times \textit{locals} \\ \textit{locals} &= \textit{vname} \rightharpoonup \textit{val} \end{aligned} \tag{3.27}$$

3.5.2 Jinja's source language semantics.

Jinja gives a small steps semantics with a judgment for the program $P :: \textit{J-prog}$ of the form $P \vdash \langle e, s\rangle \to \langle e', s'\rangle$ meaning the expression $e :: \textit{expr}$ with the state $s :: \textit{J-state}$ can reduce to e' with the state s'. Expressions are reduced progressively at each step towards a final value or an exception. The four forms of reduction rules are presented in Figure 3.9. The *normal evaluation* reduction rules are divided between the rules reducing a sub-expression of an expression and those reducing a whole expression to its final value. Whereas *exceptional* reduction rules throw and propagate exceptions. The goal of the next paragraphs is to explain the most representative or complex rules found in Jinja that are relevant to build Siaam. The complete set of reduction rules can be found in appendix A.

Normal evaluation rules forms:

$$\frac{P \vdash \langle e, s\rangle \to \langle e', s'\rangle}{P \vdash \langle c \ldots e \ldots, s\rangle \to \langle c \ldots e' \ldots, s'\rangle} \qquad \textit{sub-expression reduction}$$

$$P \vdash \langle e, s\rangle \to \langle e', s'\rangle \qquad \textit{expression reduction}$$

Exceptional evaluation rules forms:

$$P \vdash \langle e, s\rangle \to \langle \texttt{throw}\ e', s\rangle \qquad \textit{exceptional reduction}$$

$$P \vdash \langle c \ldots \texttt{throw}\ e \ldots, s\rangle \to \langle \texttt{throw}\ e, s\rangle \qquad \textit{exception propagation}$$

(where c is an expression constructor)

Figure 3.9: The four forms of rules appearing in the original Jinja small steps semantics.

Object allocation. New objects are allocated in the heap at a fresh addresses produced by the function *new-Addr* :: *heap* ⇒ *addr option*. The NEW rule initializes an object of class C at the fresh address a. The list *FDTs* of fields $((F, D), T)$ where F is the name of a field of type T declared in class D (with $C \preceq^\star D$) is computed by *has-fields*. *init-fields FDTs* builds the initial fields table (type *fields*) where each pair (F, D) maps to its default value of type T. The default values are *Bool False* for the *Boolean* type, *Intg* 0 for *Integer* and *Null* for the references. When no free address is available the exceptional reduction NEWX throws the *OutOfMemory* system exception.

$$\text{New}\ \frac{\textit{new-Addr}\ h = \lfloor a \rfloor \qquad P \vdash C\ \textsf{has-fields}\ \textit{FDTs}}{P \vdash \langle \texttt{new}\ C, (h, l)\rangle \rightarrow \langle \textit{addr}\ a, (h(a \mapsto (C, \textit{init-fields FDTs})), l)\rangle}$$

$$\text{NewX}\ \frac{\textit{new-Addr}\ h = \textit{None}}{P \vdash \langle \texttt{new}\ C, (h, l)\rangle \rightarrow \langle \texttt{THROW}\ \textit{OutOfMemory}, (h, l)\rangle}$$

Object field accesses. The notation for accessing an object's field is $e.F\{D\}$, where e is the expression reducing to the object's address, and F is the name of the field declared in class D. The class name is necessary to distinguish between fields declared with the same name in several superclasses of the accessed object. If the object expression e reduces to *null* (rules READN and WRITEN) the *NullPointer* exception is thrown. To read a field, the object's fields table *fs* is retrieved from the heap and the value v is obtained by destructing *fs* (F, D). To write v to the field $a.F\{D\}$, the heap object at a is updated so that its fields table maps (F, D) to v.

$$\text{Read}\ \frac{\textit{hp}\ s\ a = \lfloor (C, \textit{fs}) \rfloor \qquad \textit{fs}\ (F, D) = \lfloor v \rfloor}{P \vdash \langle a.F\{D\}, s\rangle \rightarrow \langle \texttt{Val}\ v, s\rangle}$$

$$\text{ReadN}\ \ P \vdash \langle \textit{null}.F\{D\}, s\rangle \rightarrow \langle \texttt{THROW}\ \textit{NullPointer}, s\rangle$$

$$\text{Write}\ \frac{h\ a = \lfloor (C, \textit{fs}) \rfloor}{P \vdash \langle a.F\{D\} := \texttt{Val}\ v, (h, l)\rangle \rightarrow \langle \textit{unit}, (h(a \mapsto (C, \textit{fs}((F, D) \mapsto v))), l)\rangle}$$

$$\text{WriteN}\ \ P \vdash \langle \textit{null}.F\{D\} := \texttt{Val}\ v, s\rangle \rightarrow \langle \texttt{THROW}\ \textit{NullPointer}, s\rangle$$

Method invocation. We now describe the various rules related to the reduction of a method call expression of the form $e.M(es)$. The first couple of rules reduces the sub-expression e corresponding to the address of the object receiving the call. First, ROBJ is applyied until e reduces to a value or an exception. In the latter case, the exceptional rule ROBJX reduces the whole remaining expression to an exception throwing expression.

$$\text{Robj}\ \frac{P \vdash \langle e, s\rangle \rightarrow \langle e', s'\rangle}{P \vdash \langle e.M(es), s\rangle \rightarrow \langle e'.M(es), s'\rangle} \qquad \text{RobjX}\ \ P \vdash \langle \texttt{throw}\ e.M(es), s\rangle \rightarrow \langle \texttt{throw}\ e, s\rangle$$

In the third rule RARGS, the receiver has been completely reduced by ROBJ, and the list of arguments *es* is being reduced. The notation $[\rightarrow]$ lifts $\rightarrow$ to the reduction of a list of expressions. Lists of expressions are reduced one element at a time, starting from the first element and progressing towards the last one. The two rules for $[\rightarrow]$ are such that in a single step, a single reduction is applied to the first non-final sub-expression of the list. RVALARG skips the values at the head of the list and REXPARG reduces the first encountered sub-expression toward its final value. The process is repeated step after step until the whole

list contains values, however if one of the sub-expressions reduces to an exception, the later is immediately propagated and replaces the call expression (RargX).

$$\text{Rargs}\ \frac{P \vdash \langle es, s\rangle [\rightarrow] \langle es', s'\rangle}{P \vdash \langle \texttt{Val}\ v.M(es), s\rangle \rightarrow \langle \texttt{Val}\ v.M(es'), s'\rangle}$$

$$\text{Rexparg}\ \frac{P \vdash \langle e, s\rangle \rightarrow \langle e', s'\rangle}{P \vdash \langle e \cdot es, s\rangle [\rightarrow] \langle e' \cdot es, s'\rangle} \qquad \text{Rvalarg}\ \frac{P \vdash \langle es, s\rangle [\rightarrow] \langle es', s'\rangle}{P \vdash \langle \texttt{Val}\ v \cdot es, s\rangle [\rightarrow] \langle \texttt{Val}\ v \cdot es', s'\rangle}$$

$$\text{RargX}\ P \vdash \langle \texttt{Val}\ v.M(map\ \texttt{Val}\ vs\ @\ (\texttt{throw}\ e \cdot es')), s\rangle \rightarrow \langle \texttt{throw}\ e, s\rangle$$

When the receiver and all the arguments have been successfully reduced to respectively an actual address and a list of values, Rcall inlines the invoke statement $addr\ a.M(map\ \texttt{Val}\ vs)$ with the method body's expression *body* preceded by the parameters assignments. The *blocks* function takes the lists of parameters names, types, and values and the called method's body expression, and produces the nested variable declaration blocks for each of the invoke parameters:

$$\begin{aligned} &blocks\ (V \cdot Vs, T \cdot Ts, v \cdot vs, e) = \{V : T;\ V := \texttt{Val}\ v;\ blocks\ (Vs, Ts, vs, e)\} \\ &blocks\ ([\,], [\,], [\,], e) = e \end{aligned} \tag{3.28}$$

In the case where the receiver sub-expression evaluates to *null*, the null pointer exception is thrown (RcallX). Also note we adapt the Rcall rule for optional method bodies as defined in *J-prog*. The case where $C\ sees\ M$ is *Native* is deferred to the next section.

$$\text{Rcall}\ \frac{\begin{array}{c} hp\ s\ a = \lfloor (C, fs)\rfloor \\ P \vdash C\ \textsf{sees}\ M : Ts \rightarrow T = \lfloor (pns, body)\rfloor\ \textsf{in}\ D \quad |vs| = |pns| \quad |Ts| = |pns| \end{array}}{P \vdash \langle addr\ a.M(map\ \texttt{Val}\ vs), s\rangle \rightarrow \langle blocks\ (this \cdot pns, Class\ D \cdot Ts, Addr\ a \cdot vs, body), s\rangle}$$

$$\text{RcallX}\ P \vdash \langle null.M(map\ \texttt{Val}\ vs), s\rangle \rightarrow \langle \texttt{THROW}\ NullPointer, s\rangle$$

3.5.3 Single-Actor semantics

In this section we extend Jinja's original source language semantics so it fits the generic single-actor semantics expected by the global semantics. The modifications are mostly about including the actor owner-ID and specifying the actor action within each reduction. We also add rules to reduce calls to native methods.

The new reduction judgment is of the form $P, w \vdash \langle e, s\rangle\ -wa\rightarrow\ \langle e', s'\rangle$ where $w :: owner$ is the current actor's owner identifier, and $wa :: J\text{-}mb\ actor\text{-}action$ is the action carried by the reduction. Like in the original semantics, P is a program of type *J-prog*, $e :: expr$ is an expression and $s :: J\text{-}state$ is an actor-local state. The general form of the normal and exceptional reduction rules are thus adapted as following:

$$\frac{P, \boxed{w} \vdash \langle e, s\rangle - \boxed{wa} \rightarrow \langle e', s'\rangle}{P, \boxed{w} \vdash \langle c \ldots e \ldots, s\rangle - \boxed{wa} \rightarrow \langle c \ldots e' \ldots, s'\rangle} \qquad \textit{sub-expression reduction}$$

$$P, \boxed{w} \vdash \langle e, s\rangle - \boxed{wa} \rightarrow \langle e', s'\rangle \qquad \textit{expression reduction}$$

$$P, \boxed{w} \vdash \langle e, s\rangle - \boxed{wa} \rightarrow \langle \texttt{throw}\ e', s\rangle \qquad \textit{exceptional reduction}$$

$$P, \boxed{w} \vdash \langle c \ldots \texttt{throw}\ e \ldots, s\rangle - \boxed{Silent} \rightarrow \langle \texttt{throw}\ e, s\rangle \qquad \textit{exception propagation}$$

The sub-expression reductions are integrally preserved, they incur no modification but the overridden judgment, the action *wa* flows unmodified from the premise to the conclusion of each rule. Generally normal and exceptional expression reductions carry the silent action *Silent* meaning they don't require any particular condition nor specify any update of the global semantics. These rules are strictly equivalent to the respective unlabeled Jinja rules. Only the object allocation and the field access rules have specific actions.

Object allocation. The object allocation rule is labeled with the *NewObj C a* actor action, so that when the global semantics picks this rule, it records the ownership relation between the current actor w and the new object of class C allocated at address a.

$$\textsc{New}\;\frac{\textit{new-Addr}\ h = \lfloor a \rfloor \qquad P \vdash C\ \textsf{has-fields}\ \textit{FDTs}}{P, w \vdash \langle \texttt{new}\ C, (h, l)\rangle \ -\textit{NewObj}\ C\ a\rightarrow \langle \textit{addr}\ a, (h \mapsto a(C, \textit{init-fields}\ \textit{FDTs}), l)\rangle}$$

Object field accesses. The field access (Read) and assignment rules (Write), respectively reading from and writing to a field of the object at address a, are now labeled with the *OwnerCheck a True* action, requiring that the object at address a belongs to the current actor w. We add the exceptional field access (ReadX) and assignment (WriteX) rules, labeled with the *OwnerCheck a False*, both throwing the *OwnerMismatch* exception to prevent from reading or writing an object that do not belongs to the current owner.

$$\textsc{Read}\;\frac{\textit{hp}\ s\ a = \lfloor (C, \textit{fs}) \rfloor \qquad \textit{fs}(F, D) = \lfloor v \rfloor}{P, w \vdash \langle a.F\{D\}, s\rangle \ -\textit{OwnerCheck}\ a\ \textit{True}\rightarrow \langle \texttt{Val}\ v, s\rangle}$$

$$\textsc{ReadX}\ \ P, w \vdash \langle a.F\{D\}, s\rangle \ -\textit{OwnerCheck}\ a\ \textit{False}\rightarrow \langle \texttt{THROW}\ \textit{OwnerMismatch}, s\rangle$$

$$\textsc{Write}\;\frac{h\ a = \lfloor (C, \textit{fs}) \rfloor \qquad h' = h(a \mapsto (C, \textit{fs}((F, D) \mapsto v)))}{P, w \vdash \langle a.F\{D\} := \texttt{Val}\ v, (h, l)\rangle \ -\textit{OwnerCheck}\ a\ \textit{True}\rightarrow \langle \textit{unit}, (h', l)\rangle}$$

$$\textsc{WriteX}\ \ P, w \vdash \langle a.F\{D\} := \texttt{Val}\ v, (h, l)\rangle \ -\textit{OwnerCheck}\ a\ \textit{False}\rightarrow \langle \texttt{THROW}\ \textit{OwnerMismatch}, s\rangle$$

Native method calls. Finally, Siaam introduces the CallNative reduction rule to handle native calls in the small-steps semantics. Just like in Call, the rule reduces the method call expression *addr* $a.M$(*map* `Val` *vs*) where a is an object of class C; but C *sees* M is *Native* meaning there is no actual expression defined for the body of M. Instead the expression in the premise is reduced by an atomic step of the native calls semantics $\rightarrow_{\mathsf{nc}}$ defined in section 3.4.6.

$$\textsc{CallNative}\;\frac{\begin{array}{c} h\ a = \lfloor (C, \textit{fs}) \rfloor \qquad P \vdash C\ \textsf{sees}\ M : \textit{Ts} \rightarrow T = \textit{Native}\ \textsf{in}\ D \\ P, w \vdash \langle a.M(\textit{vs}), h\rangle \ -\textit{wa}\rightarrow_{\mathsf{nc}} \langle \textit{vx}, h'\rangle \\ \textit{wa}' = \textit{native-wa2J}\ \textit{wa} \qquad\qquad e' = \textit{native-ret2J}\ \textit{vx} \end{array}}{P, w \vdash \langle \textit{addr}\ a.M(\textit{map}\ \texttt{Val}\ \textit{vs}), (h, l)\rangle \ -\textit{wa}'\rightarrow \langle e', (h', l)\rangle}$$

Figure 3.10: Native calls reduction.

The rule injects the call arguments and the heap into the native calls semantics and retrieves the actor action, the new heap and the value returned by the native method. The function *native-wa2J* rewrites the actor-action *wa* so that the actor-local state carried by the *Start* action is adapted to the the source language representation (see *upd-act*). Native returned

value *Ret-Val* v is translated to the value expression `Val` v, and native exception *Ret-Xcp* a to `Throw` a.

$$\begin{array}{l} \textit{native-wa2J} :: \textit{J-prog} \Rightarrow \textit{cname actor-action} \Rightarrow (\textit{expr} \times \textit{locals})\ \textit{actor-action} \\ \textit{native-wa2J}\ P\ wa = \mathsf{case}\ wa\ \mathsf{of} \\ \quad (\textit{Start}\ C\ a) \Rightarrow \mathsf{let}\ (D,\ _,\ _,\ m) = \textit{method}\ P\ C\ \textit{run};\ \lfloor(_,\ \textit{body})\rfloor = m \\ \qquad\qquad \mathsf{in}\ \textit{Start}\ (\textit{blocks}\ ([\underline{\textit{this}}],\ [D],\ [a], \textit{body}),\ \textit{empty})\ a \\ \quad |\ _ \Rightarrow wa \end{array} \tag{3.29}$$

$$\begin{array}{l} \textit{native-ret2J} :: \textit{native-ret} \Rightarrow \textit{expr} \\ \textit{native-ret2J}\ \textit{Ret-Val}\ v = \mathtt{Val}\ v \\ \textit{native-ret2J}\ \textit{Ret-Xcp}\ a = \mathtt{Throw}\ a \end{array}$$

3.5.4 Actors interleaving.

Now we plug the single-actor small steps semantics into the global semantics. To this extend, we assign concrete types *J-mb* and (*expr* × *locals*) to $'m$ and $'x$ in $('m, 'x)$*single-semantics*:

$$\textbf{type_synonym}\quad \begin{array}{l} \textit{J-semantics} = (\textit{J-mb},\ (\textit{expr} \times \textit{locals}))\ \textit{single-semantics} = \\ \qquad \textit{J-prog} \Rightarrow \textit{owner} \Rightarrow (\textit{expr} \times \textit{locals}) \times \textit{heap} \Rightarrow \\ \qquad (\textit{expr} \times \textit{locals})\ \textit{actor-action} \Rightarrow \\ \qquad (\textit{expr} \times \textit{locals}) \times \textit{heap} \Rightarrow \textit{bool} \end{array} \tag{3.30}$$

Then we provide the reduction function *J-red-act* of type *J-semantics*, which uses the single-actor reductions from the small steps semantics.

$$\begin{array}{l} \textit{J-red-act} :: \textit{J-semantics} \\ \textit{J-red-act}\ P\ w\ (e,\ l)\ h\ wa\ (e',\ l')\ h' = P, w \vdash \langle e, (l, h)\rangle\ {-wa\to}\ \langle e', (l', h')\rangle \end{array} \tag{3.31}$$

Finally we instantiate the global semantics with *J-semantics* and the *J-red-act* single-actor reduction function. The global start state for the program P and the main method M in class C called with parameters vs is written *J-start* $P\ C\ M\ vs$. It comprises only one initiated actor with the special owner-ID w_0. In the initial state, no mailboxes have been created. The initial heap h_0 contains every object reachable from vs, and the ownership relation ws_0 is populated accordingly so each object belongs to w_0. The supplied *"this"* pointer is set to *Null* to emulate a call to a static method.

$$\begin{array}{l} \textbf{interpretation}\ \textit{J-red} : \textit{global-semantics J-red-act} \\ \textit{J-start}\ P\ C\ M\ vs = \\ \quad \mathsf{let}\ (D,\ Ts,\ T,\ m) = \textit{method}\ P\ C\ M;\ \lfloor pns,\ \textit{body}\rfloor = m; \\ \qquad e = \textit{blocks}\ (\ (\underline{\textit{this}} \cdot pns),\ (\textit{Class}\ D \cdot Ts),\ (\textit{Null} \cdot vs),\ \textit{body}) \\ \quad \mathsf{in}\ ([w_0 \mapsto (e,\ \textit{empty})],\ ws_0,\ \textit{empty},\ h_0) \end{array} \tag{3.32}$$

3.6 The Siaam Virtual Machine

3.6.1 Single-Actor Machine Model.

Our virtual machine reuses and extends the bytecode defined for the Jinja VM. Instructions of the bytecode are listed in Figure 3.11, where instructions with a modified operational semantic are marked with ⋄ and new ones are marked with ⋆. We modify the field access

instructions `GetField` and `PutFields` so that an owner-check is performed to protect the accessed object. The new instructions `GetFieldOwned` and `PutfieldOwned` are the alternatives skipping the owner-checking operation. However these two instructions are restricted, only the virtual machine can decide to replace an owner-checking access with its alternative when it considers the verification is redundant or worthless. The `Invoke` instruction handles native calls to the Siaam API methods, and the `New` instruction initializes the ownership of allocated object with the current owner-ID.

The Siaam virtual machine reuses entirely the program representation of the Jinja VM. A method body (msl, mxl, ins, xt) comprises the maximum operand stack size msl, the number of local variables mxl not counting the the method's parameters and the *this* pointer, the list of instructions ins and the exception table xt. An exception table is a list of tuples (f, t, C, h, d) describing a *try/catch* block in the method body. The *try* block protects instructions at positions $f \leq pc < t$ and catches exceptions of a subclass including C. The exception handler *catch* block starts at h and d is the size of the operand stack the handler expects.

$$\textbf{type_synonym} \quad \begin{aligned} \textit{jvm-mb} &= \textit{nat} \times \textit{nat} \times \textit{instr list} \times \textit{ex-table} \\ \textit{jvm-prog} &= \textit{jvm-mb prog} \\ \textit{ex-table} &= (\textit{pc} \times \textit{pc} \times \textit{cname} \times \textit{pc} \times \textit{nat})\ \textit{list} \end{aligned} \tag{3.33}$$

The single-actor virtual machine state *jvm-state* comprises the exception flag, the heap and the stack of call frames for the current actor. The exception flag is the address of the last thrown and yet not caught exception. Each method invocation pushes a new call frame containing the operand stack, the registers for local variables, the class and method names, and the program counter indexing the method body's instructions.

$$\textbf{type_synonym} \quad \begin{aligned} \textit{jvm-state} &= \textit{xcpt-flag} \times \textit{heap} \times \textit{frame list} \\ \textit{xcpt-flag} &= \textit{addroption} \\ \textit{frame} &= \textit{opstack} \times \textit{registers} \times \textit{cname} \times \textit{mname} \times \textit{pc} \\ \textit{opstack} &= \textit{val list} \\ \textit{registers} &= \textit{val list} \end{aligned} \tag{3.34}$$

3.6.2 Operational Semantics.

We define the operational semantics of the Siaam single-actor VM in a functional style, for consistency with the Jinja VM description. Like for the source language, the virtual machine is defined by a single-actor semantics with actor actions. Later we instantiate the global semantics and plug the single-actor VM semantics into it to obtain a multi-actors virtual machine.
The Jinja VM makes one step of execution with the *exec* function taking a program and a single-actor state and computing a new state. Again we adapt that function for the single-actor operational semantics (3.35): Siaam's *exec* function takes the owner-ID of the executing actor, and returns a set of couples made of an actor-action and a single-actor state.

$$\begin{aligned} &\textit{exec} :: \textit{jvm-prog} \Rightarrow \textit{owner} \Rightarrow \textit{jvm-state} \\ &\Rightarrow ((\textit{xcpt-flag} \times \textit{frame list})\ \textit{actor-action} \times \textit{jvm-state})\ \textit{set} \end{aligned} \tag{3.35}$$

Since the function doesn't have access to the global state, it has to return the set of actions the actor is willing to take, and the respective new state if these actions were to be taken.

datatype *instr* =

`Load` *nat*	load from register
| `Store` *nat*	store into register
| `Push` *val*	push a constant
| `New` *cname* ◇	create object on heap and associates to current owner
| `Getfield` *vname cname* ◇	owner-check object and then fetch field from it
| `Putfield` *vname cname* ◇	owner-check object and then set field in it
| `GetfieldOwned` *vname cname* ⋆	fetch field from object, skip owner-check
| `PutfieldOwned` *vname cname* ⋆	set field in object, skip owner-check
| `Checkcast` *cname*	check if object is of class *cname*
| `Invoke` *mname nat* ◇	invoke instance method with *nat* parameters
| `Return`	return from method
| `Pop`	remove top element
| `IAdd`	integer addition
| `Goto` *int*	goto relative address
| `CmpEq`	equality comparison
| `IfFalse` *int*	branch if top of stack false
| `Throw`	throw exception

Figure 3.11: Instructions of the virtual machine, with modified (◇) and added (⋆) instructions.

Then the global semantics pick a satisfiable action among the set of offered alternatives and updates the actor's state accordingly. For most of the instructions, the function computes a one-element set, comprising the *Silent* action and the successor local state, which is always satisfiable. On the other hand, instructions performing an owner-checking operation offer two alternatives: either the verification succeeds and the corresponding field access side-effect is visible in the successor state, or ownership mismatches and an exception is flagged in the successor state. Similarly, most native calls carry a not-always satisfiable action, thus the *exec* function will offer both normal and exceptional continuations.

The single-actor execution function (Figure 3.12) returns the empty set only when the actor's call stack is empty, meaning w is in a final state. When the local state is flagged with an exception, the silent action is offered, and the corresponding new local state is produced with *xcpt-step*. The *xcpt-step* function looks for an exception handler in the current call frame f that matches the class of the flagged exception a. If one is found, the operand stack is adjusted to the size specified by the handler, and the program counter is updated to match the handler's first bytecode index. Otherwise the topmost frame is dropped and the previous frame is flagged with the pending exception so it may be handled by a future execution step. In normal conditions, the execution is delegated to *exec-instr*. Siaam's version of *exec-instr* is an adaptation of the original function in Jinja, where the current actor's owner-ID w now appears. Instead of returning a single local state, it now returns a set of action/state couples following the *exec* specification.

3.6.3 Instructions execution

Now we give the complete description of *exec-instr* for instructions we modify or add. For the other instructions, we simply reuse the original Jinja function (renamed *jinja-exec-instr*) and encapsulate the result in a couple with the silent action. The parameters for *exec-instr* are: the instruction I to execute, the program P, the current owner-ID w, the shared heap h, the current frame's compounds (operand stack *stk*, local variables *loc*, current method's class C_0 and method name M_0, current program points *pc*) and the rest of the call frame

$exec\ P\ w\ (xcp,\ h,\ [\,]) = \emptyset$
$exec\ P\ w\ (\lfloor a \rfloor,\ h,\ f \cdot frs) = \{(Silent,\ xcpt\text{-}step\ P\ a\ h\ f\ frs)\}$
$exec\ P\ w\ (None,\ h,\ (stk, loc, C, M, pc) \cdot frs) =$
$\quad exec\text{-}instr\ \ (instrs\text{-}of\ P\ C\ M)_{[pc]}\ P\ w\ h\ stk\ loc\ C\ M\ pc\ frs$

$exec\text{-}instr :: instr \Rightarrow jvm\text{-}prog \Rightarrow owner \Rightarrow heap \Rightarrow$
$\quad opstack \Rightarrow registers \Rightarrow cname \Rightarrow mname \Rightarrow pc \Rightarrow frame\ list \Rightarrow$
$\quad ((xcpt\text{-}flag \times frame\ list)\ actor\text{-}action \times jvm\text{-}state)\ set$

$jinja\text{-}exec\text{-}instr :: instr \Rightarrow jvm\text{-}prog \Rightarrow heap \Rightarrow$
$\quad opstack \Rightarrow registers \Rightarrow cname \Rightarrow mname \Rightarrow pc \Rightarrow frame\ list \Rightarrow jvm\text{-}state$

Figure 3.12: Siaam execution

stack *frs*.

Object allocation. The object allocation instruction retrieve a free address of the heap (*new-Addr h*), if the heap is out of addresses, a set comprising a single couple is returned, where no particular actor-action is requested but the *OutOfMemory* exception is flagged in the successive state. If address a is allocated, a new object of class C with all fields initialized to default values is created by *blank P C*. The returned set has a unique couple made of the *NewObj C a* action (always satisfiable) and the actor's successor state where the heap has been updated and a pushed on the operand stack.

$$\begin{array}{l}
exec\text{-}instr\ (\texttt{New}\ C)\ P\ w\ h\ stk\ loc\ C_0\ M_0\ pc\ frs = \\
\quad \mathsf{case}\ new\text{-}Addr\ h\ \mathsf{of} \\
\quad\ \ None \Rightarrow \{(Silent,\ (\lfloor addr\text{-}of\text{-}sys\text{-}xcpt\ OutOfMemory \rfloor,\ h,\ (stk, loc, C_0, M_0, pc) \cdot frs))\} \\
\quad |\ \lfloor a \rfloor \Rightarrow \mathsf{let}\ h' = h(a \mapsto blank\ P\ C) \\
\qquad\qquad \mathsf{in}\ \{((NewObj\ C\ a),\ (None,\ h',\ (Addr\ a \cdot stk, loc, C_0, M_0, pc+1) \cdot frs))\}
\end{array} \tag{3.37}$$

Object field accesses. The two instructions reading and writing an object's field have a common structure. The object's reference is retrieved from the top of the operand stack and *Null*-checked. If the reference is *Null*, a set with a single couple is returned, where the *NullPointer* exception is flagged and no actor-action is requested. Otherwise the accessed object must be owner-checked, thus two couples encoding the two possible outcomes of the owner verification are returned. In the first one, the actor-action *OwnerCheck a true* specifies that a belongs to w, and the side-effect of accessing the object's field is visible in the associated successor state. In the second couple, the actor-action *OwnerCheck a false* specified that a is a foreign object, and the associated successor state is flagged with the *OwnerMismatch* exception.

$$
\begin{aligned}
&\textit{exec-instr}\ (\texttt{Getfield}\ F\ C)\ P\ w\ h\ stk\ loc\ C_0\ M_0\ pc\ frs =\\
&\quad \mathsf{let}\ v \cdot stk' = stk;\ Addr\ a = v\\
&\quad \mathsf{in\ if}\ v = Null\ \mathsf{then}\ \{(Silent,\ (\lfloor \textit{addr-of-sys-xcpt NullPointer} \rfloor,\ h\ (stk, loc, C_0, M_0, pc) \cdot frs))\}\\
&\qquad \mathsf{else\ let}\ \lfloor (D, fs) \rfloor = h\ a\\
&\qquad\quad \mathsf{in}\ \{(OwnerCheck\ a\ true,\ (None,\ h,\ (the\ (fs\ (F, C)) \cdot stk', loc, C_0, M_0, pc+1) \cdot frs)),\\
&\qquad\qquad (OwnerCheck\ a\ false,\\
&\qquad\qquad\quad (\lfloor \textit{addr-of-sys-xcpt OwnerMismatch} \rfloor,\ h,\ (stk, loc, C_0, M_0, pc) \cdot frs))\}\\
&\\
&\textit{exec-instr}\ (\texttt{Putfield}\ F\ C)\ P\ w\ h\ stk\ loc\ C_0\ M_0\ pc\ frs =\\
&\quad \mathsf{let}\ v \cdot r \cdot stk' = stk;\ Addr\ a = r\\
&\quad \mathsf{in\ if}\ r = Null\ \mathsf{then}\\
&\qquad\quad \{(Silent,\ (\lfloor \textit{addr-of-sys-xcpt NullPointer} \rfloor,\ h,\ (stk, loc, C_0, M_0, pc) \cdot frs))\}\\
&\qquad \mathsf{else\ let}\ \lfloor (D, fs) \rfloor = h\ a;\ h' = h(a \mapsto (D, fs((F, C) \mapsto v)))\\
&\qquad\quad \mathsf{in}\ \{\ (OwnerCheck\ a\ true,\ (None,\ h',\ (stk', loc, C_0, M_0, pc+1) \cdot frs)),\\
&\qquad\qquad (OwnerCheck\ a\ false,\\
&\qquad\qquad\quad (\lfloor \textit{addr-of-sys-xcpt OwnerMismatch} \rfloor,\ h,\ (stk, loc, C_0, M_0, pc) \cdot frs))\}
\end{aligned}
\tag{3.38}
$$

The alternative instructions `GetfieldOwned` F C and `PutfieldOwned` F C simply delegate execution to *jinja-exec-instr*, passing the original instruction (`Getfield` F C or `Putfield` F C). The result from Jinja's operational semantics is encapsulated in a couple with the *Silent* action and returned in a set.

$$
\begin{aligned}
&\textit{exec-instr}\ (\texttt{GetfieldOwned}\ F\ C)\ P\ w\ h\ stk\ loc\ C_0\ M_0\ pc\ frs =\\
&\quad \{(Silent,\ \textit{jinja-exec-instr}\ (\texttt{Getfield}\ F\ C)\ P\ h\ stk\ loc\ C\ M\ pc\ frs)\}\\
&\\
&\textit{exec-instr}\ (\texttt{PutfieldOwned}\ F\ C)\ P\ w\ h\ stk\ loc\ C_0\ M_0\ pc\ frs =\\
&\quad \{(Silent,\ \textit{jinja-exec-instr}\ (\texttt{Putfield}\ F\ C)\ P\ h\ stk\ loc\ C\ M\ pc\ frs)\}
\end{aligned}
\tag{3.39}
$$

Method calls. Method invocation must differentiate between native and normal calls. If the resolved method has a body $\lfloor (msl, mxl, ins, xt) \rfloor$, a new frame is crafted with an empty operand stack, the class and the name of the invoked method, a program counter set to zero, and the register store containing from the lowest index to the highest: the receiver's object address r (`this` pointer), the call parameters ps in left-to-right order found in reverse order at the top of the stack, and finally the *arbitrary* value replicated mxl times to initialize local variable cells. The returned set is made of a single couple comprising the *Silent* action, an empty exception flag, the shared heap and the call stack with the new frame at the top, followed by the current frame assembled from the *exec-instr* parameters, and the rest of the call frames.

$$
\begin{aligned}
&\textit{exec-instr}\ (\texttt{Invoke}\ M\ n)\ P\ w\ h\ stk\ loc\ C_0\ M_0\ pc\ frs =\\
&\quad \mathsf{let}\ ps = rev\ (take\ n\ stk);\ r = stk_{[n]}\\
&\quad \mathsf{in\ if}\ r = Null\ \mathsf{then}\{(Silent,\ (\lfloor \textit{addr-of-sys-xcpt NullPointer} \rfloor,\ h,\ (stk, loc, C_0, M_0, pc) \cdot frs\ ))\}\\
&\qquad \mathsf{else\ let}\ Addr\ a = r;\ \lfloor (C, _) \rfloor = h\ a;\ (D,\ Ts,\ T,\ m) = method\ P\ C\ M\\
&\qquad\quad \mathsf{in\ case}\ m\ \mathsf{of}\ Native \Rightarrow\\
&\qquad\qquad\quad \{(\textit{native-wa2jvm}\ P\ wa,\\
&\qquad\qquad\qquad \textit{native-ret2jvm}\ n\ h'\ stk\ loc\ C_0\ M_0\ pc\ frs\ vx)\\
&\qquad\qquad\quad \mid wa\ vx\ h'\ .\ (wa,\ vx,\ h') \in \textit{exec-native}\ P\ w\ a\ M\ ps\ h\ \}\\
&\qquad\qquad \mid \lfloor (msl,\ mxl,\ ins,\ xt) \rfloor \Rightarrow \mathsf{let}\ f' = ([\,],\ [r] @ ps @ \textit{replicate}\ mxl\ arbitrary,\ D,\ M,\ 0)\\
&\qquad\qquad\qquad\qquad \mathsf{in}\ \{(Silent,\ (None,\ h,\ f' \cdot (stk, loc, C_0, M_0, pc) \cdot frs))\}
\end{aligned}
\tag{3.40}
$$

If the invoked method has a *Native* body, the function employs *exec-native* (Figure 3.13), the functional implementation of the *call-native* presented in section 3.4.6. Like for the

$$
\begin{aligned}
&\textit{exec-native} :: \textit{jvm-prog} \Rightarrow \textit{owner} \Rightarrow \textit{addr} \Rightarrow \textit{mname} \Rightarrow \textit{val list} \Rightarrow \textit{heap}\\
&\qquad\qquad \Rightarrow (\textit{actor-action} \times \textit{native-ret} \times \textit{heap})\\
&\textit{exec-native}\ P\ w\ a\ M\ vs\ h = \mathsf{case}\ M\ \mathsf{of}\\
&\quad \textit{currentActor} \Rightarrow \{(\textit{Silent},\ \textit{Ret-Val}\ (\mathrm{w2a}\ w),\ h)\}\\
&\ |\ \textit{start} \Rightarrow \mathsf{let}\ \lfloor(C,\ _)\rfloor = h\ a\\
&\qquad\quad \mathsf{in}\ \{\ (\textit{StartFail}\ a,\ \textit{Ret-sys-xcp OwnerMismatch},\ h),\\
&\qquad\qquad (\textit{Start}\ C\ a,\ \textit{Ret-Val unit},\ h)\ \}\\
&\ |\ \textit{setReceiver} \Rightarrow \mathsf{let}\ \textit{Addr}\ a' = \textit{hd}\ vs\\
&\qquad\qquad \mathsf{in}\ \{\ (\textit{SetReceiverFail}\ a,\ \textit{Ret-sys-xcp OwnerMismatch},\ h),\\
&\qquad\qquad\quad (\textit{SetReceiver}\ a\ (\mathrm{a2w}\ a'),\ \textit{Ret-Val unit},\ h)\ \}\\
&\ |\ \textit{put} \Rightarrow \mathsf{let}\ \textit{Addr}\ a' = \textit{hd}\ vs\\
&\qquad \mathsf{in}\ \{\ (\textit{SendFail}\ a',\ \textit{Ret-sys-xcp OwnerMismatch},\ h),\\
&\qquad\quad (\textit{Send}\ a\ a',\ \textit{Ret-Val unit},\ h),\ \}\\
&\ |\ \textit{get} \Rightarrow \{\ (\textit{ReceiveFail}\ a,\ \textit{Ret-sys-xcp OwnerMismatch},\ h),\\
&\qquad (\textit{Receive}\ a\ a',\ \textit{Ret-Val Addr}\ a',\ h)\ \}
\end{aligned}
$$

Figure 3.13: Native single step execution.

source language, the action and value returned by the native semantics must be translated into the virtual machine representation. For each tuple (wa, vx, h') returned by *exec-native*, a new action/state couple is assembled. *native-wa2jvm* converts wa into the VM action representation, just like *native-wa2J* does for the source language. *native-ret2jvm* creates a new local state from the native return value or exception vx. If the native call returns a value *Ret-Val* v, the function removes the $n+1$ parameters from the operand stack, pushes the return value v and increments the program pointer. Otherwise $vx = \textit{Ret-Xcp}\ a$, and the new local state is flagged with the exception address a.

$$
\begin{aligned}
&\textit{native-wa2jvm}\ P\ wa =\\
&\quad \mathsf{case}\ wa\ \mathsf{of}\ \textit{Start}\ C\ a \Rightarrow\\
&\qquad \mathsf{let}\ (D,\ _,\ _,\ m) = \textit{method}\ P\ C\ \textit{run};\ \lfloor(_,\ \textit{body})\rfloor = m\\
&\qquad \mathsf{in}\ \textit{Start}\ (\textit{None},\ [([\,],\ \textit{Addr}\ a \cdot \textit{replicate mxl arbitrary}, D, \textit{run}, 0)])\ a\\
&\quad |\ _ \Rightarrow wa
\end{aligned}
\tag{3.41}
$$

$$
\begin{aligned}
&\textit{native-ret2jvm}\ n\ h\ \textit{stk}\ \textit{loc}\ C_0\ M_0\ pc\ \textit{frs}\ vx =\\
&\quad \mathsf{case}\ vx\ \mathsf{of}\ \textit{Ret-Val}\ v \Rightarrow\\
&\qquad (\textit{None},\ h,\ (v\cdot\ \textit{drop}\ (n+1)\ \textit{stk},\ \textit{loc},\ C_0,\ M_0,\ pc+1) \cdot \textit{frs})\\
&\quad |\ \textit{Ret-Xcp}\ a \Rightarrow (\lfloor a\rfloor,\ h,\ (\textit{stk},\ \textit{loc},\ C_0,\ M_0,\ pc) \cdot \textit{frs})
\end{aligned}
$$

Other instructions. The execution of any other instruction I fallbacks to the default rule which applies the original Jinja VM execution step (3.43). In this case, the result from *jinja-exec-instr* is encapsulated in a couple with the silent action and returned by *exec-instr* in a set, denoting there is a unique, unconditional, successive local state for the current actor.

$$
\begin{aligned}
&\textit{exec-instr}\ I\ P\ w\ h\ \textit{stk}\ \textit{loc}\ C_0\ M_0\ pc\ \textit{frs} =\\
&\quad \{(\textit{Silent},\ \textit{jinja-exec-instr}\ I\ P\ h\ \textit{stk}\ \textit{loc}\ C_0\ M_0\ pc\ \textit{frs})\}
\end{aligned}
\tag{3.43}
$$

3.6.4 Virtual Machine Interleaving.

To instantiate the global semantics for the virtual machine, we first give a relational notation of the *exec* function noted $_, _ \vdash _ - _ \rightarrow_{\mathrm{jvm}} _$:

$$\frac{(wa, s') \in exec\ P\ w\ s}{P, w \vdash s -wa\rightarrow_{\mathrm{jvm}} s'}$$

The single-actor semantics is instantiated with the concrete types *jvm-mb* for the method body representation, and (*xcpt-flag* × *frame list*) for the local state representation:

$$\begin{aligned}&\textbf{type_synonym}\quad \textit{jvm-semantics} = (\textit{jvm-mb}, (\textit{xcpt-flag} \times \textit{frame list}))\ \textit{single-semantics} = \\ &\qquad \textit{jvm-prog} \Rightarrow \textit{owner} \Rightarrow (\textit{xcpt-flag} \times \textit{frame list}) \times \textit{heap} \\ &\qquad \Rightarrow (\textit{xcpt-flag} \times \textit{frame list})\ \textit{actor-action} \\ &\qquad \Rightarrow (\textit{xcpt-flag} \times \textit{frame list}) \times \textit{heap} \Rightarrow \textit{bool}\end{aligned} \tag{3.44}$$

Then we provide the reduction function *jvm-red-act* of type *jvm-semantics*:

$$\begin{aligned}&\textit{jvm-red-act} :: \textit{jvm-semantics} \\ &\textit{jvm-red-act}\ P\ w\ (xcp,\ frs)\ h\ wa\ (xcp',\ frs')\ h' = P, w \vdash (xcp, h, frs) -wa\rightarrow_{\mathrm{jvm}} (xcp', h', frs')\end{aligned} \tag{3.45}$$

The global semantics is instantiated with *jvm-semantics* and *jvm-red-act*. The initial global state for the multi-actors virtual machine is written *jvm-start P C M vs*. It comprises a single actor of owner-ID w_0 with a local state where no exception is flagged, a single frame in the call stack. The initial frame has an empty operand stack, the registers required for the main method are initialized with the *arbitrary* value. The registers storing the parameters for the main method are set with *vs* and *Null* for the *this* pointer to emulate a static method. The initial program counter is set to index the first instruction of the method M visible in D from C.

interpretation *J-red* : *global-semantics jvm-red-act*

$$\begin{aligned}&\textit{jvm-start}\ P\ C\ M\ vs = \\ &\quad \mathsf{let}\ (D,\ Ts,\ T,\ m) = \textit{method}\ P\ C\ M;\ \lfloor(msl,\ mxl,\ ins,\ xt)\rfloor = m; \\ &\quad \mathsf{in}\ ([w_0 \mapsto (None,\ [([\,],\ Null \cdot vs\ @\ \textit{replicate mxl arbitrary},\ D,\ M,\ 0)])],\ ws_0,\ empty,\ h_0)\end{aligned} \tag{3.46}$$

3.7 SIAAM isolation and formal proof

We formalize in this section the isolation property enforced by the Siaam abstract machine. The formalization and proof of the property have been conducted[1] with the Coq proof assistant. We use excerpts from the Coq development to present the property and its proof. The proof is not conducted directly using the operational semantics presented in the previous sections, but instead on a formal actor model that abstracts away details from the Java language. Hence, an actor is modeled just as a collection of local variables that can hold base values or object references. The evolution of a configuration of actors sharing an object heap is governed by rules that can be seen as abstractions of the global semantics rules presented in the previous sections. We call “abstract SIAAM” this abstract model and its rules of evolution.

3.7.1 Abstract SIAAM: Types

We make extensive use of a simple dictionary abstraction, called a `table`. A `set`, as defined in the Coq standard library `ListSet`, is just a list of items.

```
Variable J : Type.
Variable A : Type.

Definition tentry := prod J A.
Definition table  := set tentry.
```

We work mostly with *well-formed* tables, i.e. tables which associate a unique value to a given index. In Coq:

```
Definition wf_table (t : table) : Prop := ∀ (n : J) (v1 v2 : A),
set_In (n,v1) t →
 set_In (n,v2) t →
  v1 = v2.
```

Key operations on a table are updating and lookup, which are defined below (the term `Jeq_dec n m` amounts to an equality check between indexes `n` and `m`).

```
Fixpoint up_table (n : J) (v : A)(t : table): table :=
match t with
| [] ⇒ []
| (m,w)::ts ⇒ if Jeq_dec n m
           then (n, v) :: (up_table n v ts)
           else (m,w):: (up_table n v ts)
end.

Fixpoint tlookup (n : J) (t : table) : option A :=
match t with
| [] ⇒ None
| (m,v):: ts ⇒ if Jeq_dec n m
            then Some v
            else tlookup n ts
end.
```

Basic types in abstract SIAAM are object references (`addr`), actor identifiers (`aid`), mailbox identifiers (`mid`), object field identifiers (`fid`), local variable identifiers (`vid`), and message

1. The developments in Coq have been generously conducted by Jean-Bernard Stefani.

identifiers (`msgid`). All these different sorts of identifiers in fact wrap up an integer (a `nat`), to ensure we have a denumerable number of them at our disposal.

```
Inductive addr := mkaddr : nat → addr.
Inductive fid := mkfid : nat → fid.
Inductive vid := mkvid : nat → vid.
Inductive aid := mkaid : nat → aid.
Inductive mid := mkmid : nat → mid.
Inductive msgid := mkmsgid : nat → msgid
```

An object in abstract SIAAM is simply a table of pairs $\langle f, v \rangle$, where f is a field identifier, and v is a value:

```
Definition object := table fid value.
```

A value can be either null (`vnull`), an integer (`vnat`), a boolean (`vbool`), an object reference, an actor id or a mailbox id. The special value called *mark* (`vmark`) is used to formalize the isolation property : it is simply a distinct constant value that marks the presence of a certain piece of information in an actor or an object.

```
Inductive value : Type :=
  | vnull : value
  | vmark : value
  | vnat : nat → value
  | vbool: bool → value
  | vadd: addr → value
  | vaid: aid → value
  | vmid : mid → value.
```

We do not try in this formalization, compared to the more extensive Jinja specification, to account for Java types. We just have a compatibility relation on values (`vcompat`) that is later used to constrain what updates can take place within an object or an actor. Thus, one may update a null value with any object reference, actor id or mailbox id, but not with a boolean or an integer. The exact details of this compatibility relation are actually irrelevant for the definition of the isolation property and its proof. It can just be seen as a place-holder for typing information, should one envisage to refine the abstract SIAAM specification.

An actor in abstract SIAAM is simply construed as a set of pairs of variable ids coupled with values, identified by an actor id. We have no explicit element in an actor state to account for execution structures such as instruction stack or installed code. This in no way restricts the generality of the abstract SIAAM model since any (such)denumerable local actor state can be encoded using integer values. Contrary to the Jinja-based specification in the preceding sections, actors (and mailboxes) are *not* objects in abstract SIAAM. This is just a simplifying choice for the Coq development, which has no bearing on the isolation result.

```
Definition locals := table vid value.
Definition actor := tentry aid locals.
```

As in the previous specification based on Jinja, evolutions of a SIAAM system are modeled as transitions between configurations. Configurations (type `conf` – called global states in the preceding sections) are 4-tuples comprising: a set of actors (`acs`), an ownership table documenting the ownership relation between objects and actors (`ows`), a set of mailboxes (`mbs`), and a shared heap (`shp`):

```
Record conf : Type := mkcf { acs : actors;
                             ows : owners ;
                             mbs : mailboxes ;
```

```
                    shp : heap }.
```

A heap is a table associating an object reference (`addr`) to an object. A mailbox is essentially a queue of messages, owned by an actor. A message is just the association of a message id and an object reference, the latter referencing in fact the root object of the graph to be transferred as between a sender actor and a receiver actor, which owns the receiving mailbox.

```
Definition actors := table aid locals.
Definition heap := table addr object.

Definition message := prod msgid addr.
Definition queue := list message.

Record mbox : Type := mkmb { own : aid ; msgs : queue}.
Definition mailboxes := table mid mbox.

Definition owners := table addr (option aid).
```

3.7.2 Abstract SIAAM : Transition rules

As mentioned above, evolutions of a SIAAM system are represented in abstract SIAAM as transitions between configurations. Such transitions are governed by rules that mimic the global transition rules in the Jinja-based specification described earlier, but do away with all details pertaining to the Java language. Our abstract transition rules can be interpreted as constraints that apply when a Siaam system evolves from one global state to another. In other terms, transition rules in abstract SIAAM document under which conditions transitions between configurations are possible.

We present each rule in turn and we comment on its relation with the corresponding actor-action in the global reduction relation defined earlier in the Jinja-based specification. The terms in the rules presented hereafter closely relates to the precondition function *ok-act* and the update function *upd-act* defined earlier over the global system state.

Silent actions The rule SILENT governs all state changes that can take place in an actor that has no effect, and thus elicit *no* change, on other elements of a configuration (heap, mailboxes, ownership table, or other actors).

$$\textsc{Silent}\ \frac{\langle a, l_1\rangle \in (acs\ c_1) \qquad\qquad l_1 \dashrightarrow l_2 \qquad c_2 = \texttt{mkcf}\ (\texttt{up_actors}\ a\ l_2\ (acs\ c_1))\ (ows\ c_1)(mbs\ c_1)(shp\ c_1)}{c1\ {-a\rightarrow_{\mathtt{s}}}\ c2}$$

This rule abstracts the *silent* reductions of the single-actor semantics defined in the previous sections, i.e. the rules that carry the *Silent* action and therefore do not request any precondition nor impose any update on the global state. Such rule would have the form $P, a \vdash ((_, l_1), shp\ c_1) {-Silent\rightarrow} ((_, l_2), shp\ c_1)$. The SILENT rule also abstracts the reductions of the global semantics where the picked action is a "failing" alternative, like *SendFail*, *ReceiveFail* or *StartFail*. They all have preconditions on the global state, but these conditions are the exact negation of their "succeeding" counter-part which are abstracted by the rules in the next paragraphs. And more importantly, these actions do not require any update of the global state (no change on other elements of a configuration than the local state of the actor performing the transition).

Notice that, in a transition $c1 -a\rightarrow_{\mathtt{s}} c2$, we record the identifier `a` of the actor that is at the origin of the transition. The same is true in all other transition rules below. We use this information later on to formalize our isolation property.

In Coq, the rule is defined as follows (`up_actors` updates the `actors` table, and `c1 =s a ⇒ s2` is notation for `reds c1 a c2`):

```
Inductive reds : conf → aid → conf → Prop :=
 | reds_step : ∀ (c1 c2 : conf)(a : aid)(l l' : locals),
    set_In (a,l) (acs c1) →
      action l l' →
        c2 = mkcf (up_actors a l' (acs c1)) (ows c1) (mbs c1) (shp c1) →
          c1 =s a ⇒ c2
```

Transitions between locals, noted $l_1 \dashrightarrow l_2$ in the rule above, are defined in turn by the following rules (where `vids_from_locals` returns the set of indexes from a `locals` table, `up_locals` updates a `locals` table, and the constructor `mkcf` creates a record with the elements passed as parameters).

$$\textsc{NwBool}\ \frac{i \notin (\mathtt{vids_from_locals}\ l_1) \qquad l_2 = l_1 \cup \{\langle i, \mathtt{vbool}\ b\rangle\}}{l_1 \dashrightarrow l_2}$$

$$\textsc{NwNat}\ \frac{i \notin (\mathtt{vids_from_locals}\ l_1) \qquad l_2 = l_1 \cup \{\langle i, \mathtt{vnat}\ n\rangle\}}{l_1 \dashrightarrow l_2}$$

$$\textsc{NwNull}\ \frac{i \notin (\mathtt{vids_from_locals}\ l_1) \qquad l_2 = l_1 \cup \{\langle i, \mathtt{vnull}\rangle\}}{l_1 \dashrightarrow l_2}$$

$$\textsc{Compat}\ \frac{\langle i, v\rangle \in l_1 \qquad \langle j, w\rangle \in l_1 \qquad \mathtt{v_compat}\ v\ w \qquad l_2 = \mathtt{up_locals}\ i\ w\ l_1}{l_1 \dashrightarrow l_2}$$

$$\textsc{Forget}\ \frac{l_2 \subset l_1}{l_1 \dashrightarrow l_2}$$

The first three rules govern the creation of a new entry in an actor's locals table. The fourth rule governs the update of an actor's local variable with a value taken from another local variable. The last rule allows the deletion of an entry in an actor's locals table.

In Coq, local actions are defined by:

```
Inductive action : locals → locals → Prop :=
| a_nwbool : ∀ (l l' : locals) (i : vid)(b : bool),
        not (set_In i (vids_from_locals l)) →
              l' = set_add Aeq_lentry (i, vbool b) l →
                  action l l'
| a_nwnat : ∀ (l l' : locals) (i : vid)(n : nat),
        not (set_In i (vids_from_locals l)) →
              l' = set_add Aeq_lentry (i, vnat n) l →
                  action l l'
| a_nwnull : ∀ (l l' : locals) (i : vid),
        not (set_In i (vids_from_locals l)) →
            l' = set_add Aeq_lentry (i, vnull) l →
                action l l'
| a_compat : ∀ (l l' : locals) (i j : vid) (vi vj : value),
        set_In (i, vi) l → set_In (j, vj) l →
            v_compat vi vj → l' = up_locals i vj l →
                action l l'
| a_forget : ∀ (l l' : locals), subset l' l → action l l'.
```

Field read The rule FIELDR governs the reading of a field `f` of an object `o`, whose reference `a` is held in an actor local variable. Notice that the read operation is only allowed if the referenced object belongs to the actor `e` doing the reading (condition $\langle a, \texttt{Some}\ e\rangle \in (\texttt{ows}\ c_1)$).

$$\textsc{FieldR}\ \frac{\begin{array}{c}\langle e, l_1\rangle \in (acs\ c_1) \quad \langle i, w\rangle \in l_1 \quad \langle j, \texttt{vadd}\ a\rangle \in l_1 \quad \langle a, o\rangle \in (shp\ c_1) \quad \langle f, v\rangle \in o \\ \langle a, \texttt{Some}\ e\rangle \in (ows\ c_1) \qquad \texttt{v_compat}\ w\ v \qquad l_2 = \texttt{up_locals}\ i\ v\ l_1 \\ c_2 = \texttt{mkcf}\ (\texttt{up_actors}\ e\ l_2\ (acs\ c_1))(ows\ c_1)(mbs\ c1)(shp\ c1)\end{array}}{c_1 -e\rightarrow_{\texttt{fr}} c_2}$$

This rule abstracts the global semantics reduction picking the *OwnerCheck a True* action, offered by a READ reduction of the single-actor semantics carrying the owner-ID of actor e and performing read access to field f of the object o at address a.

In Coq :

```
Inductive redfr : conf → aid → conf → Prop :=
| redfr_step : ∀ (c1 c2 : conf)(e : aid)
                (l1 l2 : locals)(i j : vid)
                (v w : value) (a: addr)(o : object)(f: fid),
  set_In (e,l1) (acs c1) → set_In (i, w) l1 → set_In (j,vadd a) l1 →
    set_In (a,o) (shp c1) → set_In (f,v) o → set_In (a, Some e) (ows c1) →
      v_compat w v → l2 = up_locals i v l1 →
          c2 = mkcf (up_actors e l2 (acs c1)) (ows c1) (mbs c1) (shp c1) →
              c1 =fr e ⇒ c2
```

Field write The rule FIELDW governs the writing of a field `f` of an object `o`, whose reference `a` is held in actor local variable. The write operation is only allowed if the referenced object belongs to the actor `e` doing the writing.

$$\textsc{FieldW}\ \frac{\begin{array}{c}\langle e, l_1\rangle \in (acs\ c_1) \quad \langle i, w\rangle \in l_1 \quad \langle j, \texttt{vadd}\ a\rangle \in l_1 \quad \langle a, o\rangle \in (shp\ c_1) \quad \langle f, v\rangle \in o \\ \langle a, \texttt{Some}\ e\rangle \in (ows\ c_1) \qquad \texttt{v_compat}\ w\ v \\ c_2 = \texttt{mkcf}\ (acs\ c_1)\ (ows\ c_1)\ (mbs\ c1)\ (\texttt{up_heap}\ a\ (\texttt{up_object}\ f\ w\ o)\ (shp\ c1))\end{array}}{c_1 -e\rightarrow_{\texttt{fw}} c_2}$$

The rule abstracts the global semantics reduction picking the *OwnerCheck a True* action, offered by a WRITE reduction of the single-actor semantics carrying the owner-ID of actor e and performing write access to field f of the object o at address a.

In Coq:

```
Inductive redfw : conf → aid → conf → Prop :=
| redfw_step : ∀ (c1 c2 : conf)(e : aid)
                (l1 l2 : locals)(i j : vid)
                (v w : value) (a: addr)(o : object)(f: fid),
    set_In (e,l1) (acs c1) → set_In (i, w) l1 →
      set_In (j,vadd a) l1 → set_In (a,o) (shp c1) →
        set_In (f,v) o → set_In (a, Some e) (ows c1) →
          v_compat v w →
           c2 = mkcf (acs c1) (ows c1) (mbs c1) (up_heap a (up_object f w o) (shp c1)) →
            c1 =fw e ⇒ c2
```

Actor creation The rule NewA governs the creation of a new actor.

$$\textsc{NewA}\ \frac{\begin{array}{c}\langle e, l_e\rangle \in (acs\ c_1) \qquad \langle i, w\rangle \in l_e \qquad n \notin \texttt{aids_from_actors}\ (acs\ c_1) \\ (\texttt{values_from_locals}\ l_n) \subset (\texttt{values_from_locals}\ l_e) \\ k \notin (\texttt{vids_from_locals}\ l_n) \qquad \langle m, mb\rangle \in (\texttt{mbs}\ c_1) \qquad \texttt{v_compat}\ w\ v \\ acs_2 = \texttt{up_actors}\ e\ (\texttt{up_locals}\ i\ (\texttt{vaid}\ n)\ l_e)(\texttt{acs}\ c_1) \\ a_n = \langle n, \texttt{up_locals}\ k\ (\texttt{vmid}\ m)\ l_n\rangle \\ c_2 = \texttt{mkcf}\ (acs_2 \cup \{a_n\})\ (ows\ c_1)\ (mbs\ c1)\ (shp\ c1)\end{array}}{c_1 - e \rightarrow_{\texttt{nwa}} c_2}$$

The rule abstracts the Start reduction of the native call semantics in response to a call of the *Actor*.*start*() method, which offers the *Start* x a_n action. Where x is the initial state for the new actor a_n.

Notice that in abstract SIAAM, a new actor is created with a reference to an existing mailbox. This is just one possible scheme, adopted for its simplicity. The Jinja-based specification presented earlier adopts a different scheme by allowing an arbitrary initial state, but which has no bearing on the isolation property. Indeed the actor creation in Siaam could be divided in two phases, first create a new actor, allocates and initialize a new mailbox for it, start the actor, and then pass the rest of the initial state to the new actor through that mailbox.

In Coq:

```
Inductive rednwa : conf → aid → conf → Prop :=
| rednwa_step : ∀ (c1 c2 : conf)(e ne: aid) (m : mid)(mb : mbox)
                (le lne: locals)(i k: vid)(w : value),
   set_In (e,le) (acs c1) → set_In (i, w) le →
    not (set_In ne (aids_from_actors (acs c1))) →
     subset (values_from_locals lne) (values_from_locals le) →
      not (set_In k (vids_from_locals lne)) →
       set_In (m, mb) (mbs c1) →
        v_compat w (vaid ne) →
         c2 = mkcf (set_add Aeq_actor  (ne, (up_locals k (vmid m) lne))
                    (up_actors e (up_locals i (vaid ne) le) (acs c1)))
               (ows c1) (mbs c1) (shp c1) →
          c1 =nwa e ⇒ c2
```

Mailbox creation The rule NewM governs the creation of a new mailbox.

$$\textsc{NewM}\ \frac{\begin{array}{c}\langle e, l_1\rangle \in (acs\ c_1) \qquad \langle i, v\rangle \in l_1 \qquad n \notin \texttt{mids_from_mboxes}\ (mbs\ c_1) \\ \texttt{v_compat}\ v\ (\texttt{vmid}\ m) \qquad l_2 = \texttt{up_locals}\ i\ (\texttt{vmid}\ m)\ l_1 \\ c_2 = \texttt{mkcf}\ (\texttt{up_actors}\ e\ l_2\ (acs\ c_1))\ (ows\ c_1)\ ((mbs\ c1) \cup \{\langle m, \texttt{mkmb}\ e\ []\rangle\})\ (shp\ c1)\end{array}}{c_1 - e \rightarrow_{\texttt{nwm}} c_2}$$

The rule abstracts the New rule of the single-actor semantics which offers the *NewObj C m* action when the allocated object is an instance of class C and C is a sublcass of *Mailbox*.

In Coq:

```
Inductive rednwm : conf → aid → conf → Prop :=
 | rednwm_step : ∀ (c1 c2 : conf)(e : aid) (m : mid)
                 (l1 l2: locals)(i: vid)(v: value),
    set_In (e,l1) (acs c1) → set_In (i, v) l1 →
     not (set_In m  (mids_from_mboxes (mbs c1))) →
      v_compat v (vmid m) → l2 = up_locals i (vmid m) l1 →
        c2 = mkcf (up_actors e l2 (acs c1))
```

```
            (ows c1)
            (set_add Aeq_mbentry (m, mkmb e []) (mbs c1))
            (shp c1) →
        c1 =nwm e ⇒ c2
```

Object creation The rule NewO governs the creation of new object.

$$\textsc{NewO}\ \frac{\langle e, l_1\rangle \in (acs\ c_1) \quad \langle i, v\rangle \in l_1 \quad a \notin \texttt{addrs_from_owners}\ (ows\ c_1) \quad (\texttt{values_in_table}\ o) \subset (\texttt{values_from_locals}\ l_1) \quad \texttt{v_compat}\ v\ (\texttt{vadd}\ a) \quad l_2 = \texttt{up_locals}\ i\ (\texttt{vadd}\ a)\ l_1 \quad c_2 = \texttt{mkcf}\ (\texttt{up_actors}\ e\ l_2\ (acs\ c_1))\ ((ows\ c_1) \cup \{\langle a, \texttt{Some}\ e\rangle\})\ (mbs\ c1)\ ((shp\ c1) \cup \{\langle a, o\rangle\})}{c_1 -e\rightarrow_{\texttt{nwo}} c_2}$$

The rule abstracts the New rule of the single-actor semantics which offers the *NewObj C a* action when the allocated object is an instance of class C and C is not a subclass of *Mailbox*.

In Coq:

```
Inductive rednwo : conf → aid → conf → Prop :=
  | rednwo_step : ∀ (c1 c2 : conf)(e : aid) (a : addr)(o:object)
                    (l1 l2: locals)(i: vid)(v : value),
      set_In (e,l1) (acs c1) → set_In (i, v) l1 →
        not (set_In a  (addrs_from_owners (ows c1))) →
         subset (values_in_table fid value Aeq_value o)(values_from_locals l1) →
          v_compat v (vadd a) → l2 = up_locals i (vadd a) l1 →
            c2 = mkcf (up_actors e l2 (acs c1))
                    (set_add Aeq_oentry (a,Some e) (ows c1))
                    (mbs c1)
                    (set_add Aeq_sentry (a,o) (shp c1)) →
             c1 =nwo e ⇒ c2
```

Message send The rule Snd governs the emission of a message. The function `trans_owner_check` verifies that all the objects reachable from object reference a have the same given owner (here e) in the designated ownership table (here `ows c1`). The function `trans_owner_update` updates the ownership table to set the owner of all the objects reachable from a to the same owner (here `None`).

$$\textsc{Snd}\ \frac{\langle e, l\rangle \in (acs\ c_1) \quad (\texttt{vadd}\ a) \in (\texttt{values_from_locals}\ l) \quad \texttt{trans_owner_check}\ (shp\ c_1)\ (ows\ c_1)\ (\texttt{Some}\ e)\ a = \texttt{true} \quad \langle m_i, mb\rangle \in (mbs\ c_1) \quad ms \notin (\texttt{msgids_from}_{\texttt{m}}\texttt{box}\ mb) \quad \texttt{trans_owner_update}\ (shp\ c_1)\ (ows\ c_1)\ \texttt{None}\ a = \texttt{Some}\ owns \quad mb' = \texttt{mkmb}\ (own\ mb)\ (\langle ms, a\rangle :: (msgs\ mb)) \quad c_2 = \texttt{mkcf}\ (acs\ c_1)\ owns\ \texttt{up_mboxes}\ m_i\ mb'\ (mbs\ c1)\ (shp\ c1)}{c_1 -e\rightarrow_{\texttt{nwo}} c_2}$$

The rule abstracts the reduction of the *Send* m_i a action, offered when a native call to *Mailbox*.*put*() is reduced by the single-actor semantics (Send rule of the native calls semantics). In particular the `trans_owner_check` and `trans_owner_update` function correspond to the *transitive-owner-check* and *transitive-owner-transfer* functions in the Siaam formal specification.

In Coq:

```
Inductive redsnd : conf → aid → conf → Prop :=
  | redsnd_step : ∀ (c1 c2 : conf)(e : aid) (a : addr)
```

```
                 (l : locals) (ms: msgid)(mi: mid)
                 (mb mb': mbox) (owns : owners),
   set_In (e,l) (acs c1) →
    set_In (vadd a) (values_from_locals l) →
     trans_owner_check (shp c1) (ows c1) (Some e) a = true →
      set_In (mi,mb) (mbs c1) →
       not (set_In ms (msgids_from_mbox mb)) →
        Some owns = trans_owner_update (shp c1) (ows c1) None a →
         mb' = mkmb (own mb) ((ms,a)::(msgs mb)) →
          c2 = mkcf (acs c1)  owns  (up_mboxes mi mb' (mbs c1))  (shp c1) →
           c1 =snd e ⇒ c2
```

Message receive The rule RCV governs the reception of a message. The function `lasto` retrieves the first message in a queue (the last one in the list that forms a queue, since message are in-queued at the head of the queue list). The function `removelast` removes the last message of a queue.

$$\text{RCV}\;\frac{\begin{array}{c}\langle e, l_1\rangle \in (acs\ c_1) \qquad \langle m_i, mb\rangle \in (mbs\ c_1) \qquad \langle i, v\rangle \in l_1 \\ (own\ mb) = e \qquad \texttt{lasto}\ (msgs\ mb) = \texttt{Some}\ m \\ mb' = \texttt{mkmb}\ (own\ mb)\ (\texttt{removelast}\ (msgs\ mb)) \\ \texttt{trans_owner_update}\ (shp\ c_1)\ (ows\ c_1)\ (\texttt{Some}\ e)\ (\texttt{snd}\ m) = \texttt{Some}\ owns \\ \texttt{v_compat}\ v\ (\texttt{vadd}\ (\texttt{snd}\ m)) \qquad l_2 = \texttt{up_locals}\ i\ (\texttt{vadd}\ (\texttt{snd}\ m))\ l_1 \\ c_2 = \texttt{mkcf}\ (\texttt{up_actors}\ e\ l_2\ (acs\ c_1))\ owns\ \texttt{up_mboxes}\ m_i\ mb'\ (mbs\ c1)\ (shp\ c1)\end{array}}{c_1 -e\!\!\to_{\texttt{rcv}} c_2}$$

The rule abstracts the reduction of the global state where the picked action is a message reception from mailbox m_i: *Receive* m_i (*snd m*), where *snd m* is the address of the top object of the message.

In Coq:

```
Inductive redrcv : conf → aid → conf → Prop :=
 | redrcv_step : ∀ (c1 c2 : conf) (e : aid) (m : message)
                 (l1 l2: locals) (mb mb': mbox) (mi : mid)
                 (owns : owners)(i : vid)(v : value),
    set_In (e,l1) (acs c1) →
     set_In (mi,mb) (mbs c1) →
      set_In (i,v) l1 →
      (own mb) = e →
        Some m = lasto (msgs mb) →
         mb' = mkmb (own mb) (removelast (msgs mb)) →
          Some owns = trans_owner_update (shp c1) (ows c1) (Some e) (snd m) →
           v_compat v (vadd (snd m)) →
            l2 = up_locals i (vadd (snd m)) l1 →
             c2 = mkcf (up_actors e l2 (acs c1))  owns  (up_mboxes mi mb' (mbs c1))  (shp c1)
 →
              c1 =rcv e ⇒ c2
```

3.7.3 Isolation property

The SIAAM model ensures that the only means of information transfer between actors is message exchange. We can formalize this isolation property using mark values. We call

an actor a *clean* if its local variables do not hold a mark, and if all objects *reachable* from a and belonging to a hold no mark in their fields. An object o is reachable from an actor a if a has a local variable holding o's reference, or if, recursively, an object o' is reachable from a which holds o's reference in one of its fields. The isolation property can now be characterized as follows: a clean actor in any configuration remains clean during an evolution of the configuration if it never receives any message.

The theorem that formalizes this intuition is expressed as follows in Coq:

```
Theorem ac_isolation : ∀ (c1 c2 : conf) (a1 a2: actor),
  wf_conf c1 →
      set_In a1 (acs c1) →
      ac_clean (shp c1) a1 (ows c1) →
          c1 =@ (fst a1) ⇒∗ c2 →
            Some a2 = lookup_actor (acs c2) (fst a1) →
            ac_clean (shp c2) a2 (ows c2).
```

The theorem states that, in any well-formed configuration c1, an actor a1 which is clean, expressed by ac_clean (shp c1) a1 (ows c1), remains clean in any evolution of c1 that does not involve a reception by a1, which is expressed as c1 =@ (fst a1) ⇒∗ c2 and ac_clean (shp c2) a2 (ows c2), where a2 is the descendant of actor a1 in the evolution.

The relation $-@\ a\rightarrow^*$, which represents configuration evolution that does not involve a reception by the actor identified by a, is defined as the reflexive and transitive closure of the relation $-@\ a\rightarrow$, itself defined as the smallest relation satisfying the following rules of inference:

$$\frac{c_1 -e\rightarrow_{\mathtt{x}} c_2 \qquad \mathtt{x} \neq \mathtt{rcv}}{c_1 -@\ a\rightarrow c_2} \qquad\qquad \frac{c_1 -e\rightarrow_{\mathtt{rcv}} c_2 \qquad e \neq a}{c_1 -@\ a\rightarrow c_2}$$

In Coq, relation $-@\ a\rightarrow$ is defined as follows:

```
Inductive mred : aid → conf → conf → Prop :=
  | mred_silent : ∀ (c1 c2 : conf) (e a: aid),
            c1 =s e ⇒ c2 → c1 =@ a ⇒ c2
  | mred_fread : ∀ (c1 c2 : conf)(e a: aid), c1 =fr e ⇒ c2  → c1 =@ a ⇒ c2
  | mred_fwrite : ∀ (c1 c2 : conf)(e a: aid), c1 =fw e ⇒ c2  → c1 =@ a ⇒ c2
  | mred_newact : ∀ (c1 c2 : conf)(e a: aid), c1 =nwa e ⇒ c2 → c1 =@ a ⇒ c2
  | mred_newmbox : ∀ (c1 c2 : conf)(e a: aid), c1 =nwm e ⇒ c2 → c1 =@ a ⇒ c2
  | mred_newobj : ∀ (c1 c2 : conf)(e a: aid), c1 =nwo e ⇒ c2 → c1 =@ a ⇒ c2
  | mred_send : ∀ (c1 c2 : conf)(e a: aid), c1 =snd e ⇒ c2 → c1 =@ a ⇒ c2
  | mred_rcv : ∀ (c1 c2 : conf)(e a: aid), e ≠ a → c1 =rcv e ⇒ c2 → c1 =@ a ⇒ c2
```

The cleanliness of an actor is defined as follows in Coq:

```
Definition obj_clean (o : object) : Prop :=
  ∀ (f : fid)(v: value), set_In (f,v) o → v ≠ vmark.

Definition addr_clean (h : heap) (a : addr) : Prop :=
  ∀ (o : object), set_In (a,o) h → obj_clean o.

Definition trans_clean (h : heap) (a: addr) (o : aid) (owns : owners) : Prop :=
  ∀ (b : addr), oreachable h a b o owns → addr_clean h b.

Definition loc_clean (h : heap)(l: locals)(o:aid)(owns: owners): Prop :=
  ∀ (i: vid)(v:value),
    set_In (i,v) l →
        v ≠ vmark ∧ (∀ (a:addr), v = vadd a → trans_clean h a o owns).
```

```
Definition ac_clean (h : heap)(a: actor)(owns:owners): Prop :=
  loc_clean h (snd a) (fst a) owns.
```

The assertion `oreachable h a b o onws` is true, in the context of heap `h` and ownership relation `owns`, for two object references `a` and `b` if the object referenced by `b` is reachable from `a` by a chain of objects that all belong to the same actor identified by `o`. The predicate `oreachable` is defined as the smallest one satisfying the following inference rules:

$$\frac{\langle a, \texttt{Some}\ o\rangle \in owns}{\texttt{oreachable}\ h\ a\ a\ o\ owns} \qquad \frac{\langle a, \texttt{Some}\ o\rangle \in owns \quad \texttt{ofield}\ h\ a\ b \quad \langle b, \texttt{Some}\ o\rangle \in owns \quad \texttt{oreachable}\ h\ b\ c\ o\ owns}{\texttt{oreachable}\ h\ a\ c\ o\ owns}$$

Its definition in Coq is as follows:

```
Inductive oreachable : heap → addr → addr → aid → owners → Prop :=
| or_refl : ∀ (h : heap)(a : addr)(o : aid)(owns : owners),
             set_In (a,Some o) owns →
               oreachable h a a o owns
| or_trans : ∀ (h : heap)(a b c: addr)(o : aid)(owns : owners),
              set_In (a,Some o) owns →
                ofield h a b →
                  set_In (b,Some o) owns →
                    oreachable h b c o owns →
                      oreachable h a c o owns.
```

The assertion `ofield h a b` merely states that, in the context of heap `h`, object reference `b` appears as a value in a field of the object referenced by `a`. In Coq, this is formalized as :

```
Definition ofield (h : heap) (s a : addr) : Prop :=
  (∃ (o : object), set_In (s,o) h ∧ (∃ (f : fid), In (f, vadd a) o)).
```

The theorem `ac_isolation` is proved first by proving the invariance of well-formedness for configurations. This is expressed as follows in Coq:

```
Theorem red_preserves_wf : ∀ (c1 c2 : conf),
 c1 ⇒ c2 → wf_conf c1 → wf_conf c2.
```

The well-formedness of a configuration is defined as follows:

```
Definition wf_conf (c : conf) : Prop :=
  wfc_heap (shp c)
    ∧ wf_owners (ows c)
    ∧ wf_mboxes (mbs c)
    ∧ wf_actors (acs c)
    ∧ consistent_heap_owners (shp c) (ows c)
    ∧ consistent_mboxes_actors (mbs c) (acs c)
    ∧ consistent_mboxes_heap (mbs c) (shp c)
    ∧ consistent_heap_actors (shp c) (acs c).
```

The different predicates appearing in the clauses above are defined as follows:

```
Definition wf_heap (h : heap) : Prop := wf_table addr object h.

Definition complete_heap (h : heap) : Prop :=
  ∀ (a b: addr) (o : object),
    set_In (a,o) h →
        set_In b (addrs_in_fields o) →
        set_In b (addrs_from_heap h).
```

```
Definition wfc_heap (h : heap) : Prop :=  wf_heap h ∧ complete_heap h.

Definition wf_actors (acts : actors) : Prop := wf_table aid locals acts.

Definition wf_owners (owns : owners) : Prop := wf_table addr (option aid) owns.

Definition wf_mboxes (mbxs : mailboxes) : Prop := wf_table mid mbox mbxs.

Definition consistent_mboxes_actors (mbs : mailboxes) (acts : actors) : Prop :=
  subset (aids_from_mboxes mbs) (aids_from_actors acts).

Definition consistent_heap_owners (h : heap) (owns : owners) : Prop :=
  subset (addrs_from_heap h) (addrs_from_owners owns).

Definition consistent_mboxes_heap (mbs : mailboxes) (h : heap) : Prop :=
  subset (addrs_from_mboxes mbs) (addrs_from_heap h).

Definition consistent_heap_actors (h : heap) (acs : actors) : Prop :=
  subset (addrs_from_actors acs) (addrs_from_heap h).
```

Notice that the well-formedness of the ownership table in a configuration `c`, expressed by the clause `wf_owners (ows c)`, ensures that in a well-formed configuration there is only one owner for a given object.

The theorem `red_preserves_wf` is proved by induction on the derivation of the assertion `c1 ⇒ c2`. To prove the different cases, we rely mostly on simple reasoning with sets, and a few lemmas characterizing the correctness of table manipulation functions and of the `trans_owner_update` function.

Using the invariance of well-formedness for configurations, theorem `ac_isolation` is proved by induction on the derivation of the assertion `c1 =@ (fst a1) ⇒∗ c2`. To prove the different cases, we rely on a few lemmas dealing with reachability and cleanliness, for instance Lemma `oreachable_up_object_addr`, which is part of a set of lemmas that clarify how reachability evolves when the shared heap is modified, or Lemma `clean_values_upobject_clean`, which states that writing a clean value does not alter the cleanliness of other values in a configuration.

```
Lemma oreachable_up_object_addr :
  ∀ (h1 h2 : heap)(a b c x: addr)(oa : object)
       (f : fid) (o : aid) (owns:owners),
  wfc_heap h1 → set_In (a,oa) h1 →
   h2 = up_heap a (up_object f (vadd c) oa) h1 →
    oreachable h2 b x o owns →
       (oreachable h1 b a o owns ∧ oreachable h1 c x o owns) ∨ (oreachable h1 b x o owns).

Lemma clean_value_upobject_clean :
  ∀ (h1 h2 : heap)(a : addr)(oa : object)(v : value)
       (f : fid) (o : aid) (owns:owners),
  wfc_heap h1 → value_clean h1 v o owns →
    set_In (a,oa) h1 →
     h2 = up_heap a (up_object f v oa) h1 →
     value_clean h2 v o owns.
```

The last theorem, theorem `live_mark`, is a liveness property that expresses that information, in particular marks, *can* flow between actors during execution. In other terms, it shows

that the isolation property is not vacuously true. In Coq:

```
Theorem live_mark : ∃ (c1 c2 : conf)(ac1 ac2 : actor),
     c1 ⇒∗ c2 ∧ set_In ac1 (acs c1) ∧ ac_clean (shp c1) ac1 (ows c1)
 ∧ Some ac2 = lookup_actor (acs c2) (fst ac1) ∧ ac_mark (shp c2) ac2 (ows c2).
```

The proof of the theorem is simple, it consists in exhibiting configurations meeting the conditions. The one used in the proof is given below as a set of Coq definitions. It consists of three configurations `c1`, `c2`, `c3` with two actors, identified by `e1` and `e2`, that exchange a single message bearing a marked object `oa` with a single field `f` which bears a `vmark` value. We have `ac_clean h ac2 ow1`, `ac_mark h ac3 ow3`, `fst ac2 = e2`, `fst ac3 = e3`, `cf1 =snd e1 ⇒ cf2` and `cf2 =rcv e2 ⇒ cf3`.

```
Definition a: addr := mkaddr 1.
Definition f : fid := mkfid 1.
Definition oa : object := [(f, vmark)].
Definition h: heap := [(a,oa)].
Definition mi : msgid := mkmsgid 1.
Definition m : message := (mi, a).
Definition e1 : aid := mkaid 1.
Definition e2 : aid := mkaid 2.
Definition mb1 : mbox := mkmb e2 [].
Definition mb2 : mbox := mkmb e2 [m].
Definition i : vid := mkvid 1.
Definition j : vid := mkvid 2.
Definition l1 : locals := [(i, vadd a)].
Definition l2 : locals := [(j,vnull)].
Definition l3 : locals := [(j,vadd a)].
Definition ac1 : actor := (e1, l1).
Definition ac2 : actor := (e2,l2).
Definition ac3 : actor := (e2, l3).
Definition ow1 : owners := [(a, Some e1)].
Definition ow2 : owners := [(a, None)].
Definition ow3 : owners := [(a, Some e2)].
Definition acts1 : actors := [ac1, ac2].
Definition acts2 : actors := [ac1, ac3].
Definition mbi : mid := mkmid 1.
Definition mbs1 : mailboxes := [(mbi, mb1)].
Definition mbs2 : mailboxes := [(mbi, mb2)].
Definition cf1 : conf := mkcf acts1 ow1 mbs1 h.
Definition cf2 : conf := mkcf acts1 ow2 mbs2 h.
Definition cf3 : conf := mkcf acts2 ow3 mbs1 h.
```

Chapter 4

Ownership-based isolation for the JVM

Contents

4.1	**Introduction**	**77**
4.2	**Virtual Machine's core**	**78**
4.3	**Ownership-based isolation**	**92**
4.4	**Support for immutability and static variables**	**95**
4.5	**Trusted programming models**	**101**

4.1 Introduction

In this chapter we modify the JikesRVM, a research-driven Java Virtual Machine, in order to implement ownership-based isolation. We add a set of *core* primitives supplying the ownership machinery, which is then used to build *trusted* APIs. The Siaam actors programming model is available as a trusted API directly inspired by the native API proposed in the formal specification. We also developed other trusted APIs bringing ownership-based isolation to existing actors frameworks. Our work is not a basic prototype, we focused on performance optimization as often as possible, we think it worth the effort since the original virtual machine competes with industrial ones as a testbed for various state-of-the-art techniques. We discuss solutions to support object immutability, static variables and enumerated types, which have mostly not been considered by previous works on safe actor-based programming models.

4.1.1 JikesRVM

JikesRVM is a Java virtual machine written almost exclusively in Java, with the minimal bootstrapping features written in C. It comprises a class loader and two just-in-time compilers for transforming bytecode to either baseline or optimized native machine code. The baseline compiler performs a straightforward translation of the bytecode, that can be considered as "obviously correct". The optimizing compiler, on the other hand, transforms the bytecode through various intermediate representations, performing optimizations at each phase. Jikes also comes with an adaptive optimization system (AOS), that identifies "hot-spots" as the code is executed and selects recompilation plans to be performed progressively. The memory management, including several allocator and garbage collector implementations is supplied

by the Memory Manager Toolkit (MMTk). Finally, the Magic API provides low-level functionalities that wouldn't be possible to achieve using pure Java. Typical use of Magic is for accessing raw process memory through arbitrary pointers arithmetic, invoking operating system calls, or specifying "uninterruptible" sections of code.

4.1.2 Outline

Our modified Jikes virtual machine comprises a set of *core* features providing the raw ownership mechanisms that are combined into *trusted* APIs. Core features alone cannot enforce isolation, thus their usage is restricted to the implementation of high-level trusted frameworks embedded in the virtual machine. Trusted code must be carefully written so it cannot be exploited by malicious bytecode in order to violate the isolation property. The Java standard library itself is trusted as long as it doesn't manipulate any core primitive. In these conditions, dynamically loaded application code is trusted in its turn since it can only link with trusted parts of the VM.

Using the core primitives, we built the trusted Siaam actor-based programming model as specified in section 3. We also wrote a "fully user-land" implementation of the ActorFoundry API on top of the Siaam actor programming model, which can be trusted in its turn. Finally we implemented a trusted event-based actor programming model on top of the core primitives. This last model can dispatch thousand of lightweight actors over pools of threads, which enables to build high-level APIs similar to Kilim or Akka. For that matter, we use this base to supply a Kilim-like programming model. We believe the core primitives are versatile enough to support other actor-based frameworks such as Scala's actors and Akka that currently lack isolation mechanisms and must rely on developers following guidelines.

The virtual machine accepts any standard Java bytecode. Classes declaring mutable static fields and the usage of threads are both serious threats to the memory isolation and therefore should be avoided. Apart from these legitimate limitations, any legacy program or library supported by the JikesRVM is also supported by our modified version and will benefit from the ownership-based isolation. Most importantly, developers are not asked to annotate their code neither shall they post-process class files with any code weaver. Instead, the whole isolation intelligence is performed automatically at runtime. Owner-checking read and write barriers[94] are seamlessly generated on-the-fly while the bytecode is being compiled. Furthermore, the barriers overhead is significantly reduced as redundant or worthless owner-checks are optimized-out by a fast data-flow analysis (Chapter 5).

4.2 Virtual Machine's core

At the heart of the virtual machine stand a few key concepts. Each heap object has an owner, which is the address of another object. Each thread of execution has an owner-ID, which is a reference to an object implementing the special `Owner` interface. A thread can only access objects belonging to the `Owner` instance referenced by its owner-ID. Object ownership is transferable between `Owners`. As described earlier in the Siaam formal specification, the transfer applies transitively through object's field references.

The `siaam.untrusted` package gathers Siaam's features that allow one to build high-level trusted programming model APIs, but must not be accessible to applications running on top

Primitive	Description
`Owner me()`	Return the current thread's owner-id.
`Owner be(Owner w)`	Switch the current thread's owner-id to `w` and returns the previous owner-id.
`Object owner(Object o)`	Return the current owner of object `o`.
`Object withdraw(Object o)`	Withdraw ownership over objects transitively reachable from `o`.
`Object acquire(Object o)`	Acquire ownership over objects transitively reachable from `o`.
`Object concurrentAcquire(Object o)`	Like `acquire`, supports concurrent acquire operations on `o`.

Table 4.1: Siaam virtual machine's *core* primitives, static methods of `siaam.untrusted.Core`.

of the virtual machine because they can be employed to arbitrarily manipulate the identity of the current thread in manner that can violate the isolation. Table 4.1 shows the available core primitives, defined as static methods of the `siaam.untrusted.Core` class. Ownership transfer is a two step operation. First, the current owner withdraws ownership over objects in the graph starting from `o` by calling `withdraw(o)`. After that operation, objects of the graph are protected from any read or write access and cannot participate in a new withdraw operation. Later, a thread holding the same starting reference may call `acquire(o)` in order to gain ownership over the underlying objects. Concurrent attempts to acquire the same starting reference have an unspecified behavior. Therefore it is the trusted APIs responsibility to ensure that only one thread will engage an `acquire` operation over a given withdrawn graph. The `concurrentAcquire` is a variant supporting concurrent attempts to acquire the same graph, at the cost of a compare-and-swap operation. Remark that only a reference to the starting object passed to `withdraw` can be used with `acquire` or `concurrentAcquire` in order to recover ownership over the corresponding graph of objects.

4.2.1 Static contexts

The JikesRVM is fully written in Java, threads seamlessly execute application bytecode and virtual machine's internals bytecode, passing constantly from one to the other. For instance, a simple object allocation "`new C()`" may trigger the loading of some new classes, call the memory allocator, and may even lead to a full garbage collection. All these tasks are handled by the thread devoted to the application that triggered the allocation, but the methods are internal to the JikesRVM. Similarly, many methods of the standard library must reenter the virtual machine bytecode to be completed. Although Siaam isolates the application bytecode, a virtual border must be drawn so that the virtual machine's internal bytecode escapes the isolation mechanisms. Indeed threads that reenter the VM legitimately access the same common resources and there is not reason to expand the notion of ownership into the virtual machine machinery.

We chose an approach where it is possible to statically decide whether a method is in the *application context* or the *virtual machine context*. A method in the application context is instrumented with all the isolation mechanisms whereas methods in the VM context are exempt. A naive approach is to hard-wire a set of Java package names that are in the VM context, so all the others are considered to be in the application context. For instance every package starting with "`org.jikesrvm.`" belongs to the VM context. However it is not

sufficient to ensure a total distinction between the bytecode executed for the VM internals and the bytecode executed for the application because the virtual machine itself employs the standard library, hence packages starting with "`java.`" are in both contexts.

If a method can be in both context, it must be compiled in two versions, one is instrumented with the isolation mechanisms and the other for the VM context isn't. When a method is invoked, the context of the caller is used to deduce which version of the method should be called. The decision is taken statically when the invoke instruction is compiled. Given a calling context, a method resolves either to the application context or the VM context. Methods in the library packages resolve to the same context as the call-site. If the caller of a standard library method is in the application context, then the isolated version of the method is linked. Otherwise the non-instrumented version is linked. Methods in the virtual machine packages always resolve to the VM context, whatever is the calling context. Finally methods in other packages always resolve to the application context.

The original JikesRVM doesn't support contexts. It requires to clone methods of the standard library and modify the virtual machine in several places to handle the context information. We integrated the large patch by Michael Bond[20] that provides the necessary adjustments to statically clone the methods from a selection of packages. The patch was initially developed to instrument or modify the behavior of library methods called by the application without changing the behavior of these same methods when called by the virtual machine. On this occasion we fixed some bugs related to code inlining in the presence of mixed contexts.

4.2.2 Object's owners.

Siaam's ownership-based isolation requires the ability to assign, retrieve and switch the owner of any heap object. Heap objects are either arrays or scalars, but the Java Language Specification does not regulate their representation in the memory. The object model in Jikes RVM (Figure 4.2) comprises a leading *header* section and the trailing scalar object's fields or array's length and elements. The header has a fixed structure regardless of the object nature. This section is hidden to the application code, indeed it is not part of the Java Language Specification and thus it has no official existence from the application point of view, nor it is accessible through the standard Java's reflection. It is only accessible from within the VM, using the `Magic` API.

In the JikesRVM, object references are raw memory addresses that points several bytes after (offset 0 in the figure) the object's header first byte. Indeed, to avoid computing complex displacement when accessing an array's element, object's references always points-to the raw memory address of the first element, right before resides the array's length. The header can be found at a fixed negative offset from the object's reference and is truly an array of bytes that can be interpreted arbitrarily, although it is divided in word-sized slots which are attributed a relative offset. The JavaHeader section originally comprises two words. The first one, `TIB`, holds a pointer to a structure describing the type of the object. The second one, `STATUS`, contains enough bits to store a lock, a hash code state and eight bits available for garbage collectors and miscellaneous extensions. The GCHeader section may contain words as required by the selected garbage collector, and the MiscHeader may contain other words for experimental extensions.

We extend the object header with two reference-sized words: `OWNER` and `LINK`. The `OWNER`

word stores a reference to the object's owner. Compared to the centralized ownership relation maintained by the global-state in our formal model, here the relation is distributed in every object's header.

The `LINK` word is inserted for technical reasons in order to optimize the efficiency of heap graph traversal methods. We also use, again for technical efficiency, one available bit (noted `OPAQUE`) in the `STATUS`. We comment on these two technical measures later.

Notice that there is an alternative layout of the `STATUS` word containing an inlined hash-code for the object and only two free bits, however the garbage collector we employ already consumes those two bits which left no available bit for our encoding. Although we could add a supplementary header word, or piggy-back it at the least significant bit of the `OWNER` word, it would be at the expense of memory foot-print or bitwise manipulations.

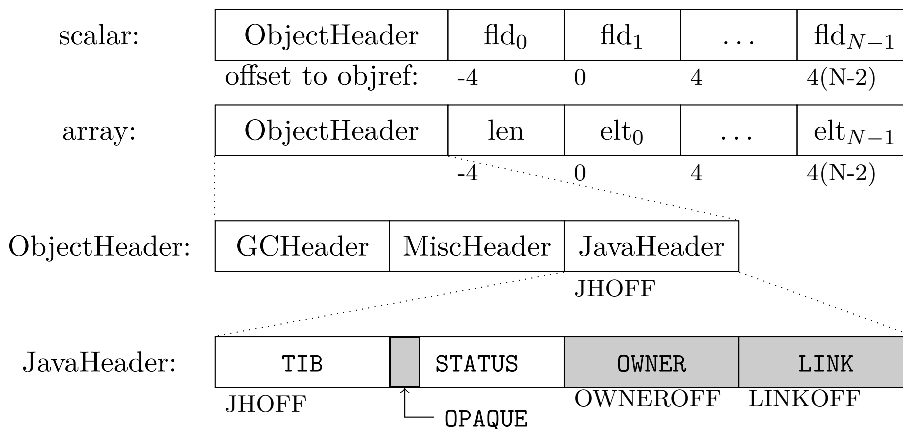


Figure 4.2: JikesRVM object model. Offsets in bytes from the object's reference are identified under each compound.

The implementation code for the `owner` primitive is shown in Figure 4.3. Given an object's reference `o` and an offset `f`, `Magic.getObjectAtOffset(o,f)` interprets the designated address-sized sequence of bytes into a Java `Object` reference.

```
1 package siaam.untrusted;
2 public abstract class Core {
3   public static Object owner(Object o) {
4     return Magic.getObjectAtOffset(o, JavaHeaderConstants.OWNEROFF);
5   }
6 }
```

Figure 4.3: Source code for the `owner` core's primitive.

Since we introduce hidden references to heap objects, we have to be careful regarding the garbage collector. Advanced garbage collectors, also known as *moving* collectors, reduce heap space fragmentation by occasionally moving objects to obtain large regions of continuous allocated or free memory. As objects are moved, references to relocated objects must be updated to the new location. The Jikes RVM integrates the `MMTk` framework that provides the garbage collection facilities. The framework however is unaware of the object-model and requires that the virtual machine collaborates in the references scanning process by indicating every memory location containing a reference. Thus we extend Jikes RVM collaboration so

that the OWNER and LINK references are also reported to MMTk.

4.2.3 Thread's owner-ID.

The virtual machine maps each Java Thread instance to an RVMThread instance, and each RVMThread to a native thread of the underlying operating system. RVMThread instances hold the VM-specific internal state of each Java thread, including the management of the thread's stack and the blocking operations. We introduce a new field Owner ownerId recording the thread's current owner-ID as illustrated in Figure 4.4. Both me and be primitives access this field. Every new thread inherits the owner-ID of its creator, but trusted APIs must enforce the following invariant:

There is at most one thread with a given owner-ID
executing bytecode in the application context. (4.1)

The invariant obviously avoid having concurrent threads operating with the same owner-ID and thus accessing the same objects. The invariant does not have to hold for bytecode in the VM context.

When the virtual machine starts, a primordial thread is allocated to execute the application's main method. This main thread is automatically provided with a unique owner-ID so it can run with the same isolation guarantee as any other thread.

```
1 package org.jikesrvm.scheduler;
2 import siaam.untrusted.*;
3 public class RVMThread {
4   [...]
5   Owner ownerId;
6 }
```

Figure 4.4: Integration of the thread's owner identifier.

Implementing both the me and be primitives is straightforward (Figure 4.5). The former returns the value of the current thread's ownerId field and the latter switches the field with the specified new owner-ID before returning the old one. A programming pattern that we will often use when implementing a trusted API consists in temporarily switching the current thread's owner-ID. The example in Figure 4.6 shows how to use a try/finally block in order to guarantee that the initial owner-ID is always recovered whatever exception may be thrown while performing operations with the alternative owner-ID.

It is worth noticing that threads are not owners in the sense that neither java.lang.Thread nor org.jikesrvm.RVMThread implements the Owner interface. But while a thread "carries" the identity of an Owner instance in its ownerId, it can access and update the fields of any object with a matching OWNER header slot. We abusively say that a thread "owns" objects it is allowed to access, and symmetrically that an object "belongs" to a thread.

4.2.4 Opaque objects.

Trusted APIs implement asynchronous communication channels with data structures storing withdrawn objects. These channels are represented by Java objects that must be shared

```
1 package siaam.untrusted;
2 public abstract class Core {
3   public static Owner me() {
4     return RVMThread.getCurrentThread().ownerId;
5   }
6   public static Owner be(Owner newId) {
7     RVMThread thread = RVMThread.getCurrentThread();
8     Owner previousId = thread.ownerId;
9     thread.ownerId = newId;
10    return previousId;
11  }
12 }
```

Figure 4.5: The `me` and `be` core primitives.

```
1 // switch to the new identity, backup the current one
2 Owner prevId = be(newId);
3 try{
4   // do something under the identity
5   // of 'newId'
6 }finally{
7   // always switch back to the previous identity
8   be(prevId);
9 }
```

Figure 4.6: Common pattern using the `be` primitive.

between the senders and the receivers, meaning they can be part of any message without disturbing the owner checking nor being affected by the ownership transfer mechanisms. In the Siaam formal specification the mailbox objects are discarded by the transitive owner-check and owner-transfer operations for the same reasons.

The Siaam core package defines the `Opaque` interface which is used to identify objects that must be discarded while exploring a message object graph. Objects of classes implementing the interface are *opaque*, they always belong to themselves and hide their content to the graph exploration operations. Mailboxes of the Siaam actor programming model API implement the `Opaque` interface, this way mailbox objects can be communicated between actors without revealing the list of pending messages they contain.

In a sense, opaque objects are actors without a thread. Threads directly call opaque object public methods and it is the opaque object's responsibility to manage concurrent invocations. Opaque objects are isolated since they own themselves from the moment they are allocated. Therefore when a thread enters an opaque object method, the method must switch the thread owner-ID so it matches the "this" pointer before accessing its internal state. Opaque objects code must be designed with care, as a thumb rule it should not expose its internal state through the invocation of an external object's method otherwise the method could exploit the current thread owner-ID to hijack the opaque object's state.

4.2.5 Ownership transfer

Consistency model. Ownership transfer is designed as a two-step process in order to ensure strict consistency of data and ownership updates involving a given `Owner`, and sequential consistency at the system level. In Siaam, `Owner`s are fully aware of any ownership modifi-

cation they are involved in. Whenever an object enters or exits an owner's domain, it must be the side effect of an explicit operation undertaken by the thread with the identity of the object's former or new owner respectively. An owner may lose the ownership over an object only when a thread carrying that owner-ID is performing a `withdraw` primitive operation. Symmetrically an owner may gain ownership of an object only when a thread with that owner-ID is performing an `acquire` operation.

The owner-local strict consistency ensures that the context of an owner `w` is stable (the set of objects within the context don't change) between two ownership transfer operations performed by a thread carrying the `w` owner-ID. Trusted programming model APIs must enforce that property by avoiding having several threads concurrently operating with the same owner-ID. Consequently, the `OWNER` header word of an object `o`, as observed by the `owner(o)` primitive, is not strictly consistent unless it matches the current thread's owner-ID. If it doesn't, `o` definitely does not belong to the current thread, but its exact owner isn't known due to potential concurrent ownership transfers involving `o`.

For instance, lets consider two concurrent threads respectively carrying the owner-ID `w` and `w'`, and an object `o`. If `w` observes that `owner(o) = w`, then `w` can definitely deduce that it owns `o`, at least until it performs an ownership withdraw operation. On the other hand, `w'` may observe `owner(o) = w` and deduce that it does not own `o` currently and definitely not until it performs an ownership acquire operation. However `w'` cannot expect the owner of `o` to be stable since `o`'s current owner may transfer the ownership to another owner at any moment.

Ownership withdraw. The operation verifies that all the objects in the transitive closure of the given starting reference belong to the current thread, and switches their ownership so that they become unusable to any owner. The withdraw operation does not modify the heap graph, it simply checks and updates the `OWNER` slot of every transferred objects. If there exists an object reachable from the starting reference that doesn't belong to the current owner, the operation is canceled and throws a `OwnerMismatch` exception.

The `withdraw` primitive is to the virtual machine what *transitive-owner-check* and *transitive-owner-transfer* are to the Siaam's formal model (see Section 3.4.4). In our formal model, the full transitive closure of the starting reference is owner-checked and then the same graph is explored again to update the ownership relation, all in a single, atomic step of the operational semantics. It is however not realistic to process in the same way in the implemented virtual machine for the following efficiency reasons:

— atomicity would require calls to `withdraw` to be globally serialized, which demands a costly system-wide synchronization.

— folding the operation into the owner-checking phase followed by the ownership update phase is costly in practice because it demands to explore the same graph of objects two times in a row.

The next paragraphs describe a transitive ownership withdraw operation that can be executed by a thread while other threads continue their current activity and potentially perform ownership transfers of their own. In practice the proposed algorithm must be translated into an iterative one because – recursive – calls are costly and the deepest message graph that could be handled would otherwise be limited by the available stack space. We detail the implementation of the iterative algorithm in the paragraph following the recursive one.

Recursive algorithm. Internally `withdraw(o)` progressively switches the ownership of objects as it explores the message graph starting from `o`. The algorithm, in its recursive form, is presented in Figure 4.7, in this paragraph we refer to lines with italic numbers preceded by ℓ as in *(ℓ3-5)*.
Transferred objects are put in a temporary ownership state where their `OWNER` slot points to the root `o` of the message passed to `withdraw` *(ℓ20)*. The reason is that we must use a value for the `OWNER` word that is guaranteed to be different from any owner-ID a thread could carry while the message exists, until an `acquire` operation receives it. That way no thread can access the objects while they are part of a message. It is not possible to chose the same value of `OWNER` for all the messages in the system because when the `withdraw` primitive explores a graph, it must be able to distinguish between (i) objects of the current owner that are encountered for the first time, (ii) objects that have already been reached and switched to the temporary ownership state *(ℓ17)*, and (iii) objects that are either part of another message or belong to another owner *(ℓ18)*. To distinguish the last two cases, it is necessary that objects that are part of different messages have different `OWNER` values. Notice that in the Siaam formal specification all messages belong to *None*, this is only possible because the message graph is entirely owner-checked before it is owner-transferred, therefore the owner-transfer phase only reaches objects that must be part of the current message.

Let `getReferenceOffsets(o)` be the function returning the list `offs` of offsets, relative to the address of object `o`, where references to other objects can be found (scalar fields or array elements). In case (i) the `OWNER` slot of the current object is switched to the root reference value *(ℓ20)* and the algorithm is applied recursively to every reference at offsets listed in `offs` *(ℓ21-24)*. In the second case the reference is discarded since it has already been taken into account earlier. Finally in case (iii) the withdraw operation must be aborted while exploring the graph, the `OWNER` slot of already reached objects must be rolled-back to their initial state (the current thread owner-ID) *(ℓ26)* before throwing the `OwnerMistmatch` exception *(ℓ9)*.

We said earlier that objects implementing the `Opaque` interface are immune to ownership transfers. When such object is encountered while exploring the graph, it is simply discarded as if it was already part of the message *(ℓ15)*. This way the algorithm never follows reference fields of the opaque objects. Therefore any object in the transitive closure of the message's starting reference that is only reachable via an opaque instance has its ownership left unmodified.

Moreover the withdraw primitive does not allow to send a message starting from an object that owns itself unless it is opaque *(ℓ5)*. The reason is that at line 17 in `withdrawrec`, the starting object would be immediately discarded since it already belongs to itself. This would leads to the situation where the message is accepted without owner-checking the objects reachable from the starting reference.

Only subtlety, the ownership transfer mechanism must not be exploited to violate the invariant (4.1). If the object pointed-to by the current owner-ID is transferred, the receiver may start a new thread carrying the same owner-ID, which clearly breaks the invariant. This situation may only arise if the current thread's owner-ID is an object belonging to itself since otherwise it cannot be transferred. Trusted APIs probably want to check that, after the ownership withdraw operation, the object pointed-to by the current owner-ID does not belong to the message starting object because it indicates that the object is part of the message. If the API detects that the owner-ID is part of the message, it may rollback the ownership transfer by calling `acquire` with the message starting reference and throw an exception to notify the application.

```
1  function withdraw(Object o) returns Object
2  begin
3    {prevent case when root object owns itself
4     and is not opaque}
5    if (owner(o) == o) and not (o instanceof Opaque)
6    then throw new IllegalArgumentException end
7
8    if withdrawrec(o, o) then return o
9    else throw OwnerMismatch end
10 end
11
12 function withdrawrec(Object o, Object root) returns boolean
13 begin
14   if (o == null) then return true end
15   if  o instanceof Opaque  then return true end {discard opaque object}
16
17   if owner(o) == root then return true end {discard already reached object}
18   if owner(o) != me() then return false end {owner mismatch}
19
20   updateOwner(o, root) {switch ownership}
21   for off in referenceFieldOffsets(o)
22   do
23     o2 := getObjectAtOffset(o, off)
24     if not withdrawrec(o2, root)  {recurse over reference fields}
25     then {rollback}
26       updateOwner(o, me())
27       return false
28     end
29   end
30 end
```

Figure 4.7: Recursive algorithm for the `withdraw` primitive. Parts that requires particular optimisation are highlighted.

Iterative implementation The optimization of the zero-copy message passing implementation is given a particular attention because the performance of the whole actor system depends on it. The three critical points of the algorithm are highlighted in Figure 4.7. First, deciding whether an object of the graph must be discarded or not should not be based on the relatively costly `instanceof` instruction that inspects VM internal structures through several indirections. Instead we read the `OPAQUE` bit of the object header where that information is encoded. Second, scanning the reference fields of an object (or the elements of an array of reference) is one of the most time consuming operation and should be optimized as much as possible. We use some techniques inspired by garbage collectors. Third, the conversion from recursion to iteration demands to translate the call stack into an explicit list structure. Furthermore the ownership acquire operation should not have to re-explore the message objects graph since the objects that are part of the message have already been discovered by the withdraw phase. We leverage the `LINK` word to support the iterative graph exploration algorithm. The produced linked list is maintained as long as the message exists so that the ownership acquire operation can efficiently traverse the objects of the message.

The iterative algorithm chains objects that are part of the message through their `LINK` word. We define the class `TransitiveClosure`(Figure 4.8) in charge of chaining objects and

listing the reference fields for each object of the list. Its `tail` member stores a reference to the last enqueued object. Objects are chained together by calling `enqueue(o)`, which sets the `LINK` word of the current tail to `o` and updates the `tail` reference to `o`, making `o` the new tail of the linked list. The entry point for the graph exploration is the method `explore` that expects the root of the graph has already been enqueued. The graph exploration algorithm traverses the linked list, starting from the message's root object. At each iteration, heap edges out of the current object are enumerated and passed to `processEdge(base, offset)` in the form of the current `base` object and the offset relative to `base`'s raw address where a reference is stored. Classes that extend `TransitiveClosure` must implement the `processEdge` method. The latter is in charge of enqueuing the reference found at the specified offset so that the graph exploration follows that edge. The constructed linked list is always terminated by the `DUMMY` object *(ℓ18)* so that every object that is part of a message has a non-`null` `LINK` word. We leverage specialized methods to efficiently enumerate the reference offsets for a given `base` object *(ℓ15)*. The technique, detailed in [46], was initially introduced in the JikesRVM to optimize the references scanning phase of the garbage collections. We simply adapted it for our transitive closure exploration class and observe about 20% speed improvement when transferring the ownership of a message.

```
1 abstract class TransitiveClosure {
2   protected abstract void processEdge(Object base, Offset offset);
3   public static final DUMMY = new Object();
4   private Object tail;
5
6   protected void enqueue(Object obj) {
7     if(tail != null) setLink(tail, obj);
8     tail = obj;
9   }
10
11  protected void explore() {
12    try{
13      Object base = tail;
14      while(base != null) {
15        call this.processEdge(base, offset) for each reference offset in base
16        base = getLink(base);
17      }
18    }finally{ enqueue(DUMMY); tail = null; }
19  }
20 }
```

Figure 4.8: The `TransitiveClosure` class.

The iterative ownership withdraw algorithm is implemented by class `ZeroCopy` (Figure 4.9) which extends `TransitiveClosure`. Its `root` field stores a reference to the root object of the current message, and its `ownerId` field refers to the current thread's owner-ID so it can be accessed efficiently. During the graph exploration, the following invariant is maintained: enqueued objects belong to the `root` object and are not opaque according to their `OPAQUE` header bit. Therefore when `processEdge` is called-back, the specified `base` object always belongs to `root` and isn't opaque. In `ZeroCopy` the `processEdge` method reads the object reference at the specified object and delegates the work to `processObject` *(ℓ29-30)*. The object `ref` received by `processObject` can be anything and may belong to any object, it is enqueued in the message list only if it meets the requirements to be part of the message. If

`ref` is `null` or opaque *(ℓ34-35)* it is discarded immediately. If `ref`'s owner has already been set to `root`, it is discarded as well since it must have already been enqueued *(ℓ36-37)*. Finally if `ref`'s owner matches the current thread's owner-ID, its ownership is switched *(ℓ39)* and the reference is enqueued in the message list *(ℓ40)* so that heap edges from `ref` will eventually be inspected. In case `ref` belong to a different owner, an exception is thrown *(ℓ42)* so the ownership transfer can be canceled.

The public entry-point to ownership withdraw is the `withdraw` method. It first makes sure that the root object is not self-owned unless it is opaque *(ℓ10)*. Then it bootstraps the graph exploration by calling `processObject` with the specified message root reference *(ℓ15)*. Unless the root is `null` or opaque *(ℓ34-35)*, its ownership is checked against the current owner-ID *(ℓ36-38)* and switched to itself before being enqueued *(ℓ39-40)*. Line 10 cannot return according to the initial test at line 10. Once bootstrapped, the graph exploration is performed by calling `explore` *(ℓ16)*. Whenever an exception is raised, it is caught and the objects that have been withdrawn so far are re-acquired to rollback the ownership relation to its initial point *(ℓ19)*. Once the graph exploration is complete, non-opaque objects that are part of the message belongs to the root object and are chained through their `LINK` header word, the root object being the head of the list.

```
1 class ZeroCopy extends TransitiveClosure {
2   private Object root;
3   private Owner ownerId;
4
5   public <T> withdraw(T root) {
6     this.ownerId = me();
7
8     // prevent case when root object owns itself
9     // and is not opaque
10    if(owner(root) == root && (!isOpaque(root)))
11      throw new IllegalArgumentException();
12
13    try{
14      this.root = root;
15      processObject(root);
16      explore();
17      return root;
18    }catch(RuntimeException e){
19      receive(root, ownerId);
20      throw e;
21    }finally{
22      this.root = null;
23      this.ownerId = null;
24    }
25  }
26
27  /* called back for each edge explored */
28  protected void processEdge(Object base, Offset offset) {
29    Object ref = Magic.getObjectAtOffset(base, offset);
30    processObject(ref);
31  }
32
33  private void processObject(Object ref) {
34    if(ref == null) return;
35    if(isOpaque(ref)) return;
36    Object refOwner = owner(ref);
37    if(refOwner == root) return;
38    if(refOwner == ownerId) {
39      updateOwner(ref, root);
40      enqueue(ref);
41    }else{
42      throw new OwnerMismatch(ownerId, refOwner, ref);
43    }
44  }
45 }
```

Figure 4.9: The `ZeroCopy` class.

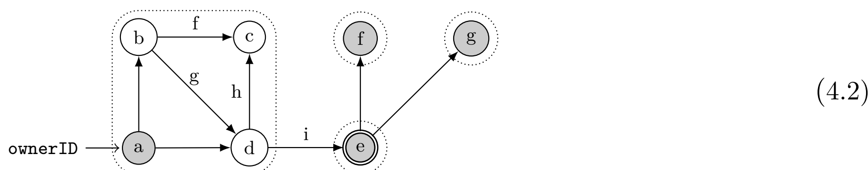
Acquire primitive. The acquire primitive, in its iterative form, is presented in Figure 4.10. It must be passed the root reference of a message, formerly supplied to and returned by the `withdraw` primitive. The primitive checks the root object owns itself, if it doesn't then the passed object is definitely not representing a receivable message *(ℓ4)*. If the specified object is an `Owner` that owns itself, it might be an active owner instead of a message root. This ambiguity is removed by checking the message invariant saying that every object that is part of a message has a non-`null` `LINK` word *(ℓ4)*. After these verifications, the message list

is traversed and the ownership of each object is switched to the current thread's owner-ID *(ℓ9)*. The LINK words are cleared as they are read *(ℓ10-12)*, until the DUMMY object is reached, marking the end of the chain.

```
1  public static <T> T acquire(T root) {
2    if(root == null) return null;
3    if(isOpaque(root)) return root;
4    if(owner(root) != root || getLink(root) == null) throw new NotReceivable(root);
5
6    Owner ownerId = me();
7    Object o = root;
8    do{
9      updateOwner(o, ownerId);
10     Object next = getLink(o);
11     updateLink(o, null)
12     o = next;
13   }while(o != TransitiveClosure.DUMMY);
14
15   return root;
16 }
```

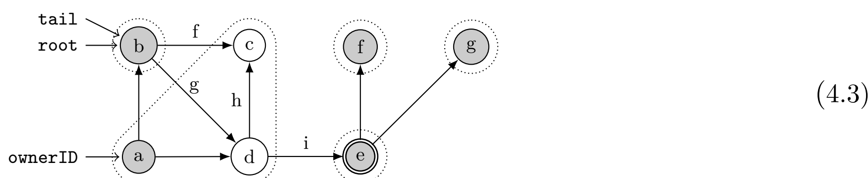
Figure 4.10: The `acquire` primitive.

Example. We illustrate the iterative ownership withdraw algorithm with the example shown in the following diagrams. Let consider the initial configuration of the heap depicted immediately below. Objects are represented by the circle nodes. Objects belonging to the same owner are surrounded by a a dotted border, the owner of an ownership domain is the only grey-colored node inside the delimited area (in this examples every owner owns itself). The current thread owner-ID is a, indicated by the pointer `ownerID`. Edges with a filled arrow represent field references, with the name of the field attached when necessary, for instance b's f field points-to c. Object e is opaque as denoted by the ring around its representative node. In this example we chose to withdraw the ownership of the message starting from b.



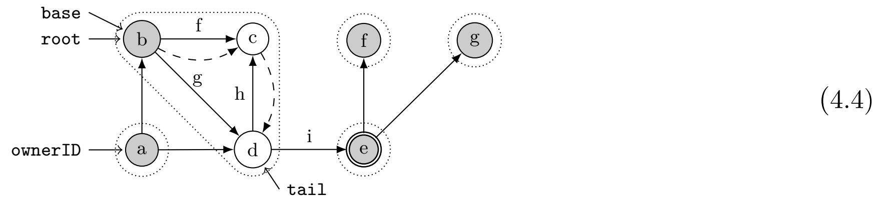
(4.2)

Below is the configuration after the initial call to `processObject(b)` at line 13 in Figure 4.9. The `root` pointer marks the message root. The `tail` pointer is initialized to the root as well after b was enqueued at line 38. Furthermore b's ownership has been switch so that it no longer belongs to the current thread. It now included in its own ownership domain.



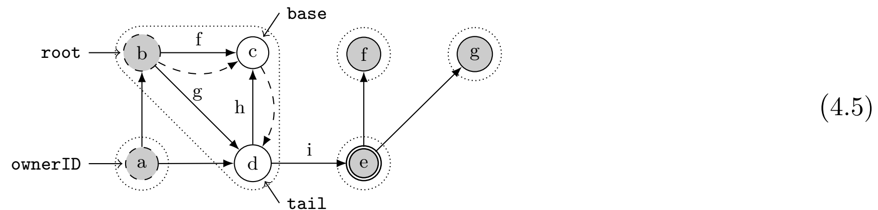
(4.3)

The next diagram illustrates the configuration right after the first iteration of `explore` complete. During this iteration the `base` pointer was pointing to b. The zero-copy `processEdge` method has been called twice. First with the offset of field f, which checked the ownership of c and switched it to b. We represent the references in the `LINK` words with dashed edges, here we see the link from b to c produced by enqueueing c. `processEdge` was then called a second time with the offset of field g. Again, d's ownership was checked, transferred into b's domain and finally enqueued after c. Therefore at the end of the iteration, the `tail` points-to d.



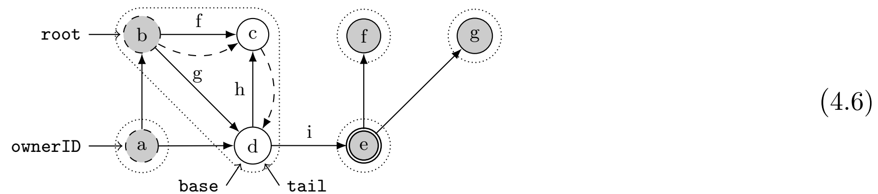
(4.4)

The second iteration of the graph exploration loop uses c as the base object since it is the next object in the `LINK` chain. There is no edge starting from c in the example configuration, maybe c has no reference field or because its reference fields are all null. The configuration at the end of the second iteration is illustrated below, only the `base` pointer moved compared to the previous configuration.



(4.5)

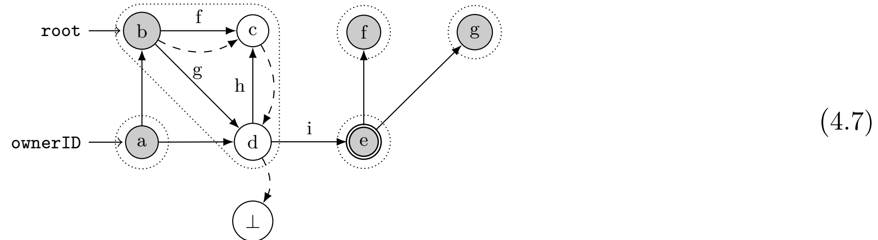
At the end of the third iteration, the configuration below shows no particular update except the `base` pointer used for the iteration was d. The `processEdge` method has been called with h's offset and then with i's offset. In the first case, c's ownership was already set to the message root b therefore no particular action was required (Figure 4.9 line 37). The edge from d to e through field i has been discarded to avoid adding e in the message list since e is opaque and should be left unmodified (line 35). As a result no new object were chained in the message list.



(4.6)

The fourth and last iteration of `explore` completes early since the new base pointer, d's `LINK` word, is null. The `finally` clause (Figure 4.8 line 18) enqueues the `DUMMY` reference — noted $\perp$ in the diagram — so that d's link word respects the message invariant. The final configuration when `withdraw` returns is depicted below. Every non-opaque object transitively reachable from the root b now belongs to b, and there is a chain from b through the `LINK`

header word that traverses these objects and ends with the dummy reference. The ownership withdraw algorithm did not change the heap graph, the edges from a to objects that no longer belong to a have been preserved.



(4.7)

Deep-copy message passing. We developed a fast deep-copy algorithm leveraging the `OWNER` and `LINK` words in order to copy arbitrary graph of objects without requiring an external mapping from original objects to copies. The deep-copy operation reuses the `TransitiveClosure` infrastructure,

In a first phase, copy objects are generated and references between originals and copies are created so that each original objects has its `LINK` word pointing-to its copy, and each copy has its `OWNER` word referring to the original. The bootstrapping consists in copying the message root object, creating the references, and then inserting the copy in the work-list.

At each iteration of the graph exploration loop, one copy object b is removed from the work-list and its reference fields are inspected. Initially copy objects fields still points-to original objects, therefore each object o pointed-to by a field f of b is original. If the original has not been copied yet, it is immediately copied and its copy c is inserted in the work-list (via c's `LINK`). Now o has its `LINK` word pointing to c and c's `OWNER` is o, thus $b.f$ must be updated with c. Once every fields of b have been inspected, b only points-to copy objects.

At the end of the first phase, every object of the original message has been copied and the copies are all linked together (plus the dummy object marking the end). In the second phase the list of copies is traversed, the ownership of each copy is set to the reference of the original message root's copy and the `LINK` word of every original object is reset to null. The deep-copy function returns the copy of the original message root, which may later be passed to the `acquire` primitive like if it was withdrawn by the `withdraw` operation.

Of course the deep-copy operation does not copy the opaque objects and performs ownership checking on the original objects.

4.3 Ownership-based isolation

4.3.1 Owner-checking barriers.

Siaam's ownership-based isolation mechanism ensures that a thread with a given owner-ID may only access fields and array elements of objects with a matching owner. This verification is performed at runtime each time an instruction uses a reference to read or write a field or an element of the pointed object. Our solution consists in performing an owner-checking barrier before every such access. These barriers are automatically inserted for the bytecode instructions listed in the first column of Table 4.11. The Java bytecode has variants of the array load and store instructions for each type of element, for instance `aaload` loads a reference element and `iastore` stores an integer. The second column in the table shows what

are the expected operands at the top of the current stack, the topmost operand being on the right. Array element indexes use an integer (*index*). Values of a `long` or `double` type occupy two word-sized slots (*value-1 value-2*) of the Jikes RVM stack, other types occupy a single slot (*value*). Finally the reference of the accessed objects (*objref*) takes one slot.

Bytecode	Jikes RVM Operand Stack (32bits slots)				Description
	slot 3	slot 2	slot 1	slot 0	
`[abcdfils]aload`			*objref*	*index*	Load an array element.
`[abcfis]astore`		*objref*	*index*	*value*	Store an array element.
`[dl]astore`	*objref*	*index*	*value-1*	*value-2*	Store an array element.
`getfield`				*objref*	Read an instance field.
`putfield`			*objref*	*value*	Write an instance field.
`putfield`		*objref*	*value-1*	*value-2*	Write an instance field.

Table 4.11: Owner-checked Java bytecodes and the expected operand stack.

We modify the Jikes RVM compilers in order to inject the `ownerCheckRead` barrier before an object is read by `?aload` or `getfield`, and the `ownerCheckWrite` barrier before an object is written by `?astore` or `putfield`. At the current stage, both barriers behave the same, we only show the source code for the write barrier in Figure 4.12. In section 4.4 we introduce immutable objects and modify the read barrier. The barriers take the accessed object's reference as a parameter and return silently if the current thread can proceed. They return immediately if the passed reference is `null` and let the virtual machine handle the null pointer exception *(ℓ2)*. Otherwise the current thread's owner-ID is compared with the accessed object owner *(ℓ5)* and an exception is thrown only if the two references do not match *(ℓ6)*.

```
1 public static void ownerCheckWrite(Object obj) {
2   if(obj == null) return;
3   Owner threadid = me();
4   Object owner = owner(obj);
5   if(owner == threadid) return;
6   throw new OwnerCheckWriteException(threadid, owner, obj);
7 }
```

Figure 4.12: Write owner-check barrier.

4.3.2 Constructors and instance fields.

When the Java virtual machine allocates an object, it clears every instance fields with a default value (`false` for booleans, `0` for numbers, `null` for references). When the object's initializer is executed, non-blank fields are implicitly set to their declaration value after the superclass initializer is implicitly invoked and returns. If a partially initialized object is passed to another owner during the construction process, the construction will fail if one of the initializers tries to access a field of the object. Although the isolation cannot be violated by passing a partially instantiated object, it can lead to unexpected behaviors of the partially initialized object's methods. In our implementation, we chose to allow passing partially initialized objects. After all the equivalent situation may arise in a classical threaded program if an object becomes accessible to a concurrent thread while it is being initialized.

```
1 class Strange extends Parent {
2   boolean b = true;
3   int i;          // blank instance field
4   Strange(Mailbox<Strange> mb) {
5     // Parent constructor is implicitly invoked here
6     // this.b is implicitly set to its initial value here
7     mb.put(this); // lost ownership over 'this'.
8     // setting this.i here would fail.
9     mb.get();     // assume the object pointed by 'this'
10                  // is in the received message, here
11                  // we gain back ownership over 'this'.
12    this.i = 1;   // succeed provided the previous assumption
13                  // otherwise fails.
14  }
15 }
```

Figure 4.13: A strange constructor that may complete in a specific case.

Java developers should already be aware of the unexpected behaviors that may arise in such situation.

The only possibility for the constructor to complete properly after passing the object is to receive it back first, which is illustrated by the special case in Figure 4.13. The same remark is applicable if one of the superclasses constructor passes the receiver object; the object must be received back before any field can be accessed. In Figure 4.13 the `Strange` initializer assumes that `this` belongs to the current actor when the `Parent` constructor returns.

4.3.3 Native methods.

Native methods are portion of programs not implemented using the Java bytecode but rather using ultimately a platform dependent binary code. These methods are not subject to the JVM management, they can access the address space of the virtual machine and interact with the underlying operating-system or the hardware with the same privileges as the JVM's process itself. Enforcing isolation in the presence of `native` methods is hardly possible since they can read and write arbitrary memory locations, including any isolated state. For these reasons, native methods are considered harmful by default, and the application bytecode should not be allowed to make native method invocations. This strict policy can be enforced during class-loading by rejecting bytecode containing native invocations.

4.3.4 Threads

In the JVM, concurrency is provided through the threading mechanism. However Siaam delegates concurrency matters to the trusted programming models. Applications should not be allowed to use other forms of concurrency than those provided by the trusted APIs. Thus any user bytecode that would create a thread of execution outside of the trusted APIs should be rejected.

4.3.5 Object finalizers

Every Java class can override the `Object.finalize()` method which is called when the garbage collector determines that there are no more references to an object. In the Jikes

RVM, a dedicated thread is in charge of calling the `finalize` method of each object before it is permanently discarded.

Finalizers are problematic because they may take any action including accessing the finalizing object and any object it may reach. Therefore isolation must be enforced, but what should be the owner-ID carried by the finalizer thread? On one hand it should match the object's owner so that the finalizer method can access the object. On the other hand an application thread may already be carrying that owner-ID and it would break the isolation invariant to have more than one thread executing with the same owner-ID. Although the finalizing object is definitely not reachable by any application thread, other objects reachable from it still might.

We define the special owner `FINALIZER_OWNER` that is only used by the virtual machine thread in charge of executing object's finalizers. The read owner-check barrier let the finalizer thread read any object. Before the `finalize` method of an object is called, its ownership is switched to `FINALIZER_OWNER` so the method can at least write fields of the "this" object without raising an owner mismatch exception.

4.4 Support for immutability and static variables

4.4.1 Final instance fields.

Should final instance fields of mutable objects be accessible to any actor? Instance fields declared `final` have their value set only once, during the object initialization, then their value don't mutate for the rest of the object's lifetime. Therefore final fields should not require to be preceded by a read owner-check when accessed. However, in the event that an object is passed before its final fields are initialized, the default values will be observable by the receiver thread. If the object is sent back to its "birth" actor, there is a chance that the constructor complete the initialization (like in the example Figure 4.13) and set the actual value of the final fields. The consequence in that very unusual situation is that a thread which first observed the object partially initialized, might observe different values of the object's final fields after its construction completes.

Therefore the question of allowing any actor to read an object's final field when that object isn't immutable must be asked. From the "birth" actor point-of-view, we see no objection since the actor is fully aware of passing the object and will be able to set the final field only if it acquires its ownership back. From the receivers point-of-view, observing an uninitialized field can lead to unexpected behavior if reading false, zero or null isn't expected. Furthermore, the receiver may suddenly read the actual value of the field (if he first passed the objects back to its "birth-place" where the initialization could continue). That observation would violate the invariant expected for a final field value. Moreover the isolation property would be violated in the sense that an unexpected side-effect could be observed.

Note that it is a known flaw of the initial Java Memory Model that has been fixed by the Java Specification Request 133 "Java Memory Model and Thread Specification Revision" [69], but from our own experimentations, the JikesRVM 3.1.2 does not implements that revision or at least not for our particular problem. In the absence of JSR133 implementation, some "safe" construction techniques are discussed in [48] where the author recommends not to publish the `this` reference during construction.

We shall formulate the same recommendation here if we allow concurrent threads to read final fields of the same object regardless of its ownership, at least until the JikesRVM is

compliant with the JSR-133 regarding final fields. Of course, allowing or forbidding unconditional read of final fields is a major programming model feature. In the current state of the virtual machine, it should be forbidden in order to guarantee strict isolation, meaning the read owner-checking barrier must be maintained before a final field is read.

4.4.2 Immutable objects

Definition. An immutable object has a fixed state after it is constructed. Because an immutable object cannot change during its lifetime, sharing such object wouldn't violates the isolation property. Immutable objects are extensively used in Java for various reasons. Strings and Integers make excellent map keys and set elements, saving memory space as different data-containers may use the same object with no synchronization issue. The ability to share immutable objects is performance critical since maps and sets are very common containers in programs. Siaam not only ensures immutables have a fixed state but also requires that they only refer to other immutables; to this extend we give the following recursive definition. A reference may only points to immutable objects if all the followings are met: *(i)* its declared type is a `final` class `T`, *(ii)* every declared and inherited non-static fields of class `T` are `final`, *(iii)* recursively, reference fields of `T` may only points to immutables or opaque objects. Instances of a class are immutable objects if the class verifies the last two points.

Example. We illustrate these concepts with the example shown in Figure 4.14. Class `A`, `B` and `I` have final non-static fields only. Class `A` isn't final, therefore the virtual machine could load new subclasses of `A` declaring non-final fields. For this reason, a reference declared of type `A` may points to a mutable object. Class `B` is final but it has a field of type `A` that might reach a mutable object, therefore instances of `B` aren't immutable in the Siaam's definition. Class `I` is final, it declares two final fields of a primitive type and inherits a final primitive and a final `I` reference (field `i`) from `A`. The circular dependency introduced by field `i` is solved by induction. Assuming references of type `I` may only reach immutable objects, then field `i` may only reach immutable objects and every fields of `I` reach immutable objects, which is correct. Obviously, instances of `I` are immutable. Although `A` isn't declared final, it is worth noticing that instances of `A` are even immutable since the class fulfill requirements *ii* and *iii*.

Initialization. Any thread can allocate and construct an immutable instance, new objects belong to the current owner until their construction complete. After a new object is allocated, the corresponding instance initializer is invoked. There are two kinds of instance initializer, *primary* constructors invoke a direct superclass constructor and *alternate* constructors invoke alternative initializer of the same class. Noticeably, given a final non-static field `V` of class `C`, the Java Language Specification[49] states *V is definitely assigned after an alternate constructor invocation*, meaning at each level of the class hierarchy, only the deepest invoked alternate initializer (a primary constructor) can and must assign the non-static final fields declared at its level. Since immutable objects only have non-static final fields, only primary instance initializers need a write access to the object. Consequently, the last method that may modify a *"wannabe"* immutable object of class `C` is a primary constructor declared by `C`.

Again we clarify these concepts with Figure 4.14 where super constructors are invoked explicitly. Class `B` declares three constructors, line 11 the *alternate* constructor invokes an

```
1 // has immutable instances.
2 // a reference of that type may reach a mutable object.
3 class A {
4   final char c; final I i;
5   A() { super(); ... } // primary
6 }
7
8 // has mutable instances.
9 final class B extends A {
10   final A a;
11   B() { this(new I()); } // alternate
12   B(A _a) { super(); this.a = _a; } // primary
13   B(long l) { super(); this.a = new I(l); } // primary
14 }
15
16 // has immutable instances.
17 // a reference of that type may only reach immutable objects.
18 final class I extends A {
19   final long l; final double d;
20   I() { this(7); } // alternate
21   I(long _l) { this(_l, 3.3); } // alternate
22   I(long _l, double _d) {
23     super(); this.l = _l; this.d = _d; } // primary
24 }
```

Figure 4.14: Various classes producing mutable or immutable instances.

alternative. At line 12 and 13, both methods are primary constructors declared by `B`. Since `B` is final, we know these two constructors are the last methods requiring a write access to this. A similar reasoning applies for `I`: the first alternate constructor line 20 invokes a second one in line 21 which in its turn invokes the primary constructor line 22. Curiously, `A`'s unique primary constructor isn't necessarily the last method requiring a write access to `this` since it may be invoked from a primary constructor declared in a direct subclass of `A`.

We modify the Jikes RVM in order to transfer the ownership of an immutable instance to the special `IMM_OWNER` owner before its last primary constructor returns. The ownership transfer is systematic when the primary constructor is declared in a final immutable class. On the other hand, when a primary constructor of a non-final class returns, we must verify at runtime that the constructed object is exactly of that class type before setting its final owner. The verification is generated by the compiler only if there is a chance that the constructed object is immutable. In our example, the primary initializer of class `A` must dynamically ensure the `this` reference points to an allocated instance of `A` before switching its ownership to `IMM_OWNER`. Indeed if the allocated object is of class `I`, the initializer of `A` is not the last initializer to require a write access to `this`. On the other hand, the primary initializer of `I` can definitely lock the state of the constructed object since `I` is final.

The ownership transfer isn't recursive, it only applies to the constructed object. Fields of an immutable object points to other immutables, therefore when a field is initialized to a non-null reference, the object pointed-to has already been constructed and indeed already belongs to `IMM_OWNER`.

Owner-check barriers. The read owner-check barrier is modified (Figure 4.15) to return silently when the owner of the checked object is IMM_OWNER *(ℓ5)*. We chose to set the IMM_-OWNER constant to null so the verification is efficiently handled without requiring to access a static field to get the actual value of IMM_OWNER at runtime — although the just-in-time compiler might be smart enough to perform that kind of constant propagation. The write barrier do not need to be changed because the Java specification enforce that a final field is never modified except when it is initialized so there should be no write barrier ever generated to check the ownership of an object owned by IMM_OWNER. Even though a write barrier is emitted, it would raise an ownership exception, as expected, since no thread owner-ID can match IMM_OWNER. Remember when the final field of an immutable object is initialized, the object still belongs to the current thread's owner-ID, therefore the write barrier is generated and should succeed on that particular occasion.

```
1 public static void ownerCheckRead(Object obj) {
2   if(obj == null) return;
3   Owner threadid = me();
4   Object owner = owner(obj);
5   if(owner == threadid || owner == IMM_OWNER) return;
6   throw new OwnerCheckReadException(threadid, owner, obj);
7 }
```

Figure 4.15: Read owner-check barrier.

Zero-copy message passing. Immutable objects can be concurrently accessed without breaking the isolation property. Regarding the zero-copy message passing scheme described in section 4.2.5, immutables must be handled like opaque objects. The invariant that only immutables are reachable from immutable objects allows to discard any immutable object without risking to miss any mutable object in the transitive closure of the message's root. Immutables' headers are flagged with the OPAQUE bit at the same time as their OWNER word is set to IMM_OWNER.

4.4.3 Arbitrary frozen objects

Frozen objects[68] are mutable objects that became immutable. They are very useful to share large structures that must initially be mutable to be elaborated. For instance, we can imagine an actor-based compiler, inside this compiler a series of actors load the source code and elaborate an intermediate representation, then the IR is frozen and passed by reference to several concurrent actors performing various analyses on the structure without modifying it. Since the structure is immutable, the objects are not owner-checked when transferred, thus the communication has an $O(1)$ cost.

The core primitive freeze(Object o) transfers the ownership of the objects transitively reachable from o to IMM_OWNER and sets their OPAQUE bit to true. The implementation of the primitive (Figure 4.16) is as simple as acquiring the ownership over the message starting from o, but instead of setting the OWNER words to the current thread's owner-ID, they are set to IMM_OWNER.

Object freezing limits the opportunities of optimizing-out owner-check barriers The optimization algorithm assumption is that if a read or write owner-check barrier succeeds at one point in the program, the checked object belongs to the current thread until an instruction

```
1 public T <T> freeze(T root) {
2   if(root == null) return null;
3   if(isOpaque(root) || owner(root) != root || getLink(root) == null)
4     throw new NotFreezable(root);
5
6   Object o = root;
7   do{
8     updateOwner(o, IMM_OWNER);
9     setOpaque(o,true);
10    Object next = getLink(o);
11    updateLink(o, null);
12    o = next;
13  }while(o != TransitiveClosure.DUMMY);
14
15  return root;
16 }
```

Figure 4.16: The `freeze` primitive.

that potentially changes that object's ownership is executed. It means, in particular, that if a field of an object is successfully read by an instruction, a write of any field of that object is expected to succeed at the next instruction, therefore it is possible to eliminate the write owner-check. Now, if we introduce frozen objects, a successful read of an object's field may succeed because that object is frozen, meaning a consecutive write of that object may fail. As a result, only a successful write owner-check barrier can guarantee that a consecutive write on the same object must succeed. By introducing frozen objects, we limit the possible optimizations and increase the runtime overhead due to owner-check barriers. Note that truly immutable objects are not a concern here because Java prevents final fields updates in the first place, so there is no write barrier to eliminate anyway.

Our experiments with the Dacapo benchmarks show that allowing frozen objects increases, by 10% in average, the total number of owner-check barriers performed by a program. The intra-procedural optimization algorithm maintains 11% of the barriers it usually removes when freezing is not possible. The measured execution time increases in average by less than 3%, which is affordable considering the benefits in term of programming.

4.4.4 Static fields and Enum types

Static fields. Class variables, or static fields, or even static variables, provide shared state that is not bound to a particular instance. In Java and most object-oriented languages, static fields are bound to a class. These variables are a serious threat to isolation since they provide a data sharing mechanism. In order to enforce isolation, the most straightforward way to deal with static fields is to systematically prevent actors from accessing them. This can be checked during class-loading, by refusing any bytecode loading or storing a value from a class variable. Or at runtime, by raising an exception when one of these bytecode is evaluated.

However we opt for a less restrictive approach where we allow the use of `final static` class variables of a primary type or referencing immutable objects. We also think it is fair to supply arbitrary immutability for arrays holding primitive values or references to immutables since they are necessary to support Java's enum types. Immediately before a class is needed

for the first time (the possible situations are detailed by the Java specification), its static initializer is executed and assigns the static fields to their initial values. Class initializers are executed by normal threads with their current owner-ID, but there is not way a thread may leak data through class variables. First, Siaam makes sure only final class variables may be initialized. Final class variables of a primitive type indeed can't be employed to share mutable state. Second, since Siaam only allows to store references to immutable objects into final class variables, it is again impossible to share mutable objects through them. Finally, Siaam allows to store references to arrays (or multi-dimensional arrays) of primitive or immutables elements. Unfortunately we cannot rely on the Java virtual machine to enforce array's immutability since Java lacks of such feature. Java's enum types could serve as immutable arrays, however enums are syntactic sugars that compile to classes and final static arrays that shall not be modified, which bring us back to where we started.

Our solution is to apply the freezing mechanism, presented in the previous section, to arrays that are reachable from the static fields of a class. The compiler generates the necessary instructions right before the end of each class initializer method. When the initializer is about to return, the static fields of the class are inspected and those of an array type are frozen. Since we allow some arrays to be read but not written, we must modify the optimization algorithm, as explained in the previous section, so that a successful array read owner-check barrier does not imply that a consecutive array write owner-check barrier will succeed. Our measures in the Dacapo benchmarks shows that allowing frozen arrays has a non-significant impact on optimization and performances. The total number of performed barriers increases by less than 2%, and less than 1% of the barriers removed in absence of frozen arrays are maintained. The average execution time is increased by approximatively less than 1%. These results are explained by the relatively sparse usage of arrays in the benchmarks and in object oriented programming languages in general. Note that using frozen scalar objects, it is possible to offer arbitrary static objects by freezing any object reachable from a final static field.

Enum types. The enum types accepted by Siaam must be immutables. The Java language provides the `enum` keyword to declare type safe enumerations. For instance `enum MyEnum { ELT1, ELT2 };` creates the enumeration `MyEnum` with two elements of type `MyEnum`. Each element of a particular enum type is an instance of the class generated by the compiler for the enum type, and the user cannot instantiate other objects of that type. The enum type class gets static final fields pointing to each generated instance. The elements are also listed in a static array of the enum class. It is even possible to declare arbitrary instance fields and methods in an enumeration, which makes elements of an enumeration not necessarily immutables. The code in Figure 4.17 illustrates how the compiler transforms an enum type declaration into a standard Java class. Instances of the `MyEnum` class is not immutable because of its non final `c` field. Consequently the generated static fields, and the static array, are not compatible with the constraints expressed in the previous paragraph. If we remove the problematic `c` field, we make `MyEnum` an immutable type and its static final fields verifies Siaam's constraints.

4.4.5 Conclusion

The relaxation of the "share-nothing" ideology, associated with the properties of immutable objects, guarantees the isolation and allows developers to use the classic program-

```
1 enum MyEnum {
2   ELT1, ELT2;
3   public int c = 0;
4   public void inc() { c++; }
5 }
```

```
1 final class MyEnum extends Enum {
2   public static final MyEnum ELT1;
3   public static final MyEnum ELT2;
4   private static final MyEnum[] $VALUES;
5   public int c = 0;
6   public void inc() { c++; }
7   private MyEnum(String name, int value) {
8     super(name, value);
9   }
10  static {
11    ELT1 = new MyEnum("ELT1", 0);
12    ELT2 = new MyEnum("ELT2", 1);
13    $VALUES = new MyEnum[]{ELT1,ELT2};
14  }
15 }
```

Figure 4.17: Java enum type example. Left: Java declaration for the `MyEnum` enum type. Right: equivalent of the compiler-generated class in Java. The accessor for the `$VALUES` static field is not presented.

ming idioms involving enumerated types and global constant definitions as well as static frozen arrays without sacrificing the performances.

Some JVMs implementing software-based processes solves the problem of static variables by simply providing each isolated application with a copy of the variables so they can be safely modified. Although we could replicate that mechanism in Siaam, holding a copy of all the static variables for each actor may consume a non-negligible part of the memory in configurations where thousands of actors are alive. It is nonetheless an issue requiring more investigations since we currently have to allow a few methods of the Java standard library to mutate static variables and access shared objects through them.

Another solution would be to create a system actor with granted access to all the static variables defined in the standard library. Other actors would communicates with this actor in order to access the variables. Similar actors could govern the mutation of static variables for user-defined classes.

4.5 Trusted programming models

In this section we presents the trusted APIs we developed, either using the untrusted core primitives or above another trusted API.

4.5.1 Siaam Actors API.

The Siaam actor-based trusted API implements the formal model given in section 3. Two classes are publicly available, `siaam.actor.Actor` and `siaam.actor.Mailbox`.

Actors. Classes extending `Actor` must implement the behavior of the actor in the `run` method. Calling the `start` method of an actor creates a dedicated thread executing the actor's `run` method. Each thread dedicated to an actor uses that latter's reference for its owner-ID. The `start` method is detailed in Figure 4.18, it proceeds as following. First the current thread withdraw its ownership over the started actor *(ℓ8)* private state. Then the current owner-ID is temporarily switched to the starting actor's reference *(ℓ14-20)*. In that

short period, the starting actor's private state is acquired *(ℓ16)* so that its ownership matches its own reference. The dedicated thread is created *(ℓ17)*, which inherits the current owner-ID. Finally at *(ℓ18)*, the dedicated thread is started, which simply calls the `run` method of the starting actor, with the starting actor object reference being the current owner-ID (Figure 4.19).

```
1 package siaam.actor;
2 import static siaam.untrusted.Core.*;
3
4 public abstract class Actor implement Owner {
5   protected abstract void run();
6
7   public final void start() {
8     Actor actor = withdraw(this);
9     if(owner(me()) != me()) {
10      acquire(actor);
11      throw new RuntimeException("Forbidden,␣sent␣current␣thread␣owner ID.");
12    }
13
14    Owner prev = be(actor);
15    try{
16      actor = acquire(actor);
17      ActorThread t = new ActorThread(actor);
18      t.start();
19    }finally{
20      be(prev);
21    }
22  }
23 }
```

Figure 4.18: The `siaam.actor.Actor` class.

```
1 final class ActorThread extends Thread { // package private
2   private final Actor actor;
3   ActorThread(Actor actor) {
4     this.actor = actor;
5   }
6   public void run() {
7     this.actor.run();
8   }
9 }
```

Figure 4.19: The `ActorThread` class definition.

Mailboxes. The `Mailbox` interface (Figure 4.20) specifies the `get` method that returns a pending message and the `put` method that sends a message.

We give the definition of the abstract mailbox `siaam.actor.AbstractMailbox` in Figures 4.21–4.24. The abstract mailbox must be backed by a message queue implementing the `enqueue` and `dequeue` methods. The backing queue must support concurrent enqueue operations. We made `AbstractMailbox` opaque so that references of mailboxes may be shared. Each mailbox is assigned a receiver actor upon construction, stored in the `receiver` field.

```
1 package siaam.actor;
2 public interface Mailbox<T> {
3   public T get();
4   public void put(T message);
5 }
```

Figure 4.20: The `siaam.actor.Mailbox` interface.

```
1 package siaam.actor;
2 import siaam.untrusted.Opaque;
3 abstract class AbstractMailbox<T> implements Mailbox<T>, Opaque {
4   protected abstract T dequeue();
5   protected abstract void enqueue(T message);
6   private final Actor receiver;
```

Figure 4.21: The `siaam.actor.AbstractMailbox` class.

The constructor sets the receiver reference, for that it must temporarily wear the owner-ID matching the constructed mailbox (`this`), since opaque objects owns themself.

```
7   AbstractMailbox(Actor receiver) {
8     Owner prev = be(this);
9     try{
10      this.receiver = receiver;
11    }finally{ be(prev); }
12  }
```

Figure 4.22: The `AbstractMailbox` constructor.

The `put` method is presented in the Figure 4.23. To deposit a message in a mailbox, the current thread must withdraw the ownership over the objects of that message using `withdraw`, then the invariant that the current owner-ID is not part of the message is verified and the operation is canceled when necessary by re-acquiring the message and throwing an exception. When the objects of the message have been successfully withdrawn, the owner-ID is temporarily switched to the mailbox's `this` reference and the message is enqueued.

```
13  public void put(T message) {
14    message = withdraw(message);
15    if(owner(me()) != me()) {
16      acquire(message);
17      throw new RuntimeException("Forbidden");
18    }
19    Owner prev = be(this);
20    try{
21      enqueue(message);
22    }finally{ be(prev); }
23  }
```

Figure 4.23: The `AbstractMailbox.put` method.

Getting a message from the mailbox requires that the current thread's owner-ID at the

entry of the `get` method matches the receiver set for the mailbox. To access the `this.receiver` field, the owner-ID must be temporarily switched to `this`, storing the previous owner-ID in the `prev` local variable. If the mailbox's receiver and the `prev` reference matches, a message is dequeued, otherwise an exception is thrown to notify that the entry thread is not the appropriate receiver. After switching the owner-ID back to `pred`, the thread acquires ownership over the received message so it can be returned to the application.

```
24   public T get() {
25     T message;
26     Owner prev = be(this);
27     try{
28       if(prev != receiver) throw new RuntimeException(
29         "Current␣thread␣is␣not␣receiver␣for␣this␣mailbox.");
30       message = dequeue();
31     }finally{ be(prev); }
32     return acquire(message);
33   }
34 }
```

Figure 4.24: The `AbstractMailbox.get` method.

We provide concrete implementations of the abstract mailbox with various backing queues, such as the Java's FIFO `LinkedBlockingQueue` which support concurrent enqueues (Figure 4.25).

```
1 public final LinkedMailbox<T> extends AbstractMailbox<T> {
2   private final LinkedBlockingQueue<T> queue;
3   public LinkedMailbox(Actor receiver) {
4     super(receiver);
5     Owner prev = be(this);
6     try{
7       queue = new LinkedBlockingQueue<T>(); // "unbounded" capacity
8     }finally{ be(prev); }
9   }
10  protected void enqueue(T message) { // assume: me() == this
11    queue.put(message);
12  }
13  protected T dequeue() { // assume: me() == this
14    return queue.take();
15  }
16 }
```

Figure 4.25: The `siaam.actor.LinkedMailbox` class.

4.5.2 ActorFoundry.

Using the trusted Siaam actors API, we build an implementation of the ActorFoundry API as described in [61]. Since it is entirely relying on trusted bytecode, it can be supplied as a third-party library instead of being integrated in the VM bytecode. Our implementation is compliant with existing ActorFoundry applications, although it currently does not support synchronization constraints.

The API provides three methods of the `osl.manager.Actor` class:

— `void send(ActorName actor, String message, Object... args)`. Sends an asynchronous message to the specified actor with the given arguments.

— `Object call(ActorName actor, String message, Object... args)`. Sends an asynchronous message to the specified actor with the given arguments and wait for a response that is returned by the primitive.

— `ActorName create(Class<? extends Actor> cls, Object... args)`. Creates a new actor from a specified class extending `osl.manager.Actor` and the specified arguments passed to its constructor. Returns the new actor's name.

Each actor created with `create` has a fixed, unique name and a dedicated thread. Actor names are opaque objects implementing the `osl.manager.ActorName` interface. Name references can be shared system-wide, they are employed as communication channels by the `send` and `call` primitives. The `message` string passed to the asynchronous communication methods must correspond to a method name of the receiver actor marked with the `osl.manager.annotation.message` annotation (otherwise the message will be discarded). Internally actor names are backed by a mailbox of the Siaam actors API, and actors are backed by a Siaam actor. The behavior of each Siaam actor is a loop that waits for a message to be available in the actor's mailbox and dispatches it using the Java Reflection API by matching the message's string and arguments types to an actor's method. The `call` method uses a temporary trusted mailbox (supplied by `siaam.actor`) to wait for the response of the receiver before returning it to the sender.

4.5.3 M:N cooperatively scheduled actors

The number of concurrent actors in the Siaam API and the ActorFoundry API is limited by the maximum OS thread count and the amount of available memory for stacks. In order to overcome this limitation, it is necessary to provide a programming model where actors execute a run-to-completion task for each received message. This way a large number of actors can be scheduled over a small number of threads. Each actor occupies a worker thread for the duration of a certain number of tasks and eventually hands-off to another actor which have pending messages. This is the mutualisation strategy adopted for instance by Kilim, O-Kilim and Akka.

We implemented a trusted API on top of the virtual machine untrusted core primitive that is able to efficiently schedule a large number of actors over one or several pools of threads. Then we adapted the API to mimic the ActorFoundry and O-Kilim official APIs so we could directly run programs from the ActorFoundry, SOTER and Kilim code-base, with Siaam's ownership-based isolation.

Chapter 5

Static analysis and its usages

Contents

5.1 Introduction **107**
5.2 Intermediate Representation **109**
5.3 Standard analyses framework. **110**
5.4 The Siaam analysis framework. **115**
5.5 Some programming models and their ownership-based semantics. **117**
5.6 The Siaam Analysis **120**
5.7 Extension to frozen objects. **140**
5.8 Usages of the Siaam analysis results **142**
5.9 Implementation **145**

5.1 Introduction

This section describes a whole-program static analysis[39] to optimize away owner-checking on field read and write barriers. The analysis is based on the observation that an instruction accessing an object's field does not need an owner-checking if the object accessed belongs to the executing actor. Any object that have been allocated or received by an actor and hasn't been passed to another actor ever since, belongs to that actor. The algorithm's output is interpreted as an under-approximation of the owner-checking removal opportunities in the analyzed program. In a program not performing any message passing, the analysis would eliminate all owner-checks.
Considering a point in the program, we say an object (or a reference to an object) is *safe* when it always belongs to the actor executing that point, regardless of the execution history. By opposition, we say an object is *unsafe* when sometimes it doesn't belong to the current actor. We extend the denomination to instructions that would respectively access a safe object or an unsafe object. Finally an instruction passing at least one unsafe object to another actor is unsafe, otherwise it is a safe message passing instruction. A safe instruction won't ever throw any OwnerException, whereas an unsafe instruction might.
The result of this analysis is employed both offline and at runtime. First, it is accurate enough to assist programmers by pinpointing potential misuses of unsafe references during the application development process. Second, it provides information to effectively eliminate redundant or worthless owner-checking, thus reducing the owner-checking barriers overhead

at runtime.

5.1.1 Outline

The algorithm is structured in two phases, first the *safe dynamic references analysis* employs a local must-alias analysis to propagate owner-checked references along the control-flow edges. It is optionally refined with an inter-procedural pass propagating safe references through method arguments and returned values. Then the *safe objects analysis* tracks safe runtime objects along call-graph's and method control-flow's edges by combining an inter-procedural points-to analysis and an intra-procedural live variable analysis. Both phases depends on the *transfered abstract objects* analysis that propagates unsafe abstract objects from the communication sites downward the call graph edges.

By combining results from the two phases, the algorithm computes conservative approximations of unsafe runtime objects and safe variables at any control-flow point in the program. The owner-check elimination for a given instruction s accessing the reference in variable V processes as following (figure 5.1). First the unsafe objects analysis is queried to know whether V may points-to an unsafe runtime object at s. If not, the instruction can skip the owner-check for V. Otherwise, the safe reference analysis is consulted to know whether the reference in variable V is considered safe at s thanks to dominant owner-checks of the reference in the control-flow graph.

The two phases of our analysis are independent, it is possible to disable one and replace it with a very conservative approximation. Disabling one phase allows faster computation but less accurate results. We implemented the safe references analysis as a code optimization pass in the Siaam virtual machine, so that intra-procedural owner-check eliminations are performed without the need of a costly whole-program analysis.

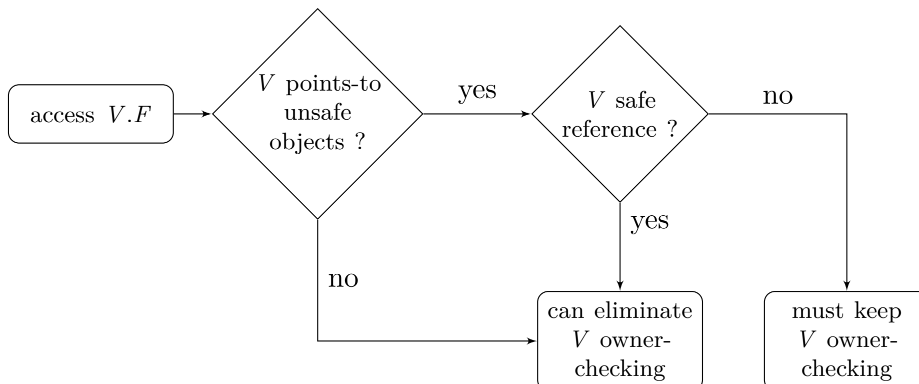


Figure 5.1: Owner-check elimination decision diagram. The left-most question is answered by the *safe objects analysis*. The right-most question is answered by the *safe references analysis*.

5.2 Intermediate Representation

5.2.1 Program representation.

The Siaam analysis framework employs an intermediate representation of the program of type *I-prog* (figure 5.2). It is handy to instantiate the parametric program type defined in section 3.3 with a new method body definition. The method body (type *I-mb*) comprises the the map from local variables names to their respective type, the list of lexical statements and the body's control-flow graph.

$$\textbf{type_synonym} \quad \begin{aligned} \textit{I-mb} &= (\textit{vname} \rightharpoonup \textit{ty}) \times \textit{statement list} \times \textit{cfg} \\ \textit{I-prog} &= \textit{I-mb prog} \end{aligned}$$

Figure 5.2: Intermediate program declaration.

5.2.2 IR language.

The intermediate representation language (Figure 5.3) is a sparse version of the virtual machine language (Figure 3.11) where the stack locations are reified by local variables, the branching instructions are translated into a control-flow graph and the instructions on primitive data types are deliberately absent because they are not relevant to our analysis.

Local variable names are of the *vname* type, parameters are indexed by natural number (type synonym *pindex*, naming convention π) from zero in the declaration order, the *this* pointer being at index 0. Lexical statements are of the following kinds: body entry-point, parameter index to local variable identity, caught exception to local variable identity, local variable assignment, object's field assignment, return statement, exception throwing. Right-hand side of assignment expressions comprise the null reference, read of variable, object allocation, read of object's field, method invocation.

The reserved local variable name $\underline{any}$ is used in place of variables of the *Void* type or any non-reference type and any literal constant or any operation on primitives. All other local variables are of a reference type *Class C*. The intermediate language doesn't include any control flow statement such as conditionals, loops or try/catch, instead the control-flow graph of every method body is available, as described in the following section.

5.2.3 Body's control-flow graph.

A control-flow graph[6, 82] (type *cfg*) for a method body is a set of directed edges $(n, n') ::$ $nat \times nat$ from the lexical statement at position n to a successor lexical statement at position n'. Each edge represents a possible path followed by the execution flow.

$$\textbf{type_synonym} \qquad cfg = (nat \times nat)\ set$$

A control-flow graph has a unique head which is an `entry` statement. There is a unique tail as well, which is a return statement. There must be only a single `entry` and a single return statement in each control-flow graph. Methods with a *Void* return type can return the *any* variable. Thus the shortest method body lexical statements list is $[\texttt{entry}, \texttt{ret}\ \textit{any}]$ with the following control-flow graph: $\{(0, 1)\}$. Note that the statements could be ordered differently in the body list, any permutation is acceptable as long as the control-flow graph is adapted accordingly.

datatype	*statement* =	
	`entry`	entry-point
	| *vname* := `Par` *pindex*	parameter identity
	| *vname* := `CaughtXcpt`	caught exception identity
	| *vname* := *rhs-expr*	local variable assignment
	| *vname.vname*{*cname*} := *var-expr*	field assignment
	| `ret` *var-expr*	return
	| `throw` *vname*	exception throw
	rhs-expr =	
	var-expr	
	| `Val` *Null*	null pointer reference value
	| `new` *cname*	object allocation
	| *vname.vname*{*cname*}	field access
	| *vname.mname*(*vname list*)	invoke method
	var-expr = `Var` *vname*	load variable

Figure 5.3: Lexical statements of the intermediate representation.

Exceptional flow edges. Each exceptional path is modeled with a control-flow edge to the first instruction of every potential trap handlers. The beginning of a trap handler is materialized by a caught exception identity statement of the form $V :=$ `CaughtXcpt`, loading the caught exception into the local variable V. For every potentially throwing instruction, there is an edge starting from each of its predecessors. If an instruction may have a side-effect before throwing an exception, there is also an edge starting after that instruction. Exceptions that are not caught by any trap handler do not generate any control-flow edge. Call statements have exceptional edges before and after themselves to model possible side-effects of the callee before an exception is raised.
Siaam introduces the *OwnerMismatch* exception that may be thrown before a field access statement, but not after since the exception prevents unwanted side-effects violating the isolation property.

5.3 Standard analyses framework.

Our analysis is client of several standard intra- and inter-procedural program analyses. It requires a call-graph representation, an inter-procedural points-to analysis, an intra-procedural liveness analysis and an intra-procedural must-alias analysis. Each of these analyses exists in many different variants offering various tradeoffs of results accuracy and algorithmic complexity, but regardless of the implementation, they provide a rather standard querying interface. In the next paragraphs, we define a standard model for each class of analysis used in Siaam. In order to instantiate the standard analyses framework, one has to provide an implementation of each model (an adapter from the actual analysis representation to the model defined by Siaam).

We declare the HOL locale *standard-framework* (5.1) fixing the concrete implementations of the standard analyses required by Siaam. The next paragraphs provides a description of an abstract model for these analyses. For the rest of the chapter, we assume all the definitions are in the context of this locale or an extension of this locale, so every function fixed in the

framework's locale is directly available.

$$
\begin{aligned}
&\textbf{locale}\ \textit{standard-framework} = \\
&\quad \text{fixes}\ \textit{targets} :: \textit{cg-targets} \\
&\quad \text{and}\ \textit{sources} :: \textit{cg-sources} \\
&\quad \text{and}\ \textit{pt} :: \textit{pointsto} \\
&\quad \text{and}\ \textit{aoc} :: \textit{absobjclass} \\
&\quad \text{and}\ \textit{maa} :: \textit{mustalias} \\
&\quad \text{and}\ \textit{live} :: \textit{liveness}
\end{aligned}
\tag{5.1}
$$

5.3.1 Program's call graph.

A call graph[50] is a directed graph that represents the invocation relationship between the program's methods. Edges of the graph go from source *call sites* to target *call graph nodes*. A single method of the program may be represented by zero, one or several nodes in the program's call-graph. Methods that cannot be reached by any execution path isn't represented in the call graph.

Sensitivity. In a context-insensitive call graph, each reachable method has exactly one node because no calling context is taken into account at the call sites. Whereas in a context-sensitive call-graph, various kind of context can be employed to distinguish calls from the same syntactic call site, thus several nodes may represent the same method, each within a different calling context. The program representation for the Siaam analysis can handle any kind of context.

Abstraction. Our model for a call-graph node (naming convention m) is a tuple $(c, C, M) ::$ *cg-node* where $M ::$ *mname* and $C ::$ *cname* are the name and class of the represented method and $c ::$ *nat* is the unique identifier of the calling context. We note N_P the finite set of nodes created in the call graph for the program P.

With the definition of a call graph node, we can identify a precise lexical statement in the context of a node. A *node statement* (naming convention s) is of the form $(m, n) ::$ *stmt* where $m ::$ *cg-node* is the specified node and n is the index of the designated lexical statement in the body of the method represented by m. For instance, the node statement $((c, \underline{List}, \underline{insert}), n)$ designates the $(n+1)$-th lexical statement in the body of method $\underline{List}.\underline{insert}$ with the calling context c. The formal notations for *program points* are *Before s* (abbreviated $\bullet s$) for the program point before statement s and *After s* (abbreviated $s\bullet$) for the program point after s.

$$
\begin{aligned}
\textbf{type_synonym}\quad & \textit{cg-node} = \textit{nat} \times \textit{cname} \times \textit{mname} \\
& \textit{stmt} = \textit{cg-node} \times \textit{nat} \\
\textbf{datatype}\quad & \textit{program-point} = \textit{Before stmt} \mid \textit{After stmt}
\end{aligned}
\tag{5.2}
$$

We now give the type definitions for the two functions *targets* :: *cg-targets* and *sources* :: *cg-sources* fixed by the analysis framework (5.1) that are employed to query the call graph. *targets* P s must return the set of call graph nodes that may be called as an effect of executing the node statement s in the program P. Symmetrically, *source* P m must return the set of node statements that may invoke the call graph node m. The two functions must guarantee that every node targeted by a given statement has that statement in its set of

sources and every statement that is a source of a given node has that node in its set of targets: $m \in \mathit{targets}\ P\ s \longleftrightarrow s \in \mathit{sources}\ P\ m$.

$$\textbf{type_synonym} \quad \begin{aligned} &\mathit{cg\text{-}targets} = \mathit{I\text{-}prog} \Rightarrow \mathit{stmt} \Rightarrow \mathit{cg\text{-}node\ set} \\ &\mathit{cg\text{-}sources} = \mathit{I\text{-}prog} \Rightarrow \mathit{cg\text{-}node} \Rightarrow \mathit{stmt\ set} \end{aligned} \tag{5.3}$$

Since we may work with node statements (*stmt*) more often than lexical statements, we introduce the following shorthands. *stmts-of* P m constructs the list of node statements for the call graph node m, it simply creates as many elements (m, n) as the count of lexical statements in the body of the method associated with m. Similarly *cfg-of* P m constructs the control-flow graph for the method associated with m where vertices are couples of statements instead of lexical statement indexes (reminder, $f`A$ is the image of set A by the function f). The function *type-of* P m V returns the type of variable V as it is declared in the body of the method associated with the call graph node m.

$$\begin{aligned} &\mathit{stmts\text{-}of} :: \mathit{I\text{-}prog} \Rightarrow \mathit{cg\text{-}node} \Rightarrow \mathit{stmt\ list} \\ &\mathit{stmts\text{-}of}\ P\ m = (\mathsf{let}\ (c,\ C,\ M) = m;\ (_,\mathit{stmts},_) = \mathit{method}\ P\ C\ M \\ &\qquad\qquad\qquad \mathsf{in}\ \mathit{map}\ (\lambda\ n\,.\ (m,n))\ [0\ ..\ {<}|\mathit{stmts}|]\ ) \\ \\ &\mathit{cfg\text{-}of} :: \mathit{I\text{-}prog} \Rightarrow \mathit{cg\text{-}node} \Rightarrow (\mathit{stmt} \times \mathit{stmt})\ \mathit{set} \\ &\mathit{cfg\text{-}of}\ P\ m = (\mathsf{let}\ (c,\ C,\ M) = m;\ (_,_,\mathit{cfg}) = \mathit{method}\ P\ C\ M \\ &\qquad\qquad\quad \mathsf{in}\ (\lambda\ (n,\ n')\,.\ (\ (m,\ n),\ (m,\ n')\ ))`\mathit{cfg}) \\ \\ &\mathit{type\text{-}of} :: \mathit{prog} \Rightarrow \mathit{cg\text{-}node} \Rightarrow \mathit{vname} \Rightarrow \mathit{ty} \\ &\mathit{type\text{-}of}\ P\ m\ V = (\mathsf{let}\ (c,\ C,\ M) = m;\ (\mathit{lcls},_,_) = \mathit{method}\ P\ C\ M\ \mathsf{in}\ \mathit{lcls}\ V) \end{aligned} \tag{5.4}$$

We note *preds* P s and *succs* P s the set of respectively the predecessors and the successors of statement s in the control-flow graph of s's node.

$$\begin{aligned} &\mathit{preds} :: \mathit{I\text{-}prog} \Rightarrow \mathit{cg\text{-}node} \Rightarrow \mathit{stmt\ set} \\ &\mathit{preds}\ P\ (m,n) = \{s' \mid (s',(m,n)) \in \mathit{cfg\text{-}of}\ P\ m\} \\ &\mathit{succs} :: \mathit{I\text{-}prog} \Rightarrow \mathit{cg\text{-}node} \Rightarrow \mathit{stmt\ set} \\ &\mathit{succs}\ P\ (m,n) = \{s' \mid ((m,n),s') \in \mathit{cfg\text{-}of}\ P\ m\} \end{aligned} \tag{5.5}$$

The head and tail statements of the control-flow graph of the node m are produced by *entry-of* :: *I-prog* ⇒ *cg-node* ⇒ *stmt* and *exit-of* :: *I-prog* ⇒ *cg-node* ⇒ *stmt* (not shown).

The notation $P \vdash \langle S \rangle_{(m,n)}$ verifies that in program P the statement (m, n) where $m = (c, C, M)$ represents the lexical statement S :: *statement* at position n in the body of $C.M$. For the succinctness of this notation we will often omit the program notation. For instance we write $\langle \underline{r1} := \mathtt{new}\ \underline{List} \rangle_{((_,\underline{List},\underline{insert}),3)}$ to designate the fourth lexical statement of method $\underline{List}.\underline{insert}$.

$$\begin{aligned} &_ \vdash \langle _ \rangle_{_} :: \mathit{I\text{-}prog} \Rightarrow \mathit{statement} \Rightarrow \mathit{stmt} \Rightarrow \mathit{bool} \\ &P \vdash \langle S \rangle_{(m,n)} \longleftrightarrow S = (\mathit{stmts\text{-}of}\ P\ m)_{[n]} \end{aligned} \tag{5.6}$$

5.3.2 Points-To Analysis.

A points-to analysis[65, 92, 57] for object-oriented programs computes an over-approximation of the runtime objects a reference variable or a reference object field may points-to. Runtime objects are usually coalesced into *abstract objects* by their allocation site, so statically it isn't possible to distinguish between the infinite number of runtime objects that can be allocated by a given statement.

Sensitivity. The sensitivity of the points-to analysis[89] is often linked to the sensitivity of the call-graph and conversely. A points-to analysis is *context-sensitive* if it computes a different points-to graph for each call-graph node. The analysis is *flow-sensitive* if it produces a different points-to graph for each statement of a method.

Abstraction. In our abstraction, a reference variable is a couple (m, V) :: *ref-var* representing the reference variable V :: *vname* in the call graph node m :: *cg-node*. An abstract object is an arbitrary name o :: *absobj*, to simplify the model we will assume that names are natural numbers in the finite set $O_P \subset \mathbb{N}$. O_P contains all the names used to identify abstract objects while analyzing the program P. The runtime objects can be coalesced in an arbitrary way by the points-to analysis. A reference field $(o, (F, C))$ designates the value of the field F :: *vname* declared in class C :: *cname* for the abstract object o.

Internally the analysis constructs a points-to graph with edges going from references to abstract objects. The general form of an edge is (x, o), where x is either a reference variable or a reference object field. We say the reference x *may points-to* the runtime objects coerced by the abstract object o, and the *points-to set* of x is the set of all successors of x in the points-to graph. Two references *may-alias* if and only if their respective points-to set have a non-empty intersection, otherwise we say the references *must-not alias* meaning they never points-to the same runtime object.

The analysis framework locale fixes the functions *pt* :: *pointsto* returning the points-to set (type *absobj set*) for a given variable reference or object field reference, and *aoc* :: *absobjclass* returning the class name of the nearest common superclass of the runtime objects coalesced into a specified abstract object.

type_synonym $absobj = nat$
$ref\text{-}var = cg\text{-}node \times vname$
$ref\text{-}field = absobj \times (vname \times cname)$

$pointsto = I\text{-}prog \Rightarrow stmt \Rightarrow (ref\text{-}var + ref\text{-}field) \Rightarrow absobj\ set$
$absobjclass = I\text{-}prog \Rightarrow absobj \Rightarrow cname$

5.3.3 Must-Alias Analysis.

The must-alias analysis[5] answers the *must-alias* question, indicating whether two references at their respective position in a method always alias a common object at runtime. Our model considers two kinds of reference: value computed by right-hand-side expression of statements, and value of local variables at a given statement. The must-alias analysis maps each reference to a *key* with the following interpretation: if two references maps to the same key then they must-alias, otherwise they may not always alias. Two variables V and V' taken at their respective position s and s' must-alias if both couples (s, V) and (s', V') map to the same key. Similarly a variable V taken at s must-alias the right-hand-side of statement s' if both (s, V) and s' map to the same key. Sometime the must-alias analysis cannot give any aliasing information about a reference and associates it with the unknown key. The value numbering approach is described in [17], inspired by SSA numbering[66]

Abstraction. The function *maa* :: *mustalias* fixed by the analysis framework locale computes the must-alias key (type *key option*) associated with a given reference that is either a call graph node statement or a variable name in the context of a node statement. The returned key is an optional natural number, when *Some* κ is returned, the must-alias information is available and the key associated with the reference is κ :: *key*. When no aliasing information is available the undefined key *None* written *Unknown* is returned. We note $K_m \subset \mathbb{N}$ the finite set of keys that can appear in the must-alias analysis of the call graph node m.

$$\begin{aligned}
\textbf{datatype}\quad & key = nat \\
& Unknown \equiv None \\
\textbf{type_synonym}\quad & mustalias = I\text{-}prog \Rightarrow (stmt + (stmt \times vname)) \Rightarrow key\ option
\end{aligned} \tag{5.7}$$

For a single example of how to use the must-alias analysis, consider the statements $\langle V_1 := V_2.F\{C\}\rangle_{s_1}$ and $\langle V_1.F\{C\} := V_3\rangle_{s_2}$ with s_2 being the sole successor of s_1 in the flow graph. If the must-alias information for V_{11} at s_1 is $\lfloor\kappa\rfloor = maa\ P\ (Inr\ (s_1, V_1))$ we deduce that V_1 must-alias every other reference associated to the key κ at the program-point before s_1. Note that V_1 is assigned at s_1 but takes its new value at the program-point after s_1, therefore the key κ reflects the value of V_1 before the assignment. Let's assume the key of the right-hand-side expression in s_1 is $\lfloor\kappa'\rfloor = maa\ P\ (Inl\ s_1)$, then we should have $maa\ P\ (Inr\ (s_2, V_1)) = \lfloor\kappa'\rfloor$ as a consequence of the assignment in s_1.

sensitivity. Field-insensitive must-alias analyses only track local variables, but field-sensitive ones are able to track object fields as well. An object field is trackable over a range of statements if it isn't updated neither by the current method nor any method that is invoked inside the range. An intra-procedural escape analysis or an inter-procedural side-effects analysis is usually employed to determine whether an object field reference can be tracked. Back to our previous example, we can consider a third statement $\langle V_4 := V_1.F\{C\}\rangle_{s_3}$ that is the sole successor of s_2. If the field of V_1 is trackable then the right-hand-side expression of s_3 have the same key as V_3 at s_2: $maa\ P\ (Inl\ s_3) = maa\ P\ (Inr\ (s_2, V_3))$, otherwise it has its own key.

For completeness, the must-alias key of the special variable *any*, which is however not of a reference type, is fixed to *Unknown*.

5.3.4 Liveness Analysis.

Liveness analysis[6] gives the set of live local variables at a specified program point. A variable is *live* at specified point of the control-flow if its value may be read before its next write. We say a variable is *dead* otherwise. More specifically, a variable is *live-in* the statement s if it is live at the program point $\bullet s$ (before), and *live-out* if it is live at the program point $s\bullet$ (after).

Our abstraction of the liveness analysis also tracks live and dead method parameters. A method parameter, identified by its index (type *pindex*), is read by parameter identity statements ($\langle _ := \texttt{Par}\ \pi\rangle_s$) but never written. A method parameter is live from the method entry-point to its last use, or immediately dead if it is never assigned to a local variable.

liveness is the type of the functions associating a set of variable names and formal parameter indexes to a program point.

$$\textbf{type_synonym} \quad liveness = I\text{-}prog \Rightarrow program\text{-}point \Rightarrow (vname + pindex)\ set \tag{5.8}$$

5.4 The Siaam analysis framework.

5.4.1 Ownership-oriented programming models semantics.

We define the *analysis-framework* locale that extends the *standard-framework* and fixes three functions expressing the ownership semantics of the programming model(s) employed in the analyzed program. The generic framework offers a mean to encode various ownership-oriented programming models semantics through the definition of functions describing the requirements and the side-effects of every program's statement in terms of object and reference ownership.

$$\begin{aligned} &\textbf{locale}\ analysis\text{-}framework = standard\text{-}framework\ + \\ &\quad \text{fixes}\ pre\text{-}conditions :: requirements \\ &\quad \text{and}\ side\text{-}effects :: effects \\ &\quad \text{and}\ pt\text{-}filter :: pointstofilter \end{aligned} \tag{5.9}$$

A statement can require the following over-approximated pre-conditions to be verified at it's entry-point in order to guarantee a normal execution, otherwise it *may* throw an exception:

— a set of local variables *must be* safe.

— a set of abstract objects *must be* safe.

For instance, let's consider the following invoke statement $\langle \underline{mb}.\underline{put}([\underline{msg}])\rangle_s$ and assume the variable $\underline{mb}$ is of type *Class* $\underline{Mailbox}$. The statement s calls the message passing method defined in the Siaam actor API, which requires that all the objects transitively reachable from the parameter $\underline{msg}$ are safe. Thus all the abstract objects transitively reachable from $\underline{msg}$ in the points-to graph of the analyzed program must appear in s's ownership preconditions. The statement $\langle V.F\{C\} := unit\rangle_{s'}$ only succeeds if the variable V is safe (regardless of the objects it may points-to), therefore the set of reference variables that must be safe before s' must comprise V.

A statement may enforce some ownership side-effects if its pre-conditions are satisfied:

— a set of local variables *must become* safe.

— a set of abstract objects *may become* unsafe.

The execution semantics associated with ownership side-effects for a given statement are:

— either the statement completes and all the side-effects happen,

— or the statement fails with an exception and none of the side-effects happen.

In the previous example, s transfers all the objects reachable from $\underline{msg}$ to another owner, the set of abstract objects that may become unsafe is over-approximated by the abstract objects in the transitive closure of $\underline{msg}$ in the program's points-to graph. The statement s' accesses a field of the object pointed-to by V and implicitly performs an owner-check of that object,

so in the control-flow point after s', V is necessarily safe, because otherwise an exception is raised and the execution doesn't flow to s''s normal successor.

We don't offer the possibility to declare a set of abstract objects that must become safe because an abstract object may represent more than one instance of runtime object. Declaring such a set would mean the statement has the side-effect of checking the ownership of every objects represented by the abstract objects, which is likely impracticable or very strange. Declaring a set of local variables that may become unsafe is also voluntarily avoided, instead the analysis will automatically manage unsafe variables according to the set of abstract objects that may become unsafe.

Ownership pre-conditions. The ownership preconditions are checked at the very end of the Siaam analysis in order to partition safe statements and unsafe statements. At the end of the Siaam analysis, the safety information for every local variables and abstract objects at every program point is available. Thus it is possible to compare the pre-conditions expressed by each statement with the safety information computed for the program point before the statement. The requirements of a statement are encoded as a set of *requirement* elements. $Safe_V\ V$ encodes that variable V must be safe, and $Safe_O\ o$ encodes that the abstract object o must be safe. A function of type *requirements* returns the set of pre-conditions associated with a given statement.

$$\begin{array}{rl} \textbf{datatype} & \textit{requirement} = \textit{Safe}_V\ \textit{vname} \mid \textit{Safe}_O\ \textit{absobj} \\ \textbf{type_synonym} & \textit{requirements} = \textit{I-prog} \Rightarrow \textit{stmt} \Rightarrow \textit{requirement set} \end{array} \tag{5.10}$$

Ownership side-effects. The ownership side-effects of a statement are encoded as a set of *effect* elements. *UnsafeEff* o encodes the effect that invalidates the safety of the abstract object o, meaning at least one runtime object represented by o may be unsafe from the program-point after the statement. The *SafeEff* κ encoding indicates that the safety of any reference with a must-alias key equal to κ is validated from the program-point after the statement. The usage of must-alias keys greatly simplify the program analysis because we are more interested in treating indifferently all references to the same object than tracking the content of every variables independently.

A function of type *effects* returns the set of effects associated with a given statement.

$$\begin{array}{rl} \textbf{datatype} & \textit{effect} = \textit{SafeEff key} \mid \textit{UnsafeEff absobj} \\ \textbf{type_synonym} & \textit{effects} = \textit{I-prog} \Rightarrow \textit{stmt} \Rightarrow \textit{effect set} \end{array} \tag{5.11}$$

Filtered points-to information. Programming models can filter-out arbitrary points-to graph edges to hide some heap paths, for instance actor's mailboxes contains references to messages but these messages should be hidden. The framework-fixed function *pt-filter* :: *pointstofilter* returns a boolean indicating if the specified edge, a head reference and a tail abstract object, should be visible to the Siaam analysis. We define the points-to analysis *filtered-pt* :: *pointsto* that filters-out edges from the standard framework's points-to graph according to *pt-filter*. Finally, *filtered-pt-trans* computes the set of abstract objects transitively reachable via visible points-to edges from a reference variable or an abstract object at a given statement.

type_synonym $\textit{pointstofilter} = \textit{I-prog} \Rightarrow \textit{stmt} \Rightarrow (\textit{ref-var} + \textit{ref-field}) \Rightarrow \textit{absobj} \Rightarrow \textit{bool}$

$$\begin{aligned}&\textit{filtered-pt} :: \textit{pointsto}\\&\textit{filtered-pt}\ P\ s\ r = \{o \in \textit{pt}\ P\ s\ r \mid \textit{pt-filter}\ P\ s\ r\ o\}\end{aligned} \tag{5.12}$$

$$\textit{filtered-pt-trans} :: \textit{I-prog} \Rightarrow \textit{stmt} \Rightarrow (\textit{vname} + \textit{absobj}) \Rightarrow \textit{absobj set}$$

5.4.2 Programming models API abstraction.

Until now we have only provided abstractions to analyze sequential parts of programs. We can model the sequence of statements an actor may execute, and associate it with customized ownership semantics. Our choice is to employ existing standard analyses instead of providing new implementations of them that would fit exactly the semantics of ownership-based concurrent programming models. We now face the problem that the standard call graph and points-to analyses must be aware of the underlying operations performed by the programming models APIs in order to faithfully reproduce their effects. Therefore we have to provide a minimal mockup implementation of each API's methods that mimic the necessary side-effects. Although analyzing the program with the full API implementation is possible, it might introduce unnecessary burden in the construction of the points-to graph and call graph.

For instance we must provide a fake *Mailbox*.*put* method that at least stores the reference of the passed message in the mailbox instance object, so that later the fake *Mailbox*.*get* method can return that reference. By doing so, the points-to analysis will correctly report that the set of abstract objects returned by *get* corresponds to the set of abstract objects passed to *put*. Notice that the filtered points-to graphs are introduced to hides the edges from the mailbox objects to the messages they contain.

Some standard analyses can even be sensitive to concurrency, usually to support multi-threaded programs. We must provide a supplementary statement *Spawn* $V.M(\textit{Vs})$ to model the invocation of $V.M(\textit{Vs})$ in a concurrent thread of execution.

datatype $\textit{statement} = \ldots \mid \textit{Spawn}\ \textit{vname}.\textit{mname}(\textit{vname list})$

The mockup implementation of Siaam's *Actor*.*start* includes the spawn statement $\langle \textit{Spawn}\ V.\underline{run}([\,])\ \rangle$ to model the call to the *run* method of actor V in a concurrent thread of execution.

5.5 Some programming models and their ownership-based semantics.

The following functions will help constructing sets of ownership side-effects. *optional-to-set* takes an optional value and produces a set. *safe-effects* produces a set of safe side-effects given a set of must-alias keys. Similarly, *unsafe-effects* produces a set of unsafe side-effects. *safeo-preconds* produces a set of safe-object preconditions given a set of abstract objects.

$$\begin{aligned}&\textit{optional-to-set} :: {'a}\ \textit{option} \Rightarrow {'a}\ \textit{set}\\&\textit{optional-to-set}\ x = \textsf{case}\ x\ \textsf{of}\ \textit{None} \Rightarrow \emptyset \mid \textit{Some}\ a \Rightarrow \{a\}\\&\textit{safe-effects} :: \textit{key set} \Rightarrow \textit{effect set}\\&\textit{safe-effects}\ \textit{keys} = (\lambda\,\kappa\,.\ \textit{SafeEff}\ \kappa)`\textit{keys}\\&\textit{unsafe-effects} :: \textit{nat set} \Rightarrow \textit{effect set}\\&\textit{unsafe-effects}\ \textit{absobjs} = (\lambda\,o\,.\ \textit{UnsafeEff}\ o)`\textit{absobjs}\\&\textit{safeo-preconds} :: \textit{key set} \Rightarrow \textit{requirements set}\\&\textit{safeo-preconds}\ \textit{absobjs} = (\lambda\,o\,.\ \textit{Safe}_O\ o)`\textit{absobjs}\end{aligned} \tag{5.13}$$

5.5.1 Implicit owner-checks

We formalize in this section the ownership-based semantics implicitly attached to the statements of the IR language. We only show the rules producing non-empty preconditions and side-effects.

Ownership requirements. Both reading from and writing to an object's field referenced by the variable V requires V to be safe:

$$\begin{aligned}&\textit{implicit-pre-conds} :: \textit{requirements}\\&\textit{implicit-pre-conds}\ P\ \langle _ := V.F\{C\}\rangle_s = \{\textit{Safe}_V\ \ V\}\\&\textit{implicit-pre-conds}\ P\ \langle V.F\{C\} := _\rangle_s = \{\textit{Safe}_V\ \ V\}\end{aligned} \tag{5.14}$$

Ownership side-effects. As a direct consequence of the previous preconditions, successfully reading from or writing to a field of the object referenced by the variable V at statement s confirms that V, hence the must-alias key of V at s, is safe. Freshly allocated objects are always safe. Finally, the *Null* value may always be considered safe. Note that none of the implicit rules generate unsafe side-effects.

$$\begin{aligned}&\textit{implicit-side-effects} :: \textit{effects}\\&\textit{implicit-side-effects}\ P\ \langle _ := V.F\{C\}\rangle_s = \textit{safe-effects}\ (\ \textit{optional-to-set}\ (\textit{maa}\ P\ (\textit{Inr}\ (s,\ V))))\\&\textit{implicit-side-effects}\ P\ \langle V.F\{C\} := _\rangle_s = \textit{safe-effects}\ (\ \textit{optional-to-set}\ (\textit{maa}\ P\ (\textit{Inr}\ (s,\ V))))\\&\textit{implicit-side-effects}\ P\ \langle _ := \texttt{new}\ C\rangle_s = \textit{safe-effects}\ (\ \textit{optional-to-set}\ (\textit{maa}\ P\ (\textit{Inl}\ s)))\\&\textit{implicit-side-effects}\ P\ \langle _ := \texttt{Val}\ \textit{Null}\rangle_s = \textit{safe-effects}\ (\ \textit{optional-to-set}\ (\textit{maa}\ P\ (\textit{Inl}\ s)))\end{aligned} \tag{5.15}$$

Filtered points-to analysis. The points-to graph filtering function associated with the implicit owner-checks accepts all edges from the original graph.

$$\begin{aligned}&\textit{implicit-pt-filter} :: \textit{pointstofilter}\\&\textit{implicit-pt-filter}\ P\ s\ r\ o = \textit{True}\end{aligned} \tag{5.16}$$

5.5.2 Siaam actor programming model.

Filtered points-to analysis. The points-to graph filter hides edges from and to mailbox objects. This way mailboxes and enqueued messages are not visible by the Siaam analysis when it traverses the filtered graph. The filter is to the static analysis what opaque objects are to the virtual machine implementation.

$$\begin{aligned}&\textit{siaam-actor-pt-filter} :: \textit{pointstofilter}\\&\textit{siaam-actor-pt-filter}\ P\ s\ (\textit{Inr}\ (o,\ (F,\ C)))\ o' = \neg\ (P \vdash (\textit{aoc}\ P\ o) \preceq^\star \underline{\textit{Mailbox}}\\&\qquad\qquad\qquad\qquad \vee\ P \vdash (\textit{aoc}\ P\ o') \preceq^\star \underline{\textit{Mailbox}}\ )\\&\textit{siaam-actor-pt-filter}\ P\ s\ (\textit{Inl}\ (m,\ V))\ o' = \neg\ (P \vdash (\textit{aoc}\ P\ o') \preceq^\star \underline{\textit{Mailbox}})\end{aligned} \tag{5.17}$$

In equation (5.17), since *aoc* returns the nearest common superclass of the objects coalesced into o, it may happen that o represents some mailboxes among other objects and

hence: $P \vdash \underline{Mailbox} \preceq^\star aoc\ P\ o$. In this situation the filter may accept edges from mailbox objects to messages they contain. Therefore we employ the result of the filter knowing it may include hidden edges. Note that with most pointer analyses, abstract objects are coalesced by allocation site and therefore have precisely the type of the allocated objects.

Ownership pre-conditions. Hereafter are presented the ownership preconditions associated with the methods of the Siaam actors API, they directly relate to *ok-act* defined in the formal model of Siaam. To start the actor pointed-to by V, the objects transitively reachable from V in the filtered points-to graph must be safe:

$$\begin{array}{l} \textit{siaam-actor-pre-conds} :: \textit{requirements} \\ \textit{siaam-actor-pre-conds}\ P\ \langle _ := V.\underline{start}([\,])\rangle_s = \\ \quad \textsf{let}\ T = \textit{type-of}\ P\ s\ V \\ \quad \textsf{in if}\ P \vdash T \leq^\star \textit{Class}\ \underline{Actor} \\ \quad\quad \textsf{then}\ \textit{safeo-preconds}\ (\textit{filtered-pt-trans}\ P\ s\ (\textit{Inl}\ V)\ )\ \textsf{else}\ \emptyset \end{array} \tag{5.18}$$

Similar preconditions apply for the emission of the message pointed-to by V':

$$\begin{array}{l} \textit{siaam-actor-pre-conds}\ P\ \langle _ := V.\underline{put}([V'])\rangle_s = \\ \quad \textsf{let}\ T = \textit{type-of}\ P\ s\ V \\ \quad \textsf{in if}\ P \vdash T \leq^\star \textit{Class}\ \underline{Mailbox} \\ \quad\quad \textsf{then}\ \textit{safeo-preconds}\ (\textit{filtered-pt-trans}\ P\ s\ (\textit{Inl}\ V')\ )\ \textsf{else}\ \emptyset \end{array} \tag{5.19}$$

Ownership side-effects. The ownership side-effects presented in this paragraph are related to *upd-act* in the formal model. Starting an actor invalidates the safety of every objects reachable from the actor's reference:

$$\begin{array}{l} \textit{siaam-actor-effects} :: \textit{effects} \\ \textit{siaam-actor-effects}\ P\ \langle _ := V.\underline{start}([\,])\rangle_s = \\ \quad \textsf{let}\ T = \textit{type-of}\ P\ s\ V \\ \quad \textsf{in if}\ P \vdash T \leq^\star \textit{Class}\ \underline{Actor} \\ \quad\quad \textsf{then}\ \textit{unsafe-effects}\ (\textit{filtered-pt-trans}\ P\ s\ (\textit{Inl}\ V)\ )\ \textsf{else}\ \emptyset \end{array} \tag{5.20}$$

Similarly message emission invalidates the safety of every objects reachable from the message starting reference:

$$\begin{array}{l} \textit{siaam-actor-effects}\ P\ \langle _ := V.\underline{put}([V'])\rangle_s = \\ \quad \textsf{let}\ T = \textit{type-of}\ P\ s\ V \\ \quad \textsf{in if}\ P \vdash T \leq^\star \textit{Class}\ \underline{Mailbox} \\ \quad\quad \textsf{then}\ \textit{unsafe-effects}\ (\textit{filtered-pt-trans}\ P\ s\ (\textit{Inl}\ V')\ )\ \textsf{else}\ \emptyset \end{array} \tag{5.21}$$

The starting reference of a received message is guaranteed to be safe. Our current static analysis is not designed to leverage the fact that all the runtime objects transitively reachable from a received message are safe as well.

$$\begin{array}{l} \textit{siaam-actor-effects}\ P\ \langle _ := V.\underline{get}([\,])\rangle_s = \\ \quad \textsf{let}\ T = \textit{type-of}\ P\ s\ V \\ \quad \textsf{in if}\ P \vdash T \leq^\star \textit{Class}\ \underline{Mailbox} \\ \quad\quad \textsf{then}\ \textit{safe-effects}\ (\ \textit{optional-to-set}\ (\textit{maa}\ P\ (\textit{Inl}\ s)))\ \textsf{else}\ \emptyset \end{array} \tag{5.22}$$

5.5.3 Reflection-based message dispatch APIs

Certain APIs use the Java reflection features to dispatch messages, hence it is necessary to modify the call graph to render explicit every possible dispatches since standard analyses frameworks cannot handle reflection in the general case (although some notable efforts [70, 18] must be cited). For instance the ActorFoundry API dispatches received messages according to the method name specified in every message. Fortunately, methods receiving messages are annotated with `@message`. To analyze ActorFoundry programs, we implemented a pre-processing phase that generates dispatching loop methods for each class of actor. Another difficulty is to cope with reflection-based actor creation. ActorFoundry actors are created from the name of their behavior class. Fortunately in every programs that we analyzed the specified class name is a constant literal, therefore the pre-processing phase is able to replace reflection-based every actor creation with a normal allocation followed by a call to the generated dispatching loop. These two tricks allow to use the standard analyses transparently with the Siaam analysis. Of course the ownership semantics for the `send` and `call` methods must be encoded, but they are equivalent to the Siaam mailboxes' `put` method, although in ActorFoundry several parameters are passed instead of a single one.

5.6 The Siaam Analysis

The analysis computes an under-approximation of the safe reference variables and an over approximation of the unsafe abstract objects at each program-point. These results are then compared with the ownership pre-conditions of each statement to determine whether a given statement can be considered safe or not. A statement can be statically declared safe if all its pre-conditions are met at its entry point.

We present a data-flow analysis that propagates the ownership side-effects of the program's statements. The ownership side-effects are declared by the programming model abstraction presented in section 5.4.1, they express sets of must-alias keys that must become safe and abstract objects that may become unsafe after a given statement.

Data-flow lattice.

Must-alias keys and abstract objects take an ownership state in the totally ordered set $\mathcal{S} = \{\mathit{Safe}, \mathit{Unsafe}\}$ where *Safe* indicates that the reference is safe and *Unsafe* that the reference is unsafe. A third state will be introduced as an extension in section 5.7. We define the join $\vee_{\mathcal{S}}$ and the meet $\wedge_{\mathcal{S}}$ operators on $\mathcal{S}$, with the total order relation $\leqslant_{\mathcal{S}}$ as follows:

$$\begin{array}{rr} \mathit{Unsafe} \leqslant_{\mathcal{S}} \mathit{Unsafe} = \mathit{True} & \mathit{Unsafe} \leqslant_{\mathcal{S}} \mathit{Safe} = \mathit{True} \\ \mathit{Safe} \leqslant_{\mathcal{S}} \mathit{Unsafe} = \mathit{False} & \mathit{Safe} \leqslant_{\mathcal{S}} \mathit{Safe} = \mathit{True} \end{array}$$

The lattice has a greatest element noted $\top_{\mathcal{S}}$ and a least element noted $\bot_{\mathcal{S}}$:

$$\top_{\mathcal{S}} = \bigvee\nolimits_{\mathcal{S}} \mathcal{S} = \mathit{Safe} \quad \bot_{\mathcal{S}} = \bigwedge\nolimits_{\mathcal{S}} \mathcal{S} = \mathit{Unsafe}$$

It can easily be showed that $(\mathcal{S}, \leqslant_{\mathcal{S}})$ is a lattice[27] by showing either commutativity, associativity, idempotence and absorption on $\wedge_{\mathcal{S}}$ and $\vee_{\mathcal{S}}$, or by showing $a \wedge_{\mathcal{S}} b$ and $a \vee_{\mathcal{S}} b$ exist for all $a, b \in \mathcal{S}$. Then we can show that $(\mathcal{S}, \leqslant_{\mathcal{S}})$ is a complete lattice by showing that $\bigvee_{\mathcal{S}} A$ and $\bigwedge_{\mathcal{S}} A$ exist for all $A \subseteq \mathcal{S}$. Since $\mathcal{S}$ comprises only two elements, we can exhibit all the

cases:

$\vee_{\mathcal{S}}, \wedge_{\mathcal{S}}$	*Unsafe*	*Safe*
Unsafe	*Unsafe*, *Unsafe*	*Safe*, *Unsafe*
Safe	*Safe*, *Unsafe*	*Safe*, *Safe*

A	$\bigvee_{\mathcal{S}}$	$\bigwedge_{\mathcal{S}}$
$\emptyset$	$\bot_{\mathcal{S}}$	$\top_{\mathcal{S}}$
$\{Unsafe\}$	*Unsafe*	*Unsafe*
$\{Safe\}$	*Safe*	*Safe*
$\mathcal{S}$	$\top_{\mathcal{S}}$	$\bot_{\mathcal{S}}$

The analysis operates over lists of ownership states. We define the set of lists of a given length l over elements of a given set A by *lists* l A. Therefore any list of ownership states of a certain length l is in the set *lists* l $\mathcal{S}$. We use indifferently the notations $z[n] \equiv z_{[n]}$ to denote the element at position n in a list $z :: {}'a\ list$.

$$\begin{aligned} &lists :: nat \Rightarrow {}'a\ set \Rightarrow ({}'a\ list)\ set \\ &lists\ l\ A = \{as \mid |as| = l \wedge set\ as \subseteq A\} \end{aligned} \tag{5.23}$$

We lift the ordering relation over $\mathcal{S}$ to operate point-wise over lists of ownership states of the same length, defined as the partial ordering noted $\sqsubseteq_{\mathcal{S}}$:

$$\begin{aligned} &(\forall z, z' :: \mathcal{S}\ list) \quad z \sqsubseteq_{\mathcal{S}} z' = ord\ z\ z' \\ &ord\ [\,]\ [\,] = True \\ &ord\ (x \cdot xs)\ (y \cdot ys) = x \leqslant_{\mathcal{S}} y \wedge ord\ xs\ ys \end{aligned} \tag{5.24}$$

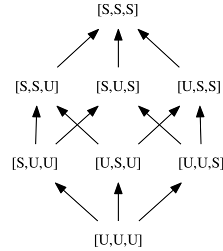


Figure 5.4: Hasse-diagram for the set *lists* 3 $\mathcal{S}$ ordered by $\sqsubseteq_{\mathcal{S}}$. "U" and "S" stand respectively for *Unsafe* and *Safe*

A set *lists* l $\mathcal{S}$ has a greatest element noted $\top^l_{\mathcal{S}}$ and an lowest element $\bot^l_{\mathcal{S}}$. They are the lists of length l filled respectively with *Safe* and *Unsafe*.

$$\top^l_{\mathcal{S}} = replicate\ l\ \top_{\mathcal{S}} \qquad\qquad \bot^l_{\mathcal{S}} = replicate\ l\ \bot_{\mathcal{S}}$$

We provide sets of $\mathcal{S}$ lists of the same length with the definitions of the least upper bound (join operator) $\bigsqcup_{\mathcal{S}}$ and the greatest lower bound (meet operator) $\bigsqcap_{\mathcal{S}}$. They respectively apply the point-wise join and meet operators defined over $\mathcal{S}$. The least upper bound of a couple of lists is written $z \sqcup_{\mathcal{S}} z'$ and produces the list where each element at position n is the greatest of those found at $z[n]$ and $z'[n]$. Symmetrically the greatest lower bound is written $z \sqcap_{\mathcal{S}} z'$ and produces the list where each element is the least of those at $z[n]$ and $z'[n]$.

It can be showed[42] that for all size l, the partially ordered set $(\mathit{lists}\ l\ \mathcal{S}, \sqsubseteq_{\mathcal{S}})$ equipped with the pointwise meet $\sqcap_{\mathcal{S}}$ and join $\sqcup_{\mathcal{S}}$ operators is a complete lattice, as a product of l complete lattices.

Definitions.

Let be N_P the set of call-graph nodes for the program P, E_P the set of every statements, O_P the set of every abstract objects in the points-to graph for P, K_m the set of must-alias keys defined in the call-graph node m. When the program is not mentioned, N stands for N_P, E for E_P, O for O_P. Program points are in the set produced by the cartesian product $\{\mathit{before}, \mathit{after}\} \times E_P$.

For the necessities of the analysis, we define the following sets of lists over ownership states. L_K^m is the set of lists of size $max(K_m) + 1$ over $\mathcal{S}$, with $m \in N_P$ and $max(K_m)$ the highest natural number identifying a must-alias key for the references in the context of that node. Similarly L_O is the set of lists of size $max(O_P) + 1$ over $\mathcal{S}$, with $max(O_P)$ the highest natural number identifying an abstract object in the points-to graph of the program.

$$L_K^m = \mathit{lists}\ (max(K_m) + 1)\ \mathcal{S} \qquad L_O = \mathit{lists}\ (max(O_P) + 1)\ \mathcal{S}$$

Consequently the least element of L_K^m is noted $\bot_K^m$, the greatest element is noted $\top_K^m$, the least and greatest elements of L_O are, respectively, $\bot_O$ and $\top_O$.

$$\begin{aligned} \bot_K^m &\equiv \bot_{\mathcal{S}}^{max(K_m)+1} & \top_K^m &\equiv \top_{\mathcal{S}}^{max(K_m)+1} \\ \bot_O &\equiv \bot_{\mathcal{S}}^{max(O)+1} & \top_O &\equiv \top_{\mathcal{S}}^{max(O)+1} \end{aligned}$$

We introduce the following helpers to build ownership states lists. $\mathit{mk\text{-}unsafe\text{-}list}\ X\ X'$ takes two sets of natural numbers and produces the list of length $max(X) + 1$ where the element at index $n \in [0.. \le max(X)]$ is Unsafe iif $n \in X'$ or Safe otherwise. The $\mathit{mk\text{-}safe\text{-}list}$ function operates the other way around, elements are unsafe by default and safe if they can be found in X'.

$$\begin{aligned} &\mathit{mk\text{-}unsafe\text{-}list} :: \mathit{nat\ set} \Rightarrow \mathit{nat\ set} \Rightarrow \mathcal{S}\ \mathit{list} \\ &\mathit{mk\text{-}unsafe\text{-}list}\ X\ X' = \mathit{map}\ (\lambda\ n\ .\ \mathsf{if}\ n \in X'\ \mathsf{then}\ \mathit{Unsafe}\ \mathsf{else}\ \mathit{Safe})\ [0.. \le max(X)] \\ &\mathit{mk\text{-}safe\text{-}list} :: \mathit{nat\ set} \Rightarrow \mathit{nat\ set} \Rightarrow \mathcal{S}\ \mathit{list} \\ &\mathit{mk\text{-}safe\text{-}list}\ X\ X' = \mathit{map}\ (\lambda\ n\ .\ \mathsf{if}\ n \in X'\ \mathsf{then}\ \mathit{Safe}\ \mathsf{else}\ \mathit{Unsafe})\ [0.. \le max(X)] \end{aligned} \tag{5.25}$$

We note $SO_p \in L_O$ the states list for the abstract objects at the program point p such that $SO_p[o]$ is the ownership state for o in the result computed by the analysis. If q is a program-point defined in the call graph node m, we note $SK_q \in L_K^m$ the list of states of the must-alias keys at q such that $SK_q[\kappa]$ is the ownership state for the key κ in the result of the analysis. The goal of the Siaam analysis is to compute SO_p and SK_p for every program-point p where the *Safe* elements are under-approximated, meaning that an abstract object or a must-alias key will be considered *Safe* if the corresponding abstract objects or variables *must* be safe at runtime.

Organization.

We describe the Siaam static analysis in progressive steps, each step brings more accuracy but also requires more knowledge about the whole program. Each version of the analysis presented in a step computes a sound result, therefore it is possible to mix results from several

versions by choosing the most adapted one for each piece of the analyzed program.

In an introductory step, we present a trivial version of the analysis that provides the most conservative results. Then we develop a fast data-flow analysis which doesn't requires the program's call graph. In the other steps the analysis progressively takes advantage of a call graph and a points-to graph, first in an intra-procedural fashion and finally using the whole program knowledge in an inter-procedural version of the analysis.

5.6.1 Most conservative analysis.

The most conservative approximation consists in considering all the references unsafe whether they are variables or object fields. It is the strategy we have used for the formal model of the Siaam actor-based concurrent programming model. With that pessimistic approximation, the algorithm associates the *Unsafe* state to every abstract objects and every must-alias keys at every program-point.

$$\forall p.\, SO_p = [\mathit{Unsafe}, \ldots, \mathit{Unsafe}] \wedge SK_p = [\mathit{Unsafe}, \ldots, \mathit{Unsafe}] \tag{5.26}$$

Verification of the statements pre-conditions. With a result obtained so rapidly, we can detail how the pre-conditions of each statement are verified at the end of the analysis. The precondition $\mathit{Safe}_O\ o$ required by the statement s is met, according to the static analysis, if $SO_{\bullet s}[o] = \mathit{Safe}$. To verify that a reference variable V is safe before the statement s (precondition $\mathit{Safe}_V\ V$), at least one of the following must be true at $\bullet s$:

1. all the objects in the points-to set of V are safe, or
2. the must-alias key of V is safe.

If the key is unknown, no aliasing information is available for the variable, and one must consider it unsafe. Otherwise the ownership state of the key κ is $SK_{\bullet s}[\kappa]$ and must be equal to *Safe*.
The function *verify-cond* verifies a single pre-condition for a given statement s. Given a statement and its pre-conditions set, *unverified-conds* returns the set of requirements that are not met. Finally the function *verify-conds* verifies that all the pre-conditions are met for the specified statement.

$$\begin{array}{l}
\mathit{verify\text{-}cond} :: \mathit{I\text{-}prog} \Rightarrow \mathit{stmt} \Rightarrow \mathit{requirement} \Rightarrow \mathit{bool} \\
\mathit{verify\text{-}cond}\ P\ s\ (\mathit{Safe}_O\ o) = (SO_{\bullet s}[o] = \mathit{Safe}) \\
\mathit{verify\text{-}cond}\ P\ s\ (\mathit{Safe}_V\ V) = \\
\quad \mathsf{let}\ (m,\ \pi) = s \\
\quad \mathsf{in\ if}\ (\lambda\ o\,.\ \mathit{verify\text{-}cond}\ P\ s\ (\mathit{Safe}_O\ o))`(\mathit{filtered\text{-}pt}\ P\ s\ (\mathit{Inl}\ (m,\ V))) = \{\mathit{True}\}\ \mathsf{then}\ \mathit{True} \\
\qquad \mathsf{else\ (case}\ (\mathit{maa}\ P\ (\mathit{Inr}\ (s,\ V)))\ \mathsf{of}\ \mathit{Unknown} \Rightarrow \mathit{False} \mid \mathit{Some}\ \kappa \Rightarrow SK_{\bullet s}[\kappa] = \mathit{Safe})
\end{array} \tag{5.27}$$

$$\begin{array}{l}
\mathit{unverified\text{-}conds} :: \mathit{I\text{-}prog} \Rightarrow \mathit{stmt} \Rightarrow \mathit{requirement\ set} \Rightarrow \mathit{requirement\ set} \\
\mathit{unverified\text{-}conds}\ P\ s\ \mathit{conds} = \{\mathit{cond} \in \mathit{conds} \mid \neg\ \mathit{verify\text{-}cond}\ P\ s\ \mathit{cond}\}
\end{array}$$

$$\begin{array}{l}
\mathit{verify\text{-}conds} :: \mathit{I\text{-}prog} \Rightarrow \mathit{stmt} \Rightarrow \mathit{requirement\ set} \Rightarrow \mathit{bool} \\
\mathit{verify\text{-}conds}\ P\ s\ \mathit{conds} = (\mathit{unverified\text{-}conds}\ P\ s\ \mathit{conds} = \emptyset)
\end{array}$$

Instantiation of the frameworks. The most conservative implementations of the standard analyses are sufficient to provide the level of approximation required by this introductory step. The points-to analysis can be reduced to its most conservative form where all the reference seems to point to the single abstract object 0 of class type $\underline{Object}$ — the root of the class hierarchy. The must-alias analysis simply associates the unknown key to every references. We don't show the definitions for the most conservative call graph functions and the most conservative live analysis since we don't need these analyses for the moment and we will prefer more accurate implementations of them later. We give life to the corresponding standard framework with the interpretation *standard-framework*$_0$.

$$\begin{array}{ll} pt_0\ P\ s\ x = \{0\} & aoc_0\ P\ 0 = \underline{Object} \\ maa_0\ P\ x\ = Unknown & \end{array}$$

$$\textbf{interpretation}\ \textit{standard-framework}_0 : \textit{standard-framework}\ \textit{targets}_0\ \textit{sources}_0\ pt_0\ aoc_0\ maa_0\ live_0 \tag{5.28}$$

Once a programming model is chosen, and the functions required to describe its ownership-based semantics are implemented, one can give an interpretation of the analysis framework. For the purpose of this example, we create a framework embedding the semantics of the Siaam actor API and the implicit owner-checks, both described in section 5.5. The example interpretation (5.29) of the analysis framework (5.9) uses the most conservative standard analyses (5.28) and the programming model abstraction functions *pm-pre-conds*, *pm-effects* and *pm-pt-filter*. The first couple of function simply aggregate respectively the ownership pre-conditions and side-effects expressed by the two semantics we chose to embed. The last one makes a points-to edge visible if both programming models semantics agree to that.

$$\begin{array}{l} \textit{pm-pre-conds} :: \textit{requirements} \\ \textit{pm-pre-conds}\ P\ s = \textit{implicit-pre-conds}\ P\ s \cup \textit{siaam-actor-pre-conds}\ P\ s \\ \\ \textit{pm-effects} :: \textit{effects} \\ \textit{pm-effects}\ P\ s = \textit{implicit-effects}\ P\ s \cup \textit{siaam-actor-effects}\ P\ s \\ \\ \textit{pm-filtered-pt} :: \textit{pointsto} \\ \textit{pm-pt-filter}\ P\ s\ r\ o = \textit{implicit-pt-filter}\ P\ s\ r\ o \wedge \textit{siaam-actor-pt-filter}\ P\ s\ r\ o \end{array} \tag{5.29}$$

$$\begin{array}{l} \textbf{interpretation}\ \textit{pm-analysis-framework} : \\ \quad \textit{analysis-framework}\ \textit{standard-framework}_0\ \textit{pm-pre-conds}\ \textit{pm-effects}\ \textit{pm-pt-filter} \end{array}$$

5.6.2 Intra-procedural *SafeEff* side-effects propagation

In this section we present a fast intra-procedural flow-sensitive analysis that only takes into account the side-effects of the form *SafeEff* V. It computes safe references without any knowledge of the “outside world”, in particular the worst assumptions must be made about the ownership state of the references at the entry-point of the control flow graph and after the call sites. Later we will progressively increase the knowledge of this outside world thanks to static whole-program analysis techniques.

Informally, the propagation of safe references is based on the observation that if a reference r is safe (owned by the thread of execution) at the program point before s, then either s is a statement passing the object pointed-to by r to another owner and r becomes unsafe after

s, or else r is also safe after s. Furthermore, if a statement s has the side-effect guaranteeing that r becomes safe, then r is owned at the program point following s. A reference r must be safe after every predecessors of a given statement in order to be considered safe before that statement, otherwise r may be unsafe.

We describe the *Safe References Analysis* as an equational *kill/gen* data-flow problem that determine: For each program point, which reference *must* have already been owned by the current thread of execution, and not later passed to another owner, on all paths to the program point. The Safe References Analysis can be seen as an available expression analysis [82] in which we are not interested in the value of expressions but in their ownership status. We often use the term *reference* in place of *must-alias key*, in our analysis they can be considered as the same concept, hence the Safe References Analysis is computing the ownership status of must-alias keys.

Equational data-flow formulation. We formulate the safe references analysis as a set of data-flow equations extracted from the analyzed method $m \in N_P$. A first class of equations relate ownership informations after a statement to ownership informations before the same statement. These equations have the form:

$$SK_{s\bullet} = f_s(SK_{\bullet s})$$

A second class of equations relate ownership informations before a statement to the informations after every predecessors of the statement:

$$SK_{\bullet s} = f'_s(\{SK_{s'\bullet} \mid s' \in preds\ P\ s\})$$

We will define the functions SRA_{entry} :: $I\text{-}prog \Rightarrow stmt \Rightarrow L^m_K$ and SRA_{exit} :: $I\text{-}prog \Rightarrow stmt \Rightarrow L^m_K$ such that:

$$\begin{aligned} SK_{s\bullet} &= SRA_{exit}\ P\ s \\ SK_{\bullet s} &= SRA_{entry}\ P\ s \end{aligned} \tag{5.30}$$

Every statement *kills* a set of references and *generates* a set of safe references. A reference is *killed* by a statement if that statement potentially transfers the ownership of the object pointed-to by the reference. The function $kill_{SRA}$ :: $I\text{-}prog \Rightarrow stmt \Rightarrow L^m_K$ produces the list where each must-alias key corresponds to an element that is either *Unsafe* if the key is killed by the statement or *Safe* otherwise. Assume $z = kill_{SRA}\ P\ s$ then the statement s kills the key κ *iif* $z[\kappa] = Unsafe$ otherwise $z[\kappa] = Safe$.
A reference is *generated* by a statement if it is guaranteed that the reference is owned by the thread of execution after the statement has been successfully evaluated. The list produced by the function gen_{SRA} :: $I\text{-}prog \Rightarrow stmt \Rightarrow L^m_K$ associates *Safe* to every element corresponding to a generated must-alias key, and *Unsafe* for all the other elements. Assume $z' = gen_{SRA}\ P\ s$ then the statement s generates the key κ *iif* $z[\kappa] = Safe$ otherwise $z[\kappa] = Unsafe$.

We now define entirely the data-flow functions for SRA (5.31). SRA_{entry} computes the ownership state of each reference at the program point before a given statement also called the statement's entry. For the method's entry point, we pessimistically assume that all the references points-to unsafe objects because we don't know whether or not the objects reachable from the parameters might have been transferred to another owner before the method is called. Otherwise the references that are safe at the entry of a statement are the

references that are safe at every of the exits from the predecessors of that statement. SRA_{exit} produces the ownership state of each reference at the program point after a given statement (statement's exit). The safe references at a statement's exit are the references that are safe at its entry minus the references it kills, plus the safe references generated by the statement. In more formal words, the "minus" operation is handled by the meet operator that finds the greatest lower bound of the ownership states for the statement's entry and the list produced by $kill_{SRA}$ for the same statement. The "plus" operation is handled by the join operator that find the least upper bound for the result of the minus operation and the list produced by gen_{SRA}.

$$
\begin{aligned}
&SRA_{entry}\ P\ s = \begin{cases} \bot_K^m & \text{if } \langle \texttt{entry} \rangle_s \\ \bigsqcap_{\mathcal{S}} \{SRA_{exit}\ P\ s' \mid s' \in preds\ P\ s\} & \text{otherwise} \end{cases} \\
&SRA_{exit}\ P\ s = (SRA_{entry}\ P\ s \sqcap_{\mathcal{S}} kill_{SRA}\ P\ s) \sqcup_{\mathcal{S}} gen_{SRA}\ P\ s
\end{aligned}
\tag{5.31}
$$

We are interested in the greatest ownership state list satisfying the equation for SRA_{entry}.

The keys killed and generated by each statement are found by querying the programming model abstraction function *side-effects* :: *effects* defined in the *analysis-framework* locale. The gen-list for a given statement s is filled with *Unsafe* and each side-effect of the form *SafeEff* κ is translated into a *Safe* element at the corresponding position in the list produced by gen_{SRA}.

$$
\begin{aligned}
gen_{SRA}\ P\ s = &\ \mathsf{let}\ (m,\ n) = s;\ keys = \{\kappa \mid SafeEff\ \kappa \in side\text{-}effects\ P\ s\} \\
&\ \mathsf{in}\ mk\text{-}safe\text{-}list\ K_m\ keys
\end{aligned}
\tag{5.32}
$$

An killing side-effect of the form *UnsafeEff* o at a statement s indicates that the abstract object o may become unsafe as a side-effect of s, therefore all the references pointing-to o are killed by s. Since there is no direct mapping between must-alias keys and abstract objects, one have to proceed in two steps to find the points-to set corresponding to a must-alias key κ:

1. find a variable V' declared in the analyzed method and a statement s in that method, such that the couple of s' and V' maps to κ according to the must-alias analysis. Such couple should exists because the key was initially found by querying the same must-alias analysis (see 5.4.1). The exception is when the key corresponds to the value of a right-hand-side expression that is never used in another statement. In that case there is no variable ever associated with the must-alias key, however to be sound in prevision of the next improvements of the analysis, the greatest points-to set must be returned.

2. retrieve the points-to set of κ which is computed by the points-to analysis for the variable V' at the statement s'.

$$
\begin{aligned}
&key\text{-}pt :: I\text{-}prog \Rightarrow cg\text{-}node \Rightarrow key\ option \Rightarrow absobj\ set \\
&key\text{-}pt\ P\ m\ Unknown = O \\
&key\text{-}pt\ P\ m\ \lfloor\kappa\rfloor = \mathsf{if}\ (\exists\ s' \in set\ stmts\text{-}of\ P\ m,\ \exists\ V' \in dom\ lcls\text{-}of\ P\ m.\ maa\ P\ (Inr\ (s',\ V')) = \lfloor\kappa\rfloor) \\
&\qquad \mathsf{then}\ filtered\text{-}pt\ P\ s'\ (Inl\ (m,\ V'))\ \mathsf{else}\ O
\end{aligned}
\tag{5.33}
$$

For a statement s with some behavior governed by the programming model abstraction, the set of killed must-alias keys corresponds to the set of keys having in their points-to set at least one of the abstract objects that become unsafe as a side-effect of s:

if *pm-governed P s*:

$$\begin{aligned} kill_{SRA}\ P\ s = \ & \mathsf{let}\ (m,\ n) = s;\ unsafes = \{o \in O \mid UnsafeEff\ o \in side\text{-}effects\ P\ s\}; \\ & \quad killed = \{\ \kappa \in K_m \mid key\text{-}pt\ P\ m\ \lfloor\kappa\rfloor \cap unsafes \neq \emptyset\ \} \\ & \mathsf{in}\ mk\text{-}unsafe\text{-}list\ K_m\ killed \end{aligned} \tag{5.34}$$

Where *pm-governed P s* verifies that the statement s has a behavior directly governed by the programming model abstraction, which might be a non-empty set of ownership pre-conditions or a non-empty set of ownership side-effects.

$$\begin{aligned} & pm\text{-}governed :: I\text{-}prog \Rightarrow stmt \Rightarrow bool \\ & pm\text{-}governed\ P\ s = pre\text{-}conditions\ P\ s \neq \emptyset \vee effects\ P\ s \neq \emptyset \end{aligned} \tag{5.35}$$

We have to introduce artificial kills for invocation statements in order to take into account objects that might become unsafe because of the ownership side-effects of the –transitively– invoked methods. For the time being we don't dispose of any knowledge of the program outside of the analyzed method, we have to assume that any object may be passed to another owner when a method is called, therefore the kill-list for a call-site in the method m is equal to $\bot_K^m$. For all the other statements, that don't have a direct ownership semantic and aren't call-site, the associated kill-list is set to the greatest element so that it has no killing effect in SRA_{exit} (5.31).

$$\begin{aligned} & \text{if}\neg pm\text{-}governed\ P\ s: \\ & kill_{SRA}\ P\ s = \begin{cases} \bot_K^m & \text{if } targets\ P\ s \neq \emptyset \\ \top_K^m & \text{otherwise} \end{cases} \end{aligned} \tag{5.36}$$

5.6.3 Using the call graph information.

In the safe references analysis presented in the previous section, while analyzing a given method, we had to make very pessimistic assumptions about the side effects of the other methods that may be called and the ownership state of the references at the method entry-point because each method was processed independently from the others.

The call graph provides information about the interactions between the methods of the program, or more precisely between the call graph nodes, since the same method can be present multiple times in the graph with different calling contexts. In this section we extend the analysis to propagate safe references along the call graph edges both downward and upward. In the downward direction the parameters that are safe at the exit-point of a method are propagated towards the callers of that same method. The same technique is employed to propagate safe returned values down to the call-site. Symmetrically, in the upward direction, safe parameters at a given call-site are propagated toward every possible callee entry-point.

We base our inter-procedural safe references propagation algorithm on a call-by-value evaluation strategy. In the call-by-value strategy, methods receive their own copy of the parameters' values so that arguments in the callers scope are left unchanged by the side-effects of the call. The receiver method can update its own copy of the arguments, but the changes don't interfere with the arguments at the call-site. Regarding reference parameters, it is worth noticing the must-alias relation between arguments at a given call-site and the corresponding arguments at the entry of the called method. Therefore we observe the following:

1. there is a mapping from must-alias keys of the call-site arguments to keys of the arguments received by the callee,

2. meaning that inside a method, whenever a reference must-alias the entry value of a parameter, it also must-alias the caller's copy of the corresponding parameter.

Lets consider briefly a call-site targeting a method, and that method is only called by that call-site. The interesting corollary from the previous observations is that pairs of corresponding arguments at the call-site and at the callee's entry-point have the same ownership state, determined at the call-site. Indeed, if the must-alias key of the k-th argument is safe at the call-site, then the key of the k-th argument at the callee entry-point is safe as well. Furthermore, at the callee's exit-point, the ownership state of the received arguments determines the ownership state of the caller's arguments when the call returns. If the must-alias key of the k-th argument is safe at the exit of the callee, then the corresponding key is safe after the call-site.

If we now rollback to a more general case where call-sites have several targets and call graph nodes have several callers, we must reevaluate the previous points. The k-th argument at a method entry is safe if the k-th argument is safe at every call-site targeting that method. And if the k-th argument is safe at the exit of every targets of a given call-site, then the corresponding key is safe after the call-site.

Propagation of the *SafeEff* side-effects towards the callers. The propagation of safe references towards the callers is done through the gen-list of the call-sites. For a call-site statement s, $gen_{SRA}\ P\ s$ is made of two components that are joined together. The first component $gen\text{-}params_{SRA}$ produces the ownership states list where some *Safe* elements are set according to the ownership side-effects of the invoked method over the passed parameters. The second component $gen\text{-}ret_{SRA}$ may comprise a *Safe* element if the invoked method returns a safe reference.

$$\begin{array}{l}\text{if } \langle V := V.M(Vs)\rangle_s \wedge \neg\ pm\text{-}governed\ P\ s \wedge targets\ P\ s \neq \emptyset\text{:}\\ gen_{SRA}\ P\ s = gen\text{-}params_{SRA}\ P\ s \sqcup_S gen\text{-}ret_{SRA}\ P\ s\end{array} \tag{5.37}$$

To compute $gen\text{-}params_{SRA}$ we first build the maps from parameter indexes to must-alias keys at call-sites and method entry-points. For an invoke statement $\langle _ := V_0.M([V_1, \ldots, V_n])\rangle_s$, *call-mapping* $P s$ produces the map from the parameter indexes to the associated must-alias key at s. The call-by-value evaluation strategy guarantees the mapping is still valid when the call returns.

$$\begin{array}{l}call\text{-}mapping :: I\text{-}prog \Rightarrow stmt \Rightarrow (pindex \rightharpoonup key)\\ call\text{-}mapping\ P\ \ \langle V' := V_0.M(Vs)\rangle_s = \mathsf{let}\ vars = V_0 \cdot Vs\\ \quad \mathsf{in}\ (\lambda\,\pi\,.\ \mathsf{if}\ \pi < |vars|\ \mathsf{then}\ maa\ P\ (Inr\ (s,\ vars_{[\pi]}))\ \mathsf{else}\ None\ )\end{array} \tag{5.38}$$

The function *entry-mapping* creates a mapping from parameter indexes to must-alias keys at the entry point of the specified call graph node. The key associated with a parameter index π is looked-up with *param-key*, that searches for a parameter identity statement $\langle _ := \texttt{Par}\ \pi\rangle_{s'}$ and returns the right-hand-side must-alias key of s' if it exists.

$$\begin{array}{l}entry\text{-}mapping :: I\text{-}prog \Rightarrow cg\text{-}node \Rightarrow (pindex \rightharpoonup key)\\ entry\text{-}mapping\ P\ m = (\lambda\,\pi\,.\,param\text{-}key\ P\ m\ \pi)\\[1em] param\text{-}key :: I\text{-}prog \Rightarrow cg\text{-}node \Rightarrow pindex \Rightarrow key\ option\\ param\text{-}key\ P\ m\ \pi = \begin{cases} maa\ P\ (Inl\ s') & \text{if } \exists s' \in set\ stmts\text{-}of\ P\ m.\langle V := \texttt{Par}\ \pi\rangle_{s'}\\ Unknown & \text{otherwise}\end{cases}\end{array} \tag{5.39}$$

We now dispose of the key-maps (type *pindex* $\rightharpoonup$ *key*) at both call-sites and method entry-points. Given a key-map and the ownership states list for a program-point, we produce a state-map (type *pindex* $\rightharpoonup \mathcal{S}$) with the function *export* (5.40), that maps indexes of parameters with a known key to ownership states.

$$\begin{aligned}
&\mathit{export} :: (\mathit{pindex} \rightharpoonup \mathit{key}) \Rightarrow \mathcal{S}\ \mathit{list} \Rightarrow (\mathit{pindex} \rightharpoonup \mathcal{S})\\
&\mathit{export}\ \mathit{keymap}\ \mathit{states} = \lambda\,\pi\,.\ \mathsf{case}\ \mathit{keymap}\ \pi\ \mathsf{of}\\
&\qquad\qquad\qquad\lfloor\kappa\rfloor \Rightarrow \lfloor \mathit{states}_{[\kappa]}\rfloor\\
&\qquad\qquad\quad\mid \mathit{Unknown} \Rightarrow \mathit{None}
\end{aligned} \tag{5.40}$$

State-maps are used as an interchange format to propagate ownership states of the arguments between an export-point and an import-point. The function *import* (5.41) translates inter-procedural state-maps into intra-procedural ownership states lists. It takes πs the list of parameter indexes to process, *statemap* the state-map for the export point obtained with *export*, *keymap* the key-map for the import point and *states* the initial states list for the import point. For each parameter index $\pi \in \pi s$, the *states* list is updated if both the state-map and the key-map are defined (not *None*) for π. If they produce respectively the ownership state σ and the must-alias key κ, the *states* list element for κ is updated with the greatest ownership state between σ and the current state.

$$\begin{aligned}
&\mathit{import} :: \mathit{pindex}\ \mathit{list} \Rightarrow (\mathit{pindex} \rightharpoonup \mathcal{S}) \Rightarrow (\mathit{pindex} \rightharpoonup \mathit{key}) \Rightarrow \mathcal{S}\ \mathit{list} \Rightarrow \mathcal{S}\ \mathit{list}\\
&\mathit{import}\ [\,]\ \mathit{statemap}\ \mathit{keymap}\ \mathit{states} = \mathit{states}\\
&\mathit{import}\ \pi \cdot \pi s\ \mathit{statemap}\ \mathit{keymap}\ \mathit{states} =\\
&\quad \mathsf{let}\ \mathit{states}' = \mathit{import}\ \pi s\ \mathit{statemap}\ \mathit{keymap}\ \mathit{states}\\
&\quad \mathsf{in}\ \mathsf{case}\ \mathit{statemap}\ \pi\ \mathsf{of}\\
&\qquad \lfloor\sigma\rfloor \Rightarrow \mathsf{case}\ \mathit{keymap}\ \pi\ \mathsf{of}\\
&\qquad\qquad\quad \lfloor\kappa\rfloor \Rightarrow \mathit{states}'[\kappa := \sigma \vee_{\mathcal{S}} \mathit{states}'_{[\kappa]}]\\
&\qquad\qquad \mid \mathit{Unknown} \Rightarrow \mathit{states}';\\
&\quad\ \mid \mathit{None} \Rightarrow \mathit{states}'
\end{aligned} \tag{5.41}$$

To propagate the ownership status of the parameters from the exit-point of a call graph node down-to one of its call-site, we first export the state-map of the exit-point using the node's entry key-map and then we import it at the call-site using the call's key-map. By applying the export-import operation to each target of a given call-site, we obtain a set of ownership states lists. The $\mathit{gen\text{-}params}_{SRA}$ component of the gen-list for that call-site is the greatest lower bound of the obtained set.

$$\begin{aligned}
&\mathit{gen\text{-}params}_{SRA}\ P\ s = \mathsf{let}\ (m,\ _) = s;\\
&\qquad\qquad \mathsf{in}\ \textstyle\bigsqcap_{\mathcal{S}}\ \{\ \mathit{import}\ [0\ ..\ \leq|\mathit{Vs}|]\\
&\qquad\qquad\qquad\qquad (\mathit{export}\ (\mathit{entry\text{-}mapping}\ P\ m')\ \mathit{SK}_{\bullet(\text{exit-of}\ P\ m')}\ )\\
&\qquad\qquad\qquad\qquad (\mathit{call\text{-}mapping}\ P\ s)\\
&\qquad\qquad\qquad\qquad \bot^{m}_{\mathcal{S}}\\
&\qquad\qquad\quad \mid m' \in \mathit{targets}\ P\ s\ \}
\end{aligned} \tag{5.42}$$

If the right-hand-side expression at the call-site has a defined must-alias key, a lower bound for the ownership status of that key when the call returns corresponds is the infimum of the ownership status for the references returned by the targeted call graph nodes. The function *ret-state* returns the ownership state associated with the variable returned by a specified node, which is $\mathit{SK}_{\bullet s}[\kappa]$ for a return statement $\langle \texttt{ret}\ V\rangle_s$ where V has the must-alias key κ, or conservatively *Unsafe* when V's key is undefined. The ownership states list produced by $\mathit{gen\text{-}ret}_{SRA}\ P\ s$ is based on $\bot^m_K$ where m is the node containing s, meaning no key is generated by default when s returns. If the must-alias analysis associates the key κ to

s, the default ownership states list is updated at position κ with the greatest lower bound of the set of *ret-state* $P\ m'$ for every node m' targeted by s in the call graph.

$$\begin{aligned} \textit{gen-ret}_{SRA}\ P\ s = & \ \mathsf{let}\ (m,\ _) = s; \\ & \ \mathsf{in\ case}\ \textit{maa}\ P\ (\textit{Inl}\ s)\ \mathsf{of} \\ & \quad \lfloor\kappa\rfloor \Rightarrow \bot^m_K[\kappa := \textstyle\bigwedge_{\mathcal{S}} \{\ \textit{ret-state}\ P\ m' \mid m' \in\ \textit{targets}\ P\ s\}] \\ & \ \mid \textit{Unknown} \Rightarrow \bot^m_K \end{aligned} \tag{5.43}$$

$$\begin{aligned} \textit{ret-state}\ P\ m = & \ \mathsf{let}\ \langle \mathtt{ret}\ V\rangle_s = \textit{exit-of}\ P\ m \\ & \ \mathsf{in\ case}\ \textit{maa}\ P\ (\textit{Inr}\ (s,\ V))\ \mathsf{of}\ \lfloor\kappa\rfloor \Rightarrow \textit{SK}_{\bullet s}[\kappa] \mid \textit{Unknown} \Rightarrow \textit{Unsafe} \end{aligned}$$

Propagation of the *SafeEff* side-effects towards the callees. Upward propagation from call-sites to node entry-points employs the export-import mechanism described in the previous paragraph. We define a new version of $\textit{SRA}_{entry}\ P\ s$ for the call graph nodes entry statements that may comprises non-*Unsafe* elements at positions corresponding to parameters' keys.

$$\begin{aligned} & \text{if}\ \ \langle \mathtt{entry}\rangle_s \wedge \neg\ \textit{pm-governed}\ P\ s \wedge \textit{sources}\ P\ (\textit{fst}\ s) \neq \emptyset: \\ & \textit{SRA}_{entry}\ P\ s = \mathsf{let}\ (m,\ _) = s;\ (_,\ C,\ M) = m;\ (_, \textit{Ts},\ _,\ _) = \textit{method}\ P\ C\ M \\ & \qquad \mathsf{in}\ \textstyle\bigsqcap_{\mathcal{S}} \{\ \textit{import}\ [0\ ..\ \leq |\textit{Ts}|] \\ & \qquad\qquad (\textit{export}\ (\textit{call-mapping}\ P\ s')\ \textit{SK}_{\bullet s'}) \\ & \qquad\qquad (\textit{entry-mapping}\ P\ m) \\ & \qquad\qquad \bot^m_K \\ & \qquad \mid s' \in \textit{sources}\ P\ m\ \} \end{aligned} \tag{5.44}$$

Refining call-sites kills. In section 5.6.2 we introduced "artificial" kills to model the undetermined side-effects of the methods transitively reached from call-sites. Thanks to the call graph information, we can now detect whether or not a method invocation may actually transfer some objects' ownership. Let $N_U \subseteq N$ be the set of call graph nodes containing at least one statement with an ownership side-effect of the form *UnsafeEff* o, and *trans-targets* the transitive closure of *targets* that produces the set of nodes that might be reached by successive call graph edges from a given invoke statement. Then we must introduce artificial kills for the call-site s only if the set of nodes reachable from s has a non-empty intersection with N_U.

$$\begin{aligned} & \text{if} \neg \textit{pm-governed}\ P\ s: \\ & \textit{kill}_{SRA}\ P\ s = \begin{cases} \bot^m_K & \text{if}\ \textit{trans-targets}\ P\ s \cap N_U \neq \emptyset \\ \top^m_K & \text{otherwise} \end{cases} \end{aligned} \tag{5.45}$$

5.6.4 Using accurate points-to information.

In the previous versions of the analysis, we assumed the standard framework fixed a very conservative points-to analysis like pt_0 presented in the *standard-framework*$_0$ interpretation (5.28). In the following steps, we consider an interpretation of the standard framework that features an accurate inter-procedural points-to analysis. The points-to analysis used thereafter is able to distinguish between several abstract objects.

The immediate benefit from using accurate points-to information is the improvement of precision in the ownership side-effects. It becomes possible to invalidate the safety of only a subset of all the abstract objects. Similarly the points-to set of a given must-alias key can be computed with more accuracy because most references don't alias every objects in a program. Thus, in the definition of $\mathit{kill}_{SRA}\ P\ s$ (5.34) we may now have $\mathit{unsafes} \neq \emptyset$, $\mathit{key\text{-}pt}\ P\ m\ \lfloor\kappa\rfloor \neq \emptyset$ and $\mathit{unsafes} \cap \mathit{key\text{-}pt}\ P\ m\ \lfloor\kappa\rfloor = \emptyset$, meaning the must-alias key κ does not points to any object invalidated by the statement. If that key is safe before s then it is still safe at $s\bullet$.

Always safe abstract objects. By inspecting the ownership side-effects of every statement in the analyzed program, it is possible to determine the set of abstract objects that may be transferred during the program execution, and conversely the set of abstract objects that are always safe because they are never passed to another owner. Using this set we obtain a greatest lower bound on *SO* for a particular program by setting the maybe-transferred abstract objects to *Unsafe* and the never-transferred objects to *Safe*.

$$\begin{aligned}
&\mathit{maybe\text{-}transferred} :: \mathit{I\text{-}prog} \Rightarrow \mathit{absobj\ set}\\
&\mathit{maybe\text{-}transferred}\ P = \{o \in O \mid \exists s \in E.\, (\mathit{UnsafeEff}\ o) \in \mathit{effects}\ P\ s\}\\
&\mathit{never\text{-}transferred} :: \mathit{I\text{-}prog} \Rightarrow \mathit{absobj\ set}\\
&\mathit{never\text{-}transferred}\ P = O \setminus \mathit{maybe\text{-}transferred}\ P\\
&\forall p.\, SO_p \sqsupseteq_{\mathcal{S}} \mathit{mk\text{-}unsafe\text{-}list}\ O\ (\mathit{maybe\text{-}transferred}\ P)
\end{aligned} \tag{5.46}$$

Consequently, we must revise the artificial kills introduced for the invocation statements in equation (5.36) and refined in equation (5.45). Instead of killing all the must-alias keys, invocation statements that actually reaches an ownership transfer now only kills the keys with a point-to set intersecting the set of maybe-transferred abstract objects.

$$\begin{aligned}
&\text{if}\ \ \neg \mathit{pm\text{-}governed}\ P\ s \wedge \mathit{trans\text{-}targets}\ P\ s \cap N_U \neq \emptyset:\\
&\mathit{kill}_{SRA}\ P\ s = \mathsf{let}\ (m,\ _) = s;\ \mathit{killed} = \{\ \kappa \in K_m \mid \mathit{key\text{-}pt}\ P\ m\ \lfloor\kappa\rfloor \cap \mathit{maybe\text{-}transferred}\ P \neq \emptyset\}\\
&\qquad\qquad\quad \mathsf{in}\ \mathit{mk\text{-}unsafe\text{-}list}\ K_m\ \mathit{killed}
\end{aligned} \tag{5.47}$$

Refinement at call-site. Using the points-to analysis it is possible to refine the set of abstract objects that may be transferred as a side-effect of a given call-site s. Notice that the objects exchanged between a call-site and the invoked method are those transitively reachable from the parameters before and after the call, plus those reachable from the returned value. We say these objects, reachable by both the caller and the callee, *escape* from the callee *through* the considered call-site. At a given call-site s, it is only meaningful to consider the ownership side-effects involving objects that escape through s, all the others are definitely generated by different execution paths.

We now express the upper bound on the set of abstract objects for which ownership side-effects must be taken into account at a call-site $\langle V' := V.M(\mathit{Vs})\rangle_s$. The function *may-escape-through* (5.48) takes an abstract object o and an invoke statement s, and produces a boolean indicating if at runtime an object represented by o may escape the called method through the parameters or the value returned at s. If true, a data-flow fact generated by the callee about the specified abstract object, must be propagated down to the specified call-site. It proceeds as following. If it can find a parameter variable V_i, with a must-alias key pointing-to o', then it looks for o in the transitive closure of o' in the points-to graphs before and

after s. The points-to graph after s is in fact the merge of the points-to graph before every successor of s. If a flow-insensitive pointer analysis is employed, the set of objects reachable from the parameters before and after the call are the same, otherwise they may differ if the called method updates the heap graph in a visible manner for the caller. The second part of the predicate search for o in the transitive closure of the returned reference in the points-to graphs after s, it uses the must-alias key of the invoke expression to find the objects o' immediately pointed-to by the returned value, and searches for o in the transitive closure of every o'. In *may-escape-through* we explore the filtered points-to graph, so that, for instance, objects reachable from mailboxes do not interfere in the process.

$$\textit{may-escape-through} :: \textit{I-prog} \Rightarrow \textit{absobj} \Rightarrow \textit{stmt} \Rightarrow \textit{bool}$$

$$\begin{array}{l}
\text{if } \neg\ \textit{pm-governed}\ P\ s \wedge \langle V' := V.M(Vs)\rangle_s: \\
\textit{may-escape-through}\ P\ o\ s = \mathsf{let}\ (m,_) = s\ \mathsf{in} \\
\quad (\exists\, V_i \in \textit{set}\ (V \cdot Vs), \\
\quad\ \exists o' \in \textit{key-pt}\ P\ m\ (\textit{maa}\ P\ (\textit{Inr}\ (s,\ V_i))\ ), \\
\qquad o \in \textit{filtered-pt-trans}\ P\ s\ (\textit{Inr}\ o') \\
\qquad \vee\ (\exists s' \in \textit{succs}\ P\ s, o \in \textit{filtered-pt-trans}\ P\ s'\ (\textit{Inr}\ o')\ )\ ) \\
\quad \vee\ (\exists o' \in \textit{key-pt}\ P\ m\ (\textit{maa}\ P\ (\textit{Inl}\ s)), \\
\qquad \exists s' \in \textit{succs}\ P\ s,\ o \in \textit{filtered-pt-trans}\ P\ s'\ (\textit{Inr}\ o')\ ) \\
\\
\text{otherwise:} \\
\textit{may-escape-through}\ P\ o\ s = \text{false}
\end{array} \tag{5.48}$$

The *SRA* kill-list for the statement s is updated so that only the objects comprised in *maybe-transferred* that escape through s are killed (nothing escapes from non call-site statements):

$$\begin{array}{l}
\text{if } \neg\ \textit{pm-governed}\ P\ s: \\
\textit{kill}_{\textit{SRA}}\ P\ s = \\
\quad \mathsf{let}\ (m,\ _) = s; \\
\qquad \textit{maybetransferred} = \{o \in \textit{maybe-transferred}\ P \mid \textit{may-escape-through}\ P\ o\ s\}; \\
\qquad \textit{killed} = \{\ \kappa \in K_m \mid \textit{key-pt}\ P\ m\ \lfloor\kappa\rfloor \cap \textit{maybetransferred} \neq \emptyset\} \\
\quad \mathsf{in}\ \textit{mk-unsafe-list}\ K_m\ \textit{killed}
\end{array} \tag{5.49}$$

5.6.5 Inter-procedural Transferred Abstract Objets analysis

In this section we generalize the notion of kills to the inter-procedural level. We built several versions of $\textit{kill}_{\textit{SRA}}$, each one more precise than the previous. They have the common point of considering an over-approximation of the set of abstract objects that may be transferred as a side effect of calling a method. We now fix the definition of $\textit{kill}_{\textit{SRA}}$ and make it depends on the Transferred Abstract Objects analysis (*TAO*). For each call graph node m, $\textit{TAO}_m$ produces the set of abstract objects potentially transferred as a side-effect of calling m.

The kill-list of keys associated with a statement s that has no ownership semantics defined by the programming model can be expressed with the unique equation (5.50), where *maybe-transferred* is the set collecting the objects transferred by targets of s in the call-graph (non call-site statements simply have no targets):

if $\neg$ *pm-governed* $P\ s$:

$$\begin{aligned}
kill_{SRA}\ P\ s = \mathsf{let}\ & maybe\text{-}transferred = \textstyle\bigcup_{\forall m \in targets\ P\ s} TAO_m; \\
& transferred\text{-}through = \{o \in maybe\text{-}transferred \mid may\text{-}escape\text{-}through\ P\ o\ s\}; \\
& killed = \{\ \kappa \in K_m \mid key\text{-}pt\ P\ m\ \lfloor\kappa\rfloor \cap transferred\text{-}through \neq \emptyset\} \\
\mathsf{in}\ & mk\text{-}unsafe\text{-}list\ K_m\ killed
\end{aligned} \tag{5.50}$$

Naive definitions. In the most conservative version of the Siaam analysis we considered that each method potentially transfers the ownership of all the abstract objects, which corresponds to fixing ${}_0TAO_m = O$ for every call graph node m.
Then thanks to the accurate call graph and point-to informations, we identified the set N_U of call graph nodes comprising some killing side-effects, which allows to define *TAO* with more precision. Calling a node m may transfer the ownership of some objects iif m or any other node transitively called by m is in N_U. The following equation allows to reproduce the results of equation (5.47):

$${}_1TAO_m = \begin{cases} maybe\text{-}transferred\ P & \text{if } (m \in N_U) \\ & \vee(\exists s \in stmts\text{-}of\ P\ m, trans\text{-}targets\ P\ s \cap N_U \neq \emptyset) \\ \emptyset & \text{otherwise} \end{cases} \tag{5.51}$$

To improve TAO_m we can collect the side-effects defined for every statement possibly executed when calling m:

$$\begin{aligned}
{}_2TAO_m = \mathsf{let}\ & nodes = \{m\} \bigcup_{\forall s \in stmts\text{-}of\ P\ m} trans\text{-}targets\ P\ s; \\
& stmts = \bigcup_{\forall m' \in nodes} stmts\text{-}of\ P\ m' \\
\mathsf{in} & \bigcup_{\forall s' \in stmts} \{\ o \mid UnsafeEff\ o \in effects\ P\ s'\ \}
\end{aligned} \tag{5.52}$$

The naive approach described in the above equation have a flaw, unsafe abstract objects inexorably accumulate downward from the leafs to the roots of the call graph. The unconditional accumulation of side-effects is a sound but still improvable over-approximation.

Refinement. In this paragraph we give a recursive definition of TAO_m made of the union of two components. The first component is the set of abstract objects directly transferred by statements of m with ownership side-effects governed by the programming model. The second component is the set of abstract objects transferred as side-effects of the call-sites in m. For each statement s of m, and each target m' of s, the contribution to TAO_m is the part of $TAO_{m'}$ that may escape through s:

$$\begin{aligned}
{}_3TAO_m = & \left(\bigcup_{\forall s \in stmts\text{-}of\ P\ m} \{o \mid UnsafeEff\ o \in side\text{-}effects\ P\ s\} \right) \\
& \cup \left(\bigcup_{\substack{\forall s \in stmts\text{-}of\ P\ m \\ \forall m' \in targets\ P\ s}} (TAO_{m'} \cap \{o \in O \mid may\text{-}escape\text{-}through\ P\ o\ s\}) \right)
\end{aligned} \tag{5.53}$$

Notice that it is possible to adapt the precision of *TAO* for each call graph node. To this extend we do not impose a specific version of $TAO_{m'}$ in the second term of ${}_3TAO_m$. For instance, if one of the call graph node is a method for which the body is absent, it is still possible to select $TAO_{m'} = {}_1TAO_{m'}$.

We are interested in the least solution (the smallest set) satisfying ${}_3TAO_m$ for each call graph node m in the call graph. It is clearly a data-flow problem which can be solved by an iterative algorithm where initially ${}_3TAO_m$ is the empty set for every nodes, and each step computes an updated TAO_m until a fixed point is reached.

5.6.6 Intra-procedural *UnsafeEff* side-effects propagation

In the previous sections we showed how to statically compute an under-approximation of the safe references in the analyzed program. We now describe a data-flow analysis, similar to *SRA* in many aspects, that produces the ownership state of each abstract object at each program point. The following Safe Abstract Objects analysis (*SOA*) is divided into an intra-procedural and an inter-procedural part. Its goal is to compute $SO_{\bullet s}$ and $SO_{s\bullet}$, the lists of abstract objects ownership states respectively before and after each statement s.

$$\begin{aligned} SO_{\bullet s} &= SOA_{entry}\ P\ s \\ SO_{s\bullet} &= SOA_{exit}\ P\ s \end{aligned} \tag{5.54}$$

The ownership states list at the entry of the statement s is produced by $SOA_{entry}\ P\ s$. We use the set of maybe-transferred abstract objects to over-approximate the unsafe abstract objects at the entry-point of the analyzed method. Before any other statement s, the data-flow information is equal to the greatest lower bound of the data-flow information after every predecessor of s, precisely: an abstract object is considered safe at the entry of a statement if it is considered safe at the exit of every predecessors of that statement.

The data-flow information $SOA_{exit}\ P\ s$ at the exit of the statement s is again expressed in the form of a kill/gen equation, where $kill_{SOA}\ P\ s$ is the ownership states list associating *Unsafe* to the elements at indexes of abstract objects killed by s and *Safe* to the others, and symmetrically $gen_{SOA}\ P\ s$ is the list associating *Safe* to the abstract objects that become safe after s and *Unsafe* to the others.

$$\begin{aligned} SOA_{entry}\ P\ s &= \begin{cases} \textit{mk-unsafe-list}\ O\ (\textit{maybe-transfered}\ P) & \text{if } \langle \texttt{entry} \rangle_s \\ \bigsqcap_{\mathcal{S}} \{ SO_{s\bullet} \mid s' \in \textit{preds}\ P\ s \} & \text{otherwise} \end{cases} \\ SOA_{exit}\ P\ s &= (SO_{\bullet s} \sqcap_{\mathcal{S}} kill_{SOA}\ P\ s) \sqcup_{\mathcal{S}} gen_{SOA}\ P\ s \end{aligned} \tag{5.55}$$

The abstract objects killed by a statement with some ownership side-effects defined by the programming model abstraction corresponds to the objects becoming unsafe as a side-effect of that statement:

$$\begin{aligned} &\text{if } \textit{pm-governed}\ P\ s\text{:} \\ &kill_{SOA}\ P\ s = \mathsf{let}\ \textit{killed} = \{\ o \in O \mid \textit{UnsafeEff}\ o \in \textit{effects}\ P\ s\} \\ &\qquad\qquad\quad \mathsf{in}\ \textit{mk-unsafe-list}\ O\ \textit{killed} \end{aligned} \tag{5.56}$$

The kill-list for a statement s with no side-effect defined by the programming model is generated using the Transferred Abstract Objects analysis and the set of objects escaping through s:

$$\begin{aligned} &\text{if } \neg\ \textit{pm-governed}\ P\ s\text{:} \\ &kill_{SOA}\ P\ s = \mathsf{let}\ \textit{maybe-transferred} = \textstyle\bigcup_{\forall m \in \textit{targets}\ P\ s} TAO_m; \\ &\qquad\qquad\qquad\quad \textit{transferred-through} = \{o \in \textit{maybe-transferred} \mid \textit{may-escape-through}\ P\ o\ s\} \\ &\qquad\qquad\quad \mathsf{in}\ \textit{mk-unsafe-list}\ K_m\ \textit{transferred-through} \end{aligned} \tag{5.57}$$

Although the ownership side-effects declared by the programming model abstraction does not allow to specify abstract objects that must become safe after a statement, we design

a recovery mechanism able to locally regenerate the safety state of the abstract objects. The recovery is based on a local liveness analysis. As a reminder, a variable (or a formal parameter) is *live* at a given program point if its value may be read before its next update, otherwise the variable is *dead*. We extend the local liveness definition to abstract objects: an abstract object is locally live at a given program point if it is transitively reachable from a live local variable or a live formal parameter. *liveobjs P p* returns the set of live abstract objects at the specified program point:

$$\begin{array}{l} \textit{liveobjs} :: \textit{I-prog} \Rightarrow \textit{program-point} \Rightarrow \textit{absobj set} \\ \textit{liveobjs}\ P\ \bullet s = \bigcup_{vn \in \textit{live}\ P\ \bullet s} \textit{filtered-pt-trans}\ P\ s\ (\text{Inr}\ (\textit{filtered-pt'}\ P\ s\ vn)) \\ \textit{liveobjs}\ P\ s\bullet = \bigcup_{s' \in \textit{succs}\ P\ s} \textit{liveobjs}\ P\ \bullet s' \end{array} \tag{5.58}$$

The auxiliary function *filtered-pt'* returns the abstract objects directly pointed-to by the given variable or the formal parameter *vn*:

$$\begin{array}{l} \textit{filtered-pt'} :: \textit{I-prog} \Rightarrow \textit{stmt} \Rightarrow (\textit{vname} + \textit{pindex}) \Rightarrow \textit{absobj set} \\ \textit{filtered-pt'}\ P\ s\ (\text{Inl}\ V) = \textit{filtered-pt}\ P\ s\ (\text{Inl}\ V) \\ \textit{filtered-pt'}\ P\ (m,\ _)\ (\text{Inr}\ \pi) = \textit{key-pt}\ P\ m\ \ (\textit{param-key}\ P\ m\ \pi) \end{array} \tag{5.59}$$

We state that an unsafe abstract object o may be recovered at statement s if it is locally dead before s and not killed by that statement. Indeed, there is only two possibilities for o to become live again: (i) a new instance of an object represented by o is allocated and becomes reachable from a live variable, (ii) an object represented by o is received by an ownership transfer communication and becomes reachable from a live variable. In both cases, the object is necessarily safe when its reference reappears. The hypothesis where after s a finite number of object fields are read in order to retrieve a reference to o is definitely swept aside since before s there is no local variable reaching o. Notice that our reasoning is valid even if some methods are invoked after s since the only objects they might reach without performing an allocation or a message reception are those live after s.

With a wealth of these observations, the safe abstract objects generated by the statement s are the objects of O which are not live before s:

$$\begin{array}{l} \textit{gen}_{SOA}\ P\ s = \mathsf{let}\ \textit{generated} = O \setminus \textit{liveobjs}\ P\ \bullet s \\ \qquad\qquad\quad \mathsf{in}\ \textit{mk-safe-list}\ O\ \textit{generated} \end{array} \tag{5.60}$$

We insist on the fact that the recovery process described above does only apply to the scope of the analyzed method. An abstract object, recovered locally and still safe at the exit of the analyzed method is not necessarily safe for the callers when the method returns. We can recover unsafe objects locally because they become unreachable (dead), however we must assume that these objects are live for the callers.

5.6.7 Propagation of the safe abstract objects towards the callees.

For each call graph node m, the inter-procedural propagation phase supplies the ownership states lists for the entry-point of m based on the ownership states computed by *SOA* at every call-site targeting m.

We already established that the abstract objects passed by a call-site s to its targets are those transitively reachable from the parameters before s. Therefore it is only relevant to propagate the ownership state for these passed abstract objects towards the targets. We use this property to limit the amount of unsafe abstract objects propagated to the entry of the

call-graph nodes. The contribution of a call-site $\langle V.M(Vs)\rangle_s$ is the list of ownership states $SO^{\uparrow}_s$ such that:

$$SO^{\uparrow}_s[o] = \begin{cases} SO_{\bullet s}[o] & \text{if } \exists V_i \in (V \cdot Vs), o \in \textit{filtered-pt-trans } P\ s\ (\textit{Inl } V_i) \\ \textit{Safe} & \text{otherwise} \end{cases} \tag{5.61}$$

The ownership states list for the entry-point of a call-graph node m is $\top_O$ for the call-graph roots, or the greatest lower bound of the set of ownership states list contributed by the call-sites targeting m:

$$SOA_{entry}\ P\ \langle \texttt{entry} \rangle_{(m,n)} = \begin{cases} \top_O & \text{if } \textit{sources } P\ m = \emptyset \\ \bigsqcap_{\mathcal{S}} \{SO^{\uparrow}_s \mid s \in \textit{sources } P\ m\} & \text{otherwise} \end{cases} \tag{5.62}$$

5.6.8 Escapement analysis for *TAO*

In this section we propose a custom escapement analysis[30, 88] that refines further the Transferred Objects Analysis. Earlier we presented a variation of *TAO* where transferred objects are filtered at each call-site according to the set of objects that may escape through the parameters and the returned value of the call-sites. Although that measure improves the accuracy of *TAO*, it is still limited in the following situation:

— call-site s passes abstract object o in parameters to call graph node m. Therefore we say that o *escapes* from m *through* s.

— the method in m allocates a new object that is also represented by o and transfers the ownership of that object. Therefore o is in TAO_m even if at runtime the transferred object is definitely different from the one received from s in the parameters.

— at call-site s, the fact that o is transferred as a side-effect of calling m is used to kill the variables pointing-to o because o seems to escape through s.

This quick example brings out the necessity for an algorithm that may prevent an abstract object to be inserted in *TAO* when it is possible to determine that the transferred (runtime) object does not *escape* the analyzed method down to its callers.

A flow-sensitive weak escapement judgement. The escapement judgement required to improve *TAO* is defined as follows. The abstract object o transitively reachable from a live variable before statement $(m, n) \in E_P$ is said to escape the call graph node m if the lifetime of at least one runtime object represented by o may exceeds the lifetime of the method represented by m. We weaken the escapement judgement by considering objects received from a message passing communication not to be aliased by callers of the analyzed method. Meaning there must be an evidence that a received object eventually escapes through a formal parameter or the return value in order to report that the object is escaping the method. In other words allocated objects and received objects may be treated the same way. As we shall show next, this weakening does not endanger the soundness of the *SRA* analysis nor the *SOA* analysis, both using *TAO*.

We note $P \vdash \textit{Esc}\ s\ o$ the fact that in program P, $o \in O_P$ is transitively reachable from a live variable before $s \in E_P$ and escapes the call graph node *fst* s, with the hereinbefore specificities.

Refinement of *TAO*. The variation of *TAO* leveraging the escapement judgement is defined by the two following rules.

The rule 4TaoDirect infers that the abstract object o may be transferred as a side-effect of calling m if there is a statement s of m, with a killing side-effect *UnsafeEff* o, for which o weakly escapes.

$$\text{4TaoDirect}\frac{s \in E_P \quad m = \mathit{fst}\ s \quad \mathit{UnsafeEff}\ o \in \mathit{side\text{-}effects}\ P\ s \quad P \vdash \mathit{Esc}\ s\ o}{P \vdash o \in {}_4\mathit{TAO}_m} \tag{5.63}$$

The rule 4TaoInvoke infers that the abstract object o may be transferred as a side-effect of calling m if there is a call-site s of m invoking another method m' and m' transfers the abstract object o. Moreover it is necessary that o escapes m' through the call-site and eventually weakly-escapes from m.

$$\text{4TaoInvoke}\frac{\begin{array}{c} s \in E_P \quad m = \mathit{fst}\ s \quad m' \in \mathit{targets}\ P\ s \quad o \in \mathit{TAO}_{m'} \\ \mathit{may\text{-}escape\text{-}through}\ P\ o\ s \qquad P \vdash \mathit{Esc}\ s\ o \end{array}}{P \vdash o \in {}_4\mathit{TAO}_m} \tag{5.64}$$

Soundness. We shall now explain why *SOA* and *SRA* are sound even with our weak escapement judgement. First it is worth remembering that *TAO* is used to invalidate the safety of abstract objects that may become unsafe and kill must-alias keys that may points-to invalidated (unsafe) abstract objects. Second it should be noticed that if an abstract object is already considered as unsafe before a call-site, then it is unnecessary to track the ownership transfer for that object through the call-site since it won't change the outcome: the abstract object is still unsafe after the call-site. Conversely, an object received in the scope of a given method must already be considered unsafe at the call-sites targeting that method where that object is live before the call. Consequently it is sound, with respect to *SOA* and *SRA*, not to include in *TAO* the received abstract objects when there is no evidence of escaping in the analyzed method because in this case the object is either not aliased or already considered unsafe by the callers.

Weak escapement semantics. In this paragraph we present the inference rules governing the flow-sensitive escapement judgement designed specifically for *TAO*.

The rule EscPar infers that every abstract object o reachable from the formal parameter n at statement s escapes the call graph node of s.

$$\text{EscPar}\frac{P \vdash \langle _ := \mathtt{Par}\ n \rangle_s \quad o \in \mathit{filtered\text{-}pt\text{-}trans}\ P\ s\ (\mathit{Inr}\ (\mathit{filtered\text{-}pt'}\ P\ s\ (\mathit{Inr}\ n)\ ))}{P \vdash \mathit{Esc}\ s\ o}$$

The rule EscRet infers that every abstract object o reachable from the method return value V at statement s escapes the call graph node of s.

$$\text{EscRet}\frac{P \vdash \langle \mathtt{ret}\ V \rangle_s \quad o \in \mathit{filtered\text{-}pt\text{-}trans}\ P\ s\ (\mathit{Inl}\ V)}{P \vdash \mathit{Esc}\ s\ o}$$

The rule EscFwd propagates escapement facts forward in the control-flow graph based on the abstract objects liveness defined in Section 5.6.6.

In order to infer that o reachable from a live-in variable at statement s does escape, the rule requires that o is already escaping in a predecessor s' of s and o must be live-out s' to ensure that it may represents the the same runtime object as in s.

$$\textsc{EscFwd}\ \frac{o \in \textit{liveobjs}\ P\ \bullet s \qquad s' \in \textit{preds}\ P\ s \qquad P \vdash \textit{Esc}\ s'\ o' \qquad o \in \textit{liveobjs}\ P\ s'\bullet}{P \vdash \textit{Esc}\ s\ o}$$

The rule EscEdg propagates escapement to abstract objects reachable from already escaping objects through new edges of the points-to graph when the pointer-analysis is flow-sensitive (because the visibility of a particular edge may be deferred up to the successors of the field assignment statement generating that edge). Note that EscEdg is not required when a flow-insensitive points-to analysis is employed because every object eventually reachable from a parameter or the returned value of the method is already discovered by EscPar and EscRet.

$$\textsc{EscEdg}\ \frac{\{o, o'\} \subseteq \textit{liveobjs}\ P\ \bullet s \qquad s' \in \textit{succs}\ s \qquad P \vdash \textit{Esc}\ s\ o' \qquad o \in \textit{filtered-pt-trans}\ P\ s'\ (\textit{Inr}\ o')}{P \vdash \textit{Esc}\ s\ o}$$

The rules EscBwd propagates escapement facts backward in the control-flow graph. In order to infer that o reachable from a live variable at s does escape, the rule requires that the object does escape from a successor s' of s. Moreover o must be live after s and before s' to make sure that it may represents the same runtime object.

$$\textsc{EscBwd}\ \frac{o \in \textit{liveobjs}\ P\ s\bullet \qquad o \in \textit{liveobjs}\ P\ \bullet s' \qquad P \vdash \textit{Esc}\ s'\ o'}{P \vdash \textit{Esc}\ s\ o}$$

Examples. In Figure 5.6 the statements of the analyzed call graph node are presented on the left, the liveness of various abstract objects is presented in the central part, where each column represents an object. In the first column we see that o_α is live from before the method entry-point to s_2, dead after s_2 until s_4 allocates an object represented by o_α that becomes live right after s_4 through variable q. Since the method returns q, the object pointed-to by q is live up to s_9 included. We see that parameter of index 0 points-to o_α, and initially its field F points-to o_β. In this example the pointer analysis is flow sensitive, therefore field assignments are only visible after the assignment statement, for instance s_5 assigns $y.G$ with z, although y is dead after the assignment, the field G of o_β eventually points-to o_3 after s_5. The similar situation happens after s_8 where both o_3 and o_6 are dead but the new edge through the field H of o_3 is visible. Object o_6 is filled with black to indicate that it is unsafe after s_7 and as long as it is live.

The right side of the figure shows the live abstract objects escaping for each statement. Let see how to infer that o_6 reachable from s_7 escapes the analyzed method. The equation in Figure 5.5 summarize the series of rules and goals used to infer $P \vdash \textit{Esc}\ s_7\ o_6$. Starting from the parameter identity statement s_1, EscPar infers $P \vdash \textit{Esc}\ s_1\ o_\beta$. Then EscFwd is applied to progress up to s_5. One application of EscEdg uses the edge G to infer $P \vdash \textit{Esc}\ s_5\ o_3$. The progression continues forward until EscEdg is applied at s_8 to infer $P \vdash \textit{Esc}\ s_8\ o_6$. The last step goes backward from there and infer the final goal.

Knowing $P \vdash \textit{Esc}\ s_7\ o_6$, we can apply 4TaoDirect to show $P \vdash o_6 \in {}_4\textit{TAO}_m$.

In order to infer that the object o_α pointed-to by q escapes, we apply EscRet at s_9 showing that o_α escapes, and then we apply EscBwd up to s_5. Notice that starting from $P \vdash \textit{Esc}\ s_1\ o_\alpha$ and forward we cannot infer that the object pointed-to by q escapes because o_α in the range s_0-s_2 does not represent the same runtime object as o_α in the range s_5-s_9.

The example in Figure 5.7 is the same sequence of instructions but the objects liveness is computed with a flow-insensitive points-to analysis. The accuracy is particularly degraded.

$$P \vdash \langle x := \texttt{Par}\ 0\rangle_{s_1}(\textsc{EscPar}) \longrightarrow P \vdash \mathit{Esc}\ s_1\ o_\beta(\textsc{EscFwd}) \longrightarrow P \vdash \mathit{Esc}\ s_2\ o_\beta$$
$$(\textsc{EscFwd}) \longrightarrow P \vdash \mathit{Esc}\ s_3\ o_\beta(\textsc{EscFwd}) \longrightarrow P \vdash \mathit{Esc}\ s_4\ o_\beta(\textsc{EscFwd}) \longrightarrow P \vdash \mathit{Esc}\ s_5\ o_\beta$$
$$(\textsc{EscEdg}) \longrightarrow P \vdash \mathit{Esc}\ s_5\ o_3(\textsc{EscFwd}) \longrightarrow P \vdash \mathit{Esc}\ s_6\ o_3(\textsc{EscFwd}) \longrightarrow P \vdash \mathit{Esc}\ s_7\ o_3$$
$$(\textsc{EscFwd}) \longrightarrow P \vdash \mathit{Esc}\ s_8\ o_3(\textsc{EscEdg}) \longrightarrow P \vdash \mathit{Esc}\ s_8\ o_6(\textsc{EscBwd}) \longrightarrow P \vdash \mathit{Esc}\ s_7\ o_6$$

Figure 5.5: Inferring $P \vdash \mathit{Esc}\ s_7\ o_6$ in the example Figure 5.6

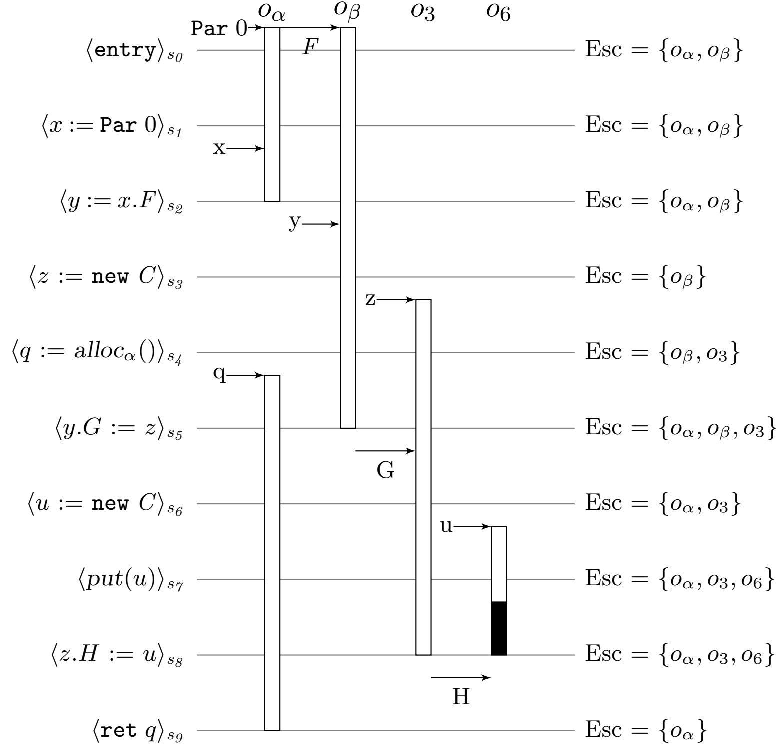


Figure 5.6: Example of escapement analysis, using an ideal flow-sensitive points-to analysis.

First the points-to graph is the same at every program point therefore many edges are visible at statements where they would not exist at runtime. Since y is live from $s_3\bullet$ to $\bullet s_5$, and q is live from $s_4\bullet$, the abstract object o_β is continuously live and so are o_3 and o_6, maintained transitively reachable by the formal parameter 0, x, y and q consecutively.

It is worth noticing that inferring $P \vdash \mathit{Esc}\ s_7\ o_6$ can be achieve either starting from the formal parameter 0, or starting from the returned reference. In the first case we have immediately: $P \vdash \langle x := \texttt{Par}\ 0\rangle_{s_1}(\textsc{EscPar}) \longrightarrow P \vdash \mathit{Esc}\ s_1\ o_6$, and then EscFwd is applied until s_7 is reached. In the second case: $P \vdash \langle \texttt{ret}\ q\rangle_{s_9}(\textsc{EscRet}) \longrightarrow P \vdash \mathit{Esc}\ s_9\ o_6$, and then EscBwd is applied until s_7 is reached.

Thrown exceptions. Until know we have brilliantly dodged the issue of exceptions. Objects reachable from a thrown exception may escape down to the call graph root unless the exception is caught in the way. The good point is that an exception and the objects reachable from that exception are only accessible from the moment the exception is caught. Therefore as long as the thrown exception "flies" down the call graph, overflew methods do not have to

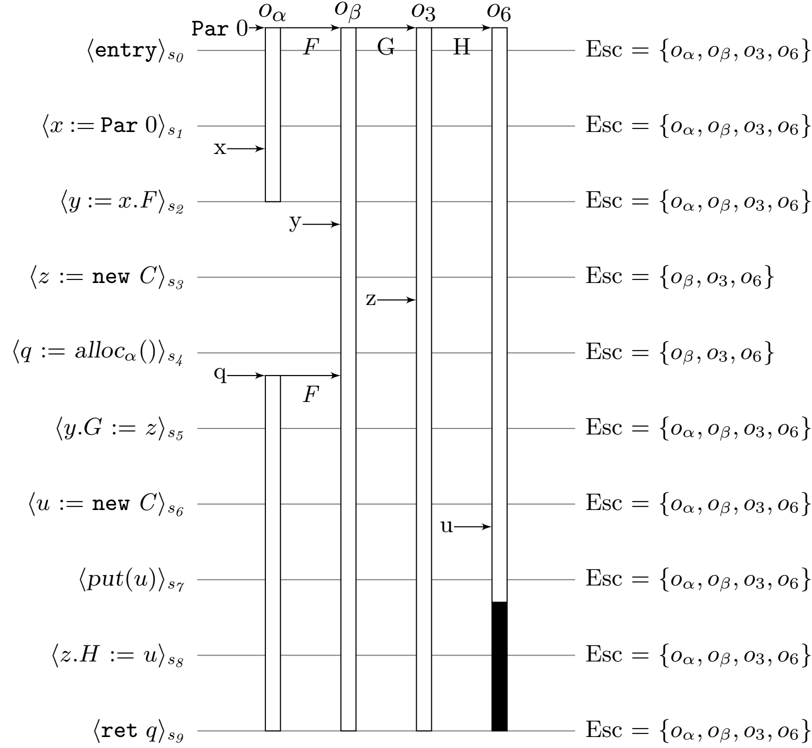


Figure 5.7: Same as example 5.6, using a flow-insensitive points-to analysis.

worry about the ownership state of the objects reachable from the exception.

In order to take exceptions into account, we shall improve the escapement analysis and the *TAO* so that ownership state of exceptions propagates from throwing statements to the matching `CaughtXcpt` statements. An inter-procedural exception analysis[29] would be helpful to establish the matchings.

5.7 Extension to frozen objects.

In the chapter devoted to the virtual machine implementation we explained how frozen arrays brings support for type enumerations and static array variables. We also described how arbitrary frozen scalar objects may limit the opportunities of optimizing-out owner-check barriers. In this section we give a quick overview of the extension of the Siaam analysis to support frozen arrays and frozen scalars.

5.7.1 The *Read* ownership state

We extend the lattice $\mathcal{S}$ with a new ownership state indicating that a reference is definitely safe for a field read operation but a field write of that reference may raise an ownership mismatch exception.

$$\mathcal{S} = \{Safe, Read, Unsafe\}$$

The ordering on $\mathcal{S}$ places *Read* between *Safe* and *Unsafe*:

$$Unsafe \leqslant_{\mathcal{S}} Read \qquad Read \leqslant_{\mathcal{S}} Safe$$

Accordingly, we define a new kind of ownership pre-condition $Read_V\ V$ denoting that the variable V is expected to be readable or safe. And a new kind of ownership side-effect $ReadEff\ \kappa$ denoting that the must-alias key κ becomes at least readable:

$$\begin{array}{ll} \textbf{datatype} & requirement = Safe_V\ vname \mid Read_V\ vname \mid Safe_O\ absobj \\ \textbf{datatype} & effect = SafeEff\ key \mid ReadEff\ key \mid UnsafeEff\ absobj \end{array} \tag{5.65}$$

Then we extend the verification function defined in equation (5.27). The readability condition on variable V is verified at statement s either if V is safe or if the must-alias key associated with V at s has the $Read$ ownership status:

$$\begin{array}{l} verify\text{-}cond\ P\ s\ (Read_V\ V) = \\ \quad \mathsf{if}\ verify\text{-}cond\ P\ s\ (Safe_V\ V)\ \mathsf{then}\ True \\ \quad \mathsf{else\ let}\ (m,\pi) = s \\ \qquad \mathsf{in\ (case}\ (maa\ P\ (Inr\ (s,\ V)))\ \mathsf{of}\ Unknown \Rightarrow False \mid Some\ \kappa \Rightarrow SK_{\bullet s}[\kappa] = Read) \end{array} \tag{5.66}$$

The definition of $genSRA$ is revised to take the new kind of side-effect into account. *mk-read-list* is the function similar to *mk-safe-list* that produces a list of ownership status where the specified indexes are set to *Read* and the remaining to *Unsafe*:

$$\begin{array}{l} gen_{SRA}\ P\ s = \mathsf{let}\ (m,\ n) = s;\ safekeys = \{\kappa \mid SafeEff\ \kappa \in side\text{-}effects\ P\ s\}; \\ \qquad\qquad\qquad readkeys = \{\kappa \mid ReadEff\ \kappa \in side\text{-}effects\ P\ s\} \\ \qquad\qquad \mathsf{in}\ (mk\text{-}safe\text{-}list\ K_m\ safekeys) \sqcup_S (mk\text{-}read\text{-}list\ K_m\ readkeys) \end{array} \tag{5.67}$$

In Section 5.6.6 we introduced a mechanism to recover the safety of the abstract objects under certain liveness conditions. We have to modify this mechanism so that abstract objects that might be frozen are recovered up to the *Read* state instead of the *Safe* state. There is no absolute solution to declare which objects might become frozen at one point in the program history, it is up to the programming model abstraction to provide that information. For instance an abstraction supporting static variables would declare the abstract objects transitively reachable from these variables as potentially frozen since it is a necessary condition to enforce the isolation property. A programming model featuring a "freeze" method to freeze arbitrary objects would proceed similarly with the frozen references.

5.7.2 A programming model with frozen arrays

To support array elements accesses we first define a special field name $\underline{ARR}$. We interpret the expression $V.\underline{ARR}\{\}$ as an access of an element of the array V at an unspecified index. We now define a new programming model abstraction that supports frozen arrays. This programming model may be used in conjunction with the default implicit owner-checks programming model (Section 5.5.1) and the siaam actor programming model (Section 5.5.2), as demonstrated in (5.29). We do not explicit how the arrays may become frozen, in this abstraction we simply assume that any array may be in a frozen state.

Ownership pre-conditions. Reading an element of the array pointed-to by V requires V to be at least readable. The condition to write an array element is the same as writing a scalar's field, the reference must be safe:

$$\begin{array}{l} frozen\text{-}arrays\text{-}pre\text{-}conds :: requirements \\ frozen\text{-}arrays\text{-}pre\text{-}conds\ P\ \langle _ := V.\underline{ARR}\{\}\rangle_s = \{Read_V\ V\} \\ frozen\text{-}arrays\text{-}pre\text{-}conds\ P\ \langle V.\underline{ARR}\{\} := _\rangle_s = \{Safe_V\ V\} \end{array} \tag{5.68}$$

Ownership side-effects. The ownership side-effects of reading from and writing to an array element are set accordingly to the preconditions expressed previously. A successful read guarantees that the array is readable and a successful write asserts the entire safety of the accessed array:

frozen-arrays-side-effects :: *effects*
$$\textit{frozen-arrays-side-effects}\ P\ \langle _ := V.\underline{ARR}\{\} \rangle_s = \textit{ReadEff}\text{`}(\ \textit{optional-to-set}\ (\textit{maa}\ P\ (\textit{Inr}\ (s,\ V)))) \quad (5.69)$$
$$\textit{frozen-arrays-side-effects}\ P\ \langle V.\underline{ARR}\{\} := _ \rangle_s = \textit{SafeEff}\text{`}(\ \textit{optional-to-set}\ (\textit{maa}\ P\ (\textit{Inr}\ (s,\ V))))$$

5.7.3 A programming model with frozen scalars

This exercise is similar in every points to the previous one. Assuming that any object may be frozen, reading an object's field requires that object to be at least in the *Read* state and guarantees this state if the access is successful.

5.8 Usages of the Siaam analysis results

5.8.1 Owner-check elimination

The analysis identifies *safe* object accesses, for such statements it is statically guaranteed that the accessed object belongs to the current thread owner-ID. A runtime system, like our virtual machine implementation, can use the fact that an access is safe in order to skip the owner-checking operation for the safe accesses. A single owner-checking operation introduces only a very small overhead at runtime, however object accesses are extremely frequent in object-oriented programs. Therefore the owner-checks elimination can provide a sensible performance gain.

The function *verify-conds* defined in equation (5.27) returns true if every precondition associated with a given statement has been statically verified by the analysis. A runtime system handles bytecode instructions and is usually not able to benefit from context-sensitive informations, thus it is necessary to project the contextual statements safety information to bytecodes safety information. We benefits from the direct mapping between the lexical statements in the method body representation (the statement list in *I-mb*) and the underlying bytecode representation. Hence the lexical statement at position n in the body of method $C.M$ corresponds to the bytecode at the same position in the bytecodes array of the method once compiled. A call graph node statement $((c, C, M), n)$ is projected to the bytecode at position n in the bytecode array of $C.M$, the calling context information c is lost in the process. The safety information for a given bytecode is the merge of the safety information of every statement projected to that bytecode. A bytecode is safe if all the statements projected into it are safe:

$$\textit{safe-bc}\ P\ C\ M\ n = \forall c, \textit{verify-conds}\ P\ ((c, C, M), n) \quad (5.70)$$

The virtual machine can use *safe-bc* to decide whether a bytecode instruction performing a field access requires to owner-check the accessed object or if the owner-check may be omited. In the formal definition of the Siaam virtual machine, Section 3.6, we included the two restricted instructions `GetfieldOwned` and `PutfieldOwned` performing field access and skipping the owner-check operation. When a bytecode instruction is statically safe, the virtual machine can replace the traditional `Getfield` (or `Putfield`) instruction with the appropriate alternative omitting the dynamic owner verification.

5.8.2 Programming assistant

The programming assistant helps the application developers understand why a given program statement is potentially unsafe and may throw an ownership exception at runtime. Developers can use this functionality to review their programs and add appropriate try/catch blocks if they estimate that an alert is legitimate. The static analysis guarantees that there will be no false negative, but in order to limit the amount of false positives it is necessary to use a combination of the most accurate standard analyses.

Let be s an unsafe statement with a non-empty set U of unverified ownership preconditions produced by *unverified-conds* P s (*pre-conditions* P s). The programming assistant tracks the program P backward starting from s to find every program points that may explain why a given pre-condition in U is not met at s. For each unsatisfied precondition, it can exhibit the various shortest execution paths that may result in an exception being raised at s.

An ownership requirement (5.10) is either that a variable is safe ($Safe_V\ V$) or that an abstract object is safe ($Safe_O\ o$). When a requirement is not satisfied before the statement s, it raises one or several questions of the form "why the abstract object o is unsafe before s?". The precondition $Safe_O\ o$ generates a single question about o. The precondition $Safe_V\ V$ generates one question for each abstract object pointed-to by V. The goal of the assistant is find answers to these questions. It traverses the control-flow backward, looks for immediate answers at each reached statement and propagates the questions further if necessary, until no new question can be generated. In fact it performs the reverse propagation of the *SOA* analysis, another way of designing the assistant would be to modify the *SOA* so it records at each program point what are the original statements responsible for the invalidation of a given abstract object.

Q&A game. Informally, in order to answer the question "why o is unsafe before s", it is necessary to generate the question "why o is unsafe after s'?" for every predecessor s' of s where o is actually unsafe after s'. We now have a second form of question. A first explanation is that o is already unsafe before s', in which case it generates the question "why o is unsafe before s'?", of the first form. Another explanation for the second form of question is that the statement actually invalidates the abstract object. If the invalidation is an immediate side-effect governed by the programming model, then we have found one explanation for the series of questions that engendered this response. If the invalidation is the side-effect of a call-site, then a third form of question must be propagated to each target m of that call-site that potentially transfers the ownership of the tracked abstract object: "what statement of m transfers the escaping abstract object o?". Finally, when s is the entry statement of the call graph node m, the question "why o is unsafe before s?" generates the equivalent question for every potential call-site targeting m and passing o through the formal parameters.

Q&A graph. We now show how to construct a graph where nodes are questions and explanations, and directed edges goes from a question to an engendered question and from a question to its explanation. The assistant generates the initial questions from unsafe statements, and once the graph is fully expanded, finding answers to a given question consists in collecting all the explanation nodes in its transitive closure.

We write $(o? \bullet s)$ the question "why o is unsafe before s?". Similarily $(o?s\bullet)$ encodes "why o is unsafe after s". "what statement of m transfers the escaping abstract object o" is

written $(o?m)$. Finally $(o!s)$ explains that o is transfered as an immediate side-effect of s. The transition $P \vdash G \rightarrow_? G'$ means that the graph G can be extended to the graph G'. The function *mkedge* $G\ q\ q'$ produces the new graph corresponding to G with an edge from q to q'. The dummy root node (0) is used as an origin for the initial questions. The empty graph is noted $\emptyset$.

Initial questions. As explained above, the programming assistant generates initial questions from the unverified preconditions. For an abstract object o, it generates a single question to know why o is unsafe before s:

$$\textsc{WhyO}\ \frac{(\mathit{Safe}_O\ o) \in \mathit{unverified\text{-}conds}\ P\ s\ (\mathit{pre\text{-}conditions}\ P\ s)}{P \vdash G \rightarrow_? \mathit{mkedge}\ G\ (0)\ (o?\bullet s)}$$

The rule WhyV must be applied for every object o in the points-to set of variable V at statement s in order to generate all the appropriate questions:

$$\textsc{WhyV}\ \frac{\begin{array}{c}(\mathit{Safe}_V\ V) \in \mathit{unverified\text{-}conds}\ P\ s\ (\mathit{pre\text{-}conditions}\ P\ s)\\ o \in \mathit{filtered\text{-}pt}\ P\ s\ (\mathit{Inl}\ V)\end{array}}{P \vdash G \rightarrow_? \mathit{mkedge}\ G\ (0)\ (o?\bullet s)}\ \forall o \in O_P$$

Graph expansion. The following rules add edges to propagates the initial questions and provide answers. The graph is fully expanded when no more rule can be applied on the current graph G to obtain a larger graph $G' \supset G$.

The rule WhyPred propagates a question from the entry of a statement to the exit of its predecessors where the specified abstract object is also unsafe.

$$\textsc{WhyPred}\ \frac{q \in G \quad q = (o?\bullet s) \quad s' \in \mathit{preds}\ P\ s \quad \mathit{SO}_{s'\bullet}[o] = \mathit{Unsafe}}{P \vdash G \rightarrow_? \mathit{mkedge}\ G\ q\ (o?s'\bullet)}\ \forall s' \in E_P$$

The rule WhyBfr propagates a question from the exit of a statement to its entry where the specified abstract object is also unsafe.

$$\textsc{WhyBfr}\ \frac{q \in G \quad q = (o?s\bullet) \quad \mathit{SO}_{\bullet s}[o] = \mathit{Unsafe}}{P \vdash G \rightarrow_? \mathit{mkedge}\ G\ q\ (o?\bullet s)}$$

The rule Expl explains that the specified object is unsafe after s because the statement has the immediate side-effect of transferring the object's ownership.

$$\textsc{Expl}\ \frac{q \in G \quad q = (o?s\bullet) \quad \mathit{UnsafeEff}\ o \in \mathit{side\text{-}effects}\ P\ s}{P \vdash G \rightarrow_? \mathit{mkedge}\ G\ q\ (o!s)}$$

The rule WhySrc propagates a question from the entry of a call graph node to the call-sites targeting that node and passing the abstract object through the formal parameters. This rule reflects the fact that o may have already been transferred before the call graph node m is called from s.

$$\textsc{WhySrc}\ \frac{\begin{array}{c}q \in G \qquad q = (o?\bullet (m,n)) \qquad P \vdash \langle \texttt{entry} \rangle_{(m,n)}\\ s \in \mathit{sources}\ P\ m \quad o \in \mathit{may\text{-}escape\text{-}through\text{-}upward}\ P\ s\end{array}}{P \vdash G \rightarrow_? \mathit{mkedge}\ G\ q\ (o?\bullet s)}\ \forall s \in E_P$$

The rule WhyTrg propagates a question from the exit of a call-site s to the invoke targets of s. The invoked method must transfer the object and the object must escape through the

call-site. This rule reflects the fact that a method transitively invoked by s transfers the ownership of o.

$$\textsc{WhyTrg}\frac{q \in G \quad q = (o?s\bullet) \quad m \in \mathit{targets}\ P\ s \quad o \in \mathit{TAO}_m \quad \mathit{may\text{-}escape\text{-}through}\ P\ o\ s}{P \vdash G \rightarrow_? \mathit{mkedge}\ G\ q\ (o?m)} \forall m \in N_P$$

The rule WhyTransTrg propagates a question towards the leafs of the call graph with respect to the escapement judgement presented in Section 5.6.8. A question is propagated from a node to one of its successor if the successor may transfer the abstract object and that abstract object eventually escapes.

$$\textsc{WhyTransTrg}\frac{q \in G \quad q = (o?m) \quad s \in \mathit{stmts\text{-}of}\ P\ m \quad m' \in \mathit{targets}\ P\ s \quad o \in \mathit{TAO}_{m'} \quad \mathit{may\text{-}escape\text{-}through}\ P\ o\ s \quad P \vdash \mathit{Esc}\ s\ o}{P \vdash G \rightarrow_? \mathit{mkedge}\ G\ q\ (o?m')} \forall m \in N_P, \forall s \in E_P$$

The last rule TransExpl find an explanation for a question of the form $(o?m)$. There must be a statement of m with an immediate side-effect invalidating o and that object must escape m.

$$\textsc{TransExpl}\frac{q \in G \quad q = (o?m) \quad s \in \mathit{stmts\text{-}of}\ P\ m \quad \mathit{UnsafeEff}\ o \in \mathit{side\text{-}effects}\ P\ s \quad P \vdash \mathit{Esc}\ s\ o}{P \vdash G \rightarrow_? \mathit{mkedge}\ G\ q\ (o!s)} \forall s \in E_P$$

Interpretation. As seen earlier, finding answers for a particular question consists in collecting every answer $(o!s)$ in the transitive closure of the question. The reverse Q&A graph exhibits execution paths from potential ownership transfer of a given object to potential misuses of that unsafe object. This information is currently reported textually, but it would takes on its full meaning once exposed in a graphical Integrated Development Environment such as Eclipse[77].

5.9 Implementation

The analyses described in this chapter have been implemented in two contexts. First as an optimization phase of the just-in-time compiler of the JikesRVM. And second as a whole-program analysis using the Soot framework.

Just-in-time owner-checks elimination in JikesRVM. The intra-procedural Safe Reference Analysis has been included in the JikesRVM optimizing compiler. Despite its relative simplicity and the very conservative assumptions the analysis have to make, it efficiently eliminates about 55% of the owner-check barriers introduced by the application bytecode and the standard library for the representative benchmarks we have tested (see Chapter 6). The implementation comprises the extension supporting frozen arrays which are required to safely handle type enumerations and static arrays.

We modified the JikesRVM classloader in order to read non-standard safety bytecode annotations generated by the external tool detailed in the next section. This way we were able to compute the whole-program static analysis offline and embed the results into the class files loaded by the virtual machine.

Whole-program static analysis in Soot. The *SRA* and the *SOA* analyses have been entirely implemented in their inter-procedural version as an offline tool written in Java – by opposition to the just-in-time optimization. The implementation contains a restricted version of *TAO* analysis, which corresponds to the third refinement ($_3TAO$, equation (5.53)) given in this document.

The tool is designed to be as generic as possible regarding the program representation and the standard analyses it uses. Programming model abstractions are extensible and generic as well. We followed the same organization as the formal description of the analysis in the current chapter, the standard analyses are provided by a third party framework and integrated through a thin interfacing module.

Our prototype is interfaced with the Soot analysis framework[64], which provides the program representation, the call-graph, the inter-procedural pointer analysis, the must-alias analysis and the liveness analysis exactly as described by the *standard-framework* in Section 5.3. The Siaam analyses accounts for about 3600 lines of code (not counting statement-less lines), and the interface with the Soot framework accounts for roughly 550 LOC.

We had to slightly modify Soot for two reasons:

1. Generate exceptional execution paths in the data-flow graphs to handle the potential ownership mismatch exceptions implicitly raised by statements accessing object fields and array elements. This was done by modifying Soot's Throw Analysis module because the framework's API does not comes with a handy way to switch the default throw analysis with a custom one. Most probably because it is unusual to add implicit exceptions to instructions of the Java language.

2. Annotate input class files with safety information without producing modified bytecodes for the methods code. The traditional workflow in Soot is to transform the input bytecode to the intermediate representation, apply analyses, transformations, optimizations and annotations on the IR, and finally generate output class files with obtained by translating the IR back to the java bytecode. Even when disabling all the transformations and the optimizations, the output bytecode still presents some differences with the input one. To be completely fair in our comparative benchmarks, we wanted the output annotated bytecode to be identical to the input bytecode in terms of instructions. Therefore we wrote a custom output module for Soot, using ASM[25] to inject our own annotations in the original input class files.

Chapter 6

Evaluation

6.1 Overall performances

6.1.1 Dacapo Benchmark

The Dacapo[14] suite offers a collection of non-trivial benchmarks based on widely used Java programs, representative of various real industrial workloads. These applications use regular Java and thus make no mailbox-based communications. The bytecode is instrumented with Siaam's owner-checkings and all threads share the same owner-ID. With this benchmark we measure the overhead of the dynamic ownership machinery, encompassing the object owner initialization and the owner-checking barriers, plus the allocation and collection costs linked to the object header modifications.
We benchmarked five configurations:

- `nosiaam` is the reference JikesRVM without modifications.
- `opts` designates the modified JikesRVM with JIT ownership elimination.
- `noopts` designates the modified JikesRVM without JIT ownership elimination, an owner-checking barrier is performed for every field access except for stack-allocated objects.
- `sopts` is the same as `opts` but the application bytecode has safety annotations issued by the offline Siaam static analysis tool.
- `sopts-nosc` is the same as `sopts` without library methods cloning and therefore without owner-check barriers generated for the standard library bytecode. Note that this configuration does not enforce isolation, it is just provided as an ideal optimization reference.

The `opts`, `noopts` and `sopts` have standard library methods duplicated with the application context version instrumented with owner-checking barriers.

We executed the version 2010-MR2 of the Dacapo benchmark, each run consists of one program of the suite executed five times for warmup and a last time for execution time measurement. Every run was repeated several times, although some programs have more variability than others, the repetition of the experience with different machines and operating systems confirms the tendency of the following results. Two workloads are benchmarked, the `default` and the `large`. We force methods to be compiled with the optimizing compiler, the adaptive optimization system is disabled, after the first warmup turn most of the bytecode is compiled. The heap size is set to 2GB, the allocation and collection system is the *generational immix*[15]. The provided results were obtained using a machine equipped with an Intel Xeon W3520 2.67Ghz processor.

The following execution time results are normalized with respect to the reference virtual machine for each program of the suite: *lower is better.* We include the geometric mean to summarize the typical overhead for each configuration. The second geometric mean given in the histograms excludes the `jython` program, although it does not changes the interpretation of the results in a significant way.

The table in Figure 6.1 shows the results for the Dacapo 2010-MR2 runs. The graphical representation is exposed in figures 6.2 for the `default` workload and 6.3 for the `large` one. A quick look at the histograms and the geometric means let us concludes that as expected the configuration without any owner-check elimination has the worst overhead, followed by the configuration where runtime JIT elimination is performed. Statically annotated bytecode provides a significant performance improvement. Removing the barriers in the standard library provides more or less significant improvement, certainly depending on the more or less intensive usage of the library.

Benchmark	opts	noopts	sopts	sopts-nosc
default workload				
`antlr`	1.20	1.32	1.09	1.11
`bloat`	1.24	1.41	1.17	1.05
`hsqldb`	1.24	1.36	1.09	1.06
`jython`	1.52	1.73	1.41	1.24
`luindex`	1.25	1.46	1.09	1.05
`lusearch`	1.31	1.45	1.17	1.18
`pmd`	1.32	1.37	1.29	1.24
`xalan`	1.24	1.39	1.33	1.35
geometric mean	1.28	1.43	1.20	1.16
large workload				
`antlr`	1.21	1.33	1.11	1.10
`bloat`	1.40	1.59	1.14	0.96
`hsqldb`	1.45	1.60	1.29	1.10
`jython`	1.45	1.70	1.45	1.15
`luindex`	1.25	1.43	1.09	1.03
`lusearch`	1.33	1.49	1.21	1.21
`pmd`	1.34	1.44	1.39	1.30
`xalan`	1.29	1.41	1.38	1.40
geometric mean	1.34	1.50	1.25	1.15

Table 6.1: Dacapo benchmark *2006-10 MR2 release.*

The main conclusion of the quantitative analysis of the results is that the modified virtual machine including JIT barrier elimination has an overhead of about 30% compared to the not-isolated reference. The JIT elimination improves the performances by slightly less than 20% compared to the `noopts` configuration. When the bytecode is annotated by the whole-program static analysis the performances are 10% to 20% better than with the runtime-only eliminations. Although we must precise here, the Dacapo benchmarks uses the Java reflection API to load classes and invoke methods, therefore the static analysis with Soot was not able to process all the bytecode with the best precision. We can expect better results with other programs for which the call graph can be entirely built with precision. Moreover the we used a context-insensitive, flow-insensitive pointer analysis, meaning the Siaam analysis could be even more accurate with sensitive standard analyses. Although `sopts` is currently backed by an offline analysis, we can expect at least equivalent results from a whole-program analysis integrated in the virtual machine.

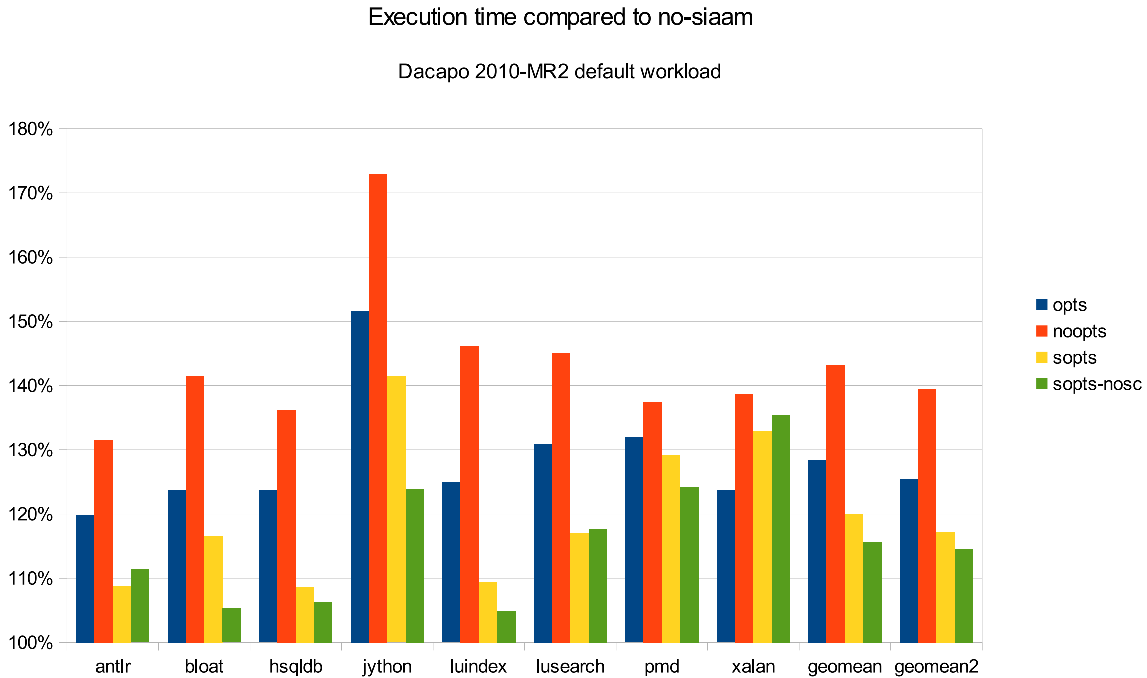


Figure 6.2:

Finally the standard library bytecode is not annotated by our tool, it is only treated by the just-in-time elimination optimization. In order to give a preview of the performances that we may expect from further optimizations of the standard library, we provide the `sopts-nosc` configuration. The results show constant overhead (w.r.t. application) with a mean of 15% overhead, which shoud be considered as an acceptable price to pay for the simplicity of developing isolated programs with Siaam.

6.2 Micro benchmarks

Methodology. We repeatedly measure the execution time of the benchmarked operation. Time is red with `System.nanoTime()` to get the most precise timestamp right before and after the operation. The benchmarking stops when a given amount of time has been totalized. The total amount of time measured divided by the execution count gives the micro-benchmark run result in operations/unit of time. The higher is the result, the faster the operation executes.

The JikesRVM AOS is disabled and methods are initially compiled with the highest level of optimizations. We select the `GenImmix` two-generations copying collector and allocate a 2GB heap. The virtual machine is compiled for the 32-bit x86 architecture with four-byte memory words and addresses size. For each benchmarking run, we execute warmup iterations up to ten seconds, and continue with the performance measurement phase totalizing one minute. We run the experiments several times on various hardware and observed only non-significative variance in the results.

Realistic messages workload. Our main workload for the micro-benchmarks is based on the study by Garner et al. [46] where they characterize the most-frequent reference fields lay-

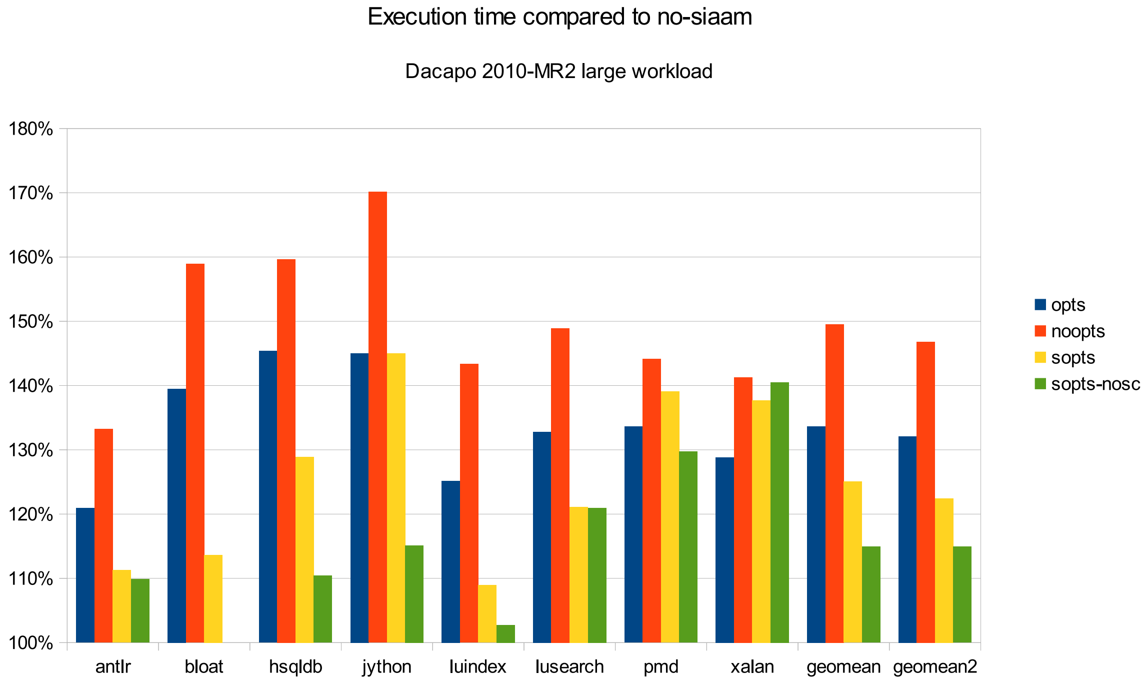


Figure 6.3:

outs in objects allocated by the SPECjvm98 and the Dacapo benchmarks. Both benchmarks includes representative, real-purpose applications such as databases, the java compiler, XML and XSLT processors, bytecode analysis tools, parser generators and more. We produce a mix of objects in the proportions presented in the latter study. The mix contains objects of 32 different fields layouts. The eight most frequent patterns totalize 93% of the mix. the most common pattern (33% of the mix) has no references, then 18% of the objects have one reference localized in their first field, the third, fourth and fifth patterns have respectively three, six and two reference fields localized at the begining of the layout. The sixth most frequent kind of object is array of references (we use one-element arrays for that pattern). Objects with references located after the 16th field represent about 0.15% of the typical mix. The analysis of the Dacapo benchmarks by Blackburn et al. [14] suggests the mean object size is sixty-two bytes. Assuming this size includes a two-words header, there is room left for thirteen four-bytes fields or array elements. Thus in our mix, object field layouts are padded with integers up to thirteen fields to ensure a mean size of about sixty-eight bytes – sixty-four plus the two header bytes required by Siaam. Although some objects in the mix are larger than sixty-eight bytes, they represent less than 0.2% of the total demography.

6.2.1 Zero-Copy vs. Deep-Copy.

Table 6.4 compares message construction performances of the deep-copy and the zero-copy methods for various message sizes with the SPECjvm98/Dacapo mix of objects. Each row fixes the number of objects in one message, ranging from ten to one million. The results in the central part of the table are expressed in number of messages that can be constructed in one second for each method. The right-most column presents how faster is the zero-copy

method compared to the deep-copy. Zero-copy is about three to almost five times faster than deep-copy. The bottom row give the mean number of messages that can be constructed in one second given the count 10^n of objects in each message. We don't take into account the results for ten objects messages because the timing information is too imprecise in that case. Under is the corresponding mean time to process one object of a message. In average, zero-copy outperforms deep-copy by a factor of four in this experiment.

	Message constructions/s		
Objects count	Deep-copy	Zero-copy	Ratio
1×10^1	2.5×10^5	8.4×10^5	×3.4
1×10^2	3.2×10^4	1.3×10^5	×4.1
1×10^3	3.5×10^3	1.3×10^4	×3.7
1×10^4	3.4×10^2	1.4×10^3	×4.1
1×10^5	3.1×10^1	1.3×10^2	×4.2
1×10^6	2.7×10^0	1.3×10^1	×4.8
1×10^n	$3.2 \times 10^{6-n}$	$13.2 \times 10^{6-n}$	×4.1
s/object	313×10^{-9}	75×10^{-9}	

Table 6.4: Message construction performances with a representative mix of objects, thirteen-fields padded layouts. On a Intel Xeon W3520 2.67GHz, 8MB cache.

We reproduced the benchmark over several machine with different processors. The results are summarized in Table 6.7. One-hundred-thousand objects per message may not be representative of the typical value in a real-purpose program, but it amortize the timing imprecision. For comparison we present results for one hundred objects per messages. The observed ratio corroborates the previous results, zero-copy outperforms deep-copy by a factor of three up to almost five.

	$\times 10^4$ message constructions/s			message constructions/s		
	100 objects			10^5 objects		
Processor	Deep-copy	Zero-copy	Ratio	Deep-copy	Zero-copy	Ratio
Core 2 E8400 3.0Ghz	5.2	15.9	×3.0	56	179	×3.2
Xeon W3520 2.67GHz	3.2	12.9	×4.0	30	133	×4.4
Xeon L5420 2.50Ghz	2.2	10.5	×4.7	19	86	×4.5
Opteron 6164 HE 1.7Ghz			×			×

Table 6.5: Message construction performances with a representative mix of objects. Comparing several processors performances.

We ran the same micro-benchmarks without padding object fields layouts, so objects are only large enough to contain their last reference field. Except for objects without reference fields (33%) which have one primitive field. Most objects in the mix have all their reference fields within six fields, so the mean object size is

6.2.2 Graph-traversal optimizations

Method specialization techniques discussed in section 4.2.5 leads to an 18% reduction in message graph exploration time compared to naive type reflection (Table 6.8). This improvement is consistent with the 21% reduction measured by Garner et al.[46] when applying method specialization to the garbage collector scanning mechanism.

	Message constructions/s		
Objects count	Deep-copy	Zero-copy	Ratio
1×10^1	3.5×10^5	1.0×10^6	×2.9
1×10^2	5.0×10^4	1.6×10^5	×3.2
1×10^3	5.6×10^3	1.7×10^4	×3.0
1×10^4	5.8×10^2	1.8×10^3	×3.1
1×10^5	5.7×10^1	1.8×10^2	×3.2
1×10^6	4.7×10^0	1.8×10^1	×3.8
1×10^n	$5.05 \times 10^{6-n}$	$16.2 \times 10^{6-n}$	×3.2
s/object	198×10^{-9}	56×10^{-9}	

Table 6.6: Message construction performances with a representative mix of objects, non-padded fields layouts. On a Intel Xeon W3520 2.67GHz, 8MB cache.

	$\times 10^5$ message constructions/s			message constructions/s		
	10 objects			10^6 objects		
Processor	Deep-copy	Zero-copy	Ratio	Deep-copy	Zero-copy	Ratio
Core 2 E8400 3.0Ghz	3.8	10.9	×2.9	4.0	17.1	×4.3
Xeon W3520 2.67GHz	3.5	10.3	×2.9	4.7	17.8	×3.8
Xeon L5420 2.50Ghz	2.2	7.8	×3.5	3.3	13.7	×4.1
Opteron 6164 HE 1.7Ghz	1.5	5.0	×3.3	2.0	7.5	×3.8

Table 6.7: Message construction performances with a representative mix of objects. Comparing several processors performances.

Objects count	(1) RVMType Reflection	(1)+(2) Hand-Inlining	(1)+(2)+(3) Specialization
1×10^4	1.7×10^3	1.9×10^3	2.0×10^3
1×10^5	1.7×10^2	1.9×10^2	2.0×10^2
1×10^6	1.6×10^1	1.9×10^1	1.9×10^1

Table 6.8: Graph-traversal optimizations performances. Intel Xeon W3520 Gulftown 2.67GHz, 8MB cache.

6.2.3 Owner Check

We measure the amount of time required for an owner-checking operation. The micro-benchmark accumulate execution time by slices of 10×10^6 owner-check operations in 10×10^4 iterations of 100 consecutive hard-coded owner-checks. So about 1% of the total executed instructions is spent incrementing and comparing the `for` loop counter. Various constant propagation and field analysis are disabled so that consecutive owner-checks are not optimized-out by the compiler.

The measurement reveals that a single owner-check consumes about $5.5ns$ on a *Intel® Core 2 Duo 2.8Ghz*. We also decompose the owner-check operation into i) reading the current owner-ID in the current thread ($2.1ns$). ii) reading the object owner from the object's header ($< 1ns$).

6.2.4 Memory consumption

The Siaam virtual machine tends to consume more heap space than the unmodified JikesRVM. First, classes of the standard Java library used by both the virtual machine

and the application are duplicated, meaning two versions of the native code generated by the just-in-time compiler must reside in memory. Second, since more methods must be compiled, more short living objects are produced by the compiler's activity. Hopefully the compilation phases usually don't span the entire application lifetime so the heap is not constantly polluted by compiler's garbage. Third, the two words we add in every object's header may have a more or less significant impact on the application's heap space consumption, depending on the average size of the allocated objects. The average object size in the Dacapo benchmarks is [14] sixty-two bytes, so we increase it by 13%. We have measured a 13% increase in the full garbage collection time, which account for the tracing of the two additional references and the memory compaction.

6.3 Owner-checking elimination

6.3.1 Actory Foundry Benchmarks

We compare the efficiency of the Siaam whole-program analysis to the SOTER algorithm. Table 6.9 contains the results that we obtained for the benchmarks reported in [81]. For each analyzed application we give the total number of owner-checking barriers and the total number of message passing sites in the bytecode. The columns "Ideal safe" show the expected number of safe sites for each criteria. The column "Siaam safe" gives the result obtained with Siaam. The analysis execution time is given in the third main colum. The last column compares the result ratio to ideal for both SOTER and Siaam. Our analysis outperforms significantly. The number of sites reported in the "message passing" colum is different in Table 6.9 compared to Negara's "passed arguments" article because we count one site when they count one or several arguments passed at that site, but we count one site when an actor is created (because actor creation is translated to Siaam actor creation) when they do not count it. Anyway when we obtain a 100% ratio to ideal this difference is meaningless.

Comparison with SOTER. SOTER relies on an inter-procedural live-analysis and a points-to analysis to infer message passing sites where by-ref semantic can applies safely. Given an argument a_i of a message passing site s in the program, SOTER computes the set of objects passed by a_i and the set of objects transitively reachable from the variables live after s. If the intersection of these two sets is empty, SOTER marks a_i as eligible for by-ref argument passing, otherwise it must use the default by-value semantic. The weakness to this pessimistic approach is that among the live objects, a significant part won't actually be accessed in the control-flow after s. On the other hand, Siaam do care about objects being actually accessed, which is a stronger evidence criterion to incriminate message passing sites. Although Siaam's algorithm wasn't designed to optimize-out by-value message passing, it is perfectly adapted for that task. For each unsafe instruction detected by the algorithm, there is one or several guilty dominating message passing sites. Our diagnosis algorithm tracks back the application control-flow from the unsafe instruction to the incriminated message passing sites. These sites represent a subset of the sites where SOTER cannot optimize-out by-value argument passing.

Diagnosis. (1) Siaam reports five problematic message passing sites. Among them, three are legit, for the two others the algorithm is tricked by the dispatching loop allowing sequences of messages that would not happen at runtime.

(2) Siaam reports five problematic ownercheckings and seven problematic message passing sites. The reported ownercheckings are explained in (3) along with five of the problematic message passing sites. The two remaining message passing sites are legitimately reported because the analyzed program passes the same array to multiple actors.

(3) The program is safe but our analysis is again tricked by the dispatching loop allowing sequences of message reception, and in some cases cannot statically infer the safety of certain parts of the actor's private state.

(4) Siaam legitimately reports one problematic message passing site. The program synchronously sends (`call`) the same object twice in a row. The problem is trivially fixed by making the object's class immutable.

	Ownercheck			Message Passing				ratio to Ideal	
	Sites	Ideal safe	Siaam safe	Sites	Ideal safe	Siaam safe	Time (sec)	Siaam	SOTER
ActorFoundry									
`threadring`	24	24	24	8	8	8	0.1	100%	100%
(1) `concurrent`	99	99	99	15	12	10	0.1	98%	58%
(2) `copymessages`	89	89	84	22	20	15	0.1	91%	56%
`performance`	54	54	54	14	14	14	0.2	100%	86%
`pingpong`	28	28	28	13	13	13	0.1	100%	89%
`refmessages`	4	4	4	6	6	6	0.1	100%	67%
Benchmarks									
`chameneos`	75	75	75	10	10	10	0.1	100%	33%
`fibonacci`	46	46	46	13	13	13	0.2	100%	86%
`leader`	50	50	50	10	10	10	0.1	100%	17%
`philosophers`	35	35	35	10	10	10	0.2	100%	100%
`pi`	31	31	31	8	8	8	0.1	100%	67%
`shortestpath`	147	147	147	34	34	34	1.2	100%	88%
Synthetic									
`quicksortCopy`	24	24	24	8	8	8	0.2	100%	100%
(3) `quicksortCopy2`	56	56	51	10	10	5	0.1	85%	75%
Real world									
`clownfish`	245	245	245	102	102	102	2.2	100%	68%
(4) `rainbow_fish`	143	143	143	83	82	82	0.2	99%	99%
`swordfish`	181	181	181	136	136	136	1.7	100%	97%

Table 6.9: ActorFoundry analyses.

Chapter 7

Conclusion

7.1 Synthesis

We presented the specification of Siaam, a simple actor programming model enforcing a strong isolation property based on a flat ownership relation. The specification is divided into an actor-local semantics which is inspired by the Java language, and a global semantics that controls and interleaves certain actor actions. The global semantics is orthogonal to the local semantics, it allows to abstract the local behavior and focus on the dynamic ownership at the global level. We formalized an abstraction of Siaam's global semantics with the Coq proof assistant and conducted the proof of our isolation property.

We have implemented our ownership-based isolation specification in the JikesRVM, a trustworthy Java virtual machine. We were able to run programs representative of industrial workloads with an acceptable overhead, ranging from 30% in average with the just-in-time intra-procedural barriers elimination to 15% using a whole-program analysis and forecasting future improvements in the way the standard Java library may be optimized. Our effort to make the ownership management as versatile as possible allowed us to provide not only the Siaam actor programming model but also some bytecode compatible APIs that mimic existing models while immediately bringing zero-copy message-passing and strong isolation.

The static analysis addresses a larger scope than the actor model specified in Chapter 2. It captures the evolving ownership relation and the essence of the various programming models that may adopt the ownership transfer as a communication mechanism. The analysis is based on an abstract description of the programming models in term of ownership side-effects. We showed briefly how to combine several programming models in a single program. Furthermore the analysis is not tied to a particular interleaving semantics as long as each logical unit of sequential execution represents a unique and fixed owner. The result of the analysis is used at runtime to eliminate redundant owner checking barriers, the evaluation shows that a fast intra-procedural analysis performed just-in-time successfully removes the half of the barriers. The whole-program version of the analysis, currently computed ahead-of-time, offers very promising results and it makes no doubt that it may be embedded in the virtual machine as well.

7.2 Future Work

7.2.1 Just-in-time whole-program analysis.

We believe that for long-running applications like web servers, databases or enterprise frameworks, a virtual machine could incrementally compute the inter-procedural analysis with more and more accuracy for hot-spot methods. Existing works on runtime adaptive optimization[9, 10] and incremental call-graph construction[90] indicates that state-of-the-art runtime systems are already providing the necessary machinery.

Furthermore, when new classes are dynamically loaded, the virtual machine can rollback the previous optimizations and start a new cycle of gradual optimization.

The method cloning technique employed in the virtual machine could also be employed to clone methods dynamically and eliminate owner-checkings based on inter-procedural criteria. For instance if the same method is called from a particular set of contexts with many unsafe parameters and from another set of contexts with only safe objects reachable from the parameters, it may be worth duplicating the method and have two version of it. This idea was already suggested in [11], while dealing with write barriers overhead the authors conclude: "[the JIT compiler] could even perform method splitting to specialize methods, so as to remove useless barriers along frequently used call paths". It is worth noticing that the aggressive inlining[55] done in the JikesRVM is a primitive form of specialization since our optimization pass eliminates owner checking barriers after the IR is inlined.

7.2.2 Cohabitation with ownership types annotations.

The Siaam model and the static analyses described in this chapter demonstrates that it is possible to build an isolated actor programming model without requiring any explicit ownership type annotation in the program's source code. Although it does not necessarily mean annotations should be forbidden in the source code, nor that ownership types analyses shouldn't be used. In most previous works where ownership types are statically verified, the authors suggest a default mode for variables, arguments, fields etc. so that developers only need to annotate places where the default mode is not appropriated because it is too constraining.

We shall study the mutual benefits of having an analysis like Siaam working in collaboration with a static ownership type verifier. Since they both compute sound approximations, it may happen that a given property is not verified by one analysis and verified by the other, in which case the first analysis can safely assume that the property holds and continue with that fact. The question is to define what properties can be checked by both analyses, or what property may only be checked by one of the analyses but would be useful to the other.

Another field of interest is to use ownership types annotations on commonly used libraries, particularly the Java standard library in our case. Modular static ownership types verifiers[34, 1] are particularly interesting because they can check pieces of code without requiring the whole-program to be available. In [53] authors report that they successfully annotated parts of the Scala standard library with significantly few annotations. Therefore it suggests that the same kind of work should be tractable on the Java library. Modular ownership verification is based on annotated interfaces specifying ownership constraints on the the arguments of the modules or software components. If a non-annotated program is to use an annotated module, it is necessary to make sure, statically or dynamically, that the non-annotated code will actually respect the module's specification. A Siaam-like analysis may help identifying

places where the constraints cannot be guaranteed, so the appropriate runtime verifications may be handled dynamically by a Siaam-like virtual machine.

7.2.3 Modular actor analysis.

In future works we shall study how to precisely analyze a single actor without considering the whole-program knowledge. The challenge is to obtain a sound call-graph and a sound pointer analysis in the absence of information about the objects that might be received by inter-actor communications. The following questions must be studied:

— For each message reception site, what is the shape of the received message? By shape we mean what topology has the graph of received objects, what can be the type of the objects in that graph.

— If a received object is of an interface class type or a non-final class type it is not possible to determine what will be its exact instance class statically. Therefore it is not possible to know what are the (ownership) side-effects of calling the methods of such object.

We can think of the following methodology:

— Close the classes hierarchy by adding an artificial final class extending each known non-final class and make each artificial class implements all the known interfaces. Also add a single artificial final class implementing all the known interfaces. Declare a single field of type Object in each of the artificial classes.

— Use the most conservative variant of the Transferred Objects Analysis to approximate the set of objects invalidated by the methods of the artificial classes.

— Transform every message reception statement so that the set of abstract objects transitively reachable from the message comprises every abstract object that might have already been sent by the actor before the reception, plus one fresh abstract object representing the reception of an object never sent nor received before.

7.2.4 Demand-driven Siaam analysis

In Section 5.8.2 we presented a programming assistant for the whole-program static analysis. The assistant explores the inter-procedural program control-flow in the reverse direction in order to find every previous program point that could lead to an ownership exception at one or several starting points. Interestingly, if we puts every object access statements in the set of starting points, the exploration goes through every program point that *must* be analyzed by the Safe Objects Analysis, all the other program points may be discarded. Therefore the graph expansion algorithm detailed in Section 5.8.2 may be used to drive the *SOA* and limit the amount of processing required by this phase of the analysis.

Appendix A

Jinja

type_synonym $\textit{jvm-state} = \textit{addr option} \times \textit{heap} \times \textit{frame list}$
$\textit{frame} = \textit{opstack} \times \textit{registers} \times \textit{cname} \times \textit{mname} \times \textit{pc}$
$\textit{opstack} = \textit{val list}$
$\textit{registers} = \textit{val list}$

$$\frac{P \vdash \langle e, s\rangle \rightarrow \langle e', s\rangle}{P \vdash \langle \texttt{Cast}\ C\ e, s\rangle \rightarrow \langle \texttt{Cast}\ C\ e', s'\rangle}$$

$$\frac{P \vdash \langle e, s\rangle \rightarrow \langle e', s\rangle}{P \vdash \langle V := e, s\rangle \rightarrow \langle V := e', s'\rangle}$$

$$\frac{P \vdash \langle e, s\rangle \rightarrow \langle e', s\rangle}{P \vdash \langle e.F\{D\}, s\rangle \rightarrow \langle e'.F\{D\}, s'\rangle}$$

$$\frac{P \vdash \langle e, s\rangle \rightarrow \langle e', s\rangle}{P \vdash \langle e.F\{D\} := e_2, s\rangle \rightarrow \langle e'.F\{D\} := e_2, s'\rangle}$$

$$\frac{P \vdash \langle e, s\rangle \rightarrow \langle e', s\rangle}{P \vdash \langle \texttt{Val}\ v.F\{D\} := e, s\rangle \rightarrow \langle \texttt{Val}\ v.F\{D\} := e', s'\rangle}$$

$$\frac{P \vdash \langle e, s\rangle \rightarrow \langle e', s\rangle}{P \vdash \langle e \ll bop \gg e_2, s\rangle \rightarrow \langle e' \ll bop \gg e_2, s'\rangle}$$

$$\frac{P \vdash \langle e, s\rangle \rightarrow \langle e', s\rangle}{P \vdash \langle \texttt{Val}\ v \ll bop \gg e, s\rangle \rightarrow \langle \texttt{Val}\ v \ll bop \gg e', s'\rangle}$$

$$\frac{P \vdash \langle e, (h, l\ (V := None))\rangle \rightarrow \langle e', (h', l')\rangle \qquad l'\ V = None \qquad \neg assigned\ V\ e}{P \vdash \langle \{V : T;\ e\}, (h, l)\rangle \rightarrow \langle \{V : T;\ e'\}, (h', l'\ (V := l\ V))\rangle}$$

$$\frac{P \vdash \langle e, (h, l\ (V := None))\rangle \rightarrow \langle e', (h', l')\rangle \qquad l'\ V = \lfloor v \rfloor \qquad \neg assigned\ V\ e}{P \vdash \langle \{V : T;\ e\}, (h, l)\rangle \rightarrow \langle \{V : T;\ V := \texttt{Val}\ v;\ e'\}, (h', l'\ (V := l\ V))\rangle}$$

$$\frac{P \vdash \langle e, (h, l(V \ \mapsto\ v))\rangle \rightarrow \langle e', (h', l')\rangle \qquad l'\ V = \lfloor v' \rfloor}{P \vdash \langle \{V : T;\ V := \texttt{Val}\ v;\ e\}, (h, l)\rangle \rightarrow \langle \{V : T;\ V := \texttt{Val}\ v';\ e'\}, (h', l'\ (V := l\ V))\rangle}$$

$$\frac{P \vdash \langle e, s\rangle \rightarrow \langle e', s'\rangle}{P \vdash \langle e.M(es), s\rangle \rightarrow \langle e'.M(es), s'\rangle}$$

$$\frac{P \vdash \langle es, s\rangle [\rightarrow] \langle es', s'\rangle}{P \vdash \langle \texttt{Val}\ v.M(es), s\rangle \rightarrow \langle \texttt{Val}\ v.M(es'), s'\rangle}$$

$$\frac{P \vdash \langle e, s\rangle \rightarrow \langle e', s'\rangle}{P \vdash \langle e;\ e_2, s\rangle \rightarrow \langle e';\ e_2, s'\rangle}$$

$$\frac{P \vdash \langle e, s\rangle \rightarrow \langle e', s'\rangle}{P \vdash \langle \texttt{if}\ (e)\ e_1\ \texttt{else}\ e_2, s\rangle \rightarrow \langle \texttt{if}\ (e')\ e_1\ \texttt{else}\ e_2, s'\rangle}$$

$$\frac{P \vdash \langle e, s\rangle \rightarrow \langle e', s'\rangle}{P \vdash \langle e \cdot es, s\rangle [\rightarrow] \langle e' \cdot es, s'\rangle}$$

$$\frac{P \vdash \langle es, s\rangle [\rightarrow] \langle es', s'\rangle}{P \vdash \langle \texttt{Val}\ v \cdot es, s\rangle [\rightarrow] \langle \texttt{Val}\ v \cdot es', s'\rangle}$$

Figure 1.1: Subexpression reduction

$$\frac{\mathit{new\text{-}Addr}\ h = \lfloor a \rfloor \qquad P \vdash C\ \textsf{has-fields}\ \mathit{FDTs}}{P \vdash \langle \mathtt{new}\ C, (h,l)\rangle \to \langle \mathit{addr}\ a, (h \mapsto a(C, \mathit{init\text{-}fields}\ \mathit{FDTs}), l)\rangle}$$

$$\frac{\mathit{hp}\ s\ a = \lfloor (D, \mathit{fs})\rfloor \qquad P \vdash D \preceq^\star C}{P \vdash \langle \mathtt{Cast}\ C\ (\mathit{addr}\ a), s\rangle \to \langle \mathit{addr}\ a, s\rangle} \qquad P \vdash \langle \mathtt{Cast}\ C\ \mathit{null}, s\rangle \to \langle \mathit{null}, s\rangle$$

$$\frac{\mathit{lcl}\ s\ V = \lfloor v \rfloor}{P \vdash \langle \mathtt{Var}\ V, s\rangle \to \langle \mathtt{Val}\ v, s\rangle} \qquad P \vdash \langle V := \mathtt{Val}\ v, (h,l)\rangle \to \langle \mathit{unit}, (h, l\,(V \mapsto v))\rangle$$

$$\frac{\mathit{binop}(\mathit{bop}, v_1, v2) = \lfloor v \rfloor}{P \vdash \langle \mathtt{Val}\ v_1 \ll \mathit{bop} \gg \mathtt{Val}\ v_2, s\rangle \to \langle \mathtt{Val}\ v, s\rangle} \qquad \frac{\mathit{hp}\ s\ a = \lfloor (C, \mathit{fs})\rfloor \qquad \mathit{fs}(F,D) = \lfloor v \rfloor}{P \vdash \langle a.F\{D\}, s\rangle \to \langle \mathtt{Val}\ v, s\rangle}$$

$$\frac{h\,a = \lfloor (C, fs)\rfloor}{P \vdash \langle a.F\{D\} := \mathtt{Val}\ v, (h,l)\rangle \to \langle \mathit{unit}, (h\,(a \mapsto (C, \mathit{fs}\,((F,D) \mapsto v)))\,, l)\rangle}$$

$$\frac{\begin{array}{c}\mathit{hp}\ s\ a = \lfloor (C, fs)\rfloor \\ P \vdash C\ \textsf{sees}\ M : Ts \to T = (\mathit{pns}, \mathit{body})\ \textsf{in}\ D \qquad |\mathit{vs}| = |\mathit{pns}| \qquad |Ts| = |\mathit{pns}|\end{array}}{P \vdash \langle \mathit{addr}\ a.M(\mathit{map}\ \mathtt{Val}\ \mathit{vs}), s\rangle \to \langle \mathit{blocks}(\mathit{this} \cdot \mathit{pns}, \mathit{Class}\ D \cdot Ts, \mathit{Addr}\ a \cdot \mathit{vs}, \mathit{body}), s\rangle}$$

$$P \vdash \langle \{V : T;\ V := \mathtt{Val}\ v;\ \mathtt{Val}\ u\}, s\rangle \to \langle \mathtt{Val}\ u, s\rangle \qquad P \vdash \langle \{V : T;\ \mathtt{Val}\ u\}, s\rangle \to \langle \mathtt{Val}\ u, s\rangle$$

$$P \vdash \langle \mathtt{Val}\ v;\ e_2, s\rangle \to \langle e_2, s\rangle$$

$$P \vdash \langle \mathtt{if}\ (\mathit{true})\ e_1\ \mathtt{else}\ e_2, s\rangle \to \langle e_1, s\rangle \qquad P \vdash \langle \mathtt{if}\ (\mathit{false})\ e_1\ \mathtt{else}\ e_2, s\rangle \to \langle e_2, s\rangle$$

$$P \vdash \langle \mathtt{while}\ (b)\ c, s\rangle \to \langle \mathtt{if}\ (b)\ (c;\ \mathtt{while}\ (b)\ c)\ \mathtt{else}\ \mathit{unit}, s\rangle$$

Figure 1.2: Expression reduction

$$\frac{\mathit{new\text{-}Addr}\ h = \mathit{None}}{P \vdash \langle \mathtt{new}\ C, (h,l)\rangle \to \langle \mathtt{THROW}\ \mathit{OutOfMemory}, (h,l)\rangle}$$

$$\frac{\mathit{hp}\ s\ a = \lfloor (D, \mathit{fs})\rfloor \qquad \neg P \vdash D \preceq^\star C}{P \vdash \langle \mathtt{Cast}\ C\ (\mathit{addr}\ a), s\rangle \to \langle \mathtt{THROW}\ \mathit{ClassCast}, s\rangle}$$

$$P \vdash \langle \mathit{null}.F\{D\}, s\rangle \to \langle \mathtt{THROW}\ \mathit{NullPointer}, s\rangle$$

$$P \vdash \langle \mathit{null}.F\{D\} := \mathtt{Val}\ v, s\rangle \to \langle \mathtt{THROW}\ \mathit{NullPointer}, s\rangle$$

$$P \vdash \langle \mathit{null}.M(\mathit{map}\ \mathtt{Val}\ \mathit{vs}), s\rangle \to \langle \mathtt{THROW}\ \mathit{NullPointer}, s\rangle$$

$$\frac{P \vdash \langle e, s\rangle \to \langle e', s'\rangle}{P \vdash \langle \mathtt{throw}\ e, s\rangle \to \langle \mathtt{throw}\ e', s'\rangle} \qquad P \vdash \langle \mathtt{throw}\ \mathit{null}, s\rangle \to \langle \mathtt{THROW}\ \mathit{NullPointer}, s\rangle$$

$$\frac{P \vdash \langle e, s\rangle \to \langle e', s'\rangle}{P \vdash \langle \mathtt{try}\ e\ \mathtt{catch}\ (C\ V)\ e_2, s\rangle \to \langle \mathtt{try}\ e'\ \mathtt{catch}\ (C\ V)\ e_2, s'\rangle}$$

$$P \vdash \langle \mathtt{try}\ \mathtt{Val}\ v\ \mathtt{catch}\ (C\ V)\ e_2, s\rangle \to \langle \mathtt{Val}\ v, s\rangle$$

$$\frac{\mathit{hp}\ s\ a = \lfloor (D, \mathit{fs})\rfloor \qquad P \vdash D \preceq^\star C}{P \vdash \langle \mathtt{try}\ \mathtt{Throw}\ a\ \mathtt{catch}\ (C\ V)\ e_2, s\rangle \to \langle \{V : \mathit{Class}\ C;\ V := \mathit{addr}\ a;\ e_2\}, s\rangle}$$

$$\frac{\mathit{hp}\ s\ a = \lfloor (D, \mathit{fs})\rfloor \qquad \neg P \vdash D \preceq^\star C}{P \vdash \langle \mathtt{try}\ \mathtt{Throw}\ a\ \mathtt{catch}\ (C\ V)\ e_2, s\rangle \to \langle \mathtt{Throw}\ a, s\rangle}$$

Figure 1.3: Exceptional expression reduction

$P \vdash \langle \texttt{Cast}\ C\ \texttt{throw}\ e, s\rangle \to \langle \texttt{throw}\ e, s\rangle$

$P \vdash \langle V := \texttt{throw}\ e, s\rangle \to \langle \texttt{throw}\ e, s\rangle$

$P \vdash \langle \texttt{throw}\ e.F\{D\}, s\rangle \to \langle \texttt{throw}\ e, s\rangle$

$P \vdash \langle \texttt{throw}\ e.F\{D\} := e_2, s\rangle \to \langle \texttt{throw}\ e, s\rangle$

$P \vdash \langle \texttt{Val}\ v.F\{D\} := \texttt{throw}\ e, s\rangle \to \langle \texttt{throw}\ e, s\rangle$

$P \vdash \langle \texttt{throw}\ e \ll bop \gg e_2, s\rangle \to \langle \texttt{throw}\ e, s\rangle$

$P \vdash \langle \texttt{Val}\ v \ll bop \gg \texttt{throw}\ e, s\rangle \to \langle \texttt{throw}\ e, s\rangle$

$P \vdash \langle \{V{:}T;\ \texttt{Throw}\ a\}, s\rangle \to \langle \texttt{Throw}\ a, s\rangle$

$P \vdash \langle \{V{:}T;\ V := \texttt{Val}\ v;\ \texttt{Throw}\ a\}, s\rangle \to \langle \texttt{Throw}\ a, s\rangle$

$P \vdash \langle \texttt{throw}\ e.M(es), s\rangle \to \langle \texttt{throw}\ e, s\rangle$

$P \vdash \langle \texttt{Val}\ v.m(map\ \texttt{Val}\ vs\ @\ (\texttt{throw}\ e \cdot es')), s\rangle \to \langle \texttt{throw}\ e, s\rangle$

$P \vdash \langle \texttt{throw}\ e;\ e_2, s\rangle \to \langle \texttt{throw}\ e, s\rangle$

$P \vdash \langle \texttt{if}\ (\texttt{throw}\ e)\ e_1\ \texttt{else}\ e_2, s\rangle \to \langle \texttt{throw}\ e, s\rangle$

$P \vdash \langle \texttt{throw}\ \texttt{throw}\ e, s\rangle \to \langle \texttt{throw}\ e, s\rangle$

Figure 1.4: Exception propagation

Appendix B

Siaam

$$\textsc{CastE}\frac{P, w \vdash \langle e, s\rangle \;-wa\rightarrow\; \langle e', s'\rangle}{P, w \vdash \langle \texttt{Cast}\ C\ e, s\rangle \;-wa\rightarrow\; \langle \texttt{Cast}\ C\ e', s'\rangle}$$

$$\textsc{StoreE}\frac{P, w \vdash \langle e, s\rangle \;-wa\rightarrow\; \langle e', s'\rangle}{P, w \vdash \langle V := e, s\rangle \;-wa\rightarrow\; \langle V := e', s'\rangle}$$

$$\textsc{ReadE}\frac{P, w \vdash \langle e, s\rangle \;-wa\rightarrow\; \langle e', s'\rangle}{P, w \vdash \langle e.F\{D\}, s\rangle \;-wa\rightarrow\; \langle e'.F\{D\}, s'\rangle}$$

$$\textsc{WriteLE}\frac{P, w \vdash \langle e, s\rangle \;-wa\rightarrow\; \langle e', s'\rangle}{P, w \vdash \langle e.F\{D\} := e_2, s\rangle \;-wa\rightarrow\; \langle e'.F\{D\} := e_2, s'\rangle}$$

$$\textsc{WriteRE}\frac{P, w \vdash \langle e, s\rangle \;-wa\rightarrow\; \langle e', s'\rangle}{P, w \vdash \langle \texttt{Val}\ v.F\{D\} := e, s\rangle \;-wa\rightarrow\; \langle \texttt{Val}\ v.F\{D\} := e', s'\rangle}$$

$$\textsc{BopLE}\frac{P, w \vdash \langle e, s\rangle \;-wa\rightarrow\; \langle e', s'\rangle}{P, w \vdash \langle e \ll bop \gg e_2, s\rangle \;-wa\rightarrow\; \langle e' \ll bop \gg e_2, s'\rangle}$$

$$\textsc{BopRE}\frac{P, w \vdash \langle e, s\rangle \;-wa\rightarrow\; \langle e', s'\rangle}{P, w \vdash \langle \texttt{Val}\ v \ll bop \gg e, s\rangle \;-wa\rightarrow\; \langle \texttt{Val}\ v \ll bop \gg e', s'\rangle}$$

$$\textsc{VarDeclE}\frac{P, w \vdash \langle e, (h, l\ (V := None))\rangle \;-wa\rightarrow\; \langle e', (h', l')\rangle \qquad l'\ V = None \qquad \neg assigned\ V\ e}{P, w \vdash \langle \{V : T;\ e\}, (h, l)\rangle \;-wa\rightarrow\; \langle \{V : T;\ e'\}, (h', l'\ (V := l\ V))\rangle}$$

$$\textsc{VarDefE}\frac{P, w \vdash \langle e, (h, l\ (V := None))\rangle \;-wa\rightarrow\; \langle e', (h', l')\rangle \qquad l'\ V = \lfloor v \rfloor \qquad \neg assigned\ V\ e}{P, w \vdash \langle \{V : T;\ e\}, (h, l)\rangle \;-wa\rightarrow\; \langle \{V : T;\ V := \texttt{Val}\ v;\ e'\}, (h', l'\ (V := l\ V))\rangle}$$

$$\textsc{VarUpdtE}\frac{P, w \vdash \langle e, (h, l(V \mapsto v))\rangle \;-wa\rightarrow\; \langle e', (h', l')\rangle \qquad l'\ V = \lfloor v' \rfloor}{P, w \vdash \langle \{V : T;\ V := \texttt{Val}\ v;\ e\}, (h, l)\rangle \;-wa\rightarrow\; \langle \{V : T;\ V := \texttt{Val}\ v';\ e'\}, (h', l'\ (V := l\ V))\rangle}$$

$$\textsc{CallE}\frac{P, w \vdash \langle e, s\rangle \;-wa\rightarrow\; \langle e', s'\rangle}{P, w \vdash \langle e.M(es), s\rangle \;-wa\rightarrow\; \langle e'.M(es), s'\rangle}$$

$$\textsc{Args}\frac{P, w \vdash \langle es, s\rangle [-wa\rightarrow] \langle es', s'\rangle}{P, w \vdash \langle \texttt{Val}\ v.M(es), s\rangle \;-wa\rightarrow\; \langle \texttt{Val}\ v.M(es'), s'\rangle}$$

$$\textsc{ArgE}\frac{P, w \vdash \langle e, s\rangle \;-wa\rightarrow\; \langle e', s'\rangle}{P, w \vdash \langle e \cdot es, s\rangle [-wa\rightarrow] \langle e' \cdot es, s'\rangle} \qquad \textsc{ArgV}\frac{P, w \vdash \langle es, s\rangle [-wa\rightarrow] \langle es', s'\rangle}{P, w \vdash \langle \texttt{Val}\ v \cdot es, s\rangle [-wa\rightarrow] \langle \texttt{Val}\ v \cdot es', s'\rangle}$$

$$\textsc{SeqE}\frac{P, w \vdash \langle e, s\rangle \;-wa\rightarrow\; \langle e', s'\rangle}{P, w \vdash \langle e;\ e_2, s\rangle \;-wa\rightarrow\; \langle e';\ e_2, s'\rangle}$$

$$\textsc{CondE}\frac{P, w \vdash \langle e, s\rangle \;-wa\rightarrow\; \langle e', s'\rangle}{P, w \vdash \langle \texttt{if}\ (e)\ e_1\ \texttt{else}\ e_2, s\rangle \;-wa\rightarrow\; \langle \texttt{if}\ (e')\ e_1\ \texttt{else}\ e_2, s'\rangle}$$

$$\textsc{WhileE}\frac{P, w \vdash \langle e, s\rangle \;-wa\rightarrow\; \langle e', s'\rangle}{P, w \vdash \langle \texttt{while}\ (e)\ e_1, s\rangle \;-wa\rightarrow\; \langle \texttt{while}\ (e')\ e_1, s'\rangle}$$

$$\textsc{ThrowE}\frac{P, w \vdash \langle e, s\rangle \;-wa\rightarrow\; \langle e', s'\rangle}{P, w \vdash \langle \texttt{throw}\ e, s\rangle \;-wa\rightarrow\; \langle \texttt{throw}\ e', s'\rangle}$$

$$\textsc{TryE}\frac{P, w \vdash \langle e, s\rangle \;-wa\rightarrow\; \langle e', s'\rangle}{P, w \vdash \langle \texttt{try}\ e\ \texttt{catch}\ (C\ V)\ e_2, s\rangle \;-wa\rightarrow\; \langle \texttt{try}\ e'\ \texttt{catch}\ (C\ V)\ e_2, s'\rangle}$$

Figure 2.1: Siaam Subexpression reduction

$$\textsc{New}\ \frac{\textit{new-Addr}\ h = \lfloor a \rfloor \qquad P \vdash C\ \textsf{has-fields}\ \textit{FDTs}}{P, w \vdash \langle \texttt{new}\ C, (h, l)\rangle\ {-}\textit{NewObj}\ C\ a{\rightarrow}\ \langle \textit{addr}\ a, (h \mapsto a(C, \textit{init-fields}\ \textit{FDTs}), l)\rangle}$$

$$\textsc{Cast}\ \frac{\textit{hp}\ s\ a = \lfloor (D, \textit{fs}) \rfloor \qquad P \vdash D \preceq^{\star} C}{P, w \vdash \langle \texttt{Cast}\ C\ (\textit{addr}\ a), s\rangle\ {-}\textit{Silent}{\rightarrow}\ \langle \textit{addr}\ a, s\rangle}$$

$$\textsc{CastN}\ P, w \vdash \langle \texttt{Cast}\ C\ \textit{null}, s\rangle\ {-}\textit{Silent}{\rightarrow}\ \langle \textit{null}, s\rangle$$

$$\textsc{Load}\ \frac{\textit{lcl}\ s\ V = \lfloor v \rfloor}{P, w \vdash \langle \texttt{Var}\ V, s\rangle\ {-}\textit{Silent}{\rightarrow}\ \langle \texttt{Val}\ v, s\rangle}$$

$$\textsc{Store}\ P, w \vdash \langle V := \texttt{Val}\ v, (h, l)\rangle\ {-}\textit{Silent}{\rightarrow}\ \langle \textit{unit}, (h, l\,(V \mapsto v))\rangle$$

$$\textsc{Bop}\ \frac{\textit{binop}(\textit{bop}, v_1, v_2) = \lfloor v \rfloor}{P, w \vdash \langle \texttt{Val}\ v_1 \ll \textit{bop} \gg \texttt{Val}\ v_2, s\rangle\ {-}\textit{Silent}{\rightarrow}\ \langle \texttt{Val}\ v, s\rangle}$$

$$\textsc{Read}\ \frac{\textit{hp}\ s\ a = \lfloor (C, \textit{fs}) \rfloor \qquad \textit{fs}(F, D) = \lfloor v \rfloor}{P, w \vdash \langle a.F\{D\}, s\rangle\ {-}\textit{OwnerCheck}\ a\ \textit{True}{\rightarrow}\ \langle \texttt{Val}\ v, s\rangle}$$

$$\textsc{Write}\ \frac{h\ a = \lfloor (C, \textit{fs}) \rfloor \qquad h' = h(a \mapsto (C, \textit{fs}((F, D) \mapsto v)))}{P, w \vdash \langle a.F\{D\} := \texttt{Val}\ v, (h, l)\rangle\ {-}\textit{OwnerCheck}\ a\ \textit{True}{\rightarrow}\ \langle \textit{unit}, (h', l)\rangle}$$

$$\textsc{Call}\ \frac{\begin{array}{c}\textit{hp}\ s\ a = \lfloor (C, \textit{fs}) \rfloor \\ P \vdash C\ \textsf{sees}\ M : \textit{Ts} \rightarrow T = \lfloor (\textit{pns}, \textit{body}) \rfloor\ \textsf{in}\ D \qquad |\textit{vs}| = |\textit{pns}| \qquad |\textit{Ts}| = |\textit{pns}|\end{array}}{P, w \vdash \langle \textit{addr}\ a.M(\textit{map}\ \texttt{Val}\ \textit{vs}), s\rangle\ {-}\textit{Silent}{\rightarrow}\ \langle \textit{blocks}\ (\textit{this} \cdot \textit{pns}, \textit{Class}\ D \cdot \textit{Ts}, \textit{Addr}\ a \cdot \textit{vs}, \textit{body}), s\rangle}$$

$$\textsc{VarDef}\ P, w \vdash \langle \{V : T;\ V := \texttt{Val}\ v;\ \texttt{Val}\ u\}, s\rangle\ {-}\textit{Silent}{\rightarrow}\ \langle \texttt{Val}\ u, s\rangle$$

$$\textsc{VarDecl}\ P, w \vdash \langle \{V : T;\ \texttt{Val}\ u\}, s\rangle\ {-}\textit{Silent}{\rightarrow}\ \langle \texttt{Val}\ u, s\rangle$$

$$\textsc{Seq}\ P, w \vdash \langle \texttt{Val}\ v;\ e_2, s\rangle\ {-}\textit{Silent}{\rightarrow}\ \langle e_2, s\rangle$$

$$\textsc{CondTrue}\ P, w \vdash \langle \texttt{if}\ (\textit{true})\ e_1\ \texttt{else}\ e_2, s\rangle\ {-}\textit{Silent}{\rightarrow}\ \langle e_1, s\rangle$$

$$\textsc{CondFalse}\ P, w \vdash \langle \texttt{if}\ (\textit{false})\ e_1\ \texttt{else}\ e_2, s\rangle\ {-}\textit{Silent}{\rightarrow}\ \langle e_2, s\rangle$$

$$\textsc{While}\ P, w \vdash \langle \texttt{while}\ (b)\ e, s\rangle\ {-}\textit{Silent}{\rightarrow}\ \langle \texttt{if}\ (b)\ (e;\ \texttt{while}\ (b)\ e)\ \texttt{else}\ \textit{unit}, s\rangle$$

$$\textsc{Try}\ P, w \vdash \langle \texttt{try}\ \texttt{Val}\ \textit{vs}\ \texttt{catch}\ (C\ V)\ e, s\rangle\ {-}\textit{Silent}{\rightarrow}\ \langle \texttt{Val}\ \textit{vs}, s\rangle$$

$$\textsc{Catch}\ \frac{\textit{hp}\ s\ a = \lfloor (D, \textit{fs}) \rfloor \qquad P \vdash D \preceq^{\star} C}{P, w \vdash \langle \texttt{try}\ \texttt{Throw}\ a\ \texttt{catch}\ (C\ V)\ e, s\rangle\ {-}\textit{Silent}{\rightarrow}\ \langle \{V : \textit{Class}\ C;\ V := \textit{addr}\ a;\ e\}, s\rangle}$$

$$\textsc{ReThrow}\ \frac{\textit{hp}\ s\ a = \lfloor (D, \textit{fs}) \rfloor \qquad \neg P \vdash D \preceq^{\star} C}{P, w \vdash \langle \texttt{try}\ \texttt{Throw}\ a\ \texttt{catch}\ (C\ V)\ e, s\rangle\ {-}\textit{Silent}{\rightarrow}\ \langle \texttt{Throw}\ a, s\rangle}$$

Figure 2.2: Siaam Expression reduction

$$\textsc{OutOfMem}\ \frac{\textit{new-Addr}\ h = \textit{None}}{P, w \vdash \langle \texttt{new}\ C, (h, l)\rangle\ -\textit{Silent}\!\rightarrow \langle \texttt{THROW}\ \textit{OutOfMemory}, (h, l)\rangle}$$

$$\textsc{BadCast}\ \frac{\textit{hp}\ s\ a = \lfloor (D, \textit{fs})\rfloor \qquad \neg P \vdash D \preceq^{\star} C}{P, w \vdash \langle \texttt{Cast}\ C\ (\textit{addr}\ a), s\rangle\ -\textit{Silent}\!\rightarrow \langle \texttt{THROW}\ \textit{ClassCast}, s\rangle}$$

$$\textsc{ReadNull}\ P, w \vdash \langle \textit{null}.F\{D\}, s\rangle\ -\textit{Silent}\!\rightarrow \langle \texttt{THROW}\ \textit{NullPointer}, s\rangle$$

$$\textsc{ReadBadOwner}\ P, w \vdash \langle a.F\{D\}, s\rangle\ -\textit{OwnerCheck}\ a\ \textit{False}\!\rightarrow \langle \texttt{THROW}\ \textit{OwnerMismatch}, s\rangle$$

$$\textsc{WriteNull}\ P, w \vdash \langle \textit{null}.F\{D\} := \texttt{Val}\ \textit{vs}, s\rangle\ -\textit{Silent}\!\rightarrow \langle \texttt{THROW}\ \textit{NullPointer}, s\rangle$$

$$\textsc{WriteBadOwner}\ P, w \vdash \langle a.F\{D\} := \texttt{Val}\ v, (h, l)\rangle\ -\textit{OwnerCheck}\ a\ \textit{False}\!\rightarrow \langle \texttt{THROW}\ \textit{OwnerMismatch}, s\rangle$$

$$\textsc{CallNull}\ P, w \vdash \langle \textit{null}.M(\textit{map}\ \texttt{Val}\ \textit{vs}), s\rangle\ -\textit{Silent}\!\rightarrow \langle \texttt{THROW}\ \textit{NullPointer}, s\rangle$$

$$\textsc{ThrowNull}\ P, w \vdash \langle \texttt{throw}\ \textit{null}, s\rangle\ -\textit{Silent}\!\rightarrow \langle \texttt{THROW}\ \textit{NullPointer}, s\rangle$$

Figure 2.3: Siaam Exceptional expression reduction

$$\textsc{CastT}\ P, w \vdash \langle \texttt{Cast}\ C\ \texttt{throw}\ e, s\rangle\ -\textit{Silent}\!\rightarrow \langle \texttt{throw}\ e, s\rangle$$

$$\textsc{StoreT}\ P, w \vdash \langle V := \texttt{throw}\ e, s\rangle\ -\textit{Silent}\!\rightarrow \langle \texttt{throw}\ e, s\rangle$$

$$\textsc{ReadT}\ P, w \vdash \langle \texttt{throw}\ e.F\{D\}, s\rangle\ -\textit{Silent}\!\rightarrow \langle \texttt{throw}\ e, s\rangle$$

$$\textsc{WriteLT}\ P, w \vdash \langle \texttt{throw}\ e.F\{D\} := e_2, s\rangle\ -\textit{Silent}\!\rightarrow \langle \texttt{throw}\ e, s\rangle$$

$$\textsc{WriteRT}\ P, w \vdash \langle \texttt{Val}\ v.F\{D\} := \texttt{throw}\ e, s\rangle\ -\textit{Silent}\!\rightarrow \langle \texttt{throw}\ e, s\rangle$$

$$\textsc{BopLT}\ P, w \vdash \langle \texttt{throw}\ e \ll \textit{bop} \gg e_2, s\rangle\ -\textit{Silent}\!\rightarrow \langle \texttt{throw}\ e, s\rangle$$

$$\textsc{BopRT}\ P, w \vdash \langle \texttt{Val}\ v \ll \textit{bop} \gg \texttt{throw}\ e, s\rangle\ -\textit{Silent}\!\rightarrow \langle \texttt{throw}\ e, s\rangle$$

$$\textsc{VarDeclT}\ P, w \vdash \langle \{V : T;\ \texttt{Throw}\ a\}, s\rangle\ -\textit{Silent}\!\rightarrow \langle \texttt{Throw}\ a, s\rangle$$

$$\textsc{VarDefT}\ P, w \vdash \langle \{V : T;\ V := \texttt{Val}\ v;\ \texttt{Throw}\ a\}, s\rangle\ -\textit{Silent}\!\rightarrow \langle \texttt{Throw}\ a, s\rangle$$

$$\textsc{CallT}\ P, w \vdash \langle \texttt{throw}\ e.M(\textit{es}), s\rangle\ -\textit{Silent}\!\rightarrow \langle \texttt{throw}\ e, s\rangle$$

$$\textsc{ArgsT}\ P, w \vdash \langle \texttt{Val}\ v.M(\textit{map}\ \texttt{Val}\ \textit{vs}\ @\ (\texttt{throw}\ e \cdot \textit{es}')), s\rangle\ -\textit{Silent}\!\rightarrow \langle \texttt{throw}\ e, s\rangle$$

$$\textsc{Throw}\ P, w \vdash \langle \texttt{throw}\ e;\ e_2, s\rangle\ -\textit{Silent}\!\rightarrow \langle \texttt{throw}\ e, s\rangle$$

$$\textsc{CondT}\ P, w \vdash \langle \texttt{if}\ (\texttt{throw}\ e)\ e_1\ \texttt{else}\ e_2, s\rangle\ -\textit{Silent}\!\rightarrow \langle \texttt{throw}\ e, s\rangle$$

$$\textsc{ThrowT}\ P, w \vdash \langle \texttt{throw}\ \texttt{throw}\ e, s\rangle\ -\textit{Silent}\!\rightarrow \langle \texttt{throw}\ e, s\rangle$$

Figure 2.4: Exception propagation

Annexe C

Abstracts

C.1 Résumé en français

Dans cette thèse nous étudions l'isolation mémoire et les mesures de communications efficaces par passage de message dans le contexte des environnements à mémoire partagée et la programmation orientée-objets. L'état de l'art en la matière se base presque exclusivement sur deux techniques complémentaires dites de propriété des objets (ownership) et d'unicité de références (reference uniqueness) afin d'adresser les problèmes de sécurité dans les programmes concurrents. Il est frappant de constater que la grande majorité des travaux existants emploient des méthodes de vérification statique des programmes, qui requirent soit un effort d'annotations soit l'introduction de fortes contraintes sur la forme et les références vers messages échangés.

Notre contribution avec SIAAM est la démonstration d'une solution d'isolation réalisée uniquement à l'exécution et basée sur le modèle de programmation par acteurs. Cette solution purement dynamique ne nécessite ni annotations ni vérification statique des programmes. SIAAM permet la communication sans copie de messages de forme arbitraire. Nous présentons la sémantique formelle de SIAAM ainsi qu'une preuve d'isolation vérifiée avec l'assistant COQ. L'implantation du modèle de programmation pour le langage Java est réalisé dans la machine virtuelle JikesRVM. Enfin nous décrivons un ensemble d'analyses statiques qui réduit automatiquement le cout à l'exécution de notre approche.

C.2 English abstract

In this thesis we study state isolation and efficient message-passing in the context of concurrent object-oriented programming. The 'ownership' and 'reference uniqueness' techniques have been extensively employed to address concurrency safety in the past. Strikingly the vast majority of the previous works rely on a set of statically checkable typing rules, either requiring an annotation overhead or introducing strong restrictions on the shape and the aliasing of the exchanged messages.

Our contribution with SIAAM is the demonstration of a purely runtime, actor-based, annotation-free, aliasing-proof approach to concurrent state isolation allowing efficient communication of arbitrary objects graphs. We present the formal semantic of SIAAM, along with a machine-checked proof of isolation. An implementation of the model has been realized

in a state-of-the-art Java virtual-machine and a set of custom static analyses automatically reduce the runtime overhead.

Bibliography

[1] Mojave and the universe type system. In Peter Müller, editor, *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*, pages 39–76. Springer Berlin Heidelberg, 2002.

[2] Gul Agha. *Actors: a model of concurrent computation in distributed systems.* MIT Press, Cambridge, MA, USA, 1986.

[3] Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Galen Hunt, and James Larus. Deconstructing process isolation. In *Proceedings of the 2006 workshop on Memory system performance and correctness*, MSPC '06, pages 1–10, New York, NY, USA, 2006. ACM.

[4] Paulo Sérgio Almeida. Balloon types: Controlling sharing of state in data types. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP'97 — Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 32–59. Springer Berlin Heidelberg, 1997.

[5] Rita Z. Altucher and William Landi. An extended form of must alias analysis for dynamic allocation. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 74–84, New York, NY, USA, 1995. ACM.

[6] Andrew W. Appel. *Modern compiler implementation in Java.* Cambridge University Press, New York, NY, USA, 1998.

[7] Joe Armstrong. A history of erlang. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 6–1–6–26, New York, NY, USA, 2007. ACM.

[8] Joe Armstrong, Robert Virding, and Mike Williams. *Concurrent programming in ERLANG (2nd ed.).* Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996.

[9] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the jalapeño jvm. *SIGPLAN Not.*, 35(10):47–65, October 2000.

[10] Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter F. Sweeney. A survey of adaptive optimization in virtual machines. In *PROCEEDINGS OF THE IEEE, 93(2), 2005. SPECIAL ISSUE ON PROGRAM GENERATION, OPTIMIZATION, AND ADAPTATION*, 2004.

[11] Godmar Back, Wilson C. Hsieh, and Jay Lepreau. Processes in kaffeos: isolation, resource management, and sharing in java. In *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 23–23, Berkeley, CA, USA, 2000. USENIX Association.

[12] Henry G. Baker. "use-once" variables and linear objects: storage management, reflection and multi-threading. *SIGPLAN Not.*, 30(1):45–52, January 1995.

[13] Clemens Ballarin. Interpretation of locales in isabelle: Theories and proof contexts. In *MATHEMATICAL KNOWLEDGE MANAGEMENT (MKM 2006), LNAI 4108*, pages 31–43. Springer-Verlag, 2006.

[14] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 169–190, New York, NY, USA, October 2006. ACM Press.

[15] Stephen M. Blackburn and Kathryn S. McKinley. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. *SIGPLAN Not.*, 43(6):22–32, June 2008.

[16] Bruno Blanchet. Escape analysis for object-oriented languages: application to java. *SIGPLAN Not.*, 34(10):20–34, October 1999.

[17] Eric Bodden, Patrick Lam, and Laurie Hendren. Instance keys: A technique for sharpening whole-program pointer analyses with intraprocedural information. Technical report, 2007.

[18] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 241–250, New York, NY, USA, 2011. ACM.

[19] Jeff Bogda and Urs Hölzle. Removing unnecessary synchronization in java. *SIGPLAN Not.*, 34(10):35–46, October 1999.

[20] Michael Bond. Static cloning of library methods for application and vm contexts (patch). `http://sourceforge.net/p/jikesrvm/research-archive/39/`, April 2013.

[21] Grady Booch. *Object-oriented analysis and design with applications (2nd ed.)*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

[22] Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free java programs. *SIGPLAN Not.*, 36(11):56–69, October 2001.

[23] John Boyland. Alias killing: Unique variables without destructive reads. In Ana M. D. Moreira and Serge Demeyer, editors, *ECOOP Workshops*, volume 1743 of *Lecture Notes in Computer Science*, pages 148–149. Springer, 1999.

[24] John Tang Boyland and William Retert. Connecting effects and uniqueness with adoption. *SIGPLAN Not.*, 40(1):283–295, January 2005.

[25] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. Asm: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002.

[26] Ciarán Bryce and Chrislain Razafimahefa. An approach to safe object sharing. *SIGPLAN Not.*, 35(10):367–381, October 2000.

[27] S. Burris and H. P. Sankappanavar. *A Course in Universal Algebra*, volume 78 of *Graduate Texts in Mathematics*. Springer, New York, 1 edition, November 1981.

[28] Richard Carlsson, Konstantinos Sagonas, and Jesper Wilhelmsson. Message analysis for concurrent programs using message passing. *ACM Trans. Program. Lang. Syst.*, 28(4):715–746, July 2006.

[29] Byeong-Mo Chang, Jang-Wu Jo, Kwangkeun Yi, and Kwang-Moo Choe. Interprocedural exception analysis for java. In *Proceedings of the 2001 ACM symposium on Applied computing*, SAC '01, pages 620–625, New York, NY, USA, 2001. ACM.

[30] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for java. In *Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '99, pages 1–19, New York, NY, USA, 1999. ACM.

[31] Jong-Deok Choi, Manish Gupta, Mauricio J. Serrano, Vugranam C. Sreedhar, and Samuel P. Midkiff. Stack allocation and synchronization optimizations for java using escape analysis. *ACM Trans. Program. Lang. Syst.*, 25(6):876–910, November 2003.

[32] Dave Clarke and Tobias Wrigstad. External uniqueness is unique enough. In *In European Conference for Object-Oriented Programming (ECOOP)*, pages 176–200. Springer-Verlag, 2003.

[33] Dave Clarke, Tobias Wrigstad, Johan Östlund, and Einar Broch Johnsen. Minimal ownership for active objects. In *Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, APLAS '08, pages 139–154, Berlin, Heidelberg, 2008. Springer-Verlag.

[34] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *In Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 48–64. ACM Press, 1998.

[35] David Gerard Clarke. *Object ownership and containment*. PhD thesis, New South Wales, Australia, Australia, 2003. AAI0806678.

[36] B. Claudel. *Mécanismes logiciels de protection mémoire*. 2009.

[37] Benoît Claudel, Fabienne Boyer, Noel De palma, and Olivier Gruber. Message Passing: A Case for Mixing Deep-Copy and Migration. Research Report RR-LIG-029, LIG, Grenoble, France, 2012.

[38] William D Clinger. Foundations of actor semantics. Technical report, Cambridge, MA, USA, 1981.

[39] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '79, pages 269–282, New York, NY, USA, 1979. ACM.

[40] Grzegorz Czajkowski and Laurent Daynàs. Multitasking without compromise: a virtual machine evolution. *SIGPLAN Not.*, 47(4a):60–73, March 2012.

[41] Grzegorz Czajkowski and Laurent Daynés. Multitasking without comprimise: a virtual machine evolution. *SIGPLAN Not.*, 36(11):125–138, October 2001.

[42] B.A. Davey and H.A. Priestley. *Introduction to Lattices and Order*. Cambridge mathematical text books. Cambridge University Press, 2002.

[43] Tamar Domani, Gal Goldshtein, Elliot K. Kolodner, Ethan Lewis, Erez Petrank, and Dafna Sheinwald. Thread-local heaps for java. *SIGPLAN Not.*, 38(2 supplement):76–87, June 2002.

[44] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in singularity os. *SIGOPS Oper. Syst. Rev.*, 40(4):177–190, April 2006.

[45] Cormac Flanagan and Stephen N. Freund. Redcard: Redundant check elimination for dynamic race detectors. In Giuseppe Castagna, editor, *ECOOP*, volume 7920 of *Lecture Notes in Computer Science*, pages 255–280. Springer, 2013.

[46] Robin J. Garner, Stephen M. Blackburn, and Daniel Frampton. A comprehensive evaluation of object scanning techniques. *SIGPLAN Not.*, 46(11):33–42, June 2011.

[47] Nicolas Geoffray, Gaël Thomas, Julia Lawall, Gilles Muller, and Bertil Folliot. Vmkit: a substrate for managed runtime environments. *SIGPLAN Not.*, 45(7):51–62, March 2010.

[48] Brian Goetz. Java theory and practice: Safe construction techniques. `http://www.ibm.com/developerworks/java/library/j-jtp0618/index.html`, June 2002.

[49] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1996.

[50] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. *SIGPLAN Not.*, 32(10):108–124, October 1997.

[51] Olivier Gruber and Fabienne Boyer. Ownership-based isolation for concurrent actors on multi-core machines. In Giuseppe Castagna, editor, *ECOOP 2013*, volume 7920, pages 281–301. Springer, July 2013.

[52] Niklas Gustafsson. Isolation in Maestro. `http://blogs.msdn.com/b/maestroteam/archive/2009/02/27/isolation-in-maestro.aspx`, 2013.

[53] Philipp Haller and Martin Odersky. Capabilities for uniqueness and borrowing. In *Proceedings of the 24th European conference on Object-oriented programming*, ECOOP'10, pages 354–378, Berlin, Heidelberg, 2010. Springer-Verlag.

[54] Chris Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski, Deyu Hu, and Thorsten von Eicken. Implementing multiple protection domains in java. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '98, pages 22–22, Berkeley, CA, USA, 1998. USENIX Association.

[55] Kim Hazelwood and David Grove. Adaptive online context-sensitive inlining. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '03, pages 253–264, Washington, DC, USA, 2003. IEEE Computer Society.

[56] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[57] Michael Hind. Pointer analysis: haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '01, pages 54–61, New York, NY, USA, 2001. ACM.

[58] John Hogg. Islands: aliasing protection in object-oriented languages. *SIGPLAN Not.*, 26(11):271–285, November 1991.

[59] Galen C. Hunt and James R. Larus. Singularity: rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, April 2007.

[60] Google Inc. The dart language. `http://www.dartlang.org`.

[61] Rajesh K. Karmani, Amin Shali, and Gul Agha. Actor frameworks for the jvm platform: a comparative analysis. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, PPPJ '09, pages 11–20, New York, NY, USA, 2009. ACM.

[62] Gerwin Klein and Tobias Nipkow. Jinja is not java. *Archive of Formal Proofs*, June 2005. http://afp.sf.net/entries/Jinja.shtml, Formal proof development.

[63] Gerwin Klein and Tobias Nipkow. A machine-checked model for a java-like language, virtual machine, and compiler. *ACM Trans. Program. Lang. Syst.*, 28(4):619–695, July 2006.

[64] Patrick Lam, Eric Bodden, Laurie Hendren, and Technische Universität Darmstadt. The soot framework for java program analysis: a retrospective. http://plg.uwaterloo.ca/~olhotak/pubs/cetus.pdf.

[65] William Alexander Landi. Interprocedural aliasing in the presence of pointers. Technical report, 1992.

[66] Christopher Lapkowski and Laurie J. Hendren. Extended ssa numbering: introducing ssa properties to languages with multi-level pointers. In *Proceedings of the 1996 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '96, pages 23–. IBM Press, 1996.

[67] Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, May 2006.

[68] K. Rustan Leino, Peter Müller, and Angela Wallenburg. Flexible immutability with frozen objects. In *Proceedings of the 2nd international conference on Verified Software: Theories, Tools, Experiments*, VSTTE '08, pages 192–208, Berlin, Heidelberg, 2008. Springer-Verlag.

[69] Tim Lindholm and William Pugh. Java(tm) memory model and thread specification revision. http://jcp.org/en/jsr/detail?id=133, 2013.

[70] Benjamin Livshits, John Whaley, and Monica S. Lam. Reflection analysis for java. In *Proceedings of the Third Asian conference on Programming Languages and Systems*, APLAS'05, pages 139–160, Berlin, Heidelberg, 2005. Springer-Verlag.

[71] Andreas Lochbihler. *A Machine-Checked, Type-Safe Model of Java Concurrency : Language, Virtual Machine, Memory Model, and Verified Compiler.* PhD thesis, Karlsruher Institut für Technologie, Fakultät für Informatik, July 2012.

[72] Jean-Phillipe Martin, Michael Hicks, Manuel Costa, Periklis Akritidis, and Miguel Castro. Dynamically checking ownership policies in concurrent c/c++ programs. *SIGPLAN Not.*, 45(1):457–470, January 2010.

[73] Apache Software Foundation. Apache CouchDB: relax. http://couchdb.apache.org/, 2013.

[74] Java Community Process. JSR 121: Application Isolation API Specification. http://www.jcp.org/en/jsr/detail?id=121, 2013.

[75] Joyent Inc. Node.js. http://nodejs.org/, 2013.

[76] The Coq development team. *The Coq proof assistant reference manual.* LogiCal Project, 2004. Version 8.0.

[77] The Eclipse Foundation. Eclipse IDE. http://www.eclipse.org, 2013.

[78] The OSGi Alliance. The OSGi Architecture. http://www.osgi.org/Technology/WhatIsOSGi, 2013.

[79] Naftaly H. Minsky. Towards alias-free pointers. In *Proceedings of the 10th European Conference on Object-Oriented Programming*, ECCOP '96, pages 189–209, London, UK, UK, 1996. Springer-Verlag.

[80] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. *SIGPLAN Not.*, 41(6):308–319, June 2006.

[81] Stas Negara, Rajesh K. Karmani, and Gul Agha. Inferring ownership transfer for efficient message passing. *SIGPLAN Not.*, 46(8):81–90, February 2011.

[82] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[83] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, ECCOP '98, pages 158–185, London, UK, UK, 1998. Springer-Verlag.

[84] Krzysztof Palacz, Jan Vitek, Grzegorz Czajkowski, and Laurent Daynas. Incommunicado: efficient communication for isolates. *SIGPLAN Not.*, 37(11):262–274, November 2002.

[85] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, December 1972.

[86] Pierre Parrend and Stéphane Frénot. Java components vulnerabilities - an experimental classification targeted at the osgi platform. *CoRR*, abs/0706.3812, 2007.

[87] The Netty Project. Netty. `http://netty.io`.

[88] Alexandru Salcianu and Martin Rinard. Pointer and escape analysis for multithreaded programs. *SIGPLAN Not.*, 36(7):12–23, June 2001.

[89] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: understanding object-sensitivity. *SIGPLAN Not.*, 46(1):17–30, January 2011.

[90] Amie L. Souter and Lori L. Pollock. Incremental call graph reanalysis for object-oriented software maintenance. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, ICSM '01, pages 682–, Washington, DC, USA, 2001. IEEE Computer Society.

[91] Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-typed actors for java. In *Proceedings of the 22nd European conference on Object-Oriented Programming*, ECOOP '08, pages 104–128, Berlin, Heidelberg, 2008. Springer-Verlag.

[92] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM.

[93] John Whaley and Martin Rinard. Compositional pointer and escape analysis for java programs. *SIGPLAN Not.*, 34(10):187–206, October 1999.

[94] Xi Yang, Stephen M. Blackburn, Daniel Frampton, and Antony L. Hosking. Barriers reconsidered, friendlier still! *SIGPLAN Not.*, 47(11):37–48, June 2012.