Iowa State University
Digital Repository

Graduate Theses and Dissertations

Iowa State University Capstones, Theses and Dissertations

2016

Toward a Concurrent Programming Model with Modular Reasoning

Mehdi Bagherzadeh
Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/etd

Part of the Computer Sciences Commons

Recommended Citation

Bagherzadeh, Mehdi, "Toward a Concurrent Programming Model with Modular Reasoning" (2016). *Graduate Theses and Dissertations*. 15659.
https://lib.dr.iastate.edu/etd/15659

This Dissertation is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.

Toward a concurrent programming model with modular reasoning

by

Mehdi Bagherzadeh

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Science

Program of Study Committee:

Hridesh Rajan, Major Professor

Samik Basu

Vasant G. Honavar

Gary T. Leavens

Robyn R. Lutz

Iowa State University

Ames, Iowa

2016

Copyright © Mehdi Bagherzadeh, 2016. All rights reserved.

DEDICATION

To my parents Akram and Javad.

TABLE OF CONTENTS

LIST OF TABLES vii

LIST OF FIGURES viii

ACKNOWLEDGEMENTS x

ABSTRACT xi

CHAPTER 1. INTRODUCTION 1

1.1 Contributions 3

1.2 Outline 4

CHAPTER 2. OVERVIEW OF RELATED WORK 6

2.1 Programming Model 6

2.2 Modular Reasoning 8

2.3 Interference Control Framework 8

2.4 Concurrent Behavioral Subtyping 9

2.5 Order Types 10

CHAPTER 3. INTERFERENCE CONTROL FRAMEWORK 12

3.1 Problem 12

3.2 Solution: Panini 14

3.2.1 Sparse Interference to Solve Pervasive Interference 15

3.2.2 Cognizant Interference to Solve Oblivious Interference 18

3.2.3 Modular Reasoning Using Panini's Interference Model 19

3.2.4 Contributions 19

3.3 Panini's Syntax 20

3.4 Operational Semantics 21

3.4.1 Dynamic Objects 22

3.4.2 Local and Global Semantics 24

3.4.3 Sequential Synchronous Local Semantics 24

3.4.4 Concurrent Asynchronous Global Semantics 27

3.4.5 Ownership Transfer Semantics 29

3.4.6 Exceptional Semantics 30

3.4.7 Initial Configuration 31

3.4.8 Actions: Conflict and Happens-Before Relations 34

3.4.9 Sharing of Capsule Instances and Futures 37

3.5 Sparse and Cognizant Interference 39

3.5.1 Sparse Interference 40

3.5.2 Cognizant Interference 41

3.6 Hoare-style Modular Reasoning 44

3.7 Static Semantics 47

3.7.1 Type Attributes 47

3.7.2 Typing Rules 48

3.8 Discussion 50

3.9 Related Work 51

CHAPTER 4. CONCURRENT BEHAVIORAL SUBTYPING 54

4.1 Background 55

4.1.1 Panini 56

4.1.2 Program Syntax 56

4.1.3 Semantics in a Nutshell 58

4.2 Key Challenges 60

4.2.1 Problem ❶: Interference Behavior Incompatibility 61

4.2.2 Modular Reasoning and Interference 61

4.2.3 Interference Behavior Incompatibility 64

4.2.4 Problem ❷: Incompatible Interface Behavior 65
4.3 Concurrent Subtyping 67
4.3.1 Formalization of Concurrent Subtyping 67
4.3.2 Sound Modular Reasoning 71
4.4 Proving Concurrent Subtyping 74
4.4.1 Encapsulated Inheritance 75
4.4.2 Proving Interference Subtyping 75
4.4.3 Proving Concurrent Subtyping 77
4.5 Semantics 77
4.5.1 Static Semantics 77
4.5.2 Operational Semantics 77
4.6 Evaluation 80
4.6.1 Akka 82
4.6.2 Java 83
4.6.3 Type Encapsulation 83
4.6.4 Findings 83
4.7 Related Work 85
CHAPTER 5. ORDER TYPES 86
5.1 Motivation 88
5.1.1 Modular Message Race Detection 88
5.1.2 Abstraction and Abstraction-eligibility 92
5.1.3 Proper Blame Assignment 95
5.2 Process Model and Program Syntax 97
5.2.1 Program Syntax 97
5.3 Order Type Syntax 99
5.3.1 Type Attributes 100
5.4 Type System 100
5.4.1 Declarations 102
5.4.2 Invocations 102

5.4.3 Composition and Blame Assignment 104
5.4.4 Abstraction 107
5.4.5 Soundness of Abstraction 109
5.4.6 Discussion 109
5.5 Related Work 110

CHAPTER 6. CONCLUSION AND FUTURE WORK 112
6.1 Interference Control Framework 112
6.2 Concurrent Behavioral Subtyping 113
6.3 Order Types 113
6.4 Future Work 114
6.4.1 Modular Reasoning about Traditionally Global Properties 114
6.4.2 Concurrent Behavioral Subtyping and Hoare-style Reasoning 115
6.4.3 Interference Closure and Interference Behavior 115
6.4.4 Applicability 116

BIBLIOGRAPHY 117

LIST OF TABLES

Table 3.1 Previous work addressing (✓) pervasive and oblivious interference. 13

Table 4.1 Number of supertypes accessing their supertypes' states. 80

LIST OF FIGURES

Figure 3.1 Capsule Counter with state x and procedure add. 15

Figure 3.2 Capsule Counter with a reference type state. 16

Figure 3.3 Capsule Client with JML-like assertion Φ on line 7. 17

Figure 3.4 Panini's core syntax, based on [138]. 20

Figure 3.5 Design and wiring declarations in capsule Main. 22

Figure 3.6 Added syntax, evaluation contexts, configurations and actions in semantics. . 23

Figure 3.7 Local and global operational semantics of Panini. 25

Figure 3.8 Panini's auxiliary functions. 31

Figure 3.9 Exceptional semantics of Panini, select rules. 32

Figure 3.10 Rules to create initial configuration of Panini programs. 33

Figure 3.11 Conflicting actions and their happens-before $\triangleright$ relations. 35

Figure 3.12 Sharing counter among client1 and client2. 37

Figure 3.13 Static verification of the behavioral contract of test. 45

Figure 3.14 Type attributes 48

Figure 3.15 Panini's select typing rules. 49

Figure 4.1 Panini's core syntax with capsule inheritance and procedure specifications. . 57

Figure 4.2 Interference points $^{\alpha}$ and interference closure $\kappa(c, Counter)$. 60

Figure 4.3 Interference behavior incompatibility breaks modular reasoning. 64

Figure 4.4 Interface behavior incompatibility breaks modular reasoning. 65

Figure 4.5 Incompatible interface behavior breaks sequential modular reasoning. 66

Figure 4.6 CxCounter is an interference subtype of Counter. 70

Figure 4.7 Select rules for Panini's Hoare logic. 72

Figure 4.8 Panini's select typing rules. 78

Figure 4.9 Local and global operational semantics with capsule inheritance. 79

Figure 5.1 Well-formed process Server with empty order types for its procedures. 88

Figure 5.2 Well-formed process Proxy with existential quantification for sv in its type. . 89

Figure 5.3 Ill-formed process Client causes a message race in Server. 90

Figure 5.4 Nondeterministic messages save and kill cause a race. 91

Figure 5.5 Abstraction of messaging behavior for local logger l. 92

Figure 5.6 Naive order type abstraction in Client1. 94

Figure 5.7 Unsound abstraction of the tainted local process instance r. 94

Figure 5.8 A message composition that could lead to a race with bad composition. 96

Figure 5.9 Bad process composition in User and bad message composition in Client1. . . 96

Figure 5.10 Core process syntax, inspired by [14, 18, 86]. 98

Figure 5.11 Syntax of order types. 99

Figure 5.12 Type attributes. 100

Figure 5.13 Select type checking rules for order types. 101

Figure 5.14 Rest of type checking rules. 102

Figure 5.15 Auxiliary functions. 103

Figure 5.16 Happens-before rules for direct and transitive inorder delivery and processing. 105

Figure 5.17 Function *copy* for constraint set, inspired by copy rule [114]. 106

Figure 5.18 Typing rules for abstraction-eligibility and abstraction 108

Figure 5.19 Abstraction function *hide* and taint function *tainted*. 108

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor Hridesh Rajan for his extraordinary advice, support and encouragement throughout my entire graduate studies. I aspire to one day be as excellent an advisor as him. I would like to thank Gary T. Leavens for great research collaborations and advice, just like a second advisor. My special thanks goes to my other committee members Samik Basu, Vasant G. Honavar and Robyn R. Lutz.

In addition to the work presented in this thesis, I have been involved in the work on *translucid contracts* [17, 20, 21] to enable modular reasoning about functional and flow behaviors of event-based systems for both normal and exceptional executions [19] in the presence of event subtyping [15, 16, 50, 53, 54]. Translucid contracts build on the Ptolemy language [140–142] which supports explicit announcement and handling of typed events and allows for event type polymorphism [60]. I have also been involved in the work on semantic characterization of message order problems in concurrent message passing software [105] and the work on contract modularization using *AspectJML* [145, 146]. My thank goes to Ali Darvish, Robert Dyer, Rex D. Fernando, Eric Lin, Yuheng Long, Sean L. Mooney, Henrique Rebêlo, Josè Sànchez, Ganesha Upadhyaya and all my other collaborators for their great collaborations on these works during Panini, Ptolemy and AspectJML projects.

My research during my graduate studies including the research in this thesis was supported in part by grants from the US National Science Foundation (NSF) under grants CCF-08-46059, CCF-10-17334, CCF-11-17937, CCF-14-23370 and CCF-15-18897.

I also like to thank Gogul Balakrishnan, Franjo Ivanŏić and Aarti Gupta for the research internship opportunity at NEC Laboratories of America.

There are many other people and organizations who have helped me over the years, and while your names may not be on this list I extend my gratitude to you as well.

Parts of Chapter 1 and Chapter 2 and most of Chapter 3 is based on our previous work published in MODULARITY 2015 [18].

ABSTRACT

Modular reasoning and concurrent programming are both necessary for scalable development of performant software. Modular reasoning improves scalability by allowing a program to be understood one module at a time. Concurrent programming improves the performance by allowing simultaneous executions of multiple computations in a single program. However, modular reasoning about a concurrent program is difficult because of its thread interference, module inheritance and nondeterministic message orders. The statement of this thesis is that *there exists a concurrent programming model that enables modular reasoning about behaviors of its programs in the presence of interference, inheritance and nondeterministic message orders* using the following three ideas.

The first idea is an *interference control framework* that enables modular reasoning in the presence of interference. The technical innovations of the interference control framework are its *sparse interference* and *cognizant interference* properties that allow for standard Hoare-style modular reasoning about a concurrent program. Sparse and cognizant interference guarantee that interference happens only at explicitly specified program points and the interference behavior is statically known, respectively.

The second idea is *concurrent behavioral subtyping* that enables modular reasoning in the presence of inheritance. The technical innovations of concurrent behavioral subtyping are a new definition of behavioral subtyping for a concurrent program in terms of standard *interface subtyping* and a novel *interference subtyping* and show that in the presence of *encapsulated inheritance* the interface subtyping is sufficient to guarantee concurrent behavioral subtyping.

The third and last idea is *order types* that enables modular reasoning in the presence of nondeterministic message orders. The technical innovations of order types are to disallow message races using *existential types* that capture unknown module dependencies, *abstraction* that hides local messaging behavior of the module in its order type and a *blame assignment* that properly blames the module with bad expression composition or bad module composition and not the module in which the race happens.

These three ideas have the potential to ease software engineering of concurrent systems by improving a developer's ability to design, implement, test and evolve their software one module at a time.

CHAPTER 1. INTRODUCTION

The concurrency revolution [158] and the need for performant software requires sequentially-trained programmers [108] to write concurrent programs. Modular reasoning [132] allows programmers to understand their programs one module at a time and in a scalable manner. This is because in modular reasoning a module is understood using its implementation and static types of other modules it refers to, independent of their dynamic types [97] and implementations. Modular reasoning is also scalable because any changes in the dynamic type or implementation of a referred module cannot invalidate an already verified property of a module that refers to it. However, modular reasoning about a concurrent program can become difficult and sometimes intractable because of the interference among its threads [67], inheritance among its modules [97] and nondeterminism in orders of its messages [162].

Interference Two modules interfere if during their concurrent execution, interleaving instructions of one module changes the program state and consequently the behavior of the other module [177]. Interference makes modular reasoning difficult because the interference can happen between any two instructions of a module and at each interference point the interference behavior is unknown. To understand a module M the programmer should consider the possibility of interference between any two instructions of the module and for each interference point carry out a global reasoning to know its interference behavior, which is a lot of non-modular reasoning work. Any changes in any module in the program can invalidate a previously verified property of the module M.

To illustrate, consider the single line of code below that sums up the values of variables x and y. There could be three interference points that occur between instructions that read the value of x, read the value of y, add the values of x and y and write the result to x.

```
x = x + y;
```

That is, this code actually looks like $^{\alpha}x = {}^{\alpha}x + {}^{\alpha}y$ where $^{\alpha}$ denotes an interference point. And there is no information about how interfering tasks at interference points $^{\alpha}$ may or may not modify the

values of x and y. To understand this single line of code with only four instructions the programmer should consider three interference points and for each interference point carry out a global reasoning to know how interference task may modify the values of x and y, which is a lot of work. Any changes in any part of the program may invalidate this understanding and require the reasoning to be redone.

Inheritance Inheritance and dynamic dispatch make modular reasoning difficult because behaviors of modules in an inheritance hierarchy can be different. To understand an invocation in a module M, the programmer should consider the receiver module and carry out a global reasoning over its inheritance hierarchy to understand modules that inherit from it, which is a lot of non-modular reasoning work. Any changes in any module in the inheritance hierarchy of the receiver can invalidate a previously verified property of the module M.

To illustrate, consider the single line of code below that invokes a procedure save of a proxy module r with a static type Proxy.

```
r.save();
```

To understand this invocation, a programmer should understand the behavior of the procedure save in Proxy and any other module in its inheritance hierarchy that may override save, which is a lot of work. Any changes in the inheritance hierarchy of Proxy can invalidate this understanding and require the reasoning to be done.

Nondeterministic message orders Nondeterministic message orders make modular reasoning intractable because nondeterminism can lead to racy messages that can arrive at any arbitrary order. To understand a module M, the programmer should consider all messages that the module sends and understand their orderings. This problem becomes exponential with message races. Any changes in the message ordering of any other module can invalidate a previously verified property of M.

To illustrate, consider the single line of code below in which the programmer intends to save the client's work in a server module before shutting down the server. This code is a simplified version of a real world bug found in previous work [93, 161].

```
client :          r.save(); s. kill ();
```

The client first sends a save message to a proxy module r. The proxy saves the client's work in a server s by sending a save message to the server, after performing some bookkeeping. The proxy implements

the *Proxy* design pattern [71] and relays the client's messages to the server. Then the client sends a kill message to the server s which shuts down the server. A procedure invocation emulates a message send [14,86]. Due to asynchrony of message sends and concurrent execution of client, proxy and server, messages save and kill sent from client can arrive at the server at any order and with no happens-before order [91] between them. That is, save and kill messages are racy and save can arrive at the server before kill or vice versa. To understand this single line of code, a programmer should consider both (i.e. *2!*) message orders. In general, for a message race involving n messages there are $n!$ message orderings that should be understood, which is intractable [161, 162].

1.1 Contributions

The idea in this thesis is that there exists a concurrent programming model that enables modular reasoning about behaviors of its programs in the presence of interference, inheritance and nondeterminism. The thesis proposes a concurrent programming model with interference control, concurrent behavioral subtyping and order types to enable modular reasoning.

Interference control framework Interference control framework enables Hoare-style modular reasoning [77] in the presence of interference by guaranteeing that interference happens only at explicitly specified program points in a module and the interference behaviors are statically and modularly known. A programmer can understand a module M using only its implementation and the static types of other modules it refers to, independent of their implementations.

Concurrent behavioral subtyping Concurrent behavioral subtyping enables supertype abstraction modular reasoning [97] in the presence of inheritance by guaranteeing that the behavior of a module does not violate the behavior of a module that it inherits from. The programmer can understand the module M using only its implementation and the static types of other modules it refers to, independent of their dynamic types and inheritance hierarchies.

Order types Order types enables tractable modular reasoning in the presence of nondeterministic message orders by disallowing message races. The programmer can understand the messaging behavior of the module M using only its implementation and the abstracted order types of modules it refers to, independent of their implementations or non-abstracted types.

1.2 Outline

Chapter 2 presents an overview of the related work on the proposed message passing programming model in Section 2.1, modular reasoning in Section 2.2, interference control framework in Section 2.3, concurrent behavioral subtyping in Section 2.4 and order types in Section 2.5.

Chapter 3 discusses and formalizes the interference control framework. Section 3.1 illustrates two problems of pervasive interference and oblivious interference that make modular reasoning difficult in the presence of interference. Section 3.2 informally presents the interference control framework and its two properties sparse and cognizant interference. Section 3.3 defines the core syntax of the proposed message passing calclulus and model without inheritance and its operational semantics discussed in Section 3.4. Section 3.5 defines formalizes and proves sparse and cognizant interference properties. Section 3.6 shows how these two properties allow for the use of sequential Hoare-style modular reasoning [77]. Section 3.7 discusses the static semantics and typing rules of the model. Section 3.8 discusses expressiveness of the proposed model and granularity of interference behaviors. Section 3.9 presents a detailed discussion of related work to interference control.

Chapter 4 discusses and formalizes concurrent behavioral subtyping. Section 4.1 presents the core syntax and semantics of of the proposed message passing model with inheritance added. Section 4.2 illustrates two problems of incompatible interface behavior and incompatible interference behavior that make modular reasoning difficult in the presence of inheritance. Section 4.3 proposes and formalizes concurrent behavioral subtyping as a combination of standard interface subtyping and new interference subtyping and presents Hoare-style supertype abstraction reasoning [97] using concurrent subtyping. Section 4.4 shows reduction of concurrent behavioral subtyping to standard interface subtyping in the presence of encapsulated inheritance [155]. Section 4.5 discusses the static and operational sematnics of our calculus with inheritance added. Section 4.6 emprically evaluates the usefulness of concurrent subtyping and encapsulated inheritance on a large set of Java and Akka [8] projects. Section 4.7 presents a detailed discussion of related work of concurrent behavioral subtyping.

Chapter 5 discusses and formalizes order types. Section 5.1 motivates the use of existential order types, order type abstraction and proper blame assignment. Section 5.2 briefly presents the core calculus and its semantics. Section 5.3 presents the core syntax of order types. Section 5.4 formalizes the

type checking and well-formedness rules for order types, order type abstraction and abstraction-eligible modules and blame assignment. It also discusses the over-approximation in the order type of a conditional expression to avoid state-space explosion, use of memory effects for a more precise race detection and avoiding of benign races. Section 5.5 presents a detailed discussion of related work to order types.

Chapter 6 discusses avenues of future work and concludes.

CHAPTER 2. OVERVIEW OF RELATED WORK

2.1 Programming Model

This thesis uses and builds on capsule-oriented programming model, an implicit concurrent message passing programming model with capsules [135–138] as concurrent abstractions. In a message passing model, a program is a set of concurrent abstractions that run concurrently and communicate by sending and receiving messages. A concurrent programming model can be explicit, implicit or automatic. In an explicit model, a programmer manually creates concurrent tasks and manages their synchronizations. In an implicit model, the programmer uses annotations or programming constructs that provide hints to the compiler for creation and synchronization of concurrent task. In an automatic model, the compiler itself is responsible for finding and exposing concurrency.

A capsule [18, 136–138] is a concurrent abstraction with mutable state [172], ownership [38], encapsulation [117], inheritance with behavioral subtyping [104], typed messages [86] and sequential execution [35]. A capsule unites its declaration and the composition of the capsules it owns [113]. Capsule-oriented programming is implemented [100] in the language Panini with a large set of multithreaded and actor programs translated into Panini [100, 105] that show comparable performances [138]. Various compilation strategies along with optimal mappings of capsules to JVM threads using their communication and computation behaviors are also studied [167, 168]. Capsules are closely related to and build on previous programming models and abstractions including processes in process calculi such as communicating sequential processes (CSP) [78] and calculus of communicating systems (CCS) [111], actors [6], active objects [94, 102, 125] and ambients [42].

A CSP [78] process is a sequential entity composed from atomic actions that synchronously communicates with another concurrently running process by sending and receiving messages over a communication channel. CCS [111] is similar but adds first class channels that can be passed between two

process over a shared channel and allow for dynamic changes in the topology of the system. An actor [6, 76] encapsulates its mutable state and a thread of execution and asynchronously communicates with another concurrently running actor by sending and receiving messages but without channels. When receiving a message an actor may update its state, change its behavior, send messages or create more actors. Variations of the actor model allow for sharing among actors [8], arbitrary actor inheritance [8, 14], concurrent execution inside the actor [152], typed messages [14] or ownership of the actor state [118]. Using typed messages, an actor defines a set of procedures [8, 14] invoked by other actors. A procedure invocation emulates a type-safe message send [86]. Using ownership an actor owns its state and control access to it. Ownership can be guaranteed using ownership types [37] or analysis [118]. An ambient [42] is an actor specialized for mobile computing with support for handling of consistency issues in a distributed network with failures and connectivity losses. An active object [94] is an object that encapsulates a thread of execution and communicates with another concurrently running active object by invoking their methods. A method invocation emulates a message send [86]. Variations of active objects allow for immutable states [151], ownership of states [37], cooperative concurrent execution inside the object [151] or grouping of active object in a cobox and restricting mutable access to them [151]. Mutable state allows for readonly sharing among objects. Cooperative concurrent execution guarantees that the state of the object is not simultaneously accessed by more than one thread. The execution of a thread must explicitly yield before the execution of another thread can start.

Ownership [38], encapsulation [117], behavioral subtyping [104], typed messages [86] and sequential execution [35] of a capsule enable modular reasoning in a concurrent program in the presence of interference, inheritance and nondeterministic message orders. Different concurrent abstractions in previous work propose various combinations of these properties but not a combination of all of them in a single abstraction. Arbitrary sharing in the absence of ownership or encapsulation, arbitrary inheritance without behavioral subtyping, untyped messages, and unsynchronized concurrent execution inside capsules can break modular reasoning guarantees of the capsule-oriented programming model and make modular reasoning about a concurrent program difficult or intractable again.

2.2 Modular Reasoning

In modular reasoning, a program is understood one module at a time using only its implementation and static types of other modules it refers to, independent of their dynamic types or implementations [95,132]. To the best of our knowledge this thesis proposes the first concurrent programming model that enables modular reasoning in the combined presence of interference, inheritance and nondeterministic message orders. Modular reasoning is enabled using an interference control framework, concurrent behavioral subtyping and order types each of which build on a rich body of previous work discussed in the remaining of this chapter.

2.3 Interference Control Framework

Two modules interfere if during their concurrent execution, interleaving instructions of one module changes the program state and consequently the behavior of the other module [177]. Interference makes modular reasoning difficult because interference can happen between any two instructions of a module and at each interference point the interference behavior is unknown. Interference control framework enables modular reasoning by making interference points and interference behaviors known syntactically, statically and modularly. Previous work either controls interference points [2, 6, 7, 84, 90, 94, 151, 154, 157, 172, 176, 177, 177] or interference behaviors [56, 68, 126, 129] but not both. Actors [6, 7, 172], active objects [94, 151], atomicity [58, 66, 171], cooperability [157, 176, 177] and automatic mutual exclusion [2, 84, 154] control interference points but not interference behaviors. Rely-guarantee reasoning [87], the work of Owicki-Gries [129] and its variations [126], concurrent separation logic [59, 127] and concurrent abstract predicates [47, 159] control the interference behavior but not the interference points.

An actor [6, 76] or an active object [94] with encapsulation and ownership guarantees a macro-step semantics [4] in which interference can only happen after message sends or receives and the rest of the implementation is atomic and free from interference. An atomic block is a block of code with the guarantee that interference cannot happen inside the block. That is, interference can happen anywhere except inside atomic blocks. Automatic mutual exclusion [84] is the inverse of atomic blocks. That is, interference can happen only at explicitly specified program points and anywhere else is atomic. Co-

operability [177] is similar with the guarantee that interference happens only at program points marked with a special expression `yield`. These technique control interference points by limiting them to explicitly specified program points but not the interference behaviors. That is, interference behavior at an interference point is not known and requires a global reasoning to know about the interference behavior.

In rely-guarantee reasoning [87] and the work of Owicki-Gries [129] a module provides a guarantee and relies on guarantees provided by other modules in its environment. Environment specification of a module is an upper bound on the interfering behavior of other modules in its environment. Concurrent separation logic [127], concurrent abstract predicates (CAP) [47] and impredicative concurrent abstract predicates (iCAP) [159] limit sharing to modules called resources. A resource can have an invariant which should hold at any program state. Interfering behavior of other modules on the resource must not invalidate its invariant. These techniques control the interference behavior but not the interference points. That is, interference can happen between any two instructions of a module.

2.4 Concurrent Behavioral Subtyping

In modular supertype abstraction reasoning [97], an invocation is understood using the static type of its receiver module. Inheritance can break modular reasoning because an invocation can be dynamically dispatched to a module or any modules inheriting from it in its inheritance hierarchy where behaviors of these modules are incompatible. Concurrent behavioral subtyping allows for modular reasoning about an invocation by guaranteeing that the behavior of a module or any of the modules that inherit from it are compatible [130]. Concurrent behavioral subtyping defines interference subtyping for a concurrent programs and combines it with standard interface subtyping for a sequential program [36,49,96,130] to provide a new definition for behavioral compatibility in a concurrent program. Concurrent behavioral subtyping is complementary to previous work that extend sequential interface subtyping for concurrent programs [9,45,47,48,67,87,88,95,104,117,127,148,165]. Use of encapsulated inheritance [155] to reduce concurrent behavioral subtyping to interface subtyping is new.

The functional behavior of a procedure of module can be described using its precondition and postcondition which are predicates on the program state that must hold before and after invocations of the procedure. Sequential behavioral subtyping guarantees [10] that the behavior of a module does not

violate the behavior of the module it inherits from by relating their preconditions and postconditions by requiring a contravariance and covariance relations between preconditions and postconditions respectively. Weak behavioral subtyping [44, 96] for sequential programs weaken the covariance requirement for postconditions. However, sequential behavioral subtyping cannot accommodate interference among concurrently running modules and therefore is not directly applicable to a concurrent program [67, 148].

Strong behavioral subtyping [104] uses explicit history constraints to accommodate interference. A history constraint is a predicate on two transitively consecutive program states [95]. To guarantee behavioral subtyping between two modules of a concurrent program, not only their preconditions and postcondition but also their history constraints should be related. A variation of strong behavioral subtyping without history constraints [104] requires that every procedure that is added to an inheriting module and modifies its state should be expressible using the procedures of the inherited module. Weak behavioral subtyping [44] for concurrent programs weakens this requirement by controlling cross-type aliasing between two modules in an inheritance hierarchy and allows for mutable subtypes. Concurrent JML [148] and Contracts for concurrency [124] exploit the semantic equivalence for executions of sequential and atomic code to allow for the use of sequential behavioral subtyping for atomic modules. Previous work integrates behavioral subtyping into concurrent programming languages and tools such as Eiffel [110, 124], Java [63, 96], JML [95], ESC/Java [69] and Spec# [22]. However, previous work cannot separate the interference and interface behavior of a module and cannot compute its interference behavior using its preconditions and postconditions.

2.5 Order Types

Nondeterministic message orders can cause message races [161] and make modular reasoning about a concurrent message passing program intractable. Order types disallow message races and make modular reasoning about message orders tractable by avoiding state-space explosion [93, 161, 162]. Order type abstraction that hides the local messaging behavior of an abstraction-eligible module is new. Similar to previous work [62, 173], the blame assignment lays the blame with the module that is the supplier of bad values and not the module that uses the bad value. Local descriptive order types complement previous work on global prescriptive behavioral types and specially session types

[25, 26, 28, 40, 43, 79–81, 116, 119, 120, 160, 178].

Message orders determine the behavior of a message passing program [93, 161] and should be understood to understand a program. A state-space exploration that searches through all possible message orders of a program could become intractable because of the state-space explosion problem. Nondeterministic message orders can cause message races and make modular reasoning intractable through an exponential explosion of state-space. Two messages are racy if they are sent transitively from the same source module and arrive at a sink module with no happens-before order [91] between them. The happens-before relation is a partial order that orders messages. Partial order reduction techniques address this problem by pruning the state-space into a representative subset of all possible message orders. However, the pruning is a global analysis that is dependent on the property of interest and cannot be easily modularized.

A session type [79, 80] is a global type that prescribes a communication protocol among its participant modules over a session. A communication protocol includes messages sent by each participant and their happens-before order. A session is an abstraction that structures the communication over a private channel [111] and isolates it from other communication. A global session type can be projected into its participant to compute their local session types. Global progress of a session type guarantees the absence of deadlocks. Session fidelity ensures that communication over a channel goes as prescribed by its session type. Session linearity guarantees the absence of message races on its enclosed channel. Variations of session types support two [79] or more session participants [80] or infer a global principal type from local types of its participants [116]. Addition of channels and sessions to structure communication over channels allows the application of session types to a variety of multithreaded [43], message passing [115, 120] and sequential [81, 119] programming models. Session types and their underlying session-based programming are also added to languages like Erlang [115], Java [81] and Python [119] and frameworks like CORBA [169]. However, session types are designed for a session-based programming model that uses channel communications and are not directly applicable to message passing concurrency models without these constructs [120]. Also session types do not support abstraction and cannot hide local messaging behavior of a module.

CHAPTER 3. INTERFERENCE CONTROL FRAMEWORK

Modular reasoning is important for scalable software development because it allows programmers and tools [68, 148] to discharge verification obligations about a module by just considering the implementation of the module and the interfaces (not implementations) of *static types* named in that module. Concurrent programming, i.e. the ability to perform two or more simultaneous computations in a single program, is also vital for creating modern software systems. We believe that concurrent programming stubbornly remains difficult and error-prone because we cannot, yet, do modular reasoning about concurrent programs.

3.1 Problem

Two fundamental obstacles make modular reasoning about a concurrent program difficult [87, 177]:

❶ the *pervasive interference* problem, and

❷ the *oblivious interference* problem.

By the *pervasive interference* we mean that in a concurrent program, between any two consecutive instructions of a concurrent task, interleaving instructions of another task could change the state of the program such that it influences the subsequent behavior of the first task [177]. This in turn means, there are too many points in a program that a programmer must analyze before they can understand the behavior of their program. For example, in the single straight line code below there could be three interference points that occur between instructions that read the value of x, read the value of y, add the two values, and write the value of x.

```
x = x + y;
```

That is, this code actually looks like $^{\alpha}\, x = {}^{\alpha}\, x + {}^{\alpha}\, y$ where $^{\alpha}$ denotes an interference point.

By the *oblivious interference* we mean that interference points do not give out any information, either concrete or abstract, about what other concurrent tasks may interfere or what are their behaviors. This in turn means, a programmer must consider all potentially concurrent tasks to determine whether their interleavings would be harmful and cause interferences (global reasoning). For example, in the straight line code above, there is no information about how interfering tasks at interference points $^{\alpha}$ may or may not modify the values of x and y.

The key difference between pervasive and oblivious interference is that the former is about locations of interferences (where) whereas the latter is concerned about behaviors of interferences (what). Though, these two are well-known concurrency problems we coin the terms "pervasive interference" and "oblivious interference" to refer to them.

Pervasive and oblivious interference together, make modular reasoning about concurrent programs difficult [68, 87, 148, 176, 177]. Several prior techniques have been proposed for controlling interference and interfering behaviors of concurrent programs. Figure 3.1 summarizes closely related works and compares them regarding pervasiveness and obliviousness of interferences. Section 3.9 details the comparison of previous work regarding pervasive and oblivious interference for all related works.

Table 3.1 Previous work addressing (✓) pervasive and oblivious interference.

	Interference	
	Pervasive	***Oblivious***
atomicity, transactional memory, cooperability, AME	✓	
actors, active objects	✓	
rely-guarantee, Owicki-Gries		✓

Atomicity, transactional memory, cooperability and automatic mutual exclusion (AME) An atomic block [58, 66, 171] is a code block which is free of interference, i.e. the code in the block behaves as if it executes sequentially. Atomicity limits the interference points to the code outside atomic blocks, however, for the code outside an atomic block interference could still happen between any two instructions. Atomic blocks do not say anything about behaviors of interfering tasks.

Cooperability [157, 176, 177] and automatic mutual exclusion [2, 84, 154] are the inverse of atomic blocks, i.e. the code is atomic and interference-free unless explicitly specified. These techniques limit interferences to explicitly specified interference points but similarly do not say anything about behaviors of interfering tasks.

Actors and active objects Actors [6, 7, 172] encapsulate data and control and communicate by passing messages. A class of actor models (similarly active objects [38, 151]) in which actors provide confinement and do not permit unfettered internal concurrency limits interference points to message sends or receives, i.e. a block of code between two message receives are atomic (macro-step semantics [7]). However, actor models that allow uncontrolled internal concurrency or arbitrary data sharing among actors still could have interference between any two instructions. Actor models do not solve the oblivious interference problem because their dynamic name bindings could make the exact static type of a receiver or sender of a message unknown. Thus, a programmer cannot tell which concurrent tasks are interfering at interference points.

Rely-guarantee and Owicki-Gries Rely-guarantee based reasoning approaches [68, 87] and Owicki-Gries's work [129] specify the behavior of the environment (other concurrently running tasks) of a task and thus limit the interference behavior. However, interferences can still happen between any two instructions of a program.

3.2 Solution: Panini

A concurrent programming model can enable more modular reasoning if it provides the following properties to a programmer:

❶ *Sparse interference*: interference points are not pervasive in a program, instead they can be explicitly identified by certain program constructs; and

❷ *Cognizant interference*: interference points are not oblivious in the program, instead each explicitly identified interference point provides an abstraction of behaviors of all potentially interfering concurrent tasks.

The language Panini presents such a programming model called *capsule-oriented programming* [136, 138]. Before discussing Panini's interference model and semantics, we provide a gentle introduc-

tion using the example in Figure 3.1. The example declares a capsule type Counter with a state x, on line 2, and a procedure add, on lines 3–5. The body of add contains the straight line code shown in the previous section, which adds y to x. A capsule instance of Counter, say c, can be declared and it behaves like a process.

In Panini, a program is a set of concurrently running capsule instances which own their states and communicate with each other through asynchronous procedure invocations. Invocation of a capsule instance procedure appends the invoked procedure to the tail of the instance's queue and returns a future [174] for its result. The single execution thread of a capsule instance dequeues its invoked procedures from the head of the queue and executes them in the order that they appear in the queue. Invocation and returning of a procedure transfers the ownership of its parameters and return values between its invoking and invoked capsule instances, respectively.

```
1  capsule Counter {
2    int x;
3    void add( int y ) {
4      x = x + y;
5    }
6  }
```

Figure 3.1 Capsule Counter with state x and procedure add.

3.2.1 Sparse Interference to Solve Pervasive Interference

Panini's semantics controls and limits sharing among two capsule instances to other capsule instances and futures. This in turn allows a Panini program to limit its interference points to after capsule procedure invocations and guarantees sparse interference. As an example, there are no interferences in the body of the procedure add of a capsule instance c of type Counter, which is in contrast with straight line code in the previous section with pervasive interference and possible interferences between any two instructions. This is because Panini's semantics guarantees that the state x is owned and is only accessible to its enclosing capsule instance c and there is only one thread of execution running c and accessing x. This in turn allows interferences in the code for the procedure add to be safely swapped out [101], as if the code runs with no interferences.

A reader familiar with synchronization features in languages like Java could perhaps achieve the same by implementing the counter as a class and marking its add method as **synchronized**, however, the capsule model saves the programmer from worrying about if there still could be interferences on line 4 and whether acquiring an object-level lock is actually sufficient to *protect* x. A lock protects a memory location if, throughout a program, every access to the location is preceded by acquiring the lock [66].

```
1  capsule Counter {
2    Number x;
3    void add( Number y ) {
4      x.add( y.value() );
5    }
6    Number value() { return x.value(); }
7  }
```

Figure 3.2   Capsule Counter with a reference type state.

To illustrate Panini's semantics more, consider the capsule Counter in Figure 3.2 which creates the possibility of sharing the state x and the capsule procedure argument y, among an instance of Counter and other capsule instances, by changing their types from integer to a reference type Number.

Again, the body of the procedure add of a capsule instance c does not have any interferences. This is because Panini's semantics guarantees that not only the state x but also its *representation* [32], i.e. the object graph rooted at x, is owned by the instance c and is only accessible from within c itself. The semantics also guarantees that upon invocation of the procedure add and throughout its execution its argument y and its representation is owned by c. This in turn allows interferences in the code for add to be safely swapped out, as if the code runs with no interferences.

Note that the alternative using **synchronized** or locks works only if proper locks are put in place to guarantee that they protect not only the state x and its representation but also the parameter y and its representation [66]. Panini's semantics does not allow lock splitting, i.e. protecting fields in a capsule by different locks.

So far we have looked at capsules with no interference points in the bodies of their procedures. To illustrate capsules with interference points consider the capsule Client in Figure 3.3. This capsule imports an external capsule instance c of type Counter, on line 1, that a capsule instance of type Client can interact with. The body of the procedure test of Client contains asynchronous invocations of procedures value and add on the receiver capsule instance c, on lines 4–6. Asynchronous invocation of value, on

line 4, appends the body of the procedure to the end of the queue for c and immediately returns the unresolved future oldVal without waiting for the execution of value. The future is resolved and ready whenever the capsule instance c dequeues the body of value from its queue and executes it. Any attempt to access an unresolved future, e.g. on line 7, blocks until the future is resolved. Lines 5–6 append the queue of c with invocations of procedures add and value, respectively. The capsule instance c dequeues its invoked procedures from its queue in the same order as they appear in the queue and executes them, i.e. first in first out (FIFO) order. For invocations of value and add, on lines 5–6, execution of value starts after the execution of add finishes.

```
1  capsule Client ( Counter c ) {
2    Number oldVal, newVal;
3    void test( Number y ) {
4      oldVal = c.value();
5      c.add( y );
6      newVal = c.value();
7      //@ assert newVal >= oldVal;  Φ
8    }
9  }
```

Figure 3.3 Capsule Client with JML-like assertion Φ on line 7.

There are three interference points in the body of test, one right after each asynchronous invocation of procedures value and add, on lines 4–6. This is because, Panini's semantics guarantees that the imported capsule instance c is the only shared resource among a capsule instance of type Client and other capsule instances in the system, with unsynchronized access. Other shared resources are futures oldVal and newVal, however, their accesses are synchronized and do not cause interference [177]. Panini's semantics also guarantees that upon invocation of add, on line 5, the ownership of its parameter y is transferred to c, and the body of test does not access y after its ownership is transferred.

As Figures 3.1, 3.2 and 3.3 illustrate, unlike pervasive interference in which interferences can happen between any two instructions in a program, Panini's semantics guarantees that in a Panini program, potential interference points are explicitly identified and they are limited to only after asynchronous procedure invocations.

3.2.2 Cognizant Interference to Solve Oblivious Interference

Panini controls and limits accessibility of states of a capsule instance to only through its procedures and dispatches procedure invocations using the static type of their receiver capsule instances. This in turn, allows a Panini program to limit the interference behavior at an interference point to the Kleene closure of procedures of the static type of the receiver of the procedure invocation and guarantees cognizant interference. The Kleene closure of a set of procedures contains the empty set and any concurrent composition of any number of procedures in the set.

For example, the interfering behavior at the interference point after the invocation of the procedure value on capsule instance c, on line 4, is contained in the Kleene closure $\kappa = \{c.value(), c.add(_)\}^*$. This is because Panini's semantics guarantees that the imported capsule instance c is the only shared resource with unsynchronized access between an instance of Client and other capsule instance in a system. Panini's semantics also guarantees that the state of the capsule instance c is only accessible through its procedures add and value. Finally Panini's semantics guarantees that the invocation of value, at the interference point on line 4, is dispatched using the static type of its receiver capsule instance c and not its subtype capsules which may have more procedures than add and value.

To interfere with c, other concurrently running capsule instances in the system can change the state of c by invoking its two procedures add and value any number of times and in any order which basically is the same as the closure κ. The closure κ is a closure of 0 or more, concurrently running, invocations of *all* procedures of the capsule instance c. The Kleene closure for value and add includes empty set $\emptyset$ and is closed under the sequential composition and execution operation ;. For example, $c.value()$, $c.value(); c.value()$, $c.add(_)$ and $c.value(); c.add(_)$ are a few elements of κ.

As Figure 3.3 illustrates, unlike oblivious interference in which the interference behavior is unknown, Panini's semantics guarantees that interference behavior for an interference point after an asynchronous procedure invocation is limited to the Kleene closure of procedures of the static type of the receiver of the invocation.

3.2.3 Modular Reasoning Using Panini's Interference Model

Panini's sparse and cognizant interference enable static modular reasoning about its concurrent programs. To modularly understand a module in a Panini program *(1)* using sparse interference, the interference points of the module can be identified *syntactically* by only considering the implementation of the module. This is because interference points are explicitly identified by asynchronous procedure invocations. *(2)* using cognizant interference, the interfering behaviors at interference points can be identified *statically* by just considering the interfaces of static types of receivers of procedure invocations at these interference points. This is because interfering behaviors are Kleene closures of procedures of receivers of procedure invocations. *(3)* for each interference point identified in *(1)* its interfering behavior identified in *(2)* could be inserted at the interference point in the module to arrive at a result module that takes interference and its behavior into account. Such a module could be modularly understood [68] using Hoare-style reasoning [77].

For example, consider static verification of the assertion Φ, on line 7 of Figure 3.3. In this example, sparse interference limits the interference points to after asynchronous invocations of procedures value and add, on lines 4–6, and cognizant interference limits the interference behavior at these interference points to the Kleene closure $\kappa = \{c.value(), c.add(_)\}^*$. Φ can be modularly verified after inserting the interfering behavior κ at each interference point in the procedure test using Hoare-style [77] reasoning.

Such reasoning is modular because the programmer only needs to consider the implementation of the procedure test and the interface of the static types it refers to, i.e. capsule type Counter. Identification of interference points and interfering behavior in test is similarly modular. Behaviors of procedures value and add say that the former does not change the value of the counter whereas the latter only increases it. Behaviors of these procedures are specified by their contracts, illustrated in Section 3.6.

3.2.4 Contributions

In summary, the contributions of this chapter are the following:

- formalization of Panini's core calculus and its semantics (Section 3.3 and Section 3.4); and
- proving sparse and cognizant properties of Panini's interference model (Section 3.5); and

- illustration of modular Hoare-style reasoning using behavioral contracts for concurrent Panini programs with interference (Section 3.6).

Section 3.8 briefly discusses Panini's expressiveness, usability and Kleene closure analysis. Section 3.9 presents related work.

3.3 Panini's Syntax

$prog ::= \overline{decl}$	program
$decl ::= \textbf{capsule}\ C\ (\overline{imp})\ \{\ design\ \ \overline{state}\ \overline{proc}\ \}$	capsule declaration
$state ::= T\ \ f\ ;$	state declaration
$proc ::= T\ p\ \ (\ \overline{form}\ )\ \{\ e^{\ \alpha}\ \}$	procedure declaration
$form ::= T\ \ x$	formal
$design ::= \textbf{design}\ \{\ \overline{ins}\ \ \overline{wire}\ \}$	design declaration
$ins ::= C\ \ i\ ;$	instance declaration
$wire ::= i\ (\overline{j})\ ;$	wiring declaration
$e ::=$	expression
$\mid i\,.\,p\ (\overline{e})^{\ \alpha}$	global procedure invocation
$\mid \textbf{self}\,.\,p\ (\overline{e})$	local procedure invocation
$\mid \textbf{self}\,.\,f$	state read
$\mid \textbf{self}\,.\,f\ := e$	state assignment
$\mid \textbf{ref}\ e$	reference
$\mid \textbf{deref}\ e$	dereference
$\mid e\ := e$	reference assignment
$\mid \textbf{let}\ x = e\ \textbf{in}\ e$	let
$\mid x$	variable
$\mid ()$	unit value

$C, D \in \mathfrak{C}$	set of capsule names	$T \in \mathfrak{T}$	set of variable types
$f \in \mathfrak{F}$	set of state names	$p \in run \cup \mathfrak{P}$	set of procedure names
$x \in \mathfrak{X}$	set of variable names	$\alpha, \beta \in \mathfrak{B}$	set of labels
$i,j,h \in \mathfrak{I}$	set of capsule instance names		

Figure 3.4 Panini's core syntax, based on [138].

Figure 3.4 shows Panini's expression-based core syntax. In this figure, the superscript $\overline{term}$ shows a sequence of zero or more terms. A Panini program is a set of capsule declarations. A capsule declaration contains a capsule name C and declares a set of imported capsule instances $\overline{imp}$, a design *design*, a set of capsule states $\overline{state}$ and capsule procedures $\overline{proc}$. A capsule instance can interact and invoke procedures

of two kinds of other capsule instances: imported and locally declared. An import declaration declares an imported capsule instance by specifying its capsule type D and capsule name i. A local design declaration declares a set of local capsule instances $\overline{ins}$ and specifies their connections together in a wiring declaration *wire*. A wiring declaration $i(\overline{j})$ assigns capsule instances $\overline{j}$ to the imported capsule instances of the capsule i. Local capsule instances of one capsule are not accessible to other capsules.

Capsule instances of a Panini program interact *only* by invoking procedures of each other. A procedure declaration of a capsule has a variable return type T, a name p, set of formal parameters $\overline{form}$ and a body e. The body of a capsule procedure is a sequence of global and local expressions. Using global expressions, a procedure can *asynchronously* invoke a procedure of another capsule instance. However, using local expressions, a procedure of a capsule can *synchronously* invoke another procedure of the same capsule, access the state of the capsule through ***self*** or allocate and access memory locations. Labels $^{\alpha}$ denote possible interference points after asynchronous procedure invocations. The sequence of expressions $e_1; e_2$ is a syntactic sugar for ***let*** $x = e_1$ ***in*** e_2 in which variable x is free in e_2.

Panini's type system, in Section 3.7, distinguishes capsule types C from variable types T. Unlike variable types, capsule types *cannot subtype each* other and thus their exact types are statically known. This in turn enables statically-bound procedure invocations in which the exact type of the receiver of a capsule is statically known.

To illustrate, Figure 3.3 declares a capsule type Client with the imported capsule c of capsule type Counter and a procedure test with formal parameter y of variable type Number. The procedure body is a sequence of three asynchronous invocations of procedures value and add which are statically dispatched on the imported capsule instance c. As another example, the capsule type Main in Figure 3.5, declares a design declaration on lines 2–6 that includes declarations of two capsule instances client and counter of types Client and Counter, respectively. The design declaration contains a wiring declaration on line 5 that connects the client and counter instances by passing counter as the client's imported capsule instance.

3.4 Operational Semantics

In Panini's operational (dynamic) semantics each concurrently running capsule instance owns its states and their representations, i.e. reference graphs reachable from its states; dynamically transfers

```
1  capsule Main() {
2    design {
3      Counter counter;     // a counter
4      Client  client ;     // a client
5      client (counter);    // a wiring
6    }
7    void run() { client . test () ; }
8  }
```

Figure 3.5 Design and wiring declarations in capsule Main.

ownership of parameters and return values of its global procedure invocations; and uses only one thread of execution to dequeue and execute its invoked procedures. These in turn result in the following properties of a Panini programs that are critical to its sparse and cognizant interference model:

1. sharing among two capsule instances is limited to their imported capsule instances and unresolved future locations;

2. states of a capsule instance and their representations are only accessible through its procedures.

Panini's interference model and its underlying properties are formalized in Section 3.5. Panini's type system is formalized in Section 3.7.

3.4.1 Dynamic Objects

Panini's dynamic semantics relies on three additional expressions l, ***resolve*** and ***OWE***, shown in Figure 3.6, that are not part of its surface syntax. The expression l represents a memory location in the store. The expression $\boldsymbol{resolve}(l, e, id, p)$ returns the result of the asynchronous invocation of procedure p with body e into future location l in the invoking capsule instance id. The ownership transfer exception ***OWE*** denotes accessing a transferred location that a capsule instance no longer owns or transferring a location in the representation of a capsule's state.

Panini's operational semantics transitions from one global (program) configuration to another, as shown in Figure 3.6. A global configuration $\mathscr{K}$ is a concurrent composition $||$ of capsule instance configurations Σ[1]. A capsule configuration Σ contains a *unique* capsule identifier id, a queue Q with an expression e under evaluation at its head, a local store S, a capsule record r and an instance mapping I. A queue is a possibly empty queue of expressions e. The local store is a mapping from locations

[1]Concurrent composition $||$ is commutative, i.e. $\Sigma \,||\, \Sigma'$ is equal to $\Sigma' \,||\, \Sigma$.

Added syntax:

$e ::= ..$

| l — memory location
| $\textbf{\textit{resolve}}(l, e, id, p)$ — resolve a future location
| $\textbf{\textit{OWE}}$ — ownership transfer exception

Evaluation contexts:

$\mathscr{E} ::= - \mid i.p(v..\mathscr{E}\, e..) \mid \textbf{\textit{self}}.p(v..\mathscr{E}\, e..)$
$\mid \textbf{\textit{self}}.f \ := \ \mathscr{E} \mid \textbf{\textit{ref}}\, \mathscr{E} \mid \textbf{\textit{deref}}\, \mathscr{E}$
$\mid \mathscr{E} \ := \ e \mid \textbf{\textit{let}}\, x \ = \ \mathscr{E}\ \textbf{\textit{in}}\, e$

Evaluation relations $\overset{a}{\hookrightarrow}$ and $\overset{a}{\rightsquigarrow}$: $\mathscr{K} \overset{a}{\hookrightarrow} \mathscr{K}'$ and $\Sigma \overset{a}{\rightsquigarrow} \Sigma'$

Domains:

$\mathscr{K} ::= \bullet \mid \Sigma \,\|\, \mathscr{K}$	global (program) configurations
$\Sigma ::= \langle P, id, e.Q, S, r, I\rangle$	capsule instance configurations
$r ::= [C.F]$	capsule records
$F ::= \{f_k \mapsto l_k\}$	state maps
$Q ::= \bullet \mid e.Q$	queues
$S ::= \{l_k \mapsto v_k\}$	local stores
$I ::= \{i_k \mapsto id_k\}$	instance maps
$v ::=$	values
l	location values
$\mid \varepsilon$	unresolved future values
$\mid \square$	transferred location values

Actions:

$a ::=$	
$\mid read(id, l)$	read
$\mid write(id, l)$	write
$\mid invoke(id, id', p, l)$	invoke
$\mid resolve(id, id', p, l)$	resolve
$\mid local(id)$	local

$l \in \mathfrak{L}$	set of locations	$id, id' \in \mathfrak{N}$	set of capsule identifiers
$k \in \mathbb{N}$	set of natural numbers	P	program declarations

Figure 3.6 Added syntax, evaluation contexts, configurations and actions in semantics.

l accessible to the capsule instance to their values v. The capsule record contains the static capsule type C of the instance and a state mapping F from capsule fields f to locations. The instance mapping maps the names of imported and locally declared capsule instances i to their identifiers id. A capsule configuration also includes the program text P which contains capsule declarations, similar to the class table in Featherweight Java, and is similarly used to look up declarations and procedure bodies when invoking a procedure.

In Panini, a value can be a location l. A value can also be an unresolved future value ε that denotes the result of an asynchronous procedure invocation before it is ready; or it can be a transferred value $\square$ denoting the value of a location whose ownership has been transferred and no longer is accessible to a capsule instance.

Panini uses a left-most inner-most call-by-value evaluation policy. Evaluation contexts, in Figure 3.6, specify the evaluation order of an expression and the evaluation position in the expression.

Execution of a Panini program produces a trace of observable actions. Actions are basic units of execution and each action represents execution of a single *indivisible* (atomic) instruction. Figure 3.6 shows a core set of actions observed during the execution of a Panini program. An action can be: a read or write of a memory location l by a capsule instance id; asynchronous invocation of a procedure p of another capsule id' with the future result l; resolving the result of an asynchronous procedure invocation into the future location l; or it can be a local action of a capsule id, such as invocation of a synchronous procedure of the capsule or dequeuing an expression from its queue.

3.4.2 Local and Global Semantics

Panini's operational semantics has two sets of evaluation rules for its local and global semantics. A local evaluation $\stackrel{a}{\rightsquigarrow}$ denotes transition from a capsule configuration to another performing the action a. A local transition in turn causes a global transition $\stackrel{a}{\hookrightarrow}$ from one program configuration to another in which capsule instances run concurrently. A *preemptive scheduler nondeterministically* chooses a capsule instances for evaluation at each point in time.

Figure 3.7 shows Panini's substitution-based operational semantics for normal execution, i.e. no exceptions thrown. Figure 3.9 shows its exceptional operational semantics. The notation used in formalizing the operational semantic is similar to previous work [177].

3.4.3 Sequential Synchronous Local Semantics

Local evaluation relation $\stackrel{a}{\rightsquigarrow}$ in a capsule instance denotes evaluation of an expression e at the head of its queue to another expression e' and performing the action a. This evaluation causes transition from a capsule configuration to another with a possibly modified queue and local store[2]. In local

[2] Evaluation of a configuration does not change its mapping I and record r.

$$\textbf{\textit{Local evaluation relation}} \overset{a}{\rightsquigarrow}: \quad \langle P, id, \mathscr{E}[e].Q, S, r, I\rangle \overset{a}{\rightsquigarrow} \langle P, id, \mathscr{E}[e'].Q', S', r, I\rangle$$

(STATE READ)

$$\langle P, id, \mathscr{E}[\textbf{\textit{self}}.f].Q, S[l=v], [C.F[f=l]], I\rangle \overset{read(id,l)}{\rightsquigarrow} \langle P, id, \mathscr{E}[v].Q, S, [C.F], I\rangle$$

(STATE ASSIGN)

$$\langle P, id, \mathscr{E}[\textbf{\textit{self}}.f := v].Q, S, [C.F[f=l]], I\rangle \overset{write(id,l)}{\rightsquigarrow} \langle P, id, \mathscr{E}[v].Q, S[l:=v], [C.F], I\rangle$$

(SELF INVOC)

$$\left\langle P[\textbf{\textit{capsule}}\ C(..)\{..\ T\ p(\overline{T\ x})\{e\}\ ..\}], id, \mathscr{E}[\textbf{\textit{self}}.p(\overline{v})].Q, S, [C.F], I\right\rangle \overset{local(id)}{\rightsquigarrow} \langle P, id, \mathscr{E}[e\ [\overline{v}/\overline{x}]].Q, S, [C.F], I\rangle$$

(REF)

$$\frac{fresh(l)}{\langle P, id, \mathscr{E}[\textbf{\textit{ref}}\ v].Q, S, r, I\rangle \overset{write(id,l)}{\rightsquigarrow} \langle P, id, \mathscr{E}[l].Q, S[l:=v], r, I\rangle}$$

(DEREF)

$$\frac{v \neq \varepsilon}{\langle P, id, \mathscr{E}[\textbf{\textit{deref}}\ l].Q, S[l=v], r, I\rangle \overset{read(id,l)}{\rightsquigarrow} \langle P, id, \mathscr{E}[v].Q, S, r, I\rangle}$$

(REF ASGN)

$$\langle P, id, \mathscr{E}[l := v].Q, S[l \neq \varepsilon], r, I\rangle \overset{write(id,l)}{\rightsquigarrow} \langle P, id, \mathscr{E}[v].Q, S[l:=v], r, I\rangle$$

(LET BINDING)

$$\langle P, id, \mathscr{E}[\textbf{\textit{let}}\ x = v\ \textbf{\textit{in}}\ e].Q, S, r, I\rangle \overset{local(id)}{\rightsquigarrow} \langle P, id, \mathscr{E}[e[v/x]].Q, S, r, I\rangle$$

(FIFO DEQUEUE)

$$\langle P, id, v.e.Q, S, r, I\rangle \overset{local(id)}{\rightsquigarrow} \langle P, id, e.Q, S, r, I\rangle$$

$$\textbf{\textit{Global evaluation relation}} \overset{a}{\hookrightarrow}\textbf{:} \quad \mathscr{K} \parallel \langle P, id, \mathscr{E}[e].Q, S, r, I\rangle \overset{a}{\hookrightarrow} \mathscr{K}' \parallel \langle P, id, \mathscr{E}[e'].Q', S', r, I\rangle$$

(CONGRUENCE)

$$\frac{\langle P, id, \mathscr{E}[e].Q, S, r, I\rangle \overset{a}{\rightsquigarrow} \langle P, id, \mathscr{E}[e'].Q', S', r, I\rangle}{\mathscr{K} \parallel \langle P, id, \mathscr{E}[e].Q, S, r, I\rangle \overset{a}{\hookrightarrow} \mathscr{K} \parallel \langle P, id, \mathscr{E}[e'].Q', S', r, I\rangle}$$

(PROC INVOC)

$$\frac{\begin{array}{c} id' = I(i) \qquad \Sigma' = \langle P, id', e'.Q', S', [C'.F'], I'\rangle \in \mathscr{K} \qquad \textbf{\textit{capsule}}\ C'(\overline{D\ j})\{..\ T\ p(\overline{T\ x})\{e\}\ ..\} \in P \\ e'' = e\ [\overline{v}/\overline{x}, I'(\overline{j})/\overline{j}] \qquad R = reach(\overline{v}, S) \qquad R' = \bigcup_{l' \in dom(F)} reach(l', S) \\ R \cap R' = \emptyset \qquad fresh(l) \qquad \mathscr{K}' = \mathscr{K} \uplus \langle P, id', e'.Q'.\textbf{\textit{resolve}}(l, e'', id, p), S' \oplus R, [C'.F'], I'\rangle \end{array}}{\mathscr{K} \parallel \langle P, id, \mathscr{E}[i.p(\overline{v})].Q, S, [C.F], I\rangle \overset{invoke(id,id',p,l)}{\hookrightarrow} \mathscr{K}' \parallel \langle P, id, \mathscr{E}[l].Q, S[l:=\varepsilon] \ominus R, [C.F], I\rangle}$$

(RESOLVE)

$$\frac{\begin{array}{c} R = reach(v, S) \qquad l' \in dom(F) \qquad R' = \bigcup_{l' \in dom(F)} reach(l', S) \\ R \cap R' = \emptyset \qquad \Sigma = \langle P, id, e.Q, S[l=\varepsilon], r, I\rangle \in \mathscr{K} \qquad \mathscr{K}' = \mathscr{K} \uplus \langle P, id, e.Q, S[l:=v] \oplus R, r, I\rangle \end{array}}{\mathscr{K} \parallel \langle P, id', \textbf{\textit{resolve}}(l, v, id, p).Q', S', [C'.F'], I'\rangle \overset{resolve(id',id,p,l)}{\hookrightarrow} \mathscr{K}' \parallel \langle P, id', v.Q', S' \ominus R, [C'.F'], I'\rangle}$$

Figure 3.7 Local and global operational semantics of Panini.

semantics, a capsule instance can access its state through ***self***, allocate and access memory locations, invoke a procedure of itself or dequeue an expression in its queue. Local evaluation is synchronous and sequential, i.e., a capsule instance only has one execution thread.

A capsule can read and write its state through the variable ***self*** in the rules (STATE READ) and (STATE ASSIGN). A capsule state is accessible through state mapping F and local store S. A capsule's state name is mapped to a location in F and then there is a mapping between the state location and its value in S. To read the value of a state in (STATE READ), the notation $F[f = l]$ means to check if the field name f maps to a location l and the notation $S[l = v]$ means to check if the value of the location in the local store is equal to v and if so returns v. Reading a state stored at the location l by a capsule instance *id* causes a $read(id, l)$ action in the execution trace of the program. To assign a value to a state in (STATE ASSIGN), the notation $S[l := v]$[3] means to replace the old value of the location l with its new value v, such that the rest of S stays intact. Similar to read, writing a state stored at the location l by capsule instance *id* causes a $write(id, l)$ action.

A capsule can also create a reference, dereference it and assign to it in the rules (REF), (DEREF) and (REF ASGN). To create a new reference with a value v in (REF), $fresh(l)$ ensures that l is a fresh location which then is mapped to its value in the local store of the capsule. A fresh location is a location that does not belong to the local store of any other capsule instance in the program, as shown in Figure 5.15. By mapping the newly allocated location l to its value in the local store S, the rule (REF) makes the capsule instance *id* the owner of the location l as well. To dereference a location l in (DEREF), its value is retrieved from the store *unless the value is equal to the unresolved future value* ε. Trying to dereference an unresolved future value causes the capsule instance to *block*. The capsule instance unblocks and can continue execution when the value of the future is resolved, i.e. is not equal to ε anymore. The blocking condition $v \neq \varepsilon$ in (DEREF) *synchronizes* access to unresolved future locations and does not allow concurrent access to them. To assign to a reference in (REF ASGN), its value is simply updated in the local store of the capsule. Again, trying to assign to an unresolved future location causes the capsule instance to block. Evaluation rules (REF), (DEREF) and (REF ASGN) perform their corresponding write, read and write actions, respectively.

A reference location manipulated by these rules resides in a capsule's local store unless its ownership is transferred to another capsule via a procedure invocation. In other words, a capsule instance cannot access locations in other capsule instances if their ownership is not transferred to the capsule through procedure invocations.

[3] Notation $S[l = v]$ does not modify S whereas $S[l := v]$ does.

A capsule instance can *synchronously* invoke a procedure of itself in (SELF INVOC). Invocation of a local procedure causes the body of the procedure to replace the procedure invocation after proper substitutions for its formal parameters $\overline{x}$ and imported capsule instances $\overline{j}$. The notation $e[\overline{v}/\overline{x}]$ means to substitute formal parameters $\overline{x}$ with their values $\overline{v}$ in e. Invocation of a local procedure of a capsule instance id causes a local action $local(id)$.

In (FIFO DEQUEUE), after evaluation of the head of the queue in a capsule instance to a value v, the next expression in the queue is moved to the head of the queue for evaluation, if the queue is not empty. Dequeue blocks until the queue of the capsule instance is not empty. Dequeuing a queue of a capsule instance causes a local action as well. Semantics of a let expression is standard.

3.4.4 Concurrent Asynchronous Global Semantics

The global evaluation relation $\overset{a}{\hookrightarrow}$ denotes concurrent local evaluations of capsule instances of a Panini program as well as their asynchronous interactions through procedure invocations.

The rule (CONGRUENCE) plays the role of a preemptive scheduler that nondeterministically chooses a capsule instance id in the global configuration $\mathcal{K}$ to take an atomic action at each point in time, according to the local semantic rules.

In (PROC INVOC), a capsule instance id asynchronously invokes the procedure p of a capsule instance with the name i. The invoking capsule finds the identifier id' for the invoked capsule name i in its instance mapping I, finds its corresponding configuration Σ' in the global configuration $\mathcal{K}$ and retrieves the body e of its invoked procedure p. It then replaces in e the formal parameters $\overline{x}$ of p with their values $\overline{v}$ and imports $\overline{j}$ of its capsule type C' with their identifiers from instance mapping I', to arrive at e''. Then it wraps e'' in a ***resolve*** expression with a fresh future location l for returning the result of the invocation to its invoking capsule id and appends the resolve expression to the *tail* of the queue Q' of the invoked capsule instance id'. The notation $\mathcal{K} \uplus \langle P, id', e'.Q'.\textbf{\textit{resolve}}(l, e'', id, p), S' \oplus R, [C'.F'], I' \rangle$ denotes overriding the configuration $\langle P, id', e'.Q', S', [C'.F'], I' \rangle$ of the capsule instance id' in the global configuration $\mathcal{K}$, where $\uplus$ is an overriding union operation. A resolve expression $\textbf{\textit{resolve}}(l, e, id, p)$ is a sugar for $\textbf{\textit{let}}\ x = e\ \textbf{\textit{in}}\ \textbf{\textit{resolve}}(l, x, id, p)$ where x is free in e. The auxiliary function *dom* returns the domain of a map like store.

Because of the asynchrony of the procedure invocation, in (PROC INVOC) the control immediately returns back to the invoking capsule *id* without waiting for the execution of the invoked procedure p. The future location l is now *shared* between the invoking and invoked capsule instances, marked in the invoking capsule instance *id* as an unresolved future location with its value ε. The invocation expression performs an $invoke(id, id', p, l)$ action.

The resolve expression ensures that the result of its evaluation is sent back to the invoking capsule instance when it is ready and is going to be accessible through the future location. In (RESOLVE), the invoked capsule instance id' returns the result v of the evaluation of its expression e to the invoking capsule *id* and assigns the value to the future location l in its store, i.e. $S[l := v]$. The resolve expression performs a $resolve(id', id, p, l)$ action.

The rule (PROC INVOC) along with (FIFO DEQUEUE) enforce the first in first out (FIFO) evaluation order of expressions in the queue of a capsule where (PROC INVOC) appends to the tail of the queue and (FIFO DEQUEUE) dequeues from its head. A Panini program terminates normally when for *each capsule* instance *id* in the program configuration the expression at the head of the queue is evaluated to a value and the queue is empty, i.e. $\langle P, id, v.\bullet, S, r, I\rangle$.

To illustrate, consider the capsule Client, in Figure 3.3, and its asynchronous invocation of the procedure value of the capsule Counter, on line 4. Upon invocation of value on the capsule instance c its body, on lines 6 of Figure 3.2, is wrapped in a resolve expression and is appended to the tail of c's queue. The control immediately returns to Client and the unresolved future NewVal is shared between the client and counter capsule instances as a placeholder for the invocation's result. Any attempt to access NewVal in the client capsule, e.g. on line 7, blocks until c dequeues the resolve expression for invocation of value and executes it to resolve the future.

Asynchronous invocation, blocking expressions and procedure execution order Asynchronous invocations of capsule procedures and blocking access to unresolved future locations impose an order on executions of invoke procedures which may or may not be the same as in a synchronous setting. To illustrate, the procedure add, on line 5 of Figure 3.3, is invoked on the capsule instance c before value is invoked on the same instance, on line 6. This in turn means the body of add is appended to the queue of c and shows up before the body of value in the queue. Consequently, the invoked procedure add is executed before value because of the FIFO execution of the queue in c. This is true, even if either add

or value or both contain blocking expressions, e.g. trying to access an unresolved future, in their body because the execution of value in c does not start before the execution of add is finished.

```
1  capsule Client ( Counter c, Counter d ) {  ..
2    void test( Number y ) {
3      ..
4      d.add( y );
5      newVal = c.value();
6      ..
7    }
8  }
```

In contrast, consider invocations of procedures add and value on different capsule instances c and d, in the above variation of Client, on lines 4–5. In this example, add and value procedures can execute in any arbitrary order because instances c and d run concurrently and can execute bodies of their invoked procedures add and value in any arbitrary order. This is true even with blocking expressions in the bodies of add or value procedures or both.

In other words, for asynchronous invocations of procedures of the same capsule instance, the rule (FIFO DEQUEUE) ensures that these procedures run in the same order they are invoked, even if they contain blocking expressions. This in turn guarantees that blocking does not disrupt modular reasoning using Kleene closures, because procedure invocations in a closure are on the same receiver instance.

3.4.5 Ownership Transfer Semantics

To control sharing, Panini's global semantics uses dynamic transfer of ownership for parameters and the return value of a procedure invocation. Transferring the ownership of a location from one capsule to another removes that location and locations reachable from it, i.e. its reach (representation), from the local store of the former instance and adds them to the local store of the latter. This in turn guarantees that an invocation of a procedure does not cause sharing of its parameters and the result between the invoking and invoked capsule instances. Panini's ownership transfer is similar to ownership transfer in Singularity [57], changing threads access sets in a multithreaded program [85] or inferred ownership transfer semantics in SOTER [118] for programs in the actor language ActorFoundry [14].

In (PROC INVOC), upon invocation of a capsule's procedure the ownership of the actual parameters $\overline{v}$ of the procedure and their reach $reach(v,S)$, in Figure 5.15, is transferred from the invoking capsule instance to the invoked capsule. The auxiliary function $\ominus$, in Figure 5.15, removes locations from a

local store of a capsule instance whereas $\oplus$ adds locations to the local store of the capsule instance. For example, in the configuration $\langle P, id, \mathscr{E}[l].Q, S[l := \varepsilon] \ominus R, [C.F], I \rangle$, locations R are removed from the local store S of the invoking capsule instance id and in $\langle P, id', e'.Q'.\textbf{\textit{resolve}}(l, e'', id, p), S' \oplus R, [C'.F'], I' \rangle$ the locations R are added to the local store S' of the invoked capsule instance id'.

After transferring a location, $\ominus$ maps the value of a transferred location to $\square$. This in turn means that the capsule instance does not own the transferred location anymore and any attempt to access it results in an exceptional state.

A state of a capsule instance or its reach cannot be transferred to another capsule instance. The condition $R \cap R' = \emptyset$ checks that there is no shared location among transferred locations and their reach R and the locations in the capsule's state or its reach R'.

In (RESOLVE), upon returning the result of an invocation, the ownership of the resolved future value, holding the result, and its reach in the local store of the invoked capsule instance are transferred to the invoking capsule instance. Similar to (PROC INVOC), the state of the invoked capsule instance cannot be transferred when returning the result of an invocation.

To illustrate, the ownership of the parameter y on line 5 of the procedure test in Figure 3.3 is transferred from the invoking instance of capsule Client to the invoked capsule instance c. The ownership of the future newVal on line 6 is transferred from invoked capsule instance c to the invoking capsule Client when the future is resolved and is ready.

3.4.6 Exceptional Semantics

A Panini program terminates abnormally, throwing an exception ***OWE***, if a capsule instance attempts to access a location whose ownership is transferred or pass or return capsule states or their representations into/from global procedure invocations. Figure 3.9 shows Panini's exceptional semantics.

In rules (X DEREF) and (X REF ASGN), attempting to dereference or assign to a location that is transferred out and is not owned by a capsule instance anymore results in throwing an ownership transfer exception ***OWE***. A transferred location is marked with the value $\square$, by the transfer operation $\ominus$. In rules (X PROC INVOC) and (X RESOLVE), attempting to pass capsule states or any location in their reach as actual parameters or results of global procedure invocations, i.e. $R \cap R' \neq \emptyset$, causes throwing the ownership exception and termination of the program. A Panini program terminates abnormally when a

(FRESH)
$$\frac{\forall \langle P, id_k, \mathscr{E}[e_k].Q_k, S_k, r_k, I_k\rangle \in \mathscr{K}, l \notin dom(S_k)}{fresh(l)}$$

(REACH)
$$\frac{v \in \{(), \square\}}{reach(v,S) = \bullet}$$

(REACH LOCATION)
$$\frac{v \notin \{(), \square\} \qquad S[v = v']}{reach(v,S) = \{(v,v')\} \cup reach(v',S)}$$

($\ominus$ LOCATION)
$$S \ominus \{(l,v)\} = S[l := \square]$$

($\ominus$ REACH)
$$\frac{\forall \{(l',v')\} \in R}{S \ominus R = (S \ominus \{(l',v')\}) \ominus (R \setminus \{(l',v')\})}$$

($\oplus$ LOCATION)
$$S \oplus \{(l,v)\} = S[l := v]$$

($\oplus$ REACH)
$$\frac{(l',v') \in R}{S \oplus R = (S \oplus (l',v')) \oplus (R \setminus \{(l',v')\})}$$

$$\begin{array}{l}
labels(v) = \emptyset \\
labels(\textbf{\textit{self}}.f) = \emptyset \\
labels(i.p(\overline{e})^{\alpha}) = \{^{\alpha}\} \cup labels(\overline{e}) \\
labels(\textbf{\textit{self}}.p(\overline{e})) = labels(\overline{e}) \\
labels(\textbf{\textit{self}}.f := e) = labels(e) \\
labels(\textbf{\textit{ref}}\ e) = labels(e) \\
labels(\textbf{\textit{deref}}\ e) = labels(e) \\
labels(e_1\ :=\ e_2) = labels(e_1) \cup labels(e_2) \\
labels(\textbf{\textit{let}}\ x\ =\ e_1\ \textbf{\textit{in}}\ e_2) = labels(e_1) \cup labels(e_2) \\
labels(T\,p\ (\overline{T\ x})\{e\ ^{\alpha}\}) = \{^{\alpha}\} \cup labels(e)
\end{array}$$

Figure 3.8 Panini's auxiliary functions.

capsule instance *id* in the program evaluates the head of its queue to ***OWE***, i.e. $\langle P, id, \textbf{\textit{OWE}}.Q, S, r, I\rangle$.

In Panini's core semantics, for simplicity and without loss of generality, exceptions are final terminating states. However, in the current prototype implementation of Panini's compiler, a violation of ownership transfer semantics is detected using a modular static analysis incorporated into its type system and is reported as a compile time warning rather than terminating the program at runtime.

3.4.7 Initial Configuration

Evaluation of a Panini program follows a phase which builds the program's *initial* global and local configurations. Figure 3.10 shows Panini's initial configuration rules. The initial configuration phase takes a Panini program and *recursively* processes design declarations of its capsules, such that for each capsule instance declaration in a design declaration it instantiates a capsule instance and for each wiring declaration it connects the declared capsule instances together.

The initial configuration phase starts with the rule (MAIN). The rule constructs a special capsule instance main of capsule Main with the identifier 0, i.e. $Construct(Main\ main, 0)$. The capsule Main is the entry point to a Panini program. The rule (MAIN) sets the capsule instance main to the global configuration $\mathscr{K}_0$ and calls functions *instantiateRec* and *wireupRec*, in rules (INSTANTIATE REC) and

(X DEREF)

$$\langle P, id, \mathscr{E}[\boldsymbol{deref}\ l].Q, S[l = \Box], r, I\rangle \overset{read(id,l)}{\rightsquigarrow} \langle P, id, \boldsymbol{OWE}.Q, S, r, I\rangle$$

(X REF ASGN)

$$\langle P, id, \mathscr{E}[l := v].Q, S[l = \Box], r, I\rangle \overset{write(id,l)}{\rightsquigarrow} \langle P, id, \boldsymbol{OWE}.Q, S, r, I\rangle$$

(X PROC INVOC)

$$\frac{R = reach(\overline{v}, S) \qquad R' = \bigcup_{l' \in dom(F)} reach(l', S) \qquad R \cap R' \neq \emptyset}{\mathscr{K} \ || \ \langle P, id, \mathscr{E}[i.p(\overline{v})].Q, S, [C.F], I\rangle \overset{invoke(id,id',p,l)}{\hookrightarrow} \langle P, id, \boldsymbol{OWE}.Q, S, [C.F], I\rangle}$$

(X RESOLVE)

$$\frac{R = reach(v, S) \qquad R' = \bigcup_{l' \in dom(F)} reach(l', S) \qquad R \cap R' \neq \emptyset}{\mathscr{K} \ || \ \left\langle P, id, \boldsymbol{resolve}(l, v, id', p).Q, S, [C.F], I\right\rangle \overset{resolve(id,id',p,l)}{\hookrightarrow} \langle P, id, \boldsymbol{OWE}.Q, S, [C.F], I\rangle}$$

Figure 3.9 Exceptional semantics of Panini, select rules.

(WIREUP REC), to recursively instantiate and connect other capsule instances of the program.

For a capsule instance declaration $C\ i$, declared in the enclosing capsule instance id and a global configuration $\mathscr{K}$, the function *instantiateRec*, defined in (INSTANTIATE REC) instantiates a capsule configuration Σ' with the identifier id' and name i, using *instantiate*; changes the instance mapping I of its enclosing capsule instance to map the name i to its identifier id', i.e. $I[i := id']$ and adds Σ' to $\mathscr{K}$ to create a new global configuration $\mathscr{K}'$. To process the capsule instance declarations $C'\ i'$ in newly created id', *instantiateRec* is recursively called with the new global configuration $\mathscr{K}'$.

For a wiring declaration $i(\overline{j})$ declared in the enclosing capsule instance id and the global configuration $\mathscr{K}$, the function *wireupRec*, defined in (WIREUP REC), connects instances i and $\overline{j}$, using *wireup*, to construct the new configuration Σ'_i for i; replaces the old configuration Σ_i with its new configuration Σ'_i in $\mathscr{K}$ to arrive at a new global configuration $\mathscr{K}'$. To process the wiring declarations $i'(\overline{j'})$ for the newly wired capsule instance id_i, *wireupRec* is recursively called with the identifier id_i and the new global configuration $\mathscr{K}'$.

Function *instantiate* defined in (INSTANTIATE) simply instantiates the capsule configuration Σ for the capsule instance declaration $C\ i$ in its enclosing capsule id. Function *wireup* in (WIREUP) connects capsule instances i and $\overline{j}$ in their enclosing capsule id.

To illustrate, consider the capsule Main in Figure 3.5. In this example, $Construct(Main\ main, 0)$

(MAIN)

$$\frac{\begin{array}{c}\mathcal{K}_0 = instantiate(0, Main\ main) = \langle P, 0, \bullet.Q, S, [Main.F], I\rangle \\ \textbf{capsule}\ Main()\{\overline{T\ f}\ \textbf{design}\{\overline{ins}\ \ \overline{wire}\}\ \ \overline{proc}\} \in CT \\ \forall\ C\ i \in \overline{ins}\ .\ \mathcal{K} = instantiateRec(\mathcal{K}_0, 0, C\ i) \qquad \forall\ i(\overline{j}) \in \overline{wire}\ .\ \mathcal{K}' = wireupRec(\mathcal{K}, 0, i(\overline{j}))\end{array}}{construct(Main\ main, 0) = \mathcal{K}'}$$

(INSTANTIATION REC)

$$\frac{\begin{array}{c}\textbf{capsule}\ C(..)\{\overline{T\ f}\ \textbf{design}\{\overline{ins}\ \ \overline{wire}\}\ \ \overline{proc}\} \in CT \\ \Sigma' = \left\langle P, id', \bullet.Q', S', [C'.F'], I'\right\rangle = instantiate(id, C\ i) \qquad \Sigma = \langle P, id, \bullet.Q, S, [C.F], I\rangle \in \mathcal{K} \\ \mathcal{K}' = \mathcal{K} \uplus \left\langle P, id, \bullet.Q, S, [C.F], I[i := id']\right\rangle \cup \Sigma' \qquad \forall\ C'\ i' \in \overline{ins}\ .\ \mathcal{K}'' = instantiateRec(\mathcal{K}', id', C'\ i')\end{array}}{instantiateRec(\mathcal{K}, id, C\ i) = \mathcal{K}''}$$

(INSTANTIATION REC-BASE)

$$\frac{\begin{array}{c}\textbf{capsule}\ C(..)\{\overline{T\ f}\}\ \ \overline{proc}\} \in CT \qquad \Sigma' = \left\langle P, id', \bullet.Q', S', [C'.F'], I'\right\rangle = instantiate(id, C\ i) \\ \Sigma = \langle P, id, \bullet.Q, S, [C.F], I\rangle \qquad \mathcal{K}' = \mathcal{K} \uplus \left\langle P, id, \bullet.Q, S, [C.F], I[i := id']\right\rangle \cup \Sigma'\end{array}}{instantiateRec(\mathcal{K}, id, C\ i) = \mathcal{K}'}$$

(WIRING REC)

$$\frac{\begin{array}{c}\Sigma = \langle P, id, \bullet.Q, S, [C.F], I\rangle \in \mathcal{K} \qquad id_i = I(i) \qquad \Sigma_i = \langle P, id_i, \bullet.Q_i, S_i, [C_i.F_i], I_i\rangle \in \mathcal{K} \\ \Sigma_i' = wireup(id, i(\overline{j})) \qquad \textbf{capsule}\ C_i(\overline{imp})\{..\ \textbf{design}\{\overline{ins}\ \ \overline{wire}\}\ \ ..\} \in CT \\ \mathcal{K}' = \mathcal{K} \uplus \Sigma_i' \qquad \forall\ i'(\overline{j'}) \in \overline{wire}\ .\ \mathcal{K}'' = wireupRec(\mathcal{K}', id_i, i'(\overline{j'}))\end{array}}{wireupRec(\mathcal{K}, id, i(\overline{j})) = \mathcal{K}''}$$

(WIRING REC-BASE)

$$\frac{\begin{array}{c}\Sigma = \langle P, id, \bullet.Q, S, [C.F], I\rangle \in \mathcal{K} \qquad id_i = I(i) \qquad \Sigma_i = \langle P, id_i, \bullet.Q_i, S_i, [C_i.F_i], I_i\rangle \in \mathcal{K} \\ \Sigma_i' = wireup(id, i(\overline{j})) \qquad \textbf{capsule}\ C_i(\overline{imp})\{..\ \textbf{design}\{\overline{ins}\}\ \ ..\} \in CT \qquad \mathcal{K}' = \mathcal{K} \uplus \Sigma_i'\end{array}}{wireupRec(\mathcal{K}, id, i(\overline{j})) = \mathcal{K}'}$$

(INSTANTIATE)

$$\frac{\begin{array}{c}fresh(id') \qquad \textbf{capsule}\ C(..)\{\overline{T\ f}\ \textbf{design}\{\overline{ins}\ \ \overline{wire}\}\ \ \overline{proc}\} \in CT \qquad F = \emptyset \qquad S = \emptyset \\ I = \emptyset \qquad Q = \bullet \qquad \forall (T\ f) \in \overline{T\ f}\ .\ fresh(l), F[f := l], S[l := ()] \qquad \Sigma = \left\langle P, id', \bullet.Q, S, [C.F], I\right\rangle\end{array}}{instantiate(id, C\ i) = \Sigma}$$

(WIREUP)

$$\frac{\begin{array}{c}\Sigma = \langle P, id, \bullet.Q, S, [C.F], I\rangle \in \mathcal{K} \qquad id_i = I(i) \qquad \Sigma_i = \langle P, id_i, \bullet.Q_i, S_i, [C_i.F_i], I_i\rangle \in \mathcal{K} \\ \forall j \in \overline{j}\ .\ id_j = I(j), \left\langle P, id_j, \bullet.Q_j, S_j, [C_j.F_j], I_j\right\rangle \in \mathcal{K} \qquad \textbf{capsule}\ C_i(\overline{imp})\{..\ \textbf{design}\{\overline{ins}\ \ \overline{wire}\}\ \ ..\} \in CT \\ \Sigma_i' = \left\langle P, id_i, \bullet.Q_i, S_i, [C_i.F_i], \forall (D\ h) \in \overline{imp}, j \in \overline{j}\ .\ I_i[h := id_j]\right\rangle\end{array}}{wireup(id, i(\overline{j})) = \Sigma_i'}$$

Figure 3.10 Rules to create initial configuration of Panini programs.

creates a capsule instance configuration for Main with the identifier 0; then it calls *instantiateRec* for the instantiation of capsule instances counter and client, declared on lines 3–4, followed by a call to *wireupRec* to connect the counter and client instances, as declared on line 5.

There is no sharing of memory locations among capsule instances in the initial configuration of a Panini program, as shown in Lemma 1.

Lemma 1 ***(No sharing of memory locations in initial configuration)*** *Let* $\Sigma = \langle P, id, \mathscr{E}[e].Q, S, r, I \rangle$ *and* $\Sigma' = \left\langle P, id', \mathscr{E}[e'].Q', S', r', I' \right\rangle$ *be two arbitrary capsule instance configurations in the initial configuration* $\mathscr{K}$ *for a Panini program P with global configuration* $\mathscr{K}$*, i.e.* $\Sigma, \Sigma' \in \mathscr{K}$*. Let* $A = dom(S) \cap dom(S')$ *be the intersection of domains of the stores S and* S'*. Then* $A = \emptyset$*.*

In other words, there is no sharing between local stores of capsule instances in the initial configuration of a program.

Proof : The proof is based on the cases of the initial configuration rules of Figure 3.10. The rule (INSTANTIATE) is the only rule allocating memory locations and mapping them in the stores of capsule instances. The rule only uses fresh locations for instantiation of each capsule instance and thus does not cause any sharing among capsule instances and thus their local stores.

3.4.8 Actions: Conflict and Happens-Before Relations

Evaluation of a Panini program, with its nondeterministic preemptive scheduler, results in a trace of interleaved actions of the types, shown in Figure 3.6, performed by different capsule instances of the program. However, as illustrated in Figures 3.1, 3.2 and 3.3, for a trace of a capsule's procedure, interleaving actions of other capsule instances can be moved in the trace, such that they only appear right after the global procedure invocations in the trace, i.e. sparse interference. This is because of Panini's conflicting and happens-before relations and the mover properties of its actions which in turn are affected by sharing semantics of Panini.

In the following, we define the execution trace of a Panini program, the conflict and happens-before relations and prove mover properties of its comprising actions using Lipton's reduction theory [101]. Definitions 1-4 are adapted from previous work [175].

Definition 1 ***(Trace)*** *An execution trace of a Panini program is a total order of actions a performed by individual capsule instances in the program's configuration when evaluating the program thorough local and global evaluation rules of Figure 3.7.*

Definition 2 ***(Adjacent and neighbor actions)*** *Two actions a and b in a trace* $\mathscr{T}$ *are adjacent if one follows immediately after the other. Two adjacent actions a and b are neighbors if they are performed by different capsule instances, i.e.* $instance(a) \neq instance(b)$*.*

The auxiliary function *instance* returns the capsule identifier of an action.

Definition 3 ***(Commuting and conflicting actions)*** *Let a_1 and a_2 be neighbor actions of capsule instances id_1 and id_2 in an execution trace $\mathscr{K}_0 \overset{a_1}{\hookrightarrow} \mathscr{K}_1 \overset{a_2}{\hookrightarrow} \mathscr{K}_2$. Then actions a_1 and a_2 commute, written as a_1 !# a_2, if swapping them in the trace results in the same final state in the trace starting with the same start state, i.e. $\mathscr{K}_0 \overset{a_2}{\hookrightarrow} \mathscr{K}_1' \overset{a_1}{\hookrightarrow} \mathscr{K}_2$. Otherwise, a_1 and a_2 conflict, written as $a_1 \# a_2$.*

There are several conflicting actions in Panini considering its semantics: read or write of an unresolved future location conflicts with the resolution of the same future location and invoke action of a procedure of a capsule instance conflicts with another invoke action on the same capsule instance.

A happens-before relation $\triangleright$ [91] orders the conflicting actions. For example, in Panini resolution of a future location l by a capsule instance id must happen-before any reads (or writes) of the location by another capsule instance id', i.e. $resolve(id,id',l,p) \triangleright read(id',l)$. Figure 3.11 shows Panini's conflicting actions and their happens-before relations. The happens-before relation is a transitively closed partial order [177].

$$\frac{\langle P,id',\mathscr{E}[e].Q,S[l=\varepsilon],r,I\rangle \in \mathscr{K}}{read(id',l)\ \#\ resolve(id,id',l,p)} \qquad \frac{\langle P,id',\mathscr{E}[e].Q,S[l=\varepsilon],r,I\rangle \in \mathscr{K}}{write(id',l)\ \#\ resolve(id,id',l,p)}$$

$$invoke(id_1,id,p,l)\ \#\ invoke(id_2,id,p',l') \qquad resolve(id,id',l,p)\ \triangleright\ read(id',l)$$

$$resolve(id,id',l,p)\ \triangleright\ write(id',l)$$

Figure 3.11 Conflicting actions and their happens-before $\triangleright$ relations.

Definition 4 ***(Right, left, both and non-mover actions)*** *Let a_1 and a_2 be neighbor actions of capsule instances id_1 and id_2 in an arbitrary execution trace $\mathscr{K}_0 \overset{a_1}{\hookrightarrow} \mathscr{K}_1 \overset{a_2}{\hookrightarrow} \mathscr{K}_2$.*

Then a_1 is a right mover if swapping a_1 with a_2 in the trace results in the same final state in the trace beginning with the same start state, i.e. $\mathscr{K}_0 \overset{a_2}{\hookrightarrow} \mathscr{K}_1' \overset{a_1}{\hookrightarrow} \mathscr{K}_2$. Conversely, a_2 is a left mover if swapping it with a_1 results in the same final state. An action that can be swapped with its both left and right neighbor actions in any trace is a both mover. Conversely, an action that cannot be swapped with neither its left nor right neighbors is a non-mover.

Panini's semantics determines mover properties of its actions. Lemma 2 specifies mover properties of Panini's actions.

Lemma 2 ***(Panini action's mover properties)*** *Let $\mathscr{T}$ be the execution trace of an arbitrary Panini program P.*

Then, in trace $\mathscr{T}$ read and write actions $read(id,l)$ and $write(id,l)$ of a capsule instance id of a memory location l are right movers; a global invocation action $invoke(id,id',p,l)$ of a procedure p from the invoking capsule id to the invoked capsule id′ and result l is a non-mover; and a resolve action $resolve(id,id',p,l)$ for this global invocation is a left mover.

Proof : The proof is based on happens-before relations of Panini's actions in Figure 3.11. The resolution of a future location must happen before any read or write of the location and thus in a trace of a program, a read action $read(id,l)$ of future location l cannot be swapped with a left neighbor action $resolve(id',id,l,_)$ resolving the same location. Thus, a read action is a right mover. The same applies to an action writing a future location. Similarly, a resolve action is a left mover. Swapping an invoke action $invoke(id',id,_,_)$ invoking a procedure of capsule *id* with another left or right neighbor invoke action $invoke(id'',id,_,_)$ on the same capsule *id* results in different program states especially different queues for the capsule instance *id*. Thus an invoke action is a non-mover. The notation _ denotes irrelevant values in actions.

Let a be an action with left and right neighbors a_l and a_r respectively in the subtrace $a_l \hookrightarrow a \hookrightarrow a_r$. We replace a with read, write, invoke and resolve actions of a capsule instance *id* to show their mover properties in an arbitrary trace with arbitrary left and right neighbor actions from other capsules.

In a subtrace $a_l \hookrightarrow read(id,l) \hookrightarrow a_r$, the read action of a location l conflicts with a left neighbor $resolve(id,id',p,l)$ action of the same location. This is because swapping the read action with its left neighbor allows reading a future location even before it is resolved. However, Panini's happens-before relations, in Figure 3.11, does not allow this by ensuring that a future location is resolved before it is read or otherwise it blocks, i.e. $resolve(id,id',p,l) \triangleright read(id,l)$. This in turn means the read action cannot be a left mover. Since resolving of a future location must happen before its read, a resolve action $resolve(id,id',p,l)$ cannot be right neighbor to the read action $read(id,l)$ and thus the read action can be safely swapped with any of its right neighbors, i.e. the read action is a right mover. The same argument

applies to a write action $write(id,l)$ and thus a write action is a right mover too.

Similarly, a resolve action $resolve(id,id',p,l)$ in a subtrace $a_l \hookrightarrow resolve(id,id',p,l) \hookrightarrow a_r$ only conflicts with read and write actions from and to the same location l, i.e. $read(id,l)$ and $write(id,l)$. Again, based on Panini's happens-before relation a read or write of a future location happens only after the future is resolved and thus the read and write actions of a future location cannot be left neighbors to their resolve actions. This in turn means that the resolve action is a left mover and not a right mover.

An invoke action $invoke(id,id',p,l)$ only conflicts with another invoke action if they both invoke procedures on the same capsule instance id', since they both modify the queue of the capsule id'. In a subtrace $a_l \hookrightarrow invoke(id,id',p,l) \hookrightarrow a_r$ the invocation action cannot be safely swapped with neither its left nor its right neighbors and thus is a non-mover. This is because they neighbors could be invocation actions on the same capsule instance id'.

It is worth to note that local actions as well as read and write of non future locations are both movers.

Finally, in (X DEREF) and (X REF ASGN), trying to dereference or assign to a transferred location not owned by the capsule instance anymore, causes the program to throw an ownership transfer exception ***OWE*** and terminate. In (X PROC INVOC) and (X RESOLVE), the program terminates by throwing an ownership transfer exception upon any attempts to leak the states of an instance or its reach by passing them or returning them from a global procedure invocation.

3.4.9 Sharing of Capsule Instances and Futures

Panini limits sharing among two capsules to their imported capsule instances and unresolved futures of their procedure invocations.

```
1  capsule Main() {
2    design {
3      Counter counter;
4      Client client1 , client2 ;
5      client1 (counter);
6      client2 (counter);
7    } ..
8  }
```

Figure 3.12 Sharing counter among client1 and client2.

Panini's semantics allows an imported capsule instance to be freely shared among other instances

as specified in a design declaration of their enclosing capsule. For example, in the design declaration of Figure 3.12, counter is shared among two client instances client1 and client2, on lines 5–6.

Panini's semantics limits sharing of memory locations to unresolved future locations, as shown by Lemma 3. A future location, which is a placeholder for the result of an asynchronous procedure invocation, is shared among its invoking and invoked capsule instances as long as it is unresolved and accesses to it are synchronized. That is, any attempt to access an unresolved future location in the invoking capsule instance blocks until the future is ready.

Let $\veebar$ denote an exclusive logical disjunction, in which at most one of the disjunct predicates can be true. Let $dom_{\square}(S)$ be the domain of the stores S minus its transferred locations with the value $\square$.

Lemma 3 ***(Sharing of unresolved future locations)*** *Let* $\Sigma = \langle P, id, \mathscr{E}[e].Q, S, r, I\rangle$ *and* $\Sigma' = \langle P, id', \mathscr{E}[e'].Q', S', r', I'\rangle$ *be two arbitrary capsule instance configurations in a global configuration* $\mathscr{K}$ *for a Panini program P, i.e.* $\Sigma, \Sigma' \in \mathscr{K}$. *Let* $\mathscr{T}$ *be the execution trace of the program P. Let* $A = dom_{\square}(S) \cap dom_{\square}(S')$ *be the intersection of domains of the stores S and S′ minus their transferred locations. Let action a and a′ be any of the read and write actions of a location l in A in the trace* $\mathscr{T}$ *by capsule instances id and id′, respectively.*

Then either $A = \emptyset$ *or:*

1. $\forall l \in A\,.\; S[l = \varepsilon] \veebar S'[l = \varepsilon]$*; and*

2. $\forall l \in A,\, a \in \{read(id, l), write(id, l)\},\, a' \in \{read(id', l), write(id', l)\}\,.\;\; a \rhd a' \veebar a' \rhd a.$

In other words, (i) the only memory locations that local stores S and S′ may share are unresolved future locations with (ii) synchronized accesses (reads and writes), i.e. with happens-before relation between their reads and writes.

Transferred locations with values $\square$ are irrelevant and thus taken out in $dom_{\square}$ of local stores, because any attempt to access them terminates the program.

Proof : The proof is by cases on Panini's normal and exceptional semantics rules in figures 3.7 and 3.9, happens-before relations in Figure 3.11 and Lemma 1. Initial configuration does not cause any sharing of memory location among capsule instances; dynamic transfer of ownership of locations among capsule instances in Panini's dynamic semantics, and especially (PROC INVOC) and (RESOLVE),

limits sharing of locations among capsules to only unresolved future locations; and rules (DEREF) and (REF ASGN) synchronize access to these shared future locations by enforcing that resolving of a future location in one capsule instance must happen-before any of its reads or writes in other capsule instances.

Initial configuration Using Lemma 1 there is no shared location among capsule instances in the initial configuration, i.e. $A = \emptyset$, and thus the lemma holds.

Dynamic semantics Rules (PROC INVOC) and (RESOLVE) transfer ownership of memory locations among local stores of the invoking and invoked capsule instances. For each procedure invocation, these rules share a fresh future location among the invoking and invoked capsule instances and set the value of the shared location in the local store of the invoking instance to ε. However, the condition $R \cap R' = \emptyset$ in these two rules prevents them to share other memory locations, upon transferring ownership of parameters of the procedure invocation or returning its result.

The rule (REF) allocates a fresh location in the local store of a capsule instance and thus does not cause any sharing.

The rules (DEREF) and (REF ASGN) block when attempting to read or write an unresolved future location with the value ε. This means for an unresolved future location l with value ε in the capsule instance Σ, its read action $a = read(id, l)$ does not unblock unless the value of the future location is resolved by the capsule instance Σ; any write action $a' = write(id', l)$ of the location l by the capsule instance *id* should happen before the location is resolved, because of ownership transfer after resolve. That is, $a' \rhd resolve(id', id, l, _) \rhd a$ which in turn means $a' \rhd a$ because of the transitivity of the happens-before relation [177]. In other words, there is a happens-before relation between a and a' and thus they are synchronized. The same applies to other combination of a and a' actions in *(ii)* in the lemma. Other rules in Panini's normal and exceptional dynamic semantics do not cause any transfer of ownership or memory allocation.

3.5 Sparse and Cognizant Interference

Panini guarantees sparse interference by limiting sharing among two capsules to other capsule instances and unresolved futures and guarantees cognizant interference by limiting accessibility of states of a capsule instance to only through its procedures and dispatching a procedure invocation on the static

type of its receiver capsule. In this section we formalize and provide proof sketches of Panini's sparse and cognizant interferences.

3.5.1 Sparse Interference

Theorem 5 formalizes Panini's sparse interference property which limits the interference points of a program to points after its global procedure invocations.

Theorem 5 ***(Sparse interference in Panini)*** *Let P be a program in Panini. Let $\mathscr{I}$ be a set of labels after global capsule invocations and after procedure bodies in P, i.e. $\mathscr{I} = \{^{\alpha} | ^{\alpha} \in labels(P),\ i.p(\overline{e})^{\alpha} \in P \vee T\ \ p(\overline{form})\{e\ ^{\alpha}\} \in P\}$ where the auxiliary function labels is defined in Figure 5.15. Then $\mathscr{I}$ is the set of all potential interference points for P.*

Proof : The proof is based on Lemma 2 where in a trace of a capsule's procedure, interfering actions of other capsule instances can be safely moved to either after the global procedure invocation actions in the trace or to the beginning or end of the execution of the trace. The interference at the beginning of the trace of a procedure can be safely moved out to the trace of its invoking procedure, either to the end of the invoking procedure's trace or after one of its global procedure invocations.

Let $read(id)$, $write(id)$, $invoke(id)$ and $resolve(id)$ stand as shorter versions of $read(id, _)$, $write(id, _)$, $invoke(id, _, _, _)$ and $resolve(id, _, _, _)$ of capsule instance id where _ denotes irrelevant values that do not matter to the discussion.

Let $\mathscr{T}_{id,p}$ denote a subtrace corresponding to execution of procedure p of capsule instance id. $\mathscr{T}_{id,p}$ starts with the first action of the procedure, a_s, and ends with the resolve action $resolve(id)$ of the procedure, *with actions of procedures of other capsule instances interleaving*. Furthermore, lets partition $\mathscr{T}_{id,p}$ to smaller subtraces $\mathscr{T}_{sub}$ such that actions $invoke(id)$, $resolve(id)$ and a_s only end up at the beginning or end of the subtrace. Subtraces $\mathscr{T}_{sub}$ do not overlap and their concatenation results in $\mathscr{T}_{id,p}$. A subtrace $\mathscr{T}_{sub}$ has one of the following four shapes: (1) it starts and ends with invoke actions $invoke(id)$ with zero or more read and write actions of id, i.e. $read(id)$ or $write(id)$ in between (2) it starts with a_s and ends with an invoke action $invoke(id)$ with read and write actions of id in between; (3) it start with an $invoke(id)$ and ends in $resolve(id)$ with read and write actions of id in between; or (4) it start with a_s

and end with $resolve(id)$ with read and write actions of id in between. In any of these subtraces, actions of other capsules are interleaving.

In a subtrace $\mathscr{T}_{sub}$ of form (1), using Lemma 2 any read and write actions $read(id)$ and $write(id)$ can be right swapped such that they form a transaction with the invoke action at the end of the subtrace. A transaction is a sequence of actions of a capsule instance that behaves as if executed sequentially with no interference. This in turn means, all neighboring actions of other capsule instances in $\mathscr{T}_{sub}$ move to after the invoke action at the start of the subtrace. In a subtrace of form (2), right swapping read and write actions $read(id)$ and $write(id)$ causes them to form a transaction with the invoke action at the end of the subtrace. Consequently, all neighboring actions in this subtrace moves to before the a_s action, i.e. before the execution of the body of the procedure p. In a subtrace of form (3) right swapping of $read(id)$ and $write(id)$ causes them to form a transaction with $read(id)$ and thus all neighboring actions of other capsules move to after the invoke at the beginning of the subtrace. Finally in a subtrace of form (4), right swap of $read(id)$ and $write(id)$ causes the whole subtrace to form a transaction and all neighboring actions move to before the execution of the procedure p. In other words, all the neighboring actions interleaving with actions of procedure p of capsule instance id can be moved to either before the execution of the procedure or to after invocation actions $invoke(id)$ of the procedure. The interference at the beginning of the trace of the procedure p can be safely moved out to the trace of its invoking procedure, either to the end of the invoking procedure's trace or after one of its global procedure invocations. This argument could be repeated for execution of all procedure bodies in the trace $\mathscr{T}$ of a Panini program P.

3.5.2 Cognizant Interference

Theorem 6 formalizes Panini's cognizant interference property which limits the interfering behavior at each global invocation interference point to Kleene closure of behaviors of procedures in the static type of the invocation's receiver.

Theorem 6 ***(Cognizant interference in Panini)***

Let $\Sigma = \langle P, id, \mathscr{E}[i.p_k(\overline{e})^{\alpha}; e].Q, S, r, I\rangle$ be the configuration for capsule instance id in a global configuration $\mathscr{K}$, such that the capsule is about to evaluate the sequence expression $i.p_k(\overline{e})^{\alpha}; e$ at the head

of its queue with a single interference point $^{\alpha}$*, i.e, there is no other interference in other expressions* $\overline{e}$ *and e in the sequence. Let* $\Sigma' = \langle P, id', \mathscr{E}[e'].Q'.\textbf{\textit{resolve}}(l, e'_k, id, p_k), S', [C'.F'], I' \rangle \in \mathscr{K}$ *be the capsule configuration for capsule name i right at the interference point* $^{\alpha}$*, i.e. right after execution of invocation* $i.p_k(\overline{e})$ *in* Σ*. Let* C' *be the static capsule type for i with declared procedures* p_1 *..* p_n*, i.e.* ***capsule*** $C'(..)$ $\{$***design*** $\overline{state}$ $T_1\ p_1(..)\{e_1\}$ .. $T_k\ p_k(..)\{e_k\}$ .. $T_n\ p_n(..)\{e_n\}\} \in CT$*. Also let* $e'_1, ..e'_n$ *be bodies of procedure* $p_1, .., p_n$ *with their formal parameters substituted with their values from their invocation sites and their local capsule names substituted with their identifiers from instance mappings* I'*. Also let* θ *be the interfering behavior of other capsule instances in the program at the interference point* $^{\alpha}$*.*

Then, θ *is the Kleene closure of behaviors of procedures of the capsule* id'*, i.e.* $\theta = \{e'_1, .., e'_n\}^*$*.*

Proof : The proof is by cases on local and global evaluation rules in Figure 3.7 and that Panini limits sharing of memory locations to unresolved future locations with synchronized access, in Lemma 3, and limits accessibility of the state of a capsule instance to only through global invocation of its procedures, in Lemma 4.

Let $\textbf{\textit{resolve}}(e'_1)$ stand as abbreviation for $\textbf{\textit{resolve}}(_, e'_1, _, _)$ in which _ denotes irrelevant values. Capsule instances *id'* and *id* are shared among other capsule instances in the global configuration $\mathscr{K}$ and thus could be susceptible to interference. Using Lemma 4, the state of *id'* can only be accessed and modified through invocation of its procedures.

Case analysis for dynamic semantic rules:

(PROC INVOC): at the interference point $^{\alpha}$, Panini's global procedure invocation along with preemptive and nondeterministic scheduler in (CONGRUENCE) allows any arbitrary number (zero or more) of procedures of *id'* to be invoked and their bodies, with their formal parameters substituted with their values and ***self*** substituted with *id'*, to be appended to its queue Q'. Consequently the queue of *id'* will be of the form $\mathscr{E}[e].Q'.\textbf{\textit{resolve}}(l, e'_k, id).\{\textbf{\textit{resolve}}(e'_1), .., \textbf{\textit{resolve}}(e'_n)\}^*$, in which zero or more bodies of the invoked procedures are appended to the end of the queue.

(FIFO DEQUEUE): at the interference point $^{\alpha}$, Panini's dequeue rule (FIFO DEQUEUE) along with the scheduler (CONGRUENCE) allow an arbitrary number of resolve expressions of procedure bodies to be dequeued from Q' and evaluated.

(OTHER): the global rule (RESOLVE) or other local rules (STATE READ), (STATE ASSIGN), (REF), (DEREF), (REF ASGN), (LET BINDING) and (SELF INVOC), do not cause any invocation of procedures of the capsule id' and thus are irrelevant to interfering behavior at $^{\alpha}$.

For the capsule instance *id*, using Lemma 4, its state can only be accessed and modified through invocation of its procedures. At interference point $^{\alpha}$, any invocation of procedures of *id* using (PROC INVOC) is appended to the end of its queue Q for later dequeuing using (FIFO DEQUEUE) and local sequential execution and thus does not interfere at $^{\alpha}$. Other dynamic semantic rules do not invoke any procedure on *id* and thus do not interfere at $^{\alpha}$.

Thus, according to the aforementioned case analysis of the dynamic semantics rules, the interfering behavior of other capsule instances at the interference point $^{\alpha}$ is $\theta = \{e'_1, .., e'_n\}^*$.

Lemma 4 ***(Global accessibility through procedures)*** *Let* $\Sigma = \langle P, id, \mathscr{E}[e].Q, S, [C.F], I\rangle$ *and* $\Sigma' = \left\langle P, id', \mathscr{E}[e'].Q', S', [C'.F'], I'\right\rangle$ *be arbitrary capsule instance configurations in a global configuration* $\mathscr{K}$*, i.e.* $\Sigma, \Sigma' \in \mathscr{K}$*. Let C be the static capsule type of id with states* $\overline{f}$ *and procedures* $p_1 .. p_n$*, i.e.* ***capsule*** $C(..)$ $\{$***design*** $\overline{T\ f}\ \ U_1\ p_1(..)\{e_1\}\ ..\ U_n\ p_n(..)\{e_n\}\} \in P$.

Then during evaluation of program P, the capsule instance id' *can access (read and write) states* $\overline{f}$ *of the instance id only through its procedures* p_1 .. p_n*, and not directly through memory locations.*

Proof : The proof follows from Lemma 3 that guarantees the only shared locations among capsules are unresolved future locations and the rules (PROC INVOC) and (RESOLVE) in Figure 3.7 that prevent transfer of ownership of states of a capsule or its reach during procedure invocations and resolving of their results.

Using Lemma 3, there is no shared location among local stores S and S' of capsule instances, except future locations for returning the result of procedure invocations among capsules.

Let l be a shared future location among capsules *id* and id' when id' invokes a procedure of *id*. The future location cannot be used to modify the state of *id* because, upon the invocation the rule (PROC INVOC) guarantees that the future location is fresh and thus does not point to any state of *id*; during the execution of the procedure body, the rules (DEREF) and (REF ASGN) ensure that any attempts to access the location in *id* blocks until the future is resolved; and after the future location is resolved, (RESOLVE) ensures that the resolved future does not point to any state of *id* or its reachable locations.

3.6 Hoare-style Modular Reasoning

Standard Hoare logic [77] does not take interference into account and cannot be used right out of the box to reason about concurrent programs [68, 148]. Panini's sparse and cognizant interference enable use of Hoare logic in the presence of interference by making interference points of a Panini program and their interfering behaviors statically known.

In Panini's Hoare logic, the only rule that needs to take into account interference is the rule for global procedure invocations, because sparse interference limits interference points to after global procedure invocations. Other rules can be used as if they are interference-free. To take interference into account it is sufficient to consider the interfering behavior at each interference point [68].

To illustrate, consider the Hoare triple $\{Pre\}\ i.p(\overline{v})\ \{Post\}$ which says that if the execution of the global procedure invocation $i.p(\overline{v})$ starts in a state satisfying the predicate *Pre* and procedure p of the capsule instance i executes and terminates, it terminates in a state satisfying the predicate *Post*. To take into account the interference, the triple becomes $\{Pre\}\ i.p(\overline{v})^{\alpha}\ \{Post\}$ in which $^{\alpha}$ is an interference point. The new triple says if the execution of the procedure p of the capsule instance i starts in a state satisfying *Pre* and is interfered with by execution of some interfering tasks at $^{\alpha}$, if execution of p terminates, it terminates in a state satisfying the predicate *Post*.

Pure predicates In Hoare logic, predicates *Pre* and *Post* must be side effect free and if they invoke a procedures the procedures must be pure. A procedure is pure if it does not change the state of its program. In Panini the state of a program not only includes local stores of its capsule instances but also their queues. That is, a Panini procedure is pure if it does not change stores or queues of capsule instances in the program, including itself. To meet such a requirement a pure procedure in a capsule instance can only read states or invoke other pure procedures of its enclosing capsule instance via ***self***. Predicates *Pre* and *Post* can only invoke pure procedures of their enclosing capsule instance and cannot invoke pure procedures of other capsule instances. This is because invocation of a pure procedure of another capsule instance adds the invoked procedure to the queue of invoked capsule instance and changes the state of the system, i.e. the predicate is not pure.

Interference-free predicates In a Hoare triple $\{Pre\}\ e\ \{Post\}$ in a capsule instance in Panini, predicates *Pre* and *Post* are free from interference. This is because, these pure predicates only read

states of their enclosing capsule instance and invoke only its pure procedures. Corresponding actions for reading states and invocation of self procedures are both movers and thus any interference in predicates can be moved out, as if it appears to happen before evaluation of *Pre* or after the evaluation of *Post*.

```
1  capsule Client ( Counter c ) {
2    Number newVal, oldVal;
3    //@ requires y.value() >= 0
4    //@ ensures newVal >= oldVal
5    void test( Number y ) {
6      oldVal = c.value();
7      c.add( y );
8      newVal = c.value();
9    }
10 }
11 capsule Counter {
12   Number x;
13   //@ requires y.value() >= 0
14   //@ ensures self.value() >= \old( self .value());
15   void add( Number y ) { .. }
16   //@ pure
17   Number value() { .. }
18 }
```

Figure 3.13 Static verification of the behavioral contract of test.

Behavioral contracts In Panini, a behavioral contract for a procedure specifies the precondition and postcondition of a procedure. Figure 3.13 illustrates the contracts for procedure test of capsule Client with its pre and postconditions, on lines 3–4. The contract says if the execution of the procedure test starts in a state satisfying the precondition $y.value() >= 0$ and its execution terminates, it terminates in a state satisfying the postcondition $newVal >= oldVal$. Similarly, the contract for procedure add of capsule Counter, on lines 13–14, says it only increases the value of the counter provided that the parameter y is a positive number. Finally, the contract for value, on line 16, says it is a pure procedure and does not change the state of its enclosing capsule or any other capsule in the program. The precondition and postcondition of a behavioral contract are free from interference.

Modular reasoning Hoare-style reasoning can be used to statically verify the contract of procedure test. To illustrate, consider static verification of the method test in the capsule Client. The Hoare triple

representing such verification looks like the following:

$$\begin{aligned}
\Gamma \models \{&y.value() \ >= \ 0\} \\
&\textbf{\textit{self}}.oldVal = c.value(); \\
&c.add(y) \\
&\textbf{\textit{self}}.newVal = c.value(); \\
\{&\textbf{\textit{self}}.newVal \ >= \ \textbf{\textit{self}}.oldVal\}
\end{aligned}$$

The notation $\Gamma \models \{Pre\}\ e\ \{Post\}$ denotes that the Hoare triple $\{Pre\}\ e\ \{Post\}$ is valid in the typing environment Γ.

Following Panini's sparse interference, interferences only happens after global procedure invocations. Thus, the Hoare triple becomes like the following to take into account the interference, with $^{\alpha}$ denoting the interference points:

$$\begin{aligned}
\Gamma \models \{&y.value() \ >= \ 0\} \\
&\textbf{\textit{self}}.oldVal = c.value()\ ^{\alpha}\ ; \\
&c.add(y)\ ^{\alpha}\ ; \\
&\textbf{\textit{self}}.newVal = c.value()\ ^{\alpha}\ ; \\
\{&\textbf{\textit{self}}.newVal \ >= \ \textbf{\textit{self}}.oldVal\}
\end{aligned}$$

Following Panini's cognizant interference, the interfering behavior at the interference points in the above Hoare triple is $\theta = \{c.value(), c.add(_)\}^{*}$, i.e. the Kleene closure of procedure of the static capsule type Counter for the receiver c of the global procedure invocations:

$$\begin{aligned}
\Gamma \models \{&y.value() \ >= \ 0\} \\
&\textbf{\textit{self}}.oldVal = c.value(y);\ \{c.value(), c.add(_)\}^{*}\ ; \\
&c.add(y);\ \{c.value(), c.add(_)\}^{*}\ ; \\
&\textbf{\textit{self}}.newVal = c.value();\ \{c.value(), c.add(_)\}^{*}\ ; \\
\{&\textbf{\textit{self}}.newVal \ >= \ \textbf{\textit{self}}.oldVal\}
\end{aligned}$$

The above triple could be easily verified assuming $c.value()$ is pure and $c.add(_)$ only increases the counter, according to their contracts. This is because the closure $\{c.value(), c.add(_)\}^{*}$ either maintains the value of the counter c or increases it, which in turn means $\textbf{\textit{self}}.newVal >= \textbf{\textit{self}}.oldVal$. Such reasoning is modular because it only uses the implementation of Client and the interface (contract)

of procedures of the capsule Counter it refers to. The notation _ stands for the parameter of the procedure add. The exact value of this parameter is not statically known, however, using the precondition of add, on line 13, we know it is a positive number.

Without sparse and cognizant interference, one must consider the possibility of interferences with unknown behaviors between any two instructions of a program and its contracts [148].

Adding a procedure, say subtract, to the capsule Counter, in Figure 3.13, which decreases the counter changes the interference behaviors to the Kleene closure $\{value(), add(_), subtract(_)\}^*$. Using this closure, one cannot verify the postcondition of the procedure add anymore. However, this is not a limitation and is not specific to Panini. A similar situation happens in other reasoning techniques including rely-guarantee [67, 87], Owicki-Gries work [129], etc.

Similar to sequential reasoning, completeness of our reasoning is proportional to completeness of procedure specifications. That is, using incomplete specifications we can still reason about whatever specifications specify. For example, using the contract of add in Figure 3.13 we can reason about values of a counter, however, we cannot reason about other values which are part of its state but not mentioned in the contract.

3.7 Static Semantics

Panini's type system distinguishes between two kinds of types: variable types and capsule types. Unlike variable types that can subtype each other, capsule types cannot. This in turn, allows the exact type of the receiver of a global capsule invocation to be statically known. The type system also ensures that capsule instances cannot be passed as parameters or returned as return values of procedure invocations by requiring procedure parameters and return values to be of variable types.

3.7.1 Type Attributes

Panini's typing rules use type attributes of Figure 5.12. In this figure variable types are unit and reference types and capsule types are capsule names declared in a Panini program P.

The typing judgment $\Gamma, \Pi \vdash e : \theta$ says that for a program P in the typing environment Γ and store typing environment Π, the expression e has the type θ. The typing environment Γ maps variable names

to variable types T and capsule names to capsule types C and the store typing environment Π maps locations to their variable types.

θ ::=	type attributes
$\mid T$	variable types
$\mid C$	capsule types
T ::=	
$\mid$ ***unit***	unit types
$\mid$ ***int***	int types
$\mid$ ***ref****(T)*	reference types
$\Gamma ::= \{x_k : T_k,\ i_k : C_k\}$	variable typing environment
$\Pi ::= \{l_k : T_k\}$	store typing environment
$\Gamma, \Pi \vdash e : \theta$	typing judgement

$k \in \mathbb{N}$ set of natural numbers

Figure 3.14 Type attributes

The notation $P \vdash \theta$ in Panini's typing rules denotes that θ is a well-formed variable or capsule type and the notation $\vdash_C$ denotes well-typedness in the context of the declaration of a capsule type C.

3.7.2 Typing Rules

Figure 3.15 shows Panini's select typing rules. (T-CAPSULE DECL) type checks a capsule declaration. It ensures that the declarations of the capsule's imports, design, states and procedures are well typed. An import declaration is well typed if its imported names are of valid capsule types, i.e. $P \vdash D$. A well typed design declaration has well typed instance and wiring declarations. Similar to import declarations, an instance declaration is well typed if its declared name is of valid capsule type, i.e. $P \vdash G$. A state declaration is well typed if it declares a state name of a variable type, i.e. $P \vdash T$. This is because capsule instances cannot be part of state of other capsule instances. Wiring and procedure declarations should type check in the context of imported and locally declared capsule instances in the design declaration.

(T-PROC DECL) type checks a procedure declaration in the context of a capsule type C. It ensures that capsule instances cannot be declared as formal parameters or the return value of a procedure, by requiring them to be of variable types. This in turn prevents capsule instances to be passed to or returned

$$\textsc{(T-Capsule Decl)}\quad \frac{\begin{array}{c}\textbf{\textit{capsule}}\ C(\overline{D\ i})\{\ \textbf{\textit{design}}\{\overline{G\ h}\ \overline{wire}\}\ \overline{T\ f}\ \overline{proc}\ \} \in P \qquad \forall proc \in \overline{proc}\,.\ P,\ \overline{i:D},\ \overline{h:G}, \textbf{\textit{self}} : C \vdash_C proc \\ \forall wire \in \overline{wire}\,.\ P,\ \overline{i:D},\ \overline{h:G} \vdash wire \qquad \forall D \in \overline{D}\,.\ P \vdash D \qquad \forall G \in \overline{G}\,.\ P \vdash G \qquad \forall T \in \overline{T}\,.\ P \vdash T\end{array}}{P \vdash C}$$

$$\textsc{(T-Proc Decl)}\quad \frac{\Gamma,\Pi,\overline{x:T},\textbf{\textit{self}} : C \vdash e : T' \qquad P \vdash T' \qquad \forall T \in \overline{T}\,.\ P \vdash T}{\Gamma,\Pi \vdash_C T'\ p(\overline{T\ x})\{e\}}$$

$$\textsc{(T-Wiring Decl)}\quad \frac{C = \Gamma(i) \qquad \textbf{\textit{capsule}}\ C(\overline{D\ h})\{..\} \in P \qquad \forall j_k \in \overline{j}, D_k \in \overline{D}\,.\ D_k = \Gamma(j_k)}{\Gamma,\Pi \vdash i(\overline{j})}$$

$$\textsc{(T-Proc Invoc)}\quad \frac{C = \Gamma(i) \qquad \textbf{\textit{capsule}}\ C(..)\{..\ T'\ p(\overline{T\ x})\{e'\}\ ..\} \in P \qquad \forall e_k \in \overline{e}, T_k \in \overline{T}\,.\ \Gamma,\Pi \vdash e_k : T_k}{\Gamma,\Pi \vdash i.p(\overline{e}) : T'}$$

$$\textsc{(T-Self Invoc)}\quad \frac{\Gamma,\Pi \vdash \textbf{\textit{self}} : C \qquad \textbf{\textit{capsule}}\ C(..)\{..\ T'\ p(\overline{T\ x})\{e'\}\ ..\} \in P \qquad \forall e_k \in \overline{e}, T_k \in \overline{T}\,.\ \Gamma,\Pi \vdash e_k : T_k}{\Gamma,\Pi \vdash \textbf{\textit{self}}.p(\overline{e}) : T'}$$

$$\textsc{(T-State-Read)}\quad \frac{\Gamma,\Pi \vdash \textbf{\textit{self}} : C \qquad \textbf{\textit{capsule}}\ C(..)\{..T\,f..\} \in P}{\Gamma,\Pi \vdash \textbf{\textit{self}}.f : T} \qquad \textsc{(T-State-Assign)}\quad \frac{\Gamma,\Pi \vdash e : T \qquad \textbf{\textit{capsule}}\ C(..)\{..T\,f..\} \in P}{\Gamma,\Pi \vdash \textbf{\textit{self}}.f := e : T}$$

$$\textsc{(T-Resolve)}\quad \frac{\Gamma,\Pi \vdash e : T \qquad T = \Pi(l)}{\Gamma,\Pi \vdash \textbf{\textit{resolve}}(l,e,id,p) : T} \qquad \textsc{(T-Deref)}\quad \frac{\Gamma,\Pi \vdash e : \textbf{\textit{ref}}(T)}{\Gamma,\Pi \vdash \textbf{\textit{deref}}\,e : T} \qquad \textsc{(T-Reference)}\quad \frac{\Gamma,\Pi \vdash e : T}{\Gamma,\Pi \vdash \textbf{\textit{ref}}\ e : \textbf{\textit{ref}}(T)} \qquad \textsc{(T-}\square\textsc{)}\quad \frac{P \vdash T}{P \vdash \square : T}$$

$$\textsc{(T-}\varepsilon\textsc{)}\quad \frac{P \vdash T}{P \vdash \varepsilon : T} \qquad \textsc{(T-VarType1)}\quad \frac{T \in \{\textbf{\textit{int}},\textbf{\textit{unit}}\}}{P \vdash T} \qquad \textsc{(T-VarType2)}\quad \frac{T = \textbf{\textit{ref}}(T') \qquad P \vdash T'}{P \vdash T} \qquad \textsc{(T-Subsume)}\quad \frac{\Gamma,\Pi \vdash e : T' \qquad T' <: T}{\Gamma,\Pi \vdash e : T}$$

Figure 3.15 Panini's select typing rules.

from procedure invocations in (T-PROC INVOC). The rule also checks that the type of the body e of the procedure is the same as its return type. Subtyping can be easily added using the subsumption rule (T-SUBSUME) if needed.

(T-PROC INVOC) type checks a global procedure invocation. It ensures that the receiver i of the invocation is of capsule type C and its actual parameters and return value are of variable types. It also checks that the receiver's capsule type contains the invoked method p and the actual and formal parameters of the procedures have the same type. $\Gamma(i)$ returns the type of a capsule name i in the typing environment Γ. Type checking of local procedure invocations in (T-SELF INVOC) is similar.

(T-WIRING DECL) type checks a wiring declaration by ensuring that types of capsule names $\overline{j}$ passed to a wiring declaration are *the same as* capsule types declared in the import declaration of capsule type C. This is because capsule types do not subtype each other.

(T-RESOLVE) type checks a resolve expression. It ensures that the variable type of the expression e is of the same variable type of the location that is going to hold the value of e. $\Pi(l)$ returns the variable type of the location l in the typing environment Π.

(T-ε) and (T-$\square$) type check unresolved future value ε and $\square$ value of transferred locations, respectively. These two values can have any arbitrary variable type T. (T-REF) type checks a reference creation expression. It ensures that capsule names cannot be stored in the store, by requiring e to be of a variable type. The same is true in (T-REF ASGN).

Soundness Proof of Panini's type soundness follows standard progress and preservation arguments and thus is omitted.

3.8 Discussion

Expressiveness, usability, scalability and concurrency granularity Panini [137, 138] has been implemented [100] and tried out [100, 105] tried out on hundreds of thousands of lines of code of concurrent programs including translations of JavaGrande, NPB and StreamIt benchmarks and actor programs from Basset, Habanero and Jetlang covering a variety of patterns including master/worker, pipeline, event based coordination, loop parallelism. Previous work shows that Panini programs usually perform as well as their corresponding multithreaded programs [100, 138]. These experiences have not run into any issues with granularity of capsules or Panini's ownership model. We will not report on these experience as this chapter focuses on the formalization of Panini's semantics, interference model and modular reasoning.

Closure analysis Analysis of a Kleene closure can grow with the number of procedures in a capsule. However, Panini's cognizant interference is still a significant improvement over oblivious interference in which interfering behavior is completely unknown. Closure analysis could be further improved for example by eliminating pure procedures from closures or use of procedure invocation protocols to eliminate invalid invocation sequences and similar directions which are part of our future work plans.

Also, we conjecture that number of a capsule's procedures on average will be on par with average number of methods in a class, which is not a large number. For all Java projects in SourceForge as of Sep 2013, this average is about 8 methods, as obtained using Boa [51, 52, 55, 143].

3.9 Related Work

Actors and active objects This chapter builds on the actor model [6]. Some variants of the actor model, such as Erlang [172], guarantee confinement, i.e. no shared locations among actors, and use a single thread of execution per actor. Actors of this variant address the pervasive interference via their macro-step semantics [7] which limits interference points to message receive sites in the code. However, variants of the actor model which do not guarantee confinement, e.g. Scala Actors [74], or allow multiple unsynchronized execution threads per actor instance, e.g. Habanero [83], could still suffer from pervasive interference. Actor models and their variants also do not address the oblivious interference problem due to their dynamic binding of actor names and message names (in some cases), e.g. ActorFoundry [14] does not allow the static type of actor instances to be known statically.

Active objects [94], similar to actors, encapsulate their state and control. Variants of active objects that guarantee confinement and synchronized access to memory in active objects, such as JCoBox [151], address the pervasive interference problem. Several techniques including ownership type systems [38] or immutable data [151] can be used to enforce confinement. Again, dynamic binding of names could lead to oblivious interference.

Atomicity, transactional memory, cooperability and automatic mutual exclusion (AME) Transactional memory [92] is a concurrent programming model that enforces atomic blocks at runtime. There are also a variety of static [66] and dynamic [58] analyses to detect atomicity violations. Atomic sets [171] put fields of objects into atomic sets such that access to the fields in these sets is guaranteed to be atomic. These techniques are not concerned about oblivious interference and only partially address the pervasive interference by limiting the interference points to outside of specified atomic blocks. However, interference outside atomic blocks and between atomic blocks can still be pervasive [177]. An atomic block is an interference-free block of code.

Automatic mutual exclusion [2, 84, 154] inverts the model of atomic blocks in transactional memory

such that code is run atomically unless explicitly specified using yield expressions. Similarly, cooperative reasoning [176, 177] and observationally cooperative multithreading [157] limit interleaving points to yield expressions. Task Types [90] enforce pervasive atomicity, i.e. every piece of code must be in some atomic block, through a data-centric technique for specification of shared objects and syntactically explicated accesses to share objects. These techniques address the pervasive interference, however, they are not concerned about the oblivious interference problem.

Rely-guarantee and Owicki-Gries's work In rely-guarantee reasoning [87] a module satisfies a guarantee condition after each instruction and in turn can assume the rely condition satisfied by the environment. Previous work [68] leverages rely-guarantee reasoning for thread-modular verification of multithreaded programs in which rely and guarantee conditions are specified for threads and their environments. Similarly, in Owicki and Gries's work [129] and its variations [126] each instruction is annotated by an interference-free assertion which must hold locally in the presence of concurrent interfering tasks. These techniques extend Hoare logic and address the oblivious interference problem via environment assumptions, however, they still assume pervasive interference between each two instructions. Atomic actions [56] combine atomicity and rely-guarantee reasoning through iterative use of abstraction and reduction to infer atomic blocks. Rely-guarantee addresses oblivious interference, however, interferences outside atomic blocks and between each two atomic blocks can still be pervasive.

Concurrent separation logic and abstract predicates In concurrent separation logic [127] accesses to a shared resource are synchronized through mutual exclusion and guaranteed to preserve the resource invariants. Use of permissions enables read-sharing [31]. Concurrent separation logic can be treated as a specialization of rely-guarantee for well-synchronized programs [59]. Concurrent abstract predicates [47, 159] are self-stable predicates that combine separation logic with permissions to enable fine-grained modular reasoning about concurrent programs presenting a fiction of disjointness over shared resources. Resource specifications in these techniques limit the interference behavior and address oblivious interference, however, interferences among each two accesses to a shared resource can still be pervasive [177].

Aspect-orientation Interference between aspects and base code, its pervasiveness and obliviousness is discussed in aspect-oriented programming languages [61, 147]. However, solutions for interference problems of these sequential languages are not directly applicable to concurrent programming models.

Data race freedom There is a vast number of previous work on finding, fixing and preventing data races [30, 32, 33, 65, 144]. However, the absence of data races does not guarantee the absence of interferences and errors due to interferences [66].

Reconciliation of concurrency and modularity Panini and its capsule-oriented programming model follow previous work [106, 107, 135, 139] that suggests that modularity and concurrency goals are intertwined and could be reconciled.

CHAPTER 4. CONCURRENT BEHAVIORAL SUBTYPING

Behavioral subtyping is an important property for extensible software design, as it allows a type hierarchy to be extended without invalidating an already verified property [9, 96, 104, 130, 132]. This property, initially proposed for sequential programs, has been augmented to handle *thread-based* concurrency e.g. by adding history constraints [104], environment and resource specifications [47, 67, 87], and atomicity [148] and locking specifications [11, 148].

In this chapter, inspired by the recently found success of *module-like* linguistics abstractions for concurrency e.g. actors [6, 8], active objects [35, 38, 86, 151], capsules [18, 136, 138], etc., that promote more disciplined concurrency (compared to threads), we ask whether these linguistic abstractions can help promote a concurrent programming discipline where behavioral subtyping is the norm.

To explore these goals, we propose *concurrent behavioral subtyping* — or *concurrent subtyping* for short, a novel complementary foundation to previous work [9, 45, 47, 87, 96, 104, 130] on behavioral subtyping. Given a subtype σ and its supertype τ, concurrent subtyping defines behavioral subtyping as the combination of the following subtyping relations:

- ***interference subtyping***: the interference behaviors of the subtype σ and its supertype τ are compatible [130]; and
- ***interface subtyping***: the interface behaviors of σ and τ are compatible.

Interference and interface subtyping address interference and interface behavior incompatibilities between the supertype τ and its subtype σ and enable modular reasoning. Interface subtyping is similar to methods rule in standard behavioral subtyping [104].

Separation of the behavioral subtyping relation into these two, perhaps orthogonal, relations has a significant positive impact. For some linguistic abstractions, e.g. those that promote more disciplined concurrency, we find that the language design can imply interference subtyping and only the standard

interface subtyping [104] needs to be proven. Surprisingly, our empirical results finds that a majority of real world programs e.g. 354/416 programs written using Akka framework [8], a library for actors, can reap these benefits.

Contributions and Scope In summary, this chapter makes the following contributions:

- it proposes and formalizes *concurrent behavioral subtyping* as the combination of *interference subtyping* and *interface subtyping*.
- it shows how concurrent subtyping can enable modular supertype abstraction reasoning [97].
- it shows that with type encapsulation and encapsulated inheritance concurrent subtyping can be guaranteed using only the standard interface subtyping [96, 104].
- it presents an empirical study that examines the effectiveness of separating behavioral subtyping into two relations by measuring prevalence of scenarios where linguistic features can guarantee interference subtyping.

Throughout this chapter, we remain focussed on subtyping of functional behavior [95] in a concurrent program. Subtyping of either synchronization behavior (inheritance anomaly [109]) or communication behavior (session types [73]) is not the focus of this chapter.

4.1 Background

Modular reasoning in the presence of subtyping and dynamic dispatch is sound only if behaviors $\mathscr{B}_\sigma$ and $\mathscr{B}_\tau$ of the subtype σ and the supertype τ are compatible [9, 104]. In a concurrent program, arbitrary inheritance between a subtype σ and its supertype τ can cause incompatibilities [96, 104, 130] between behaviors of σ and τ and consequently break modular reasoning in a type η that interacts with τ and a type η' that τ interferes with.

To illustrate these two problems, we use the concurrent programming language Panini that provides module-like linguistic abstractions for concurrency [18, 136, 138]. The main features that we build upon are also present in a host of other concurrent languages with actors [6, 8, 74] such as Akka [8] and ActorFoundry [14] and Erlang [12], and active objects [35, 38] such as Creol [86] and JCoBox [151]

and even modeling languages such as ABS [46]. However, a framework for modular reasoning about Panini has also been developed by Bagherzadeh and Rajan [18] that we build upon here.

4.1.1 Panini

In Panini, a capsule is a type that encapsulates its state by making it accessible only through invocations of its procedures; a capsule instance executes its invoked procedures sequentially while running concurrently with other capsule instances in a program.

4.1.2 Program Syntax

Figure 4.1 shows the expression-based core syntax of Panini inspired by [18] and extended with inheritance. $\overline{term}$ denotes a sequence of zero or more terms and $[term]$ denotes an optional term.

A program is a set of capsule declarations that each contains a capsule name C, a supercapsule name D and declares a set of imported capsule instances $\overline{import}$, a design declaration *design*, a set of states $\overline{state}$ and procedures $\overline{proc}$. A capsule instance can asynchronously invoke procedures of its imported or locally declared capsule instances. An import declaration d has a capsule type D and name c. A design declaration declares a set of local capsule instances $\overline{instance}$ and connects them together in a wiring declaration *wiring*. A wiring declaration $c(\overline{d})$ exports capsule instances $\overline{d}$ to the imported capsule instances of c. Imported and exported capsule instances are shared.

Global asynchronous and local synchronous expressions A capsule encapsulates its state by making it available only through asynchronous invocation of its procedures [117]. A procedure declaration has a variable return type T, a name p, set of formal parameters $\overline{form}$ and a body e. The body of a capsule procedure is a sequence of *global asynchronous* and *local synchronous* expressions. In a global expression, a capsule can *asynchronously* invoke a procedure of another capsule including its supercapsule through the pseudo-variable ***super***. In asynchronous invocation the invoking capsule does not block for the execution of the invoked procedure to finish and does not wait for its result. A future [174] is returned as the result of the asynchronous invocation as a placeholder for a value which need not be immediately available. The value of a future can be claimed by blocking on it when needed using a ***deref*** expression. Due to concurrent execution of capsule instances in a program, an interference can happen

$prog ::= \overline{decl}$	program
$decl ::=$ ***capsule*** $C\ (\overline{import})$ ***extends*** D { $design\ \overline{state}\ \overline{proc}$ }	capsule declaration
$import ::= D\ d$	import declaration
$state ::= T\ s$;	state declaration
$proc ::=$ [$spec$] $T\ p$ ($\overline{T\ x}$) { e }	procedure declaration
$design ::=$ { $\overline{instance}\ \overline{wiring}$ } ;	design
$wiring ::= c\ (\overline{d})$	wiring
$instance ::= D\ d$	instance declaration
$e ::=$	*expression:*
| c.$p\ (\overline{x})\ ^{\alpha}$ | ***self***.$p\ (\overline{x})$ | ***super***.$p\ (\overline{x})$	procedure invocation
| ***self***.s | ***self***.s := x	state access
| ***ref*** x | ***deref*** x | $x := e$	references
| ***let*** x ***be*** e ***in*** e | ***if*** x ***then*** e ***else*** e	let, conditional
| ***deref*** x	future resolution
| x	variable
| $()$	unit value
$spec ::=$ ***requires*** ep ***ensures*** ep	specification
$ep ::=$	*pure predicates:*
| **true**	true
| x	variable read
| c.$p\ (\overline{x})$ | ***self***.$p\ (\overline{x})$	procedure invocation
| ep && ep | ***not*** ep	conjunction, negation
| ep == ep | $ep < ep$	equality, less than
| ***old*** (ep)	old predicate

$C, D \in \mathfrak{C}$	capsule type names	$T \in \mathfrak{T}$	variable types
$s \in \mathfrak{F}$	state names	$p \in \mathfrak{P}$	procedure names
$x \in \mathfrak{X}$	variable names	$c,d \in \mathfrak{I}$	instance names
$^{\alpha} \in \mathfrak{L}$	labels		

Figure 4.1 Panini's core syntax with capsule inheritance and procedure specifications.

at interference point $^{\alpha}$ after an asynchronous procedure invocation. In a local expression, a capsule can *synchronously* invoke a procedure of itself or access its state through the pseudo-variable ***self***.

The type system distinguishes between a capsule type C and a variable type T. Sets of possible values for capsule types and variable types are disjoint.

Atomic procedure specifications A procedure specification ***requires*** ep_1 ***ensures*** ep_2 includes a precondition predicate ep_1 and a postcondition ep_2 that both are pure and side-effect free. A specification predicate can be **true**, read variable x or it can invoke pure procedures of its enclosing capsule or other

capsules that the enclosing capsule encapsulates. Specification predicates can be compared for their values, conjoined or negated. The ***old*** predicate in a procedure specification refers to the state of its enclosing capsule at the beginning of the procedure execution. Procedure specifications are syntactically similar to JML specifications but are semantically different because they are atomic [18]. The specification ***requires*** ep_1 ***ensures*** ep_2 is atomic iff ep_1 and ep_2 predicates are free from interference. Procedure specifications are for partial correctness and do not specify termination [77, 95].

Type specifications There are no type specifications like invariants or history constraints [95, 104]. An invariant, similar to previous work [95, 103] can be desugared and conjoined into preconditions and postconditions of procedures of a capsule. Unlike previous work [103], concurrent subtyping does not require history constraints to guarantee behavioral subtyping.

4.1.3 Semantics in a Nutshell

A Panini program is a set of concurrently running capsule instances with the following semantics:

$\mathscr{P}_1$ ***Encapsulation:*** A capsule encapsulates its state and its *representation*, non shared local capsule instances, supercapsule and parameters of its procedures and their representations. The capsule makes what it encapsulates accessible only through asynchronous invocations of its procedures. The representation of a state is the object graph reachable from the state.

$\mathscr{P}_2$ ***Sequential execution:*** A capsule instance has a single thread of execution that dequeues and executes its invoked procedures from its queue sequentially [35]. The execution of an invoked procedure does not start before the execution of the currently running invoked procedure finishes [83].

$\mathscr{P}_3$ ***Typed messages:*** The procedures of a capsule constitute the set of messages it can accept [8, 14]. Invocation of a procedure of a capsule is the type-safe equivalent of sending a message to the capsule [86].

Encapsulation provided by a capsule can be guaranteed using various techniques such as copying [151], uniqueness [112] or ownership and ownership transfer [38, 39, 118]. Panini uses the latter to guarantee encapsulation.

$\mathscr{P}_4$ ***Ownership and ownership transfer:*** A capsule owns what it encapsulates. Upon a procedure invocation, the ownership of parameters of the procedure (and their representations) is transferred from the invoking to the invoked capsule instance; upon the return of the invocation result, the ownership of the result (and its representation) is transferred from the invoked to the invoking capsule instance.

Semantic properties $\mathscr{P}_1$–$\mathscr{P}_4$ are sufficient for defining concurrent subtyping. $\mathscr{P}_1$–$\mathscr{P}_4$ in turn guarantee properties $\mathscr{P}_5$–$\mathscr{P}_6$:

$\mathscr{P}_5$ ***Interference control:*** Interference can only happen after a procedure invocation on a shared capsule instance and the interference behavior at that interference point is bound by the behavior of its interference closure.

$\mathscr{P}_6$ ***Specification atomicity:*** Procedure specifications in a capsule are atomic if they only access the encapsulated state of the capsule.

Intuitively, the interference control and specification atomicity hold because the state encapsulated by a capsule can only be accessed by invocations of sequentially executing procedures of the capsule. The detailed formal proofs of these properties can be found in previous work [18].

Asynchronous execution increases the possibility of concurrency [86] and first in first out (FIFO) execution of invoked procedures of a capsule decreases the unnecessary nondeterminism in message passing concurrency [163]. However, $\mathscr{P}_7$–$\mathscr{P}_8$ are not necessary for concurrent subtyping.

$\mathscr{P}_7$ ***Asynchronous invocation:*** Asynchronous invocation of a procedure of a capsule instance puts the procedure in the capsule's queue and immediately returns a future value without waiting for the execution of the procedure.

$\mathscr{P}_8$ ***FIFO execution:*** The single execution thread of a capsule instance dequeues and executes its invoked procedures from its queue in a first in first out (FIFO) order.

Ownership guarantees Panini's sharing model.

$\mathscr{P}_9$ ***Sharing:*** The only shared entities among two capsule instances are either other capsule instances or unresolved future results of their procedure invocations. Access to an unresolved shared future is synchronized.

$\mathscr{P}_{10}$ ***Encapsulated inheritance:*** A subcapsule can access its supercapsule's state only through invocations of its procedures and not directly [155].

In summary, $\mathscr{P}_1$–$\mathscr{P}_4$ will be sufficient to define concurrent subtyping.

Illustration To illustrate, Figure 4.2 declares a capsule Counter with a state r and procedures inc and val to increment the counter by 1 and return its value. Another capsule Client imports a shared instance c of Counter, line 1, and in its main procedure asynchronously invokes Counter procedures inc and val and assigns their results to futures v1 and v2. The specification of inc in counter says that if the procedure execution starts in a program state that satisfies its precondition `true`, and the execution terminates, then it terminates in a state that satisfies its postcondition $c.val() == \textbf{\textit{old}}(c.val()) + 1$, i.e. inc increments the counter by one. The specification of val says that it is pure and does not change the value of the counter.

```
1  capsule Client(Counter c) {
2    int v1, v2;

4    //@ requires true;
5    //@ ensures v2 >= v1 + 1;                 Φ
6    void main() {
7      v1 = c.val() ;  α  κ(c,Counter) = {c.inc(),c.val()}*
8      c.inc() ;  α  κ(c,Counter) = {c.inc(),c.val()}*
9      v2 = c.val() ;  α  κ(c,Counter) = {c.inc(),c.val()}*
10   }
11 }
12 capsule Counter {
13   int r;

15   //@ requires true;
16   //@ ensures val() == old(val())+1;
17   void inc() { r = r + 1; }

19   //@ pure
20   int val() { return r; }
21 }
```

Figure 4.2 Interference points $^{\alpha}$ and interference closure $\kappa(c, Counter)$.

4.2 Key Challenges

Incompatibilities between behaviors of subtype σ and supertype τ can be divided into the following two fundamental problems:

❶ ***interference behavior incompatibility:*** the implementation of the subtype σ does not satisfy the interference behavior of its supertype τ; and

❷ ***interface behavior incompatibility:*** the implementation of an overriding procedure in the subtype σ does not satisfy the behavior of the procedure it overrides in its supertype τ.

The interference behavior of τ denotes its behavior when the execution of an instance of τ interferes with the concurrent execution of an instance of some type η. The interface behavior of τ is the behavior it exposes via its procedures and is used by η to interact with τ. An instance of η interacts with τ by invoking its procedures.

4.2.1 Problem ❶: Interference Behavior Incompatibility

A subcapsule σ with an interference behavior that is incompatible with the interference behavior of its supercapsule τ breaks modular reasoning in a capsule η that τ interferes with.

To illustrate, consider modular verification of the assertion Φ in Figure 4.2. Φ says that the implementation of the procedure main in the capsule Client satisfies its specification. To prove Φ, one should prove that if the execution of main starts in a program state that satisfies its precondition **true**, and the execution terminates, then it terminates in a program state that satisfies its postcondition $v2 >= v1 + 1$.

4.2.2 Modular Reasoning and Interference

Concurrency gives rise to thread interference and therefore any reasoning technique about a concurrent program should take into account the possibility of interference [18, 47, 87]. We first provide a short background on how modular reasoning is carried out in Panini in the presence of interference.

One way to take interference that is caused by concurrent execution of instances of Client and Counter into account when proving Φ is:

- to identify the program points in the implementation of main in which interference can happen (interference points); and
- for each interference point identify its interference behavior.

Panini's encapsulation and sequential execution of capsules guarantee that in a Panini program:

$\mathscr{G}_1$ interference can happen only after a procedure invocation on a *shared* capsule instance; and

$\mathscr{G}_2$ interference behavior at the procedure invocation (interference point) is limited to the Kleene closure of invocations of procedures of the static type of the invocation's receiver. We refer to this closure as interference closure.

The interference closure is an upper bound on the interference and therefore its behavior provides an upper bound on the interference behavior. Intuitively, this hold because the state encapsulated by a capsule instance can only be accessed by invocations of sequentially executing procedures of the capsule. More details and proofs about Panini's interference control above can be found in previous work [18].

Using these two properties one can easily take interference into account when proving Φ. To illustrate, Figure 4.2 shows the Counter and Client capsules with their interference points and interference closures. Using $\mathscr{G}_1$, there are no procedure invocations in the body of the procedure inc and therefore it is free from interference points and is atomic. A block of code is atomic if it is free from interference. Similarly, the procedure val is atomic. However, there are three interference points $^{\alpha}$ in the procedure main of Client because there are three procedure invocations on the shared capsule instance c, on lines 7–9. In Panini, an imported capsule instance is a shared capsule instance. Using $\mathscr{G}_2$, the interference closure at $^{\alpha}$ after the first invocation c.val(), on line 7, is the Kleene closure $\kappa(c,Counter) = \{c.inc(), c.val()\}^*$ of *all* procedures inc and val of the *static type* Counter of the receiver c. The closure $\kappa(c,Counter)$ is a set that is either empty $\emptyset$ or contains any sequential composition ; of any number of procedure invocations c.val() and c.inc() in any order. Invocations c.inc(), c.inc();c.val(), c.val();c.inc() or c.inc();c.inc() are examples of few elements that belong to the $\kappa(c,Counter)$. Other interference closures in main are computed similarly.

After identifying interference points $^{\alpha}$ and interference closures $\kappa(c,Counter)$ in main, one can construct the following Hoare-like triple $\mathscr{H}_{\Phi}$. The standard Hoare triple does not take interference into account [148]. To take interference into consideration, interference closures $\kappa(c,Counter)$ are inserted at interference points $^{\alpha}$ in the body of main. $\mathscr{H}_{\Phi}$ says that in the typing environment Γ if the execution of main starts in a program state that satisfies its precondition **true**, and the execution terminates, then it terminates in a program state that satisfies its postcondition $v2 >= v1 + 1$. This is exactly what the assertion Φ wants to prove. The static typing environment Γ for Client maps the counter instance c to

its static type Counter where the annotation *@import* denotes that c is a capsule instance shared between Client and other capsule instances in the system.

$$\mathscr{H}_\Phi : \Gamma \models \{\texttt{true}\} \qquad \text{precondition}$$

$$\begin{array}{ll} v1 = c.val(); & \{c.inc(), c.val()\}^*; \\ c.inc(); & \{c.inc(), c.val()\}^*; \\ v2 = c.val(); & \{c.inc(), c.val()\}^*; \end{array}$$

$\kappa(c, Counter)$: increase or not change c

$(\texttt{true}, c.val() >= \textbf{\textit{old}}(c.val()))$

$$\{v2 >= v1+1\} \qquad \text{postcondition} \quad \boxed{\checkmark}$$

$$\Gamma \vdash c : @import\ Counter$$

To prove Φ, recall that specifications of procedures of Counter say that inc increases the counter by 1 and val is pure and does not change the counter. Using these specifications, an oracle can conclude that the interference closure $\kappa(c, Counter) = \{c.val(), c.inc()\}^*$ either increases the counter by an arbitrary value or does not change it (but does not decrease it). The behavior of $\kappa(c, Counter)$ can be explained using the specification $(\texttt{true}, c.val() >= \textbf{\textit{old}}(c.val()))$ which says if the interference in the counter c starts in a program state that satisfies the precondition **true**, and its execution terminates, then it terminates in a program state that satisfies the postcondition $c.val() >= \textbf{\textit{old}}(c.val())$. The ***old*** predicate in this specification refers to the state of the counter c before the start of the interference. The behavior (ep_1, ep_2) is short for ***requires*** ep_1 ***ensures*** ep_2. Using this interference behavior and specification of Counter one can prove that the postcondition of $\mathscr{H}_\Phi$ and therefore the assertion Φ holds. In proving Φ, one should note that read accesses to values of futures v1 and v2 in the postcondition $v2 >= v1+1$ do not unblock until the executions of val in the counter c are executed and finished. Panini's first in first out execution model guarantees that in the postcondition of $\mathscr{H}_\Phi$, the execution of the first asynchronous invocation c.val() finishes before the start of the execution of c.inc() which itself finishes before the execution of the second invocation c.val() starts.

Next, we illustrate interference and interface behavior incompatibility problems of modular reasoning in the presence of interference and inheritance.

4.2.3 Interference Behavior Incompatibility

To illustrate, consider the addition of the subcapsule CxCounter in Figure 4.3 to the inheritance hierarchy of Counter. CxCounter adds an extra procedure dec that decreases the counter by one. The incompatibility between interference behaviors of CxCounter and Counter invalidates the assertion Φ, that was previously verified in the absence of CxCounter. In the presence of CxCounter and dynamic dispatch, invocations c.val() and c.inc() in the body of the procedure main in Client can be dispatched to an instance c of CxCounter instead of Counter. However, the interference closure $\kappa(c, CxCounter) = \{c.val(), c.inc(), c.dec()\}^*$ for c of type CxCounter is different from the interference closure $\kappa(c, Counter) = \{c.val(), c.inc()\}^*$ for c of type Counter and their behaviors are not compatible. Specifications $(\texttt{true}, \texttt{true})$ and $(\texttt{true}, c.val() >= \textbf{\textit{old}}(c.val()))$ denote the behaviors of interference closures $\kappa(c, CxCounter)$ and $\kappa(c, Counter)$, respectively. The behavior of $\kappa(c, Counter)$ either increases the counter or not change it, whereas the behavior of $\kappa(c, CxCounter)$ with its newly added dec procedure can not only increase or not change the counter but also decrease it by any arbitrary value, which is incompatible with the behavior of $\kappa(c, Counter)$.

```
22  capsule CxCounter extends Counter {
23    int i;
24    void dec() {  r = r - 1; }
25  }
```

Figure 4.3 Interference behavior incompatibility breaks modular reasoning.

A non modular verifier with the knowledge about the subcapsule CxCounter would construct the following Hoare triple $\mathscr{H}'_\Phi$ instead of the previous triple $\mathscr{H}_\Phi$. However, unlike $\mathscr{H}_\Phi$, the postcondition of $\mathscr{H}'_\Phi$ and therefore the assertion Φ cannot be proved anymore. The incompatibility of the interference behavior of the subcapsule CxCounter with the interference behavior of its supercapsule Counter breaks modular reasoning in the capsule Client that the supercapsule Counter interferes with. The dynamic typing environment Π of Client maps its counter instance c to its dynamic type CxCounter.

$$\mathscr{H}'_{\Phi} : \Gamma \models \{\texttt{true}\}$$

$$\begin{array}{ll} & \overbrace{\hphantom{\{c.inc(), c.val(), c.dec()\}^*}}^{\text{decrease, increase or not change c}} \\ v1 = c.val(); & \{c.inc(), c.val(),\ c.dec()\ \}^* \\ c.inc(); & \{c.inc(), c.val(),\ c.dec()\ \}^* \\ v2 = c.val(); & \underbrace{\{c.inc(), c.val(),\ c.dec()\ \}^*}_{(\texttt{true}, \texttt{true})} \end{array}$$

$$\{v2 >= v1 + 1\} \qquad \text{✗}$$

$$\Gamma \vdash c : @import\ Counter$$

$$\Pi \vdash c : CxCounter$$

4.2.4 Problem ❷: Incompatible Interface Behavior

A subcapsule σ with an interface behavior that is incompatible with the interface behavior of its supercapsule τ breaks modular reasoning in

1 a type η that τ interferes with; and

2 a type η that interfaces with τ by invoking its procedures.

Incompatible interface behavior causes incompatible interference behavior We first discuss the case in which the incompatibility of interface behaviors of σ and τ breaks the modular reasoning in a type η that τ interferes with. To illustrate, reconsider the assertion Φ that was previously verified in Figure 4.2 in the absence of CxCounter and consider the addition of the subcapsule CxCounter in Figure 4.4 to the inheritance hierarchy of Counter. CxCounter overrides the procedure inc of Counter.

```
22  capsule CxCounter extends Counter { ..
23    void inc() { r = 0;  }
24  }
```

Figure 4.4 Interface behavior incompatibility breaks modular reasoning.

After the addition of the subcapsule CxCounter, the incompatibility between the interface behaviors of CxCounter and Counter invalidates the assertion Φ. In the presence of CxCounter and dynamic dispatch, invocations c.inc() in the procedure main in Client can be dispatch to an instance c of CxCounter instead

of Counter. However, the interface behavior $\iota(c, CxCounter) = \{inc(\mathtt{true}, c.val() == 0), val(pure)\}$ for an instance c of CxCounter is not compatible with the interface behavior $\iota(c, Counter) = \{inc(\mathtt{true}, c.val() == \boldsymbol{old}(c.val()) + 1, val(pure)\}$ for c of Counter for the procedure inc. The interface behavior of CxCounter includes behaviors $(\mathtt{true}, c.val() == 0)$ and *pure* of its procedures *inc* and *val*. The behavior $(\mathtt{true}, c.val() == 0)$ of the procedure *inc* specifies its precondition **true** and the postcondition $c.val() == 0$. Unlike inc in Counter that increases the counter by one, inc in CxCounter resets the counter. This interface behavior incompatibility, causes incompatibility between the behavior of the interference closure $\kappa(c, CxCounter) = \{c.val(), c.inc()\}^*$ for CxCounter and the behavior of the interference closure $\kappa(c, Counter) = \{c.val(), c.inc()\}^*$ for Counter. Behaviors $(\mathtt{true}, c.val() == 0)$ and $(\mathtt{true}, c.val() >= \boldsymbol{old}(c.val()))$ denote behaviors of $\kappa(c, CxCounter)$ and $\kappa(c, Counter)$. Unlike $\kappa(c, Counter)$ that either increases the counter or not change it, $\kappa(c, CxCounter)$ with its overriding procedure inc either resets the counter or not change it.

Interface incompatibility breaks modular reasoning The case in which the incompatibility of interface behaviors of σ and τ breaks the modular reasoning in a type η that interacts with τ is standard and is recognized by previous work [9, 104]. To illustrate it in the context of a concurrent langauage, consider Figure 4.5 that declares a variation of Client that, unlike Figure 4.2, declares a local and non shared instance c of Counter, in its design declaration on line 2. In Panini, a local capsule instance is declared in a design declaration.

```
1  capsule Client {  ..
2    design {Counter c; }

4    //@ requires true;
5    //@ ensures v2 == v1 + 1;              Φ
6    void main() {
7      v1 = c.val() ;
8      c.inc() ;
9      v2 = c.val() ;
10   }
11 }
```

Figure 4.5 Incompatible interface behavior breaks sequential modular reasoning.

Using $\mathscr{G}_1$, the procedure main in Client is atomic because there is no procedure invocation on a shared capsule instance in its body anymore. In the absence of a subcapsule CxCounter, one can verify the new

assertion Φ by constructing the following Hoare triple $\mathscr{H}_\Phi$:

$$
\begin{array}{lll}
\mathscr{H}_\Phi : \Gamma \models \{\texttt{true}\} & & \\
& v1 = c.val(); & \overbrace{\text{no interference}}^{\kappa(c, Counter) = \emptyset} \\
& c.inc(); & \text{no interference} \\
& v2 = c.val(); & \text{no interference} \\
& \{v2 == v1 + 1\} & \qquad\qquad \boxed{\checkmark} \\
\\
\Gamma \vdash c : @local\ Counter & &
\end{array}
$$

The procedure main is atomic and therefore the interference closure $\kappa(c, Counter)$ and its behavior are trivially empty. Atomicity of main allows the use of standard sequential reasoning [77] for the verification of $\mathscr{H}_\Phi$. $\mathscr{H}_\Phi$ and consequently Φ holds because specifications of inc and val in Counter say that they only increase the counter by one or not change it.

However, after the addition of CxCounter in Figure 4.3, the incompatibility between interface behaviors of CxCounter and Counter invalidates the assertion Φ. This is because unlike the behavior $(\texttt{true}, c.val() == \textbf{\textit{old}}(c.val()) + 1)$ of inc in $\iota(c, Counter)$ which increases the counter, the behavior $(\texttt{true}, c.val() == 0)$ of inc in $\iota(c, CxCounter)$ resets the counter.

4.3 Concurrent Subtyping

Concurrent subtyping holds if both interference and interface behaviors of a subtype σ are compatible with interference and interface behaviors of its supertype τ. Concurrent subtyping addresses incompatible interference and interface behavior problems and enables modular reasoning. In this section, we formalize concurrent subtyping.

4.3.1 Formalization of Concurrent Subtyping

Concurrent subtyping defines behavioral subtyping as the combination of interference and interface subtyping. Definition 7 and Definition 8 define interference closure and concurrent subtyping.

Definition 7 ***(Interference closure)*** *Let C be a capsule type with procedures $T_1\ p_1\ (\overline{x_1})\ ..\ T_n\ p_n\ (\overline{x_n})$ in its interface, including procedures inherited from its supercapsules. Let c be a capsule instance with*

the static type C, i.e. $\Gamma \vdash c : C$*, and a possibly different dynamic type* C'*. Then, the interference closure of c, written as* $\kappa(c,C)$*, is the Kleene closure of sequential composition ; of any number of invocations of all procedures in the interface of its static type C in any arbitrary order:*

$$\kappa(c,C) = \{c.p_1(_), c.p_2(_), .., c.p_n(_)\}^*$$

The notation _ denotes an unspecified arbitrary value which is of the expected type of the procedure argument and does not invalidate the procedure specification.

The closure $\kappa(c,C)$ denotes an upper bound on the interference on an instance c that a concurrently running instance of D can cause.

Definition 8 ***(Concurrent subtyping)*** *Let* C' *and C be two capsule types in a Panini program where* C' *is a subcapsule of C in its inheritance hierarchy, written as* $C' \prec C$*. Let c be a capsule instance of static type C in a static typing environment* Γ*, i.e.* $\Gamma \vdash c : C$*, and of dynamic type* C' *in the dynamic typing environment* Π*, i.e.* $\Pi \vdash c : C'$*. Let* ep_1 *and* ep_2 *be atomic specification predicates. Let the closure* $\kappa(c,C)$ *be the interference closure of C. Then,* C' *is a concurrent subtype of C iff:*

1 C' *is an interference subtype of C:*

$$\forall ep_1, ep_2 \in ep\,\boldsymbol{.}\quad \Gamma \vdash \{ep_1\}\ \kappa(c,C)\ \{ep_2\} \quad \Rightarrow \quad \Gamma \vdash \{ep_1\}\ \kappa(c,C')\ \{ep_2\}$$

2 C' *is an interface subtype of C:*

$$\forall p \in \mathfrak{P}, ep_1, ep_2 \in ep,\ override([C', T\ p(\bar{x})\{e'\}], [C, T\ p(\bar{x})\{e\}])\,\boldsymbol{.}$$
$$\Gamma \vdash \{ep_1\}\ e\ \{ep_2\} \quad \Rightarrow \quad \Gamma \vdash \{ep_1\}\ e'\ \{ep_2\}$$

A subcapsule C' is a concurrent subtype of its supercapsule C if C' is the interference subtype and interface subtype of C. C' is an interference subtype of its supercapsule C only if the interference behavior of C' is compatible with the interference behavior of C. Similarly, C' is an interface subtype of C if the

interface behavior of C' is compatible with the interference behavior of C. Similar to standard behavioral subtyping [9, 44, 96, 104], concurrent subtyping does not provide any guarantees about program termination. The function *override* checks if a procedure p in C' is overriding the procedure p in C.

An oracle knows the behavior (ep_1, ep_2) of an interference closure $\kappa(c,C)$ of a type C. The behavior (ep_1, ep_2) for an overridden procedure of C can be its behavior *spec* specified by its programmer.

4.3.1.1 Interference Subtyping

In Definition 8, the subcapsule C' is an interference subtype of its supercapsule C if the interference behavior of its interference closure $\kappa(c,C')$ is compatible with the behavior of the interference closure $\kappa(c,C)$ of its supercapsule C:

$$\forall ep_1, ep_2 \in ep \,\text{.}$$
$$\Gamma \vdash \{ep_1\}\ \kappa(c,C)\ \{ep_2\} \quad \Rightarrow \quad \Gamma \vdash \{ep_1\}\ \kappa(c,C')\ \{ep_2\}$$

To be compatible, the interference closure $\kappa(c,C')$ of the subcapsule C' should preserve *any behavior* (ep_1, ep_2) that the interference closure $\kappa(c,C)$ of the supercapsule C satisfies. The Hoare judgement $\Gamma \vdash \{ep_1\}\ \kappa(c,C)\ \{ep_2\}$ above denotes the satisfaction of the interference behavior (ep_1, ep_2) by the interference closure $\kappa(c,C)$ in the static typing environment Γ [77]. The interference closure $\kappa(c,C)$ satisfies its behavior (ep_1, ep_2) if the execution of the closure starts in a program state that satisfies ep_1, and it terminates, then it terminates in a state that satisfies ep_2.

To illustrate, the subcapsule CxCounter in Figure 4.3 is not the interference subtype of its supercapsule Counter:

$$\Gamma \vdash \{\mathtt{true}\}\ \kappa(c, Counter)\ \{c.val() >= \textbf{\textit{old}}(c.val())\}$$
$$\not\Rightarrow$$
$$\Gamma \vdash \{\mathtt{true}\}\ \kappa(c, CxCounter)\ \{c.val() >= \textbf{\textit{old}}(c.val())\}$$

This is because the behavior $(\mathtt{true}, \mathtt{true})$ of $\kappa(c, CxCounter) = \{c.inc(), c.val(), c.dec()\}^*$ of CxCounter is not compatible with the behavior $(\mathtt{true}, c.val() >= \textbf{\textit{old}}(c.val()))$ for $\kappa(c, CxCounter) = \{c.inc(), c.val()\}^*$ of its supercapsule Counter.

However, the subcapsule CxCounter in Figure 4.6 is an interference subtype of Counter. This is because both interference closures $\kappa(c, Counter)$ and $\kappa(c, CxCounter)$ have the same and therefore trivially compatible behaviors $(\texttt{true}, c.val() >= \textbf{\textit{old}}(c.val()))$:

$$\Gamma \vdash \{\texttt{true}\}\ \kappa(c, Counter)\ \{c.val() >= \textbf{\textit{old}}(c.val())\}$$

$$\Rightarrow$$

$$\Gamma \vdash \{\texttt{true}\}\ \kappa(c, CxCounter)\ \{c.val() >= \textbf{\textit{old}}(c.val())\}$$

It is interesting to see that CxCounter is an interference subtype of Counter while their inc procedures have different behaviors. inc in CxCounter increases the counter by 2 whereas inc in Counter increases the counter by 1.

```
22  capsule CxCounter extends Counter { ..
23    void inc() { r = r + 2; }
24  }
```

Figure 4.6 CxCounter is an interference subtype of Counter.

4.3.1.2 Interface Subtyping

In Definition 8, the subcapsule C' is an interface subtype of its supercapsule C if the behavior of the body e' for each of its overriding procedures is compatible with the behavior of the body e of the procedure it overrides:

$$\forall p \in \mathfrak{P}, ep_1, ep_2 \in ep, override([C', T\ p(\overline{x})\{e'\}], [C, T\ p(\overline{x})\{e\}])\ \centerdot$$
$$\Gamma \vdash \{ep_1\}\ e\ \{ep_2\} \quad \Rightarrow \quad \Gamma \vdash \{ep_1\}\ e'\ \{ep_2\}$$

The Hoare judgement $\Gamma \vdash \{ep_1\}\ e\ \{ep_2\}$ above denotes satisfaction of the procedure behavior (ep_1, ep_2) by the expression e in the static typing environment Γ. The expression e satisfies the behavior (ep_1, ep_2) if its execution starts in a program state that satisfies ep_1, and it terminates, then it terminates in a program state that satisfies ep_2. Interface subtyping, similar to the methods rule in standard behavioral subtyping [9, 44, 104] relates the explicit preconditions and postconditions of an overriding and overridden procedure in a subtype and its supertype by requiring a contravariance and covariance relations between preconditions and postconditions respectively. Variations of interface subtyping [96] with weaker covariance requirements for postconditions can also be used.

To illustrate, the subcapsule CxCounter in Figure 4.4 is not the interface subtype of its supercapsule Counter:

$$override([CxCounter, \textbf{\textit{void}}\ inc()\{r = 0\}],$$
$$[Counter, \textbf{\textit{void}}\ inc()\{r = r+1\}]).\ \ \Gamma \vdash \{\texttt{true}\}\ r = r+1\ \{c.val() == \textbf{\textit{old}}(c.val())+1\}$$
$$\not\Rightarrow$$
$$\Gamma \vdash \{\texttt{true}\}\ r = 0\ \{c.val() == 0\}$$

This is because the behavior $(\texttt{true}, c.val() == 0)$ of the overriding procedure inc in CxCounter is not compatible with the behavior $(\texttt{true}, c.val == \textbf{\textit{old}}(c.val()) + 1)$ of its overridden procedure inc in its supercapsule Counter.

However, the subcapsule CxCounter in Figure 4.3 is an interface subtype of Counter trivially because it does not override any procedure of Counter.

Neither interference nor interface subtyping alone are sufficient for concurrent subtyping A subcapsule C' can be an interference subtype of its supercapsule C but not its interface subtype. Similarly, C' can be an interface subtype of C but not its interference subtype. This shows that neither interference subtyping nor interface subtyping alone is sufficient to guarantee concurrent subtyping.

To illustrate, the subcapsule CxCounter in Figure 4.6 is an interference subtype of its supercapsule Counter but is not its interface subtype. Similarly, the CxCounter in Figure 4.3 is an interface subtype of Counter but not its interference type.

4.3.2 Sound Modular Reasoning

Now we show a Hoare-style logic [77] that uses concurrent subtyping to enable modular supertype abstraction reasoning [97] in the presence of interference and inheritance. In supertype abstraction reasoning, an invocation of a procedure is understood using the static type C of its receiver capsule instance and independent of its possibly different dynamic type C' used to dispatch the invocation. Figure 4.7 shows select rules in our Hoare logic. The reasoning rules for atomic expressions that are free from interference are similar to sequential reasoning rules in previous work [3,77] and are omitted.

In the reasoning rules, the judgement $\Gamma \vdash \{ep_1\}\ e\ \{ep_2\}$ denotes that the Hoare triple $\{ep_1\}\ e\ \{ep_2\}$ is provable using the static typing environment Γ. The reasoning rules use a fixed capsule table CT

(V-PROC INVOC)

$$\frac{\begin{array}{c}\Gamma \vdash c : C \qquad \textbf{\textit{capsule}}\ C\ \textbf{\textit{extends}}\ ..\ \{..\ spec\ T\ p\ (\overline{x})\{e\}\ ..\ \} \in CT \\ spec = \textbf{\textit{requires}}\ ep_1\ \textbf{\textit{ensures}}\ ep_2 \qquad ep'_1 = ep_1[\overline{v}/\overline{x}, c/\textbf{\textit{self}}] \\ ep'_2 = ep_2[\overline{v}/\overline{x}, c/\textbf{\textit{self}}] \qquad \Gamma \vdash \{ep'_2\}\ \kappa(c,C)\ \{ep''_2\} \qquad \Gamma \vdash \{ep'_1\}\ c.p(\overline{v}); \kappa(c,C)\ \{ep''_2\}\end{array}}{\Gamma \vdash \{ep'_1\}\ c.p(\overline{v})^{\alpha}\ \{ep''_2\}}$$

(V-SELF INVOC)

$$\frac{\begin{array}{c}\Gamma \vdash \textbf{\textit{self}} : C \qquad \textbf{\textit{capsule}}\ C\ \textbf{\textit{extends}}\ ..\{..\ spec\ T\ p\ (\overline{x})\{e\}\ ..\ \} \in CT \\ spec = \textbf{\textit{requires}}\ ep_1\ \textbf{\textit{ensures}}\ ep_2 \qquad ep'_1 = ep_1[\overline{v}/\overline{x}] \qquad ep'_2 = ep_2[\overline{v}/\overline{x}]\end{array}}{\Gamma \vdash \{ep'_1\}\ \textbf{\textit{self}}.p(\overline{v})\ \{ep'_2\}}$$

(V-SUPER INVOC)

$$\frac{\begin{array}{c}\Gamma \vdash \textbf{\textit{self}} : C' \qquad \Gamma \vdash C' \prec C \qquad \textbf{\textit{capsule}}\ C\ \textbf{\textit{extends}}\ ..\{..\ spec\ T\ p\ (\overline{x})\{e\}\ ..\ \} \in CT \\ spec = \textbf{\textit{requires}}\ ep_1\ \textbf{\textit{ensures}}\ ep_2 \qquad ep'_1 = ep_1[\overline{v}/\overline{x}] \qquad ep'_2 = ep_2[\overline{v}/\overline{x}]\end{array}}{\Gamma \vdash \{ep'_1\}\ \textbf{\textit{super}}.p(\overline{v})\ \{ep'_2\}}$$

(V-CONSEQUENCE)

$$\frac{ep_1 \Rightarrow ep'_1 \qquad ep'_2 \Rightarrow ep_2 \qquad \Gamma \vdash \{ep'_1\}\ e\ \{ep'_2\}}{\Gamma \vdash \{ep_1\}\ e\ \{ep_2\}}$$

Figure 4.7 Select rules for Panini's Hoare logic.

which is a set of program's capsule declarations [82]. The notation $ep[\overline{v}/\overline{x}]$ denotes replacing variables $\overline{x}$ with values $\overline{v}$.

The inference rule (V-PROC INVOC) reasons about the invocation of a procedure on a capsule instance c with the static type C and a possibly different dynamic type. The rule says that the behavior of the invocation of a procedure p is the behavior (ep'_1, ep'_2) of the procedure combined with the behavior (ep'_2, ep''_2) of the interference closure $\kappa(c,C)$ for its receiver c. This is because in Panini interference can happen after each procedure invocation point $^{\alpha}$ and the interference behavior is bound by the behavior of the interference closure $\kappa(c,C)$. (ep'_1, ep'_2) is the behavior (ep_1, ep_2) of the procedure p after substitutions for its parameters and the pseudo-variable ***self***. The key in this rule is that, independent of the dynamic type of the receiver which maybe different from C, the rule uses the static type C of the receiver c to retrieve the behavior (ep'_1, ep'_2) of the procedure p and compute the interference behavior (ep'_2, ep''_2) of the receiver. Reasoning about the procedure invocation using the static type and independent of the dynamic type of its receiver is sound only because concurrent subtyping guarantees both interference

and interface subtyping.

(V-SELF INVOC) reasons about the invocation of a procedure of a capsule itself. The rule says that the behavior of the self invocation of a procedure p of a capsule C is the same as the behavior (ep'_1, ep'_2) of the procedure, after appropriate substitutions. Unlike (V-PROC INVOC), the behavior of the self invoked procedure does not need to be combined with any interference behavior. This is because the invocation of a self procedure is executed synchronously. Similarly, (V-SUPER INVOC) reasons about the invocation of a procedure in a supercapsule. The rule says that the behavior of the invocation of a procedure p of a supercapsule C is the same as the behavior (ep'_1, ep'_2) of the procedure, after appropriate substitutions. Again there is no combination of the procedure behavior with any interference behavior. This is because in Panini a capsule owns and controls access its supercapsule, similar to its local capsule instances, and can access its supercapsule's state free from interference. by its (V-CONSEQUENCE) is the standard consequence rule [77].

4.3.2.1 Soundness

Modular reasoning using the Hoare logic in Figure 4.7 is sound because:

- the behavior of a capsule C is divided into interference and interface behaviors; and
- concurrent behavioral subtyping guarantees that a subcapsule C' is both interference and interface subtype of its supercapsule C; and consequently
- interference behaviors of C and C' are compatible for a capsule D that C interfers with and interface behaviors of C and C' are compatible for a capsule D that interact with C by invoking its procedures.

Therefore, any property Φ reasoned about and verified using the behavior of the supercapsule C and the rules in Figure 4.7 will not be invalidated by any of its existing subcapsules C' or any subcapsules C' added in the future. Theorem 9 formalizes soundness of our reasoning Hoare logic.

Theorem 9 ***(Soundness of reasoning)*** *The reasoning Hoare logic in Figure 4.7 is sound. That is, any Hoare tripe provable using this Hoare logic, written as* $\Gamma \vdash \{ep_1\}\ e\ \{ep_2\}$*, is a valid Hoare triple, written as* $\Gamma \models \{ep_1\}\ e\ \{ep_2\}$*.*

Proof Sketch: The proof is based on the induction over the expressions in the syntax in Figure 4.1 and uses interference and interface subtyping guarantees of concurrent subtyping. The judgement $\Gamma \models \{ep_1\}\ e\ \{ep_2\}$ denotes that the provable triple $\{ep_1\}\ e\ \{ep_2\}$ is valid in Γ if for every state $\int$ that agrees with type environment Γ, if ep_1 is true in a program state $\int$, i.e. $\int \models ep_1$, and if the execution of e terminates in a state $\int'$, then $\int' \models ep_2$. The validity is for partial correctness and does not say anything about the termination [153].

4.4 Proving Concurrent Subtyping

To prove that a subcapsule C' is a concurrent subtype of its supercapsule C requires the following proofs according to Definition 8:

1 ***Interference subtyping***: all overriding and extra procedures of C' provide a proof that C' is an interference subtype of C; and

2 ***Interface subtyping***: every overriding procedure of C' provides a proof that C' is an interface subtype of C.

The following key observations allow for an alternative way to prove concurrent subtyping:

$\mathscr{O}_1$ in the presence of type encapsulation [117] and encapsulated inheritance [155], to prove that C' is an interference subtype of C it is sufficient to prove that C' is a standard interface subtype [45,104] of C.

$\mathscr{O}_2$ consequently and more importantly, to prove that C' is a concurrent subtype of C it is sufficient to prove that C' is an interface subtype of C.

The standard interface subtyping and its variations [10,104] relate behaviors of an overriding procedure in a subtype and the procedure it overrides in the supertype by requiring contravariance and covariance relations on their preconditions and postconditions respectively. Variations of interface subtyping [44, 96] weaken the covariance rule for postconditions.

Using $\mathscr{O}_2$, in a concurrent language with type encapsulation and encapsulated inheritance, a programmer can simply prove the standard interface subtyping and in return reap the benefits of concurrent

subtyping without the need to be worried about interference and interference subtyping. This in turn could help foster concurrent programming languages in which behavioral subtyping is the norm.

4.4.1 Encapsulated Inheritance

Encapsulated inheritance makes the state and its representation of a supercapsule C accessible to its subcapsule C' only through invocations of procedures of C [155]. This is in contrast to standard inheritance mechanisms in languages such as Java in which a subclass can directly access fields of its superclass. Definition 11 defines encapsulated inheritance. Definition 10 defines extended state of a capsule used in Definition 11.

Definition 10 ***(Extended state)*** *Let C' be a capsule with the declaration* **capsule** $C'(\overline{import})$ ***extends*** C $\{design\ \overline{state}\ \overline{proc}\}$*. The extended state of C', written as $XState(C')$, is a set of memory locations including its state $\overline{state}$ and its representation, extended state of its supercapsule C' and extended states of local non shared capsule instances declared in its design declaration.*

A locally declared instance $G\ g$ in C' is not shared if there is no wiring declaration $h(g)$ in *design* declaration of C' that exports g to h. Extended state of C' is the set of memory locations that are not shared by C' and can be accessed by the capsule without any interference. Imported or exported capsule instance of C' are shared and therefore are not part of its extended state.

To illustrate, the extended state of the capsule `Client` in Figure 4.2 includes its states `v1` and `v2` but does not include states of its imported counter instance `c`. However, the extended state of `Client` in Figure 4.5, in addition to `v1` and `v2`, includes the states of the locally declared counter instance `c`.

Definition 11 ***(Encapsulated inheritance)*** *Let C' and C be two capsule types where C' is a subtype of C, i.e. $C' \prec C$. Let $XState(C)$ be the extended state of C. Then, C' is a subcapsule of C over an encapsulated inheritance hierarchy, written as $C' \prec_{\mathscr{E}} C$, iff there is no direct access in procedures of C' to the extended state of C.*

4.4.2 Proving Interference Subtyping

In the presence of type encapsulation and encapsulated inheritance, a subcapsule C' is the interference subtype of its supercapsule C if C' is an interface subtype of C. Theorem 12 formalizes this.

Theorem 12 ***(Interference subtyping is guaranteed by encapsulated inheritance and interface subtyping)*** *Let C' and C be two capsule types where C' is a subtype of C over its encapsulated inheritance hierarchy, i.e. $C' \prec_{\mathscr{E}} C$; Let C' be an interface subtype of C. Then C' is an interference subtype of C.*

Proof : Let c be a capsule instance of static type C, i.e. $\Gamma \vdash c : C$ and dynamic type C', i.e. $\Pi \vdash c : C'$. According to Definition 8, to prove that C' is an interference subtype of C we need to prove that:

$$\forall ep_1, ep_2 \in ep \,\bullet\, \Gamma \vdash \{ep_1\}\ \kappa(c,C)\ \{ep_2\} \quad \Rightarrow \quad \Gamma \vdash \{ep_1\}\ \kappa(c,C')\ \{ep_2\}$$

Recall that the interference closure $\kappa(c,C')$ of C' is a set of sequential compositions of any number of invocations of the procedures of C' in any order. A procedure in the interface of the subcapsule C' can be: *(1)* a procedure p declared in its supercapsule C and not overridden in C' or *(2)* an overrding procedure $p_{o'}$ in C' or *(3)* an extra procedure p_x. For *(1)*, an invocation of p dispatched to C' is included in $\kappa(c,C)$ that satisfies the behavior (ep_1, ep_2) and therefore such invocation cannot invalidate (ep_1, ep_2). For *(2)*, the assumed interface subtyping guarantees that the behavior of $p_{o'}$ in C' is compatible with the behavior of the procedure p_o it overrides in C and therefore an invocation of $p_{o'}$ preserves the behavior (ep_1, ep_2) that $\kappa(c,C)$ satisfies. For *(3)*, encapsulated inheritance guarantees that to access the extended state $XState(C)$ of the supercapsule C, p_x must go through procedures of C where invocations of these procedures is included in $\kappa(c,C)$ which satisfies (ep_1, ep_2). p_x can directly access $XState(C') \backslash XState(C)$, however, Lemma 5 shows that $XState(C)$ is disjoint from $XState(C') \backslash XState(C)$ and therefore any modifications to $XState(C') \backslash XState(C)$ by invocation of p_x does not invalidate the behavior (ep_1, ep_2) over $XState(C)$.

A downcall from a supertype to an overriding procedure of its subtype via the pseudo-variable ***self*** is not allowed.

Lemma 5 ***(Disjoint extended states)*** *Let C and C' be two different capsule types with extended states $XState(C)$ and $XState(C')$. If C and C' do not belong to the same inheritance hierarchy, i.e. $C \not\prec_{\mathscr{E}} C'$ and $C' \not\prec_{\mathscr{E}} C$, then their extended states are disjoint, written as $XState(C)\ \#\ XState(C')$. Otherwise, if they are on the same inheritance hierarchy, i.e. $C' \prec_{\mathscr{E}} C$, then the extended state C is disjoint from the extended state of C' minus the extended state of C, i.e. $XState(C)\ \#\ (XState(C') \backslash XState(C))$.*

Proof Sketch: The proof is based on type and inheritance encapsulation of capsules.

4.4.3 Proving Concurrent Subtyping

In the presence of type encapsulation and encapsulated inheritance, a subcapsule C' is the concurrent subtype of its supercapsule C iff C' is an interface subtype of C. Theorem 13 formalizes this.

Theorem 13 ***(Concurrent subtyping is guaranteed by encapsulated inheritance and interface subtyping)*** *Let C' and C be two capsule types where C' is a subcapsule of C over its encapsulated inheritance hierarchy, i.e. $C' \prec_{\mathscr{E}} C$. Let C' be an interface subtype of C. Then C' is a concurrent subtype of C.*

Proof : The proof is straightforward using Theorem 12 and Definition 8.

Proving interface subtyping To prove that a subcapsule C' is an interface subtype of its sueprcapsule C one can use various techniques proposed by previous work [9, 104] including specification inheritance [45]. Specification inheritance combines behaviors of C and C' by disjoining preconditions and conjoining postconditions of an overriding procedure and the procedure it overrides. The combined specification becomes the new specification of the subcapsule C'. The key in enforcing interface subtyping using specification inheritance is that specification inheritance *automatically* guarantees that C' is an interface subtype of C. This in turn may help more in making behavioral subtyping in concurrent languages prevalent. Supporting specification inheritance for capsules is straightforward, especially since their procedure specifications are atomic.

4.5 Semantics

4.5.1 Static Semantics

Figure 4.8 adds capsule subtyping to Panini's type system. with most other rules being straightforward and similar to rules without capsule subtyping discussed in Chapter 3.

4.5.2 Operational Semantics

Figure 4.9 shows dynamic semantics of Panini. The new rule (SUPER INVOC) delegates an invocation to a supercapsule as an asynchronous invocation and in turn guarantees encapsulated inheritance. Other rules are similar to Panini's operational semantics without capsule inheritance.

(T-PROC INVOC)

$$\frac{\Gamma,\Pi \vdash c : C \qquad T'\ p(\overline{T\ x})\{e\} = procBody(C,p) \qquad \forall x_k \in \overline{x}, T_k \in \overline{T}\ \textbf{.}\ \Gamma,\Pi \vdash \overline{x_k} : \overline{T_k}}{\Gamma,\Pi \vdash c.p(\overline{x}) : T'}$$

(T-RESOLVE)

$$\frac{\Gamma,\Pi \vdash e : T \qquad T = \Pi(l)}{\Gamma,\Pi \vdash \textbf{\textit{resolve}}(l,e,id,p) : T}$$

(T-SELF INVOC)

$$\frac{\Gamma,\Pi \vdash \textbf{\textit{self}} : C \quad \textbf{\textit{capsule}}\ C(..)\ \textbf{\textit{extends}}\ D\{..\ T'\ p(\overline{T\ x})\{e\}\ ..\} = CT(C) \qquad \forall x_k \in \overline{x}, T_k \in \overline{T}\ \textbf{.}\ \Gamma,\Pi \vdash \overline{x_k} : \overline{T_k}}{\Gamma,\Pi \vdash \textbf{\textit{self}}.p(\overline{x}) : T'}$$

(T-SUPER INVOC)

$$\frac{\Gamma,\Pi \vdash \textbf{\textit{super}} : C \quad \textbf{\textit{capsule}}\ C(..)\ \textbf{\textit{extends}}\ ..\{..\ T'\ p(\overline{T\ x})\{e\}\ ..\} = CT(C) \qquad \forall x_k \in \overline{x}, T_k \in \overline{T}\ \textbf{.}\ \Gamma,\Pi \vdash \overline{x_k} : \overline{T_k}}{\Gamma,\Pi \vdash \textbf{\textit{super}}.p(\overline{x}) : T'}$$

(T-SUB VAR)

$$\frac{\Gamma,\Pi \vdash e : T' \qquad T' <: T}{\Gamma,\Pi \vdash e : T}$$

(T-SUB CAP)

$$\frac{\Gamma,\Pi \vdash c : C' \qquad C' \prec C}{\Gamma,\Pi \vdash c : C}$$

(T-CAP EXTEND)

$$\frac{\textbf{\textit{capsule}}\ C'(..)\ \textbf{\textit{extends}}\ C\{..\ T'\ p(\overline{T\ x})\{e\}\ ..\} = CT(C')}{\Gamma,\Pi \vdash C' \prec C}$$

(T-SUPER ST READ)

$$\frac{\Gamma,\Pi \vdash \textbf{\textit{self}} : C \qquad \textbf{\textit{capsule}}\ C(..)\ \textbf{\textit{extends}}\ D\{..T s..\} = CT(C)}{\Gamma,\Pi \vdash \textbf{\textit{self}}.s : T}$$

(T-SUPER ST ASGN)

$$\frac{\Gamma,\Pi \vdash \textbf{\textit{self}} : C \qquad \Gamma,\Pi \vdash x : T \qquad \textbf{\textit{capsule}}\ C(..)\ \textbf{\textit{extends}}\ D\{..T s..\} = CT(C)}{\Gamma,\Pi \vdash \textbf{\textit{self}}.s := x : T}$$

(T-DEREF)

$$\frac{\Gamma,\Pi \vdash x : \textbf{\textit{ref}}(T)}{\Gamma,\Pi \vdash \textbf{\textit{deref}}\ x : T}$$

(T-REFERENCE)

$$\frac{\Gamma,\Pi \vdash x : T}{\Gamma,\Pi \vdash \textbf{\textit{ref}}\ x : \textbf{\textit{ref}}(T)}$$

(T-ASSIGN)

$$\frac{\Gamma,\Pi \vdash x : T \qquad \Gamma,\Pi \vdash e : T}{\Gamma,\Pi \vdash x := e : T}$$

(T-LET)

$$\frac{\Gamma,\Pi \vdash e_1 : T \qquad \Gamma,\Pi \vdash e_2 : T}{\Gamma,\Pi \vdash \textbf{\textit{let}}\ x : T\ \textbf{\textit{be}}\ e_1 \textbf{\textit{in}}\ e_2 : T}$$

(T-IF)

$$\frac{\Gamma,\Pi \vdash x : int \qquad \Gamma,\Pi \vdash e_1 : T \qquad \Gamma,\Pi \vdash e_2 : T}{\Gamma,\Pi \vdash \textbf{\textit{if}}\ x\ \textbf{\textit{then}}\ e_1\ \textbf{\textit{else}}\ e_2 : T}$$

(T-$\Box$)

$$\frac{CT \vdash T}{CT \vdash \Box : T}$$

(T-ε)

$$\frac{CT \vdash T}{CT \vdash \varepsilon : T}$$

(T-PROC DECL)

$$\frac{\Gamma,\Pi \vdash C \prec D \quad \Gamma,\Pi,\overline{x:T},\textbf{\textit{self}} : C, \textbf{\textit{super}} : D \vdash e : T' \qquad \Gamma,\overline{x:T} \vdash spec : \textbf{\textit{int}} \qquad \Gamma,\Pi \vdash T' \qquad \forall T \in \overline{T}\ \textbf{.}\ \Gamma,\Pi \vdash T}{\Gamma,\Pi \vdash_C [spec]\ T'\ p(\overline{T\ x})\{e\}}$$

(T-WIRING DECL)

$$\frac{\Gamma,\Pi \vdash c : C \qquad \textbf{\textit{capsule}}\ C(\overline{G\ g})\ \textbf{\textit{extends}}\ D\{..\} = CT(C) \qquad \overline{G} = \Gamma(\overline{g})}{\Gamma,\Pi \vdash c(\overline{g})}$$

(T-CAPSULE DECL)

$$\frac{\forall proc \in \overline{proc}\ \textbf{.}\ \overline{g:G}, \overline{h:H}, \textbf{\textit{self}} : C, \textbf{\textit{super}} : D \vdash_C proc \quad \forall wire \in \overline{wire}\ \textbf{.}\ \overline{g:G}, \overline{h:H} \vdash wire \qquad \forall G \in \overline{G}\ \textbf{.}\ \vdash G \qquad \forall H \in \overline{H}\ \textbf{.}\ \vdash H \qquad \vdash D \qquad \forall T \in \overline{T}\ \textbf{.}\ \vdash T}{\vdash \textbf{\textit{capsule}}\ C(\overline{G\ g})\ \textbf{\textit{extends}}\ D\{\ \textbf{\textit{design}}\{\overline{H\ h}\ \overline{wire}\}\ \overline{T\ s}\ \overline{proc}\ \}}$$

Figure 4.8 Panini's select typing rules.

Local evaluation relation $\rightsquigarrow$: $\langle id, \mathcal{E}[e].Q, S, r, I\rangle \rightsquigarrow \langle id, \mathcal{E}[e'].Q', S', r, I\rangle$

(STATE READ)
$$\langle id, \mathcal{E}[\textbf{\textit{self}}.f].Q, S[l=v], [C.F[f=l]], I\rangle \rightsquigarrow \langle id, \mathcal{E}[v].Q, S, [C.F], I\rangle$$

(STATE ASGN)
$$\langle id, \mathcal{E}[\textbf{\textit{self}}.f=v].Q, S, [C.F[f=l]], I\rangle \rightsquigarrow \langle id, \mathcal{E}[v].Q, S[l:=v], [C.F], I\rangle$$

(SELF INVOC)
$$\frac{\textbf{\textit{capsule}}\ C(..)\ \textbf{\textit{extends}}\ C'\ \{..\ T\ p(\overline{T\ x})\{e\}\ ..\} = CT(C)}{\langle id, \mathcal{E}[\textbf{\textit{self}}.p(\overline{v})].Q, S, [C.F], I\rangle \rightsquigarrow \langle id, \mathcal{E}[e\ [\overline{v}/\overline{x}]].Q, S, [C.F], I\rangle}$$

(REF)
$$\frac{fresh(l)}{\langle id, \mathcal{E}[\textbf{\textit{ref}}\ v].Q, S, r, I\rangle \rightsquigarrow \langle id, \mathcal{E}[l].Q, S[l:=v], r, I\rangle}$$

(DEREF)
$$\frac{v \neq \varepsilon}{\langle id, \mathcal{E}[\textbf{\textit{deref}}\ l].Q, S[l=v], r, I\rangle \rightsquigarrow \langle id, \mathcal{E}[v].Q, S, r, I\rangle}$$

(REF ASGN)
$$\langle id, \mathcal{E}[l:=v].Q, S[l \neq \varepsilon], r, I\rangle \rightsquigarrow \langle id, \mathcal{E}[v].Q, S[l:=v], r, I\rangle$$

(LET)
$$\langle id, \mathcal{E}[\textbf{\textit{let}}\ x\ \textbf{\textit{be}}\ v\ \textbf{\textit{in}}\ e].Q, S, r, I\rangle \rightsquigarrow \langle id, \mathcal{E}[e[v/x]].Q, S, r, I\rangle$$

(FIFO DEQUEUE)
$$\langle id, v.e.Q, S, r, I\rangle \rightsquigarrow \langle id, e.Q, S, r, I\rangle$$

Global evaluation relation $\hookrightarrow$: $\mathcal{K} \mid\mid \langle id, \mathcal{E}[e].Q, S, r, I\rangle \hookrightarrow \mathcal{K}' \mid\mid \langle id, \mathcal{E}[e'].Q', S', r, I\rangle$

(PROC INVOC)
$$\frac{\begin{array}{c} id' = I(i) \quad \Sigma' = \left\langle id', e'.Q', S', [C'.F'], I'\right\rangle \in \mathcal{K} \quad \textbf{\textit{capsule}}\ C'(\overline{D\ j})\ \textbf{\textit{extends}}\ ..\ \{..\ T\ p(\overline{T\ x})\{e\}\ ..\} \in P \\ e'' = e\ [\overline{v}/\overline{x}, I'(\overline{j})/\overline{j}] \quad R = reach(\overline{v}, S) \quad R' = \bigcup_{l' \in dom(F)} reach(l', S) \\ R \cap R' = \emptyset \quad fresh(l) \quad \mathcal{K}' = \mathcal{K} \uplus \left\langle id', e'.Q'.\textbf{\textit{resolve}}(l, e'', id, p), S' \oplus R, [C'.F'], I'\right\rangle \end{array}}{\mathcal{K} \mid\mid \langle id, \mathcal{E}[i.p(\overline{v})].Q, S, [C.F], I\rangle \hookrightarrow \mathcal{K}' \mid\mid \langle id, \mathcal{E}[l].Q, S[l:=\varepsilon] \ominus R, [C.F], I\rangle}$$

(RESOLVE)
$$\frac{\begin{array}{c} R = reach(v, S) \quad R' = \bigcup_{l' \in dom(F)} reach(l', S) \\ R \cap R' = \emptyset \quad \Sigma = \langle id, e.Q, S[l=\varepsilon], r, I\rangle \in \mathcal{K} \quad \mathcal{K}' = \mathcal{K} \uplus \langle id, e.Q, S[l:=v] \oplus R, r, I\rangle \end{array}}{\mathcal{K} \mid\mid \left\langle id', \textbf{\textit{resolve}}(l, v, id, p).Q', S', [C'.F'], I'\right\rangle \hookrightarrow \mathcal{K}' \mid\mid \left\langle id', v.Q', S' \ominus R, [C'.F'], I'\right\rangle}$$

(SUPER INVOC)
$$\frac{\begin{array}{c} id' = I(\textbf{\textit{super}}) \\ \Sigma' = \left\langle id', e'.Q', S', [C'.F'], I'\right\rangle \in \mathcal{K} \quad \textbf{\textit{capsule}}\ C'(\overline{D\ j})\ \textbf{\textit{extends}}\ ..\{..\ T\ p(\overline{T\ x})\{e\}\ ..\} = CT(C') \\ e'' = e\ [\overline{v}/\overline{x}, I'(\overline{j})/\overline{j}] \quad R = reach(\overline{v}, S) \quad R' = \bigcup_{l' \in dom(F)} reach(l', S) \\ R \cap R' = \emptyset \quad fresh(l) \quad \mathcal{K}' = \mathcal{K} \uplus \left\langle id', e'.Q'.\textbf{\textit{resolve}}(l, e'', id, p), S' \oplus R, [C'.F'], I'\right\rangle \end{array}}{\mathcal{K} \mid\mid \langle id, \mathcal{E}[\textbf{\textit{super}}.p(\overline{v})].Q, S, [C.F], I\rangle \hookrightarrow \mathcal{K}' \mid\mid \langle id, \mathcal{E}[l].Q, S[l:=\varepsilon] \ominus R, [C.F], I\rangle}$$

Figure 4.9 Local and global operational semantics with capsule inheritance.

Table 4.1 Number of supertypes accessing their supertypes' states.

Akka

revisions	*pairs*	*subtypes*	*%accessing*	*accesses*	*writes*	*%writing*	*Total%*
1-5	354	30	8.47	81	29	35.80	*3.02*
6-11	163	9	5.52	28	7	25.00	*1.38*
12-24	249	33	13.25	51	18	35.29	*4.67*
25-49	212	15	7.08	110	42	38.18	*2.70*
50-99	75	3	4.00	6	2	33.33	*1.33*
>=100	274	25	9.12	92	34	36.96	*3.37*
Total	*1327*	*115*	*8.67*	*368*	*132*	*35.87*	*3.11*

Java

revisions	*pairs*	*subtypes*	*%accessing*	*accesses*	*writes*	*%writing*	*Total%*
1-5	2,452,901	589,050	24.01	7348,741	25,77,021	35.07	*8.42*
6-11	846,579	197,856	23.37	2551,068	915,763	35.9	*8.39*
12-24	992,184	234,234	23.61	3015,861	1,039,681	34.47	*8.13*
25-49	822,406	207,718	25.26	2624,272	968,774	36.92	*9.32*
50-99	963,136	235,178	24.42	3,038,645	111,4326	36.67	*8.95*
>=100	4,488,687	1,057,102	23.55	13,072,998	4,911,229	37.5	*8.84*
Total	*2,521,138*	*105,658*	*23.68*	*31,651,585*	*11,526,794*	*36.42*	*8.69*

4.6 Evaluation

One of our key findings is that in the presence of type encapsulation and encapsulated inheritance, the standard interface subtyping alone is sufficient to guarantee concurrent subtyping. To evaluate the usefulness of such finding we study:

$\mathscr{E}_1$ the extent to which interface subtyping can be used to guarantee concurrent subtyping, which in turn is determined by

$\mathscr{E}_2$ the prevalence of encapsulated inheritance and type encapsulation.

Dataset We measure violations of encapsulated inheritance and type encapsulation in a large set of randomly chosen Akka and Java projects from GitHub. Out data set includes 416 Akka and 554,864 Java projects. We consider a Java project that uses uses Akka's Java library [8] as an Akka project. There are 2003 actors in our Akka projects in addition to non actor classes and 21,320,654 Java classes in our Java projects.

Encapsulated inheritance We use two proxies to measure violations of encapsulated inheritance:

$\mathscr{M}_1$ actual violation of encapsulated inheritance by a subtype σ that accesses fields of its immediate or transitive supertype τ directly without using their methods; and

$\mathscr{M}_2$ possible violation of visibility-based encapsulated inheritance [155] in a type τ if it has non private or non static fields that could be accessible to any potential subtype σ.

In visibility-based encapsulation, private fields of a supertype τ are not directly accessible to its subtypes σ. Visibility-based encapsulation only encapsulates and protects the state of a type and not its representation. However, in languages like Java and Akka that do not check for encapsulation by default, programmers are taught to use visibility-based encapsulation to guarantee encapsulation [27,156]. Visibility-based encapsulation is not sufficient to guarantee encapsulation, however, may be a good starting measure to understand programmer's intent.

Type encapsulation We use the following proxy to measure the violation of type encapsulation:

$\mathscr{M}_3$ possible violations of visibility-based encapsulation [89] in a type τ if it has non private or non protected or static fields that could be accessible to any potential type η that may interact with τ.

In visibility-based encapsulation, private and protected fields of a type τ are not directly accessible to another type that interacts with τ.

Granularity We measure violations of encapsulated inheritance and type encapsulation for two type and project granularity levels. A project breaks encapsualted inheritance if there is at least one pair of supertype τ and σ in the project that break encapsulated inheritance. Similarly, the project breaks type encapsulation if there is at least one type C that its type encapsulation can be broken.

Maturity We divide the projects of our study into revision categories based on the number of their revisions. These include a category for projects with 1 to 5 revisions, and categories for 6–11, 12–24, 25–49, 50–99 and more than 100. Revision categories allow us to understand our findings for both more mature and less mature projects. We consider a project with more revisions to be more mature [122].

Boa We use Boa [51,52,55,143] to conduct our studies. Boa is a data mining infrastructure that includes an ultra-large corpus of open source projects and an infrastructure for mining and understanding software repositories.

Figure 4.1 shows our measurements for $\mathscr{M}_1$. In this figure the column *pairs* denotes the number of supertype and subtype pairs; *subtypes* denotes the number of subtypes that access their supertypes and *%accessing* shows the percentage of *subtypes* to *pairs*. The column *accesses* is the total number of accesses from *subtypes* to supertypes with the number of *writes* accesses specified and the remaining

are the read accesses. *%writing* shows the percentage of write accesses to the total accesses. *%Total* is the multipication of *%accessing* and *%accessing*.

4.6.1 Akka

Akka [8, 166] is a popular concurrent message passing framework in which a sequential actor encapsulates its state and communicates by other concurrently running actors by messages.

4.6.1.1 Encapsulated Inheritance

Type level $\mathscr{M}_1$ shows that about 8.67% of subactors access fields of their superactors from which about 35.87% access a superactor to write its fields and the remaining accesses are read. A read access can be automatically refactored to an invocation of a side effect free and pure accessor method added to the superactor to preserve encapsulated inheritance. The addition of these accessor methods does not change the interference closure and the interference behavior of the superactor. Therefore, only about 3.11% of subactors may break the encapsulated inheritance of their superactors. The 3.11% is the product of percentage of subactors that may access their superactors (8.67%) and the percentage of such accesses being a write access (35.87%).

$\mathscr{M}_2$ shows that at most 3.94% (79 out of 2003) of subactors can break encapsulated inheritance of their superactors by having non-private non-static fields that can be accessed and modified by a subactor. $\mathscr{M}_1$ is lower than $\mathscr{M}_2$ because $\mathscr{M}_1$ measures the actual violations that happen in the code whereas $\mathscr{M}_2$ measures an upper bound on the possible that may happen in the future as well.

Project level Using $\mathscr{M}_1$ there are $\boxed{11}$ out of 416 projects that violate encapsulated inheritance in their types. Using $\mathscr{M}_2$ there are at most $\boxed{38}$ out of 416 projects that could possibly violate the encapsualted inheritance.

4.6.1.2 Type Encapsulation

Similar to $\mathscr{M}_2$, $\mathscr{M}_3$ measures an upper bound on the number of type encapsulation violations.

Type level Type encapsulation of at most 2.65% (53 out of 2003) of actors can be broken by directly accessing their public or protected non static fields.

Project level At most $\boxed{24}$ projects out of 416 can violate type encapsulation.

Queries The Boa programs for encapsulated inheritance and type encapsulation measurements can be found at *http://boa.cs.iastate.edu/boa/?q=boa/job/public/34857* and *http://boa.cs.iastate.edu/boa/?q=boa/job/public/34861*.

4.6.2 Java

We conduct a similar study of measuring $\mathscr{M}_1$–$\mathscr{M}_3$ on our set of Java projects.

4.6.2.1 Encapsulated Inheritance

Type level About 23.68% of subclasses access fields of their superclass among which about 36.42% of their accesses are write. Therefore, about 8.69% of subclasses actually break the encapsulated inheritance of their superclasses.

Project level Using $\mathscr{M}_1$ there are 90,567 projects out of 554,864 that actually violate their encapsulated inheritance. Using $\mathscr{M}_2$ there are at most 131,104 projects that could possibly violate the encapsulated inheritance.

4.6.3 Type Encapsulation

Type level Type encapsulation of at most 11.8% (2,521,138 out of 21,320,654) can be violated by directly accessing their public or protected fields non static fields.

Project level At most there are 105,707 projects out of 554,864 that can violate type encapsulation.

Queries The Boa programs for encapsulated inheritance and type encapsulation measurements can be found at *http://boa.cs.iastate.edu/boa/?q=boa/job/public/34857* and *http://boa.cs.iastate.edu/boa/?q=boa/job/public/34859*.

4.6.4 Findings

Akka At the project level, using $\mathscr{M}_1$ and $\mathscr{M}_3$, there are at most 62 (38+24) Akka projects out of 416 that *can* violate either encapsulated inheritance or type encapsulation. This in turn means 354 (85.10%) projects abide by encapsulated inheritance and type encapsulation and therefore their concurrent subtyping can be guranteed using only standard interface subtyping. If we use $\mathscr{M}_2$, instead of $\mathscr{M}_1$, then there are 35 (11+24) projects (instead of 52) that *actually* violate their encapsulated inheritance or type

encapsulation and therefore the remaining 381 (91.6%) are abiding. At the type level 3.11% of types can violate either encapsulated inheritance or type encapsulation and the remaining 96.89% of types abide by both type and inheritance encapsulation.

Different distribution of types among projects causes different outcomes in the percentages of projects and type that violate encapsulated inheritance or type encapsulation.

Java At the project level, using $\mathscr{M}_1$ and $\mathscr{M}_3$, there are at most 236,811 (131,104+105,707) projects out of 554,864 that can violate either encapsulated inheritance or type encapsulation. This in turn means that 318,053 (57.32%) projects abide by encapsulated inheritance and type encapsulation and therefore their concurrent subtyping can be guranteed using only standard interface subtyping. If we use $\mathscr{M}_2$, instead of $\mathscr{M}_1$, then there are 196,274(90,567+105,707) projects, instead of 236,811, that violate encapsulated inheritance or type encapsulation and therefore the remaining 358590 (64.63%) are abiding. At type level 8.69% of types can violate either encapsulated inheritance or type encapsulation and the remaining 91.31% abide by them.

Summary More than three quarters of Akka projects and more than half of Java projects in our study abide by encapsualted inheritance and type encapsulation and their concurrent subtyping can be guranteed using interface subtyping. Interestingly, trends of our findings regarding violations of encapsualted inheritance and type encapsulation stay stable and do not fluctate a lot between more mature and less mature projects.

4.6.4.1 Threats to Validity

Similar to other experimental analysess there are threats to the validity of our findings. Our dataset includes a large set of randomly chosen Java and Akka projects not implemented by authors, however, it may not be representative of all concurrent programs that could be written in all mainstream languages that allow concurrent programming. Not all Java programs in our study are concurrent programs. Our analysis using visibility-based encapsulated inheritance and visibility-based type encapsulation is mostly syntactic and does not completely take into account aliasing semantics of Java programs.

4.7 Related Work

Previous work studies behavioral subtyping for sequential and concurrent programming languages.

Sequential Previous work [36, 49, 96, 130] in sequential languages guarantees behavioral subtyping by relating behaviors of overriding and overridden procedures in a subtype σ and its supertype τ through standard interface subtyping by enforcing contravariance and covariance requirements on preconditions and postconditions of these procedures. There is no interference during execution of a sequential program and therefore previous work in this category is not directly applicable to concurrent programming langauges.

Concurrent One category of previous work [148] extends behavioral subtyping of sequential language to programs in concurrent languages in which both procedures of a type and their specifications are atomic and free from interference. Another category of previous work [45, 95, 117] uses history constraints [9, 104] to take the interference into account and relate history constraints and invariants of the supertype τ and its subtype σ to gurantee behavioral subtyping. Other previous work [67] uses rely-guarantee [87] or its reformulation in deny-guarantee reasoning [48] in which the interference caused by one type on another is encoded as an environment specification in a rely predicate. Another category of previous work [47] extends concurrent separation logic [127] with specifications that specify shared resources and the interference behavior on the resource as its invariant. Some work in this category [88, 165] specify the access protocol of a shared resource in terms of permissible order of operations instead of the interference behavior. These specifications are fine-grained enough to talk about memory locations and read and write permissions for these locations. This chapter is complementary and adds to previous work its novelty that it separates the interference behavior and the interface behavior of a type and distinguishes between interference and interface behavior compatibility and subtyping. Another novelty of this chapter is that it uses encapsulation and encapsulated inheritance to reduce behavioral subtyping in a concurrent language to the standard interface subtyping [9].

Integration into program logics and languages Some previous work integrates variations of behavioral subtyping mentioned above into sequential or concurrent programming languages and tools such as Eiffel [110, 124], Java [63, 96], JML [95], ESC/Java [69] and Spec# [22].

CHAPTER 5. ORDER TYPES

The concurrency revolution [158] has renewed interest in message passing concurrency (MPC) [4, 43, 81, 115, 120, 169] because of its modularity [5, 18, 80, 98, 99] and scalability [168] benefits [8, 12, 74]. A significant challenge in MPC program design is reasoning about message orders. Message races and their nondeterministic message orders make reasoning about message orders intractable [29, 41, 93, 105, 108, 161] and lead to bugs that are hard to find and fix.

In this chapter, we develop *order types*, a novel type system to modularly detect message races and to explain their causes. The order type of a process is a local and descriptive causal type [123] that encodes its messaging behavior in terms of asynchronous message it sends and their happens-before relations. Compared to the rich body of previous work on session types [25, 26, 28, 40, 43, 79–81, 119, 120, 160, 178], discussed in Section 5.5, that focuses on specification and verification of system-level coordination properties, order types focus on process-level interactions aimed at facilitating bottom-up MPC program design. Our calculus, Panini, introduces three innovations in type system design for MPC programs. A Panini capsule is equivalent of a process that runs concurrently with other processes.

Bottom-up MPC program design through existential typing In bottom-up MPC program design processes are designed, coded and tested independently, and then composed to form the complete program. Programmers relies on the interfaces (types) of other processes that the process communicates with, but not on the concrete process instance. Order types facilitate this design style. The order type of the process can encode its dependency on a process instance with an unknown value via the introduction of an existentially quantified process variable. An existential process variable is eliminated statically when the process is composed with other processes and values of its composing processes are known.

Inter-process communication abstraction and abstraction eligibility Order type abstraction can decrease complexity by abstracting away inter-process communication that is local to a process. A local process instance is encapsulated by its enclosing process which controls access to it [38, 118, 136].

Naive abstraction can allow message races to hide behind abstraction. Abstraction of the order type of a process is sound if the process is abstraction-eligible. A process is abstraction-eligible if order types of sequential compositions of invocation of any of its two procedures are well-formed, i.e. do not lead to any message race. Two messages are racy if they are sent directly or transitively from a process and arrive at another process without any happens-before relation between them. Abstraction improves information hiding by preventing the exposure of local messaging behavior and increases the scalability of the type checking by decreasing the number of messages and their happens-before relations in an order type.

Blame Assignment for MPC Programs A message race in a process happens because of bad composition of messages and processes, each of which can happen at different processes. An ill-formed order type properly blames the process that causes the race and is responsible for bad composition of processes or messages as the supplier of bad values. or the process at which the race happens is not blamed because it is the users of the bad value and not its supplier. Blame assignment in order types is similar to higher order contract [62, 173] in which the blame lays with the provider of the bad value independent of who invokes the first-class function.

Contributions In summary, this chapter makes the following contributions:

- order types to enable modular reasoning about messaging behavior in a compositional bottom-up program design;
- well-formedness of order types to disallow message races; and
- abstraction of order types to soundly hide local messaging behavior of abstraction-eligible processes and improve information hiding and the scalability of type checking;
- blame assignment to properly blame a process responsible for process composition or message composition as the supplier of bad values; and
- type checking rules for an asynchronous concurrent process model with encapsulation [38] and structured sharing [118].

5.1 Motivation

Order types enable the modular detection of a message race in bottom-up program design and proper blame assignment to explain its cause. Abstraction hides the local messaging behavior.

5.1.1 Modular Message Race Detection

A message race happens when a process receives two messages from another process with no happens-before relation [91] between them. The happens-before relation is a partial order that orders messages within and between processes. Message races and their nondeterministic message orders make reasoning about message orders intractable [29, 41, 93, 105, 108, 161] and lead to subtle bugs that are hard to find, reproduce and fix.

The order type of a process can detect and disallow message races in bottom-up program design and construction. The order type of a process is a local descriptive type that encodes its messaging behavior. The messaging behavior of a process includes the messages it directly sends and their happens-before relations. A process is well-formed if the order types of its procedures are well-formed. The order type of a procedure is well-formed if the messages it sends directly or transitively do not lead to a message race. The following steps check the well-formedness of the order type of a procedure:

1 a constraints set of happens-before relationships between messages that the procedure directly or transitively sends is constructed;

2 the constraint set is checked to be well-formed and thus free from message races.

```
1 capsule Server() {
2   /* σ3 = • */
3   void save() { /* no message send */ }
4   /* σ4 = • */
5   void kill () { /* no message send */ }
6 }
```

Figure 5.1 Well-formed process Server with empty order types for its procedures.

To illustrate, consider a bottom-up program design which starts off with the declaration of a Server process in Figure 5.1. The server declares two procedures save and kill to save its client's work and shutdown, respectively. Order types $\sigma_3 = \bullet$ and $\sigma_4 = \bullet$ of save and kill are the empty order type $\bullet$

because these procedures do not send any messages in their implementations. Types σ_3 and σ_4 are trivially well-formed and therefore free from message race. This is because there is no message send in these types and therefore their constraint sets $\Delta_3 = \bullet$ and $\Delta_4 = \bullet$ are empty messages. The order type of a procedure can be easily constructed by exposing the message sends in its implementation.

Server is written in an asynchronous concurrent programming model with encapsulation [38, 118, 136], structured sharing [136] and transitive inorder delivery and processing [8, 80, 105]. A program in this model is a set of concurrently running processes that communicate by sending messages asynchronously. Procedures of a process denote type-safe messages that the process can receive and a procedure invocation is the equivalent of a message send [8,14,86]. A process encapsulates its local processes and controls access to them. Sharing is structured around importing and exporting process instances. Messages are delivered and processed in the same order they are sent. This process model overlaps with the underlying models for variations of active objects [38], actors [118] and Panini [18, 136, 138] message passing concurrency models.

```
7  capsule Proxy(Server sv) {
8    /* σ1 = ∃sv : Server . sv!save */
9    void save() { sv.save(); }
10   /* σ2 = ∃sv : Server . sv!kill */
11   void kill () { sv. kill (); }
12 }
```

Figure 5.2 Well-formed process Proxy with existential quantification for sv in its type.

To continue the bottom-up program design, consider the declaration of a Proxy process in Figure 5.2. The proxy uses Server and acts an interface to it, according to the *Proxy* design pattern [71]. The proxy declares the same procedures save and kill as the server and relays the client's messages to the server. On line 7, the proxy imports the Server process instance sv that it is dependent on and uses. The proxy does not know the value of the process variable sv. The order type $\sigma_1 = \exists sv : Server\,.\;sv!save$ for the procedure save describes its messaging behavior in which it sends a save message to the server sv. The existentially quantified process variable sv encodes the fact that the proxy does not know the value of the process variable sv that it is using. The type σ_1 is well-formed and free from message races because its constraint set $\Delta_1 = \exists sv : Server\,.\;sv!save \rhd \bullet$ is well-formed. The constraint set Δ_1 contains happens-before relations between messages that save directly or transitively sends. Order type σ_1 shows

that save sends a $\exists sv : Server \,.\, sv!save$ message directly. Receipt of this message by Server process sv causes the execution of its procedure save with the order type $\sigma_3 = \bullet$. The procedure save in Server does not send any messages and its constraint set $\Delta_3 = \bullet$ is empty, as discussed previously. That is, the direct message send $\exists sv : Server \,.\, sv!save$ in proxy's save leads to a transitive message send $\bullet$ in server's save. In the semantics of a message passing program, a message send always happens-before its receipt and processing. Therefore, the message send $\exists sv : Server \,.\, sv!save$ in proxy happens-before its receipt in server with the message send $\bullet$. This leads to the existentially quantified *semantic constraint* $\exists sv : Server \,.\, sv!save \triangleright \bullet$. This is the only constraint caused by the proxy's save and forms its constraint set $\Delta_1 = \exists sv : Server \,.\, sv!save \triangleright \bullet$ which is trivially well-formed. Construction of Δ_1 for σ_1 is modular. This is because Proxy uses its implementation and the order type σ_3 (interface) of the Server that it refers to and not its implementation. In modular reasoning, a module is understood using only its implementation and the interfaces (not implementations) of other modules it refers to [132]. Similarly, proxy's kill is well-formed.

```
13 capsule Client() {
14   design { Server s; Proxy r; r(s); }
16   /* σ = r!save;s!kill */
17   void main() { r.save(); s. kill (); }
18 }
```

Figure 5.3 Ill-formed process Client causes a message race in Server.

To continue the bottom-up program design, consider the declaration of a Client process in Figure 5.3 that uses both Proxy and Server processes. On line 15, the client instantiates local server and proxy instances s and r that it uses and exports s to the import sv of the proxy. The export of s to proxy assigns s to its import sv and shares the server s between the client and the proxy. The client declares a procedure main that intends to save the client's work in the server s and then shut down the server. The procedure main sends a save message to the proxy r followed by a kill message to the server s. The order type $\sigma = r!save; s!kill$ of main reflects the sequence of these message sends. Proxy in turn sends a save message to the server s. Client, proxy and server processes run concurrently. Unlike existential order types σ_2 and σ_3 in Proxy, σ is not existential. This is because the client instantiates process variables s and r and knows their concrete values.

Due to asynchrony of message sends and concurrent execution of client, proxy and server processes,

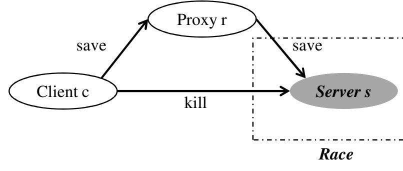


Figure 5.4 Nondeterministic messages save and kill cause a race.

messages *save* and *kill* sent from client can arrive at the server s at any order and with no happens-before relation between them. In other words, these messages are racy. Depending on the arrival order of save and kill , the client can have two different behaviors:

1 ***Desirable behavior:*** with message ordering $save \triangleright kill$ in which the save message arrives at the server before the kill message, the server *correctly* saves the client's work before it shuts down;

2 ***Undesirable behavior:*** with the opposite ordering $kill \triangleright save$ the server *incorrectly* shuts down before saving client's work.

To reason about message ordering in this program, one should consider both (i.e. 2!) message orders $save \triangleright kill$ and $kill \triangleright save$ before they can understand the program behavior. In general, in a program with a message race involving n messages, there are $n!$ message orderings that should be understood, which is intractable [162].

Because of the message race, the order type $\sigma = r!save; s!kill$ of client and its constraint set Δ are not well-formed. Construction of Δ proceeds as the following. First, the message send $r!save$ in σ causes the invocation and execution of the proxy's procedure save with the order type $\sigma_1 = \exists sv : Server\,.\; sv!save$ and its corresponding constraint set $\Delta_1 = \{\exists sv : Server\,.\; sv!save \triangleright \bullet\}$. The client knows the value of existentially quantified process variable sv in σ_1 because it instantiates the server instance s and exports and assigns it to sv in proxy r. This allows for the elimination of existential quantifier for sv and replacing it its known value s in the context of client. After quantification elimination, $\sigma_1 = s!save$ and $\Delta_1 = \{s!save \triangleright \bullet\}$. Second, the message send $s!kill$ in σ causes the invocation and execution of the server's procedure kill with its order type $\sigma_4 = \bullet$ and corresponding constraint set $\Delta_4 = \bullet$. Third, due to the inorder delivery, sending of $r!save$ of σ in client happens before sending of $s!kill$ and causes the constraint $\Delta_0 = \{r!save \triangleright s!save\}$ in the client. Putting Δ_0–Δ_2 together through

their union we arrive at the constraint set Δ below:

$$\Delta = \Delta_4 \cup \Delta_1 \cup \Delta_0 =$$

$$\{r!save \rhd s!save,\ s!save \rhd \bullet,\ s!kill \rhd \bullet,\ r!save \rhd s!kill\}$$

Δ is not well-formed because there exists two messages $s!save$ and $s!kill$ (in gray) in it with no happens-before relation between them. Consequently, the order type σ of main in Client is not well-formed. The ill-formedness of σ detects and avoid this message race *without any global or local specifications written by programmers* [80]. Detection of this race is modular because in its type checking Client uses its implementation and the order types σ_1 and σ_4 of the Proxy and Server processes and is independent from their implementations.

5.1.2 Abstraction and Abstraction-eligibility

Order type abstraction can decrease complexity by abstracting away local messaging behavior of a process. This is sound because a local process instance is encapsulated by its enclosing process which controls access to it [38, 118, 136].

```
7  capsule Proxy1(Server sv) {
8    design { Logger l; }
9    /* σ'_1 = ∃sv : Server . l!log ;s!save */
10   void save(){ l.log(); sv.save(); }
11   /* σ_2 = ∃sv : Server . s!kill */
12   void kill () { sv. kill (); }
13 }
14 capsule Client() {
15   design { Server s; Proxy1 r; r(s); }
16   /* σ' = r!save;s!kill */
17    void main() { r.save(); s. kill (); }
18 }
```

Figure 5.5 Abstraction of messaging behavior for local logger l.

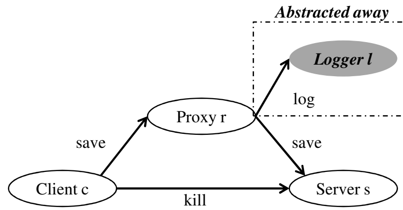
To illustrate abstraction, consider a variation of Proxy in Figure 5.5 called Proxy1. Proxy1's procedure logs bookkeeping information by sending a log message to an instance of a Logger process before relaying the save message to the server s. The order type $\sigma_1' = \exists sv : Server\,.\ l!log;sv!save$ reflects this messaging behavior. Unlike the server sv which is imported with an unknown value the logger l is locally declared in Proxy1 and its value is known.

The addition of the invocation l.log() in Proxy1's save procedure adds the message send $l!log$ to its order type $\sigma_1' = \exists sv : Server\,.\; l!log; sv!save$. The order type σ_1' with $l!log$ is larger than the order type $\sigma_1 = \exists sv : Server\,.\; sv!save$ for save in Proxy without it. To check the well-formedness of σ_1' the constraint set $\Delta_1' = \{l!log \triangleright \bullet, \exists sv : Server\,.\; sv!save \triangleright \bullet, \exists sv : Server\,.\; l!log \triangleright sv!save\}$ should be constructed and checked for well-formedness. The constraint set Δ_1' with three happens-before relations is larger than $\Delta = \{\exists sv : Server\,.\; sv!save \triangleright \bullet\}$ with only one happens-before relation and consequently its construction and checking takes more time.

Similarly, to check the well-formedness of σ' in the Client that uses Proxy1 the following constraint set Δ' should be constructed and checked for well-formedness:

$$\Delta' = \{\boxed{r!save \triangleright l!log}, \boxed{l!log \triangleright \bullet}, r!save \triangleright s!save, s!save \triangleright \bullet,$$
$$\boxed{l!log \triangleright s!save}, s!kill \triangleright \bullet, r!save \triangleright s!kill\}$$

There is a race between $s!save$ and $s!kill$ (in gray) in Δ' for Proxy1 and therefore σ' is ill-formed. However, the same race exists in Δ and σ for Proxy without the logger. Compared to Proxy, sending a log message to the local logger l in Proxy1 and consequently in σ' and its corresponding happens-before relations in Δ' does not contribute to any more races. Therefore the local process instance l, its messages and its happens-before relations can be abstracted away from the order type of the procedures of Proxy1.



After abstraction $\sigma' = \exists s : Server\,.\; s!save$ (equal to σ in Proxy) and $\Delta' = \{r!save \triangleright s!save, s!save \triangleright \bullet,$ $s!kill \triangleright \bullet, r!save \triangleright s!kill\}$ (equal to Δ in Proxy). Abstraction decreases size of σ' by reducing the number of its messages and decreases the size of Δ' by reducing the number of its happens-before relations. Construction and checking of abstracted Δ' takes less time than the original Δ' without abstraction. Abstraction only applies to local process instances and not imported process instances such as sv.

Abstraction-eligibility A naive abstraction can hide message races. To illustrate, consider a variation of Client declared in Figure 5.7 called Client1. On line 14, Client1 locally declares server and proxy process instances s and r. By abstracting the local process instance, the order type $\sigma_0 = r!save$ of the

procedure rSave becomes $\sigma_0 = \bullet$. Similarly, $\sigma = s!kill$ for sKill becomes $\sigma = \bullet$ after abstraction. Now consider a process using an instance c of Client1 in an invocation sequence c.rSave();c. sKill (). Using order types $\sigma_0 = r!save$ and $\sigma = s!kill$ before the abstraction one can conclude that there is a race between save and kill messages sent to the server. However, this race will not be detected and the invocation will be well-formed using the abstracted order types $\sigma_0 = \bullet$ and $\sigma = \bullet$ are used. Here abstraction of local process instances s and r hides the race, which is unsound.

```
13 capsule Client1() {
14   design { Server s; Proxy r; r(s); }
15   /* σ0 = r!save becomes σ0 = • after abstraction */
16   void rSave(){ r.save(); }
17   /*  σ = s!kill becomes σ = • after abstraction  */
18   void sKill () {s. kill (); }
19 }
```

Figure 5.6 Naive order type abstraction in Client1.

Similarly, abstraction of local process instance r in Client2 is not sound. This is because, in process using an instance c2 of Client2 with access to s the invocation sequence c2.rsave(); s. kill () will be ill-formed before the abstraction of Client2 and well-formed after. The local proxy instance r cannot be abstracted away because the shared imported server instance s flows to it. Flowing of an imported share process into a local process taints the local and prevents its abstraction. This is because the local can directly or indirectly send messages to the shared process which will be hide by abstraction.

```
13 capsule Client2(Server s) {
14   design { Proxy r; r(s); }
15   /* σ0 = r!save becomes σ0 = • after abstraction */
16   void rSave(){ r.save(); }
17 }
```

Figure 5.7 Unsound abstraction of the tainted local process instance r.

Order type abstraction is sound if the following conditions hold:

1 abstraction only abstracts aways message sends and their happens-before relations to untainted local process instances; and

2 the enclosing process is abstraction-eligible.

An untainted local is a local process instance with no import process instance or untainted local instance

exported to it. The abstraction is sound because an untainted locally declared process and its local processes instances are encapsulated by its enclosing process and are not directly accessible to the outside processes. These processes are only accessible through procedures of the enclosing process.

A process is abstraction-eligible if the following conditions hold:

1 the order types of its procedures are well-formed before abstraction;

2 a sequence composition ; of invocations of any of its two procedures before abstraction does not lead to message races.

To illustrate, Client1 is not abstraction-eligible. Both procedures rSave and sKill are well-formed. The invocation sequence c.sKill(); c.rSave() leads to the constraint set $\{c!sKill \triangleright s!kill, c!rSave \triangleright r!save, r!save \triangleright s!save, c!sKill \triangleright c!rSave\}$ which is well-formed in the presence of transitive inorder delivery and processing. However, the invocation sequence c.rSave();c.sKill() leads to a race and is not well-formed. Unlike Client1, Proxy1 is abstraction-eligible. This is because order types $\sigma_1' = l!log; s!save$ and $\sigma_2 = s!kill$ of its procedures save and kill are well-formed. And sequences of invocations of these two procedures lead to well-formed constraint sets. Similarly, Server is abstraction-eligible. Client is not abstraction-eligible because its procedure main leads to a race and its order type is not well-formed.

5.1.3 Proper Blame Assignment

Proper blame assignment enables programmers to quickly ascertain racy processes to be fixed or replaced. A message race in a process happens because of bad composition of messages and processes, each of which can happen at different processes. An ill-formed order type properly blames the process that causes the race and is responsible for bad composition of processes or messages as the supplier of bad values.

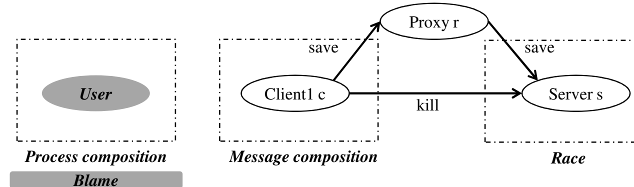
To illustrate, consider the previous Client process in Figure 5.3. The bad composition of messages r.save() and s.kill() on line 15 in client's procedure main leads to a race in Server process s. That is, the race happens in the server but we blame the client. The bad message composition in client contributes to the race but it is not the true reason behind the race. The main reason for the race is the bad process composition Proxy r(s) in Client. This process composition shares the server s between the Client and

Proxy process r by exporting client's s to proxy's imported sv. The bad process composition is blamed for the race because it is the supplier of the bad value.

```
13 capsule Client3(Server s, Proxy r) {
14   /* σ = ∃s : Server, r : Proxy . r!save; s!kill */
15    void main() { r.save(); s. kill (); }
16 }
```

Figure 5.8 A message composition that could lead to a race with bad composition.

To illustrate the difference between bad message and process compositions, consider a variation of the client called Client3 in Figure 5.8. Procedures main in both Client and Client2 compose r.save() and s. kill () messages similarly. However, while Client is ill-formed and causes a race in the server, Client3 is well-formed. The difference between these two is their process compositions. The constraint set $\Delta = \{\exists r : Proxy\, .\, r!save \triangleright \exists sv : Server\, .\, sv!save, \exists sv : Server\, .\, ,sv!save \triangleright \bullet, \exists s : Server\, .\, s!kill \triangleright \bullet, \exists r : Proxy, s : Server\, .\, r!save \triangleright s!kill\}$ for Client3 is well-formed unless $r.sv$ is equal to s. That is, Δ is well-formed if server sv of proxy r is not the same as the server s in the client. Client3 does not provide such process composition and therefore is well-formed whereas Client provides such composition and is ill-formed.



Bad process and message compositions may happen at different processes. To illustrate, consider the process User in Figure 5.9. Client3 provides a bad message composition but not a process composition that leads to a race. User provides a bad process composition for its Client3 process c and will cause a race. User is to blame for this race and not Client1.

```
17 capsule User() {
18   Design { Server s; Relay r; Client3 c, r(s); c(s,r); }
19 }
```

Figure 5.9 Bad process composition in User and bad message composition in Client1.

A bad message composition and a bad process composition both are necessary to cause a race and neither are sufficient alone. To illustrate, User in the absence of the procedure main in Client3 does not lead to a race; similarly, main in Client3 in the absence of User does not lead to a race.

5.2 Process Model and Program Syntax

In the process model Panini, a concurrent message-passing program consists of a set of processes that run concurrently with the following properties:

- a process communicates with another process by asynchronous invocations of its procedures. Procedures of a process denote the set of messages a process can receive and a procedure invocation is the type-safe equivalent of a message send [86].
- a process is sequential [78] and owns its states and local process instances. A sequential process has a single thread of execution that finishes the execution of one invoked procedure before staring another. A process owns [38, 75] its state and local processes by controlling access to them and make them accessible only through its procedures.
- a process can create other processes and share them in a structured way. Structured sharing is built around exporting and importing process instances.
- messages are sent in the same order they appear in the program text and are processed in the same order they are received [80];

This process model overlaps with the underlying models for variations of active objects [38] in Creol [86] and JCoBox [151], actors [118] in ActorFoundry [13] and SALSA [170] and Panini [18, 136, 138] message passing concurrency models. Unlike process models based on channels and sessions [43, 78, 81, 111, 120], this process model *does not* require neither channels for process communications nor sessions to structure and isolate communications [80].

5.2.1 Program Syntax

Figure 5.10 shows the expression-based core syntax of our process model. In this figure $\overline{term}$ denotes a sequence of zero or more terms.

A program is a set of process declarations. A process declares a set of local process instances and states along with procedures to access them. A process imports a set of process instances that are shared between the process and the outside world and exports a set of process instances that are shared between the process and its local process instances. A local declaration instantiates a process instance

$prog ::= \overline{decl}$	program
$decl ::= capsule\ C\ (\overline{import})\ \{\ design\ \ \overline{state}\ \overline{proc}\ \}$	process declaration
$import ::= D\ i$	imported instance
$state ::= T\ \ s\mathbf{;}$	state declaration
$proc ::=\ T\ p\ \ (\ \overline{T\ x}\ )\ \{\ e\ \}$	procedure declaration
$design ::= \textbf{\textit{design}}\ \{\ \overline{ins}\ \overline{wire}\ \}$	design declaration
$ins ::= C\ \ i$	local instance declaration
$wire ::= i\ \ \overline{j}$	wiring declaration
$e ::=$	*expression:*
$\mid i\,\mathbf{.}\,p\ (\overline{x})\ \ \mid \textbf{\textit{self}}\,\mathbf{.}\,p\ (\overline{x})$	procedure invocation
$\mid \textbf{\textit{self}}\,\mathbf{.}\,s \mid \textbf{\textit{self}}\,\mathbf{.}\,s\ \mathbf{:=}\ x$	state access
$\mid \textbf{\textit{let}}\ x{:}T\ \textbf{\textit{be}}\ e\ \textbf{\textit{in}}\ e$	let
$\mid \textbf{\textit{if}}\ x\ \textbf{\textit{then}}\ e\ \textbf{\textit{else}}\ e$	conditional
$\mid \textbf{\textit{while}}\ x\ \textbf{\textit{do}}\ e$	loop
$\mid x$	variable
$\mid ()$	unit value
$\mid n$	integer value

$C, D \in \mathfrak{C}$	set of processes	$T \in \mathfrak{T}$	set of variable types
$s \in \mathfrak{F}$	set of states	$p \in \mathfrak{P}$	set of procedures
$x \in \mathfrak{X}$	set of variables	$i,j,h \in \mathfrak{I}$	set of process instance

Figure 5.10 Core process syntax, inspired by [14, 18, 86].

and a wiring declaration composes it by exporting its composing process instances to its imports. The local declaration can compose process instances. In a procedure, a process can *asynchronously* invoke a procedure of another process or *synchronously* invoke a procedure of itself through the pseudo-variable ***self***. A let expression sequences two expressions. The let expression can compose message sends. Other expressions are standard. A local process instance or state of a process cannot be accessed directly by another process. A process instance cannot be an argument of a procedure or its return value and cannot be assigned to its local variables.

To illustrate, in Figure 5.2, the declaration of the process Proxy imports a Server instance sv and declares two procedures save and kill ; in Figure 5.3, Client instantiates Server process s and instantiates and composes Proxy instance r by exporting s to the import sv of the proxy; in Figure 5.3, Client declares a procedure main that sequences and composed to procedure invocations (message sends).

5.3 Order Type Syntax

$\sigma ::=$	*order type:*
$\mid \bullet$	empty
$\mid \tau$	singleton
$\mid \sigma ; \sigma$	sequence
$\mid [\sigma]^*$	recursive
$\tau ::=$	*action:*
$\mid \zeta \; i!p$	message
$\zeta ::=$	*message modifier:*
$\mid \epsilon$	none
$\mid \exists i : C \,.$	existential

$i,j \in \mathfrak{I}$	set of process instances	$p \in \mathfrak{M}$	set of messages
$C \in \mathfrak{C}$	set of process types	$\gamma \in \mathfrak{V}$	set of type variables

Figure 5.11 Syntax of order types.

Figure 5.11 shows the syntax for order types. An order type can be empty, a singleton message send, a sequence of two order types and a recursive order type. An empty order type encodes the type of an expression that sends no messages. A singleton order type encodes the type of an expression that sends a single message to a receiver i with an unknown or known concrete value. By existentially qualifying the receiver process variable, an existential singleton order type encodes sending a message to a receiver that its value is not known to the enclosing process. In contrast, in a regular singleton order type, the concrete value of the receiver is known. A sequence order type encodes the order type of a sequence of expressions as they appear in the program text. A recursive order type encodes the order type of a recursive expression. We take the equirecursive view for recursive order types in which a type and its unfolding are equal [133].

To illustrate, in Figures 5.1 and 5.2, the procedure save of Server has the empty order type empty $\sigma_3 = \bullet$; save of Proxy has the existential order type $\sigma_1 = \exists sv : Server \,.\; sv!save$; and in Figure 5.7, the procedure sKill of Client1 has the regular order type $\sigma = s!kill$.

5.3.1 Type Attributes

Figure 5.12 defines the type attributes used in the type checking rules for order types.

$\theta ::=$	*standard type:*
$\mid T$	variable type
$\mid C$	process type
$T ::=$	*variable type:*
$\mid$ ***unit***	unit
$\mid$ ***int***	integer
$\mid \overline{T} \rightarrow T$	procedure type
$\Gamma ::= \emptyset \mid \Gamma,\ x : T \mid \Gamma,\ i : \eta\ C$	typing environment
$\eta ::=$	*type qualifier:*
$\mid$ *@local*	local
$\mid$ *@import*	import
$\Psi ::= \emptyset \mid \Psi,\ i = j \mid \Psi,\ i.j = h$	aliasing environment
$\Delta ::= \emptyset \mid \Delta,\ \tau \triangleright \tau$	constraint set
$\Gamma \vdash e : \sigma\ ^{\theta}$	typing judgement

Figure 5.12 Type attributes.

The type system distinguishes between a variable type T and a process type C. A variable type is standard and can be a unit, integer or function type. A process type is a process declared in a program. The typing environment Γ maps a variable to its type. A qualifier for a process type denotes if the process variable is locally declared or imported. The aliasing environment Ψ maps a process variable to its aliases. The constraint set Δ is a set of happens-before relations between message sends. The typing judgement $\Gamma \vdash e : \sigma\ ^{\theta}$ denotes that in the typing environment Γ the expression e has the order type σ and the process or variable type θ.

5.4 Type System

The order type of a procedure encodes the messages a procedure sends *directly* and the *textual* ordering among them as they appear in the procedure text. The order type is well-formed if the messages

Invocation

(T-PROC INVOC)

$$\frac{\Gamma \vdash i : \bullet^{@local\ C} \qquad \forall x \in \overline{x}, T' \in \overline{T'} \,.\, \Gamma \vdash x : \bullet^{T'} \qquad \sigma^{\overline{T'} \to T} = pType(C,p)}{\Gamma \vdash\ i.p(\overline{x}) : i!p^{\,T}}$$

(T-PROC INVOC-$\exists$)

$$\frac{\Gamma \vdash i : \bullet^{@import\ C} \qquad \forall x \in \overline{x}, T' \in \overline{T'} \,.\, \Gamma \vdash x : \bullet^{T'} \qquad \sigma^{\overline{T'} \to T} = pType(C,p)}{\Gamma \vdash\ i.p(\overline{x}) : \exists i : C \,.\, i!p^{\,T}}$$

Composition

(T-LET)

$$\frac{\Gamma, x : T \vdash e' : \sigma'^{\,T'} \qquad \Gamma \vdash e'' : \sigma''^{\,T''} \qquad \Psi = aliasEnv(C) \qquad \Gamma, \Psi \vdash \sigma'; \sigma''}{\Gamma \vdash_C \textbf{\textit{let}}\ x : T\ \textbf{\textit{be}}\ e'\ \textbf{\textit{in}}\ e'' : \sigma'; \sigma''^{\,T''}}$$

(T-WIRING DECL)

$$\frac{\begin{array}{c}\Gamma \vdash i : \bullet^{C} \qquad \textbf{\textit{capsule}}\ C(\overline{D\ h})\ \{design\ \overline{state}\ \overline{proc}\} = CT(C) \qquad \forall j_k \in \overline{j}, D_k \in \overline{D} \,.\, \Gamma \vdash j_k : \bullet^{D_k} \\ p \in \overline{proc} \qquad \sigma^{\overline{T'} \to T} = pType(C,p) \qquad \Psi = aliasEnv(G) \qquad \Psi' = \downarrow_i \Psi \qquad \Gamma, \Psi' \vdash \sigma\end{array}}{\Gamma \vdash_G i(\overline{j})}$$

Declaration (without abstraction)

(T-PROCESS DECL)

$$\frac{\textbf{\textit{capsule}}\ C(\overline{import})\ \{design\ \overline{state}\ \overline{proc}\} = CT(C) \qquad \forall p \in \overline{proc} \,.\, \Gamma \vdash p \qquad \forall i(\overline{j}) \in \overline{wire} \,.\, \Gamma \vdash_C i(\overline{j})}{\Gamma \vdash C}$$

(T-PROC DECL)

$$\frac{\Gamma, \overline{x' : T'} \vdash e : \sigma^{\,T}}{\Gamma \vdash_C T\ \ p(\overline{T'\ x'})\{e\} : \sigma^{\overline{T'} \to T}}$$

Conditional

(T-COND)

$$\frac{\Gamma \vdash x : \bullet^{\textbf{\textit{int}}} \qquad \Gamma \vdash e' : \sigma'^{\,T} \qquad \Gamma \vdash e'' : \sigma''^{\,T}}{\Gamma \vdash \textbf{\textit{if}}\ x\ \textbf{\textit{then}}\ e'\ \textbf{\textit{else}}\ e'' : \sigma'; \sigma''^{\,T}}$$

Subsumption

(T-SUB)

$$\frac{\Gamma \vdash e : \sigma'^{\,T'} \qquad \Gamma \vdash \sigma' \preccurlyeq \sigma \qquad \Gamma \vdash T' <: T}{\Gamma \vdash e : \sigma^{\,T}}$$

Well-formedness

(W-TYPE)

$$\frac{\begin{array}{c}\Delta = copy(\sigma, \Psi', \Gamma) \qquad \tau_1, \tau_2 \in dom\,\Delta \cup rng(\Delta) \\ \tau_1 = \zeta_1\ i_1!p_1 \qquad \tau_2 = \zeta_2\ i_2!p_2 \qquad \forall \tau_1, \tau_2, \Psi \vdash i_1 = i_2 \,.\, \Delta \vdash \tau_1 \rhd \tau_2\ \vee\ \Delta \vdash \tau_2 \rhd \tau_1\end{array}}{\Gamma, \Psi \vdash \sigma}$$

Figure 5.13 Select type checking rules for order types.

it sends *transitively* do not lead to a race. Messages that a procedure sends transitively and their *semantic* happens-before relations are included in the constraint set of its order type. For an ill-formed order type, blame assignment properly blames the supplier of bad values. Order type abstraction hides local messaging behavior and improves the scalability of type checking. Figure 5.13 shows select type checking rules for order types. The rest of more standard typing rules can be found in Figure 5.14.

(T-SELF INVOC)
$$\frac{\Gamma \vdash \textbf{\textit{self}} : \bullet^{\,C} \qquad \forall x \in \overline{x}, T' \in \overline{T'} \,.\, \Gamma \vdash x : \bullet^{\,T'} \qquad \sigma^{\,\overline{T'} \to T} = pType(C, p)}{\Gamma \vdash \textbf{\textit{self}}.p(\overline{x}) : \sigma^{\,T}}$$

(T-LOOP)
$$\frac{\Gamma \vdash x : \bullet^{\,\textbf{\textit{int}}} \qquad \Gamma \vdash e : \sigma^{\,T}}{\Gamma \vdash \textbf{\textit{while}}\ x\ \textbf{\textit{do}}\ e : [\sigma]^{*T}}$$

(T-VAR)
$$\frac{(x : T) \in \Gamma}{\Gamma \vdash x : \bullet^{\,T}}$$

(T-STATE READ)
$$\frac{\Gamma \vdash \textbf{\textit{self}} : \bullet^{\,C} \qquad \Gamma \vdash_C f : \bullet^{\,T}}{\Gamma \vdash \textbf{\textit{self}}.f : \bullet^{\,T}}$$

(T-STATE ASGN)
$$\frac{\Gamma \vdash \textbf{\textit{self}} : \bullet^{\,C} \qquad \Gamma \vdash_C f : \bullet^{\,T} \qquad \Gamma \vdash x : \bullet^{\,T}}{\Gamma \vdash \textbf{\textit{self}}.f = x : \bullet^{\,T}}$$

(T-UNIT)
$$\Gamma \vdash () : \bullet^{\,\textbf{\textit{unit}}}$$

(T-INT)
$$\Gamma \vdash n : \bullet^{\,\textbf{\textit{int}}}$$

(T-INST)
$$\frac{(i : \eta\ C) \in \Gamma}{\Gamma \vdash i : \bullet^{\,\eta\ C}}$$

Figure 5.14 Rest of type checking rules.

5.4.1 Declarations

A program type checks if its process declarations type check. Without abstraction, type checking of a process declaration and its procedures are straightforward. In the rule (T-PROCESS DECL), a process declaration type checks if its procedure and local process declarations type check. In (T-PROC DECL), a procedure declaration type checks if its implementation type checks. Among the expressions in the body of a procedure, type checking rules for a procedure invocation, let and conditional expressions are more interesting.

5.4.2 Invocations

There are two rules for procedure invocations depending on if the concrete value of its receiver is know in the enclosing process.

Known receiver value The rule (T-PROC INVOC) type checks an invocation with a known receiver value. The enclosing process locally declares and instantiates the receiver i and therefore knows its concrete value and dynamic type. The order type of a variable is the empty order type $\bullet$. The type $i!p^{\,T}$

(AUX-*aliasEnv*)

$$\frac{\Psi = \emptyset \qquad \textbf{capsule}\ C(\overline{import})\ \{\overline{D\ i(\overline{j})}\ \overline{state}\ \overline{proc}\} = CT(C) \qquad \forall D\ i(\overline{j}) \in \overline{local}\ .\ \textbf{capsule}\ D(\overline{H\ h})\ \{..\} = CT(D), j \in \overline{j}, h \in \overline{h}\ .\ \Psi = \Psi, i.h = j}{aliasEnv(C) = \Psi}$$

(AUX-*xAliasEnv*)

$$\frac{\textbf{capsule}\ C(\overline{D\ i})\ \{\overline{local}\ \overline{state}\ \overline{proc}\} = CT(C) \qquad \textbf{capsule}\ D(\overline{G\ \ g})\ \{..\} = CT(C) \qquad \forall (D\ i(\overline{j})) \in \overline{local}, isTainted(i)\ .\ \Psi = \Psi,\ i.\overline{g} = \overline{j}}{xAliasEnv(C) = \Psi}$$

(AUX-*xTypeEnv*)

$$\frac{\textbf{capsule}\ C(\overline{D\ i})\ \{\overline{local}\ \overline{state}\ \overline{proc}\} = CT(C) \qquad \forall (D\ i) \in (\overline{D\ i})\ .\ \Gamma = \Gamma, i : @import\ D \qquad \forall (H\ h(\overline{g})) \in \overline{local}, isTainted(h)\ .\ \Gamma = \Gamma, h : @local\ H}{xTypeEnv(C) = \Gamma}$$

(AUX-TYPEENV)

$$\frac{\textbf{capsule}\ C(\overline{D\ i})\ \{\overline{local}\ \overline{state}\ \overline{proc}\} = CT(C) \qquad \Gamma = \emptyset \qquad \forall (D\ i) \in (\overline{D\ i})\ .\ \Gamma = \Gamma, i : @import\ D \qquad \forall (H\ h(\overline{g})) \in (\overline{H\ h(\overline{g})})\ .\ \Gamma = \Gamma, h : @local\ H}{typeEnv(C) = \Gamma}$$

(AUX-IMPORTS)

$$\frac{\textbf{capsule}\ C(\overline{D\ i})\ \{..\} = CT(C)}{imported(C) = \overline{i}}$$

Head

(EMP)	$head(\bullet)$	$=$	$\bullet$
(SIN)	$head(i!p)$	$=$	$i!p$
(SIN-$\exists$)	$head(\exists i : C\ .\ i!p)$	$=$	$\exists i : C\ .\ i!p$
(SEQ)	$head(\sigma_1; \sigma_2)$	$=$	$head(\sigma_1)$
(REC)	$head([\sigma]^*)$	$=$	$head(\sigma)$

Tail

(EMP)	$tail(\bullet)$	$=$	$\bullet$
(SIN)	$tail(i!p)$	$=$	$i!p$
(SIN-$\exists$)	$tail(\exists i : C\ .\ i!p)$	$=$	$\exists i : C\ .\ i!p$
(SEQ)	$tail(\sigma_1; \sigma_2)$	$=$	$tail(\sigma_2)$
(REC)	$tail([\sigma]^*)$	$=$	$tail(\sigma)$

Figure 5.15 Auxiliary functions.

encodes that *(1)* the invocation sends a message p to a receiver i and *(2)* the concrete value of the receiver is known. T is the return type of the invocation. An invocation type checks if its invoked procedure p type checks and its arguments and formal parameters are of the same type. The rule (SUB) allows for subsumption among both variable and order types. The function *pType* returns the type of a procedure.

Unknown receiver value The rule (T-INVC-$\exists$) type checks an invocation with an unknown receiver value. The enclosing process imports the receiver i and therefore does not know its concrete value and dynamic type. The existential type $\exists i : C\,.\ i!p^{\,T}$ encodes that *(1)* the invocation sends a message p to a receiver i of type C and *(2)* the concrete value of the receiver is unknown. Unknown value of an imported process instance will be known and the existential quantifier will be eliminated when processes are composed and its concrete value is exported.

The order type of an invocation represents the message it sends.

5.4.3 Composition and Blame Assignment

Bad compositions of process and messages leads to a race. Processes can be composed in a local process declaration and messages can be composed in a let expression.

5.4.3.1 Process Composition

The rule (T-WIRING DECL) type checks a local declaration in a process G. The declaration of a process instance of type C type checks if the order types of its procedures are well-formed. An order type is well-formed if its constraint set is well-formed, i.e. $\Gamma \vdash \Delta$. A constraint set is well-formed if it is free from message races. The auxiliary function *copy* constructs the constraint set of an order type. The local declaration of i instantiates and then composes it by exporting composing process instances $\overline{j}$ to its imports $\overline{h}$. These exports make $i.\overline{h}$ and $\overline{j}$ aliases in the enclosing process G. The auxiliary function *aliasEnv* constructs this aliasing environment. The aliasing environment is statically known because of structured sharing that happens only through imports and exports during process compositions. The projection auxiliary function $\downarrow$ makes the aliasing environment of the enclosing process G compatible with the point of view of the process C. For an aliasing relation $i.h = j$ in the aliasing environment of G, its i-projection in C becomes $h = j$.

5.4.3.2 Semantics Happens-before Relations

The constraint set of an order type is a set of semantic happens-before relations between messages that the order type sends transitively. Semantic happens-before relations stem from the following:

$\mathscr{R}_1$ a message send in a sender process happens-before its receipt and processing in its receiver process; and

$\mathscr{R}_2$ messages are sent in the same order they appear in the program text and they are delivered in the same order sent. Messages are processed in the same order they are delivered. These happens-before relations are guaranteed by inorder delivery and processing of messages.

Happens-before relations $\mathscr{R}_1$ and $\mathscr{R}_2$ are used in the construction of a constraint set and $\mathscr{R}_2$ relations are used in checking its well-formedness.

$$\text{(H-TRANS)}\quad \frac{\Delta \vdash \tau_1 \rhd \tau_2 \qquad \Delta \vdash \tau_2 \rhd \tau_3}{\Delta \vdash \tau_1 \rhd \tau_3} \qquad \text{(H-INORDER)}\quad \frac{\Delta \vdash i!p_1 \rhd i!p2 \qquad \Delta \vdash i!p_1 \rhd j!q1 \qquad \Delta \vdash i!p_2 \rhd j!q_2}{\Delta \vdash j!q_1 \rhd j!q2}$$

$$\text{(H-TRANS-INORDER)}\quad \frac{\Delta \vdash i!p_1 \rhd j!q1 \qquad \Delta \vdash i!p_2 \rhd .. \rhd j!q_2 \qquad \Delta \vdash i!p_1 \rhd i!p2}{\Delta \vdash j!q_1 \rhd j!q2}$$

Figure 5.16 Happens-before rules for direct and transitive inorder delivery and processing.

Happens-before relations is a partial order with transitive antisymmetric properties. Figure 5.16 shows the rules for the happens-before relation. The standard rule (H-TRANS) encodes the transitivity of the relation. The rule (H-INORDER) encodes direct inorder delivery and processing. Messages p_1 is sent to a receiver i followed by a message p_2, i.e. $i!p_1 \rhd i!p_2$. Inorder delivery guarantees that i receives p_1 before p_2. The process i processes message p_1 during which it directly sends a message q_1 to a receiver process j. Sending of p_1 message to i happens before its processing, i.e. $i!p_1 \rhd j!p_1$. Similarly, $i!p_2 \rhd j!q_2$. Inorder processing guarantees that i processes p_1 before p_2. Therefore, message send $j!q_1$ happens-before $j!q_2$, i.e. $j!q_1 \rhd j!q_2$. The rule (H-TRANS-INODER) encodes transitive inorder delivery and processing [8, 105]. (H-TRANS-INODER) is similar to (H-INORDER) except that i sends its message q_2 to j indirectly, i.e. $i!p_2 \rhd \ldots \rhd j!q_2$.

5.4.3.3 Constraint Set

Figure 5.17 shows the function *copy* used to constructs the constraint set of an order type.

$$
\begin{array}{lll}
(\textsc{Emp}) & copy(\bullet,\Psi,\Gamma) & = \bullet \\
(\textsc{Sin}) & copy(i!p,\Psi,\Gamma) & = i!p \,\triangleright\, copy(\sigma,\Psi'',\Gamma') \\
 & & \quad \textbf{\textit{if}}\ \ \Gamma \vdash i : C \\
(\textsc{Sin}\exists\textsc{Em}) & copy(\exists i : C \,.\, i!p,\Psi,\Gamma) & = j!p \triangleright copy(\sigma,\Psi'',\Gamma') \\
 & & \quad \textbf{\textit{if}}\ \ \Psi' \vdash i = j,\ \ \Gamma \vdash j : @local\ C \\
(\textsc{Sin}\exists) & copy(\exists i : C \,.\, i!p,\Psi,\Gamma) & = \exists i : C \,.\, i!p \,\triangleright\, copy(\sigma,\Psi'',\Gamma') \\
 & & \quad \textbf{\textit{if}}\ \ \Psi' \nvdash i = j \\
(\textsc{Sin}\exists\textsc{Al}) & copy(\exists i : C \,.\, i!p,\Psi,\Gamma) & = \exists j : C \,.\, j!p \,\triangleright\, copy(\sigma,\Psi'',\Gamma') \\
 & & \quad \textbf{\textit{if}}\ \ \Psi' \vdash i = j,\ \ \Gamma \vdash j : @import\ C \\
(\textsc{Seq}) & copy(\sigma_1;\sigma_2,\Psi,\Gamma) & = copy(\sigma_1,\Psi,\Gamma) \cup copy(\sigma_2,\Psi,\Gamma) \cup (tail(\sigma_1) \triangleright\ head(\sigma_2)) \\
(\textsc{Rec}) & copy([\sigma]^*,\Psi,\Gamma) & = copy(\sigma,\Psi,\Gamma)
\end{array}
$$

$$
\begin{array}{ll}
\textbf{\textit{where}} & \\
pType(p,C) : \sigma\ \overline{T'} \to T & \Psi' = \downarrow_i \Psi \\
\Psi'' = \Psi' \uplus xAliasEnv(C) & \Gamma' = \Gamma \uplus xTypeEnv(C)
\end{array}
$$

Figure 5.17 Function *copy* for constraint set, inspired by copy rule [114].

The constraint set of an empty order type is just an empty order type with no happens-before relations. In (SIN), a singleton order type $i!p$ is replaced by with an $\mathscr{R}_1$ happens-before relation between $i!p$ and the *copy* of the order type σ' of the invoked procedure p. In the recursive invocation of *copy*, the input typing environment Γ is augmented with a typing environment that contains the externally visible typing information of the recipient process C. The externally visible typing information of a capsule include its import declarations and tainted local declarations. Similarly, the input aliasing environment Ψ is augmented with an aliasing environment that contains the externally visible aliasing information for C. The auxiliary functions *xTypeEnv* and *xAliasEnv* constructs the externally visible typing and aliasing environments of a process. The externally visible aliasing information of a capsule includes aliasing of imported process instances and imports of its tainted local declarations.

In (SIN∃ELM), the concrete value of the existentially quantified process variable i is locally declared in the aliasing environment and is known. The quantified variable is replaced by its concrete value. Otherwise, in (SIN∃AL), the existential quantifier is replaced with its alias which could be a process instance with an unknown imported value.

Constraints for a sequence order type $\sigma_1;\sigma_2$ is the union of constraints of its constituents types σ_1

and σ_2. An $\mathscr{R}_2$ happens-before relation is added to denote that the last message send in σ_1 happens-before the first message send in σ_2. Auxiliary functions *head* and *tail* return the first and last message send in an order type. The constraint set for a recursive order type is equal to constraints of its once-folding. In once folding a type variable of a recursive order type is replaced with the empty order type $\bullet$. This is because inorder delivery and processing guarantees that message in one unfolding of the recursive order type are sent, delivered and processed before the messages in its next unfolding.

The function *copy* is similar to transitive application of the copy rule for method specifications [114] in object-oriented programming models. The copy rule for method specifications replaces a method invocation with the specification of the method, after proper substitutions of variables in the specification with their known values.

5.4.3.4 Well-formedness and Blame

An order type is well-formed if its constraint set is well-formed with no message races. In the rule (W-TYPE) a constraint set is well-formed if for any two messages τ_1 and τ_2 sent to the same process i, there should be a happens-before relation $\triangleright$ between them. That is, either τ_1 happens-before τ_2 or τ_2 happens-before τ_1. Note that these messages can be regular or quantified messages.

In rules (T-WIRING DECL) and (T-LET), if the constraint set of a not well-formed then the order type is ill-formed because of a race in its implementation.

5.4.4 Abstraction

For an abstraction-eligible process, abstraction allows hiding of the local messaging behavior of its procedures in their order types. Figure 5.18 shows typing rules for abstraction. Judgements $\Gamma \vdash e : \sigma^{\,T}$ and $\Gamma \overset{a}{\vdash} e : \sigma^{\,T}$ denote type checking before and after abstraction.

Unlike Figure 5.13, the rule (T-PROCESS DECL) in Figure 5.18 type checks a procedure declaration with abstraction. A process C type checks if it type checks in the absence of abstraction, i.e. $\Gamma \vdash C$, and it is abstraction-eligible. A process is abstraction eligible if sequences of invocations of any of its two procedures p_1 and p_2 are do not lead to a race. That is, order types of $\sigma_1;\sigma_2$ and $\sigma_2;\sigma_1$ of these invocations are well-formed. The sequence order type $\sigma_1;\sigma_2$ is the type of a sequence of invocations in

(T-PROCESS DECL)

$$\frac{\Gamma \vdash C \qquad \textbf{\textit{capsule}}\ C(\overline{import})\ \{design\ \overline{state}\ \overline{proc}\} = CT(C) \qquad p_1, p_2 \in \overline{proc} \\ \sigma_1^{\overline{T_1'} \rightarrow T_1} = pType(p_1, C) \qquad \sigma_2^{\overline{T_2'} \rightarrow T_2} = pType(p_2, C) \qquad \Psi = aliasEnv(C) \qquad \Gamma, \Psi \vdash \sigma \qquad \Gamma, \Psi \vdash \sigma'}{\Gamma \overset{a}{\vdash} C}$$

(T-PROC DECL)

$$\frac{\Gamma \overset{abs}{\vdash} C \qquad \Gamma \vdash_C T\ p(\overline{T'\ x'})\{e\} : \sigma^{\overline{T'} \rightarrow T} \qquad \chi = imported(C) \cup tainted(C) \qquad \sigma' = hide(\sigma, \chi)}{\Gamma \overset{a}{\vdash}_C\ T\ p(\overline{T'\ x'})\{e\} : \sigma'^{\ \overline{T'} \rightarrow T}}$$

Figure 5.18 Typing rules for abstraction-eligibility and abstraction

Hide

(EMP)	$hide(\bullet, \chi)$	$=$	$\bullet$	
(SIN)	$hide(i!p, \chi)$	$=$	$i!p$	***if*** $i \in \chi$
(SIN)	$hide(i!p, \chi)$	$=$	$\bullet$	***if*** $i \notin \chi$
(SIN-∃)	$hide(\exists i : C\ .\ i!p, \chi)$	$=$	$\exists i : C\ .\ i!p$	***if*** $i \in \chi$
(SIN-∃)	$hide(\exists i : C\ .\ i!p, \chi)$	$=$	$\bullet$	***if*** $i \notin \chi$
(SEQ)	$hide(\sigma_1; \sigma_2, \chi)$	$=$	$hide(\sigma_1, \chi); hide(\sigma_2, \chi)$	
(REC)	$hide([\sigma]^*, \chi)$	$=$	$[hide(\sigma, \chi)]^*$	

Tainted

(NON)	$tainted(i(), C)$	$=$	$\emptyset$
(COM)	$tainted(i(\bar{j}), C)$	$=$	i ***if*** ***capsule*** $C(\overline{G\ g})\ \{\overline{\textbf{design}\{ins}\ \overline{wire}\}\ ..\} = CT(C)$, $\exists j \in \bar{j}\ .\ j \in \overline{g}\ \vee$ $\exists H\ j(\bar{k}) \in \overline{wire}\ .\ tainted(j(\bar{k}, C) \neq \emptyset$
(ALL)	$tainted(\overline{wire}, C)$	$=$	$\forall wire \in \overline{wire}\ .\ \bigcup tainted(wire, C)$

Figure 5.19 Abstraction function *hide* and taint function *tainted*.

which p_1 is invoked before p_2 is invoked. Similarly, $\sigma_1; \sigma_2$ denotes the type of the alternative invocation sequence. For a process that is abstraction-eligible, order type of its procedures can be abstracted.

The rule (T-PROC DECL) type checks a procedure declaration in the presence of abstraction. The order type of the procedure can be abstracted if its enclosing process is abstraction-eligible. Abstraction hides the local messaging behavior of the procedure from its order type. The local messaging behavior includes messages sent to untainted local process instances of the enclosing process. A local process instance is untainted if it is composed of only untainted instance. A local process instance is tainted if it is composed of an imported process instance or another tainted local instance. A process instance

is tainted if it is composed from an imported or tainted process instance. Auxiliary functions *imported* and *tainted* return imported and tainted process instances of a process. The auxiliary function *hide* exposes message sends of an order type to its imported and tainted process instances and hides the rest. Figure 5.19 shows the definition of *hide*.

To illustrate, the local server process s in Client in Figure 5.3 is untainted because it is not composed of any other process; the local proxy process r is untainted because it is composed of untainted server s. However, the proxy r in Client2 in Figure 5.7 is tainted because it is composed of the imported server s.

5.4.5 Soundness of Abstraction

Abstraction of local messaging behavior of a process C is sound because if it cannot hide a message race. Local messaging behavior includes message sends to locally declared process instances. There are two types of local process instances: untainted and tainted. The abstractions hides message sends to an untainted local process instance. Abstraction is sound because of the following:

1 ***Untainted:*** Encapsulation guarantees that untainted local process instance i and its composing process instances $\overline{j}$ can only be accessed through the procedures of C.

- if message sends to i or $\overline{j}$ lead to a race in a procedure of C, the race will manifest in type checking of the procedure and the procedure does not type check.
- if message sends to i or $\overline{j}$ over multiple procedures of C lead to a race, the race will manifest in checking for abstraction-eligibility and the process does not type check.

2 ***Tainted:*** Messaging behavior for the tainted local variable i and its imported process instances $\overline{j}$ is not abstracted away.

- If message sends to shared $\overline{j}$ in procedures C and procedures of another process D lead to a race, the race will manifest in a process that composes these together and the process does not type check.

5.4.6 Discussion

Random delivery and processing Direct and transitive inorder delivery and processing are not necessary and can be added or removed to support different delivery and processing models. Actors

[6] usually support a random delivery and processing model in which messages can be delivered and processed in any arbitrary order. To model random delivery and processing it is sufficient to remove (H-INORDER) and (H-TRANS-INORDER) from the set of happens-before rules in Figure 5.16.

Conditionals The rule (T-COND) type checks an if expression. Similar to (T-LET), the order type of the expression sequences the order type of its subexpressions e' and e''. This is an over-approximation because the runtime evaluates either e' or e'' and not both. However, this over-approximation is necessary to keep the type checking tractable. To illustrate, consider an alternative order type $[\sigma', \sigma'']$ that encodes σ' and σ'' as exclusive alternatives. To encode the exclusion between σ' and σ'', two separate constraint sets should be constructed and checked to for the well-formedness of $[\sigma', \sigma'']$. In other words, two (i.e. 2^1) constraint sets should be constructed and checked for well-formedness of the type. In general, for an order type with n alternative order types in it, 2^n constraint sets should be constructed and checked, which is intractable.

Message receive A message receive $\zeta\ i?p$ can be encoded as a wait on the future result of a message send $\zeta\ i!p$. A message receive simply adds two happens-before relations to the constraint set of its enclosing procedure. The first happens-before relation is $\zeta\ i!p \triangleright i?p$ which says that a message send happens before a receive on that message. The second happens-before relation is $tail(\sigma) \triangleright \zeta\ i?p$ which says that the processing of message p in process i with the order type σ happens before the message receive. Addition of message receives are straightforward and thus are omitted.

Memory effects and benign races Precision of race detection can be improved by integrating memory effects of messages into the type system. Using effects, a message race happens when a process receives two messages with conflicting memory effects from another process with no happens-before relations between them. Two memory access conflict if they both access the same memory location and one of them is a write access [64]. Addition of an *@benign* annotation to a process which causes a benign and harmless message race can allow the process to type check.

5.5 Related Work

Previous work in the following categories are related to this chapter.

Behavioral session types and contracts Order types, similar to session types [25, 26, 28, 40, 43, 79–81, 119, 120, 160, 178] denote the messaging behavior of a process but explore a different design space. A session type *prescribes* a *global* communication protocol [116] in a session-based top-down programming model in which communication is carried over channels and is structured by sessions [79]. Addition of channels and sessions to structure communication over channels allows the application of session types to a variety of multithreaded [43], message passing [115, 120] and sequential [81, 119, 169] programming models. Session types and their underlying session-based programming are also added to languages like Erlang, Java and Python. However, an order type is a *local* type that *describes* the *abstracted* messaging behavior of a process as specified by its program text in a bottom-up programming model without channels and sessions. Prescriptive and global nature of session types and descriptive and local nature of order types can complement each other.

State-space exploration Systematic verification and testing techniques explore the entire state-space of a concurrent message-passing program including all ordering of its messages, which is intractable. The exploration can be optimized and become tractable using partial order reduction techniques [93, 161]. Previous work [162] uses message race relations to improve a partial reduction technique for dynamic testing of actor programs. Order types make modular reasoning about a message passing program tractable by statically disallowing message races and explain their causes.

In dynamic testing of actor programs, previous work [162] uses message race relations to prune out part of state-space. These techniques are usually global and any changes in a program may invalidate their reasoning. Order types, unlike global state-space exploration techniques, are modular.

Multithreaded programming Previous work prevents low [64, 65] and high level data races [66, 171] in multithreaded programs and and guarantees its deterministic execution [29]. However, data race detection techniques are not directly applicable to detect message races in message-passing models.

CHAPTER 6. CONCLUSION AND FUTURE WORK

The concurrency revolution and the need for performant software requires sequentially trained software engineers to write concurrent software. Engineering of concurrent software could be made easier with modular reasoning. Using modular reasoning a system is understood one module at a time and changes in implementations of other modules cannot invalidate the understanding about this module as long as interfaces of those modules stay the same. This thesis identifies three challenges that can make modular reasoning about a concurrent software difficult or intractable: arbitrary thread interference, arbitrary module inheritance and arbitrary message orders. This thesis proposes and formalizes an implicit concurrent message passing programming model with interference control, concurrent behavioral subtyping and order types to enable modular reasoning about a concurrent software in the presence of arbitrary interference, inheritance and message orders.

6.1 Interference Control Framework

The interference control framework enables modular reasoning by guaranteeing that in a concurrent program interference happens only at explicitly specified program points and at each interference point the interference behavior is known. This is in contrast to the state of the art programming models in which interference can happen between any two instructions of a software and interference behaviors are unknown. The interference control framework allows the standard and well-known Hoare-style [77] modular reasoning for sequential programs to be adapted to understand a concurrent program in the presence of interference. To reason about a concurrent program first its interference points and their interference behaviors are computed syntactically and modularly; second interference is taken into account by inserting interference behaviors at their corresponding interference points; and third sequential Hoare-style reasoning is applied to understand the behavior of the program. Our adaptation of standard

Hoare-style reasoning could make it easier for developments tools, such as KeY [23], that integrate construction of sequential software and formal verification [24], to adapt to concurrent software.

6.2 Concurrent Behavioral Subtyping

Concurrent behavioral subtyping enables modular reasoning by guaranteeing that the behavior of a module is compatible with the behavior of the module it inherits from. This is in contrast to state of the art where there is no relation between behaviors of two modules in an inheritance hierarchy. Concurrent behavioral subtyping allows the standard modular supertype abstraction reasoning [97] to be adapted to understand a concurrent program and specially its invocations in the presence of inheritance and dynamic dispatch. In supertype abstraction reasoning, an invocation can be understood using the static type of its receiver, independent of its dynamic types. Our adaptation of supertype abstraction reasoning could make it easier for programming logics [23, 117, 128, 131, 134] that support supertype abstraction reasoning for sequential software to adapt to concurrent software.

Concurrent behavioral subtyping provides a new definition for behavioral subtyping [103] using standard behavioral subtyping and interference subtyping.

Our definition of concurrent behavioral subtyping allows concurrent behavioral subtyping to be reduced to standard and well-known interface subtyping [45, 96, 104] in the presence of encapsulated inheritance [155]. In encapsulated inheritance a module accesses the state of the module it inherits from through its procedures and not directly. Our empirical studies of 554,864 Java and 416 Akka projects in GitHub reveal that more than 90% of these projects respect encapsulated inheritance. This means for majority of the projects in this study concurrent subtyping can be guaranteed using only standard interface subtyping [96, 104]. Interface subtyping can be easily guaranteed syntactically using techniques like specification inheritance [45] that automatically combine behavior specifications of modules in an inheritance hierarchy.

6.3 Order Types

Order types guarantee that a well-formed concurrent program is free from message races. Freedom from message races allows for tractable modular reasoning about message orders. Existential order type

of a module allows encoding of its unknown dependencies and enables bottom-up program design and construction. Abstraction decreases complexity by hiding local messaging behavior for abstraction-eligible modules, which in turn could improve the scalability of type checking. Blame assignment enables programmers to quickly ascertain racy modules to be fixed or replaced by properly blaming the module responsible for a bad module composition or bad message composition and not the module in which the race happens. Disallowing message races can not only enable modular reasoning but also prevent bugs due to these races [93, 163] which are hard to find and fix.

6.4 Future Work

There are several avenues for future work that we would like to investigate.

6.4.1 Modular Reasoning about Traditionally Global Properties

In this thesis, we enable modular reasoning about concurrency, a property that is traditionally thought to be global. An avenue of future work is to study how to enable modular reasoning about other traditionally global properties. Security and privacy [149, 150, 164] and communication properties [80, 121] are few examples of these properties. The challenge in reasoning about these properties is that they could lead to global reasoning techniques and analyses that may not be as scalable as modular reasoning techniques or lead to global specifications that may impede bottom-up software construction. To address this challenge we anticipate the use of linguistic constructs that allow security and communication information of interest to be explicated in the interface of a module. Such interfaces include sufficient information to allow sound modular reasoning about the property of interest and at the same time not expose the unnecessary implementation details of a module. Abstraction could prevent exposure of unnecessary implementation details and also decrease the complexity of checking these properties. Construction of these interfaces should not extensively add to the burden of the programmer of the module. Automatic inference of these interfaces along with light-weight programmer annotations could be used to reduce the programmer's burden.

Another way to address the reasoning challenge, in top-down software construction, is to integrate local order types and global behavioral types (global specifications) such as session types [80, 116].

Session types allow modular reasoning about communication properties for enforcing of communication protocols and prevention of deadlocks. The challenge in using session types is that they are designed for channel-based programming using sessions that is not supported in mainstream programming languages. To address this challenge we anticipate to support session-based programming in our programming model. With addition of sessions, the global session type for participants of a protocol can be automatically inferred using local order types of its participants [116].

6.4.2 Concurrent Behavioral Subtyping and Hoare-style Reasoning

Another avenue of future work is to study the expressiveness of concurrent behavioral subtyping. We anticipate to investigate if concurrent subtyping is expressive enough to encode behavioral subtyping for both sequential and concurrent software with and without interference. Standard behavioral subtyping [10], strong behavioral subtyping [104] and weak behavioral subtyping [44] are few examples of behavioral subtyping for sequential and concurrent software.

Another avenue of future work is to study if our Hoare-style modular reasoning could be augmented to encode previous reasoning techniques. Thread-modular [67], rely-guarantee [87], deny-guarantee [48] and concurrent abstract predicates [47] are examples of these reasoning techniques. The challenge in encoding these reasoning techniques is that some are built on top of low-level logics such as concurrent separation logic [127] and use fine-grained access permissions [34] that are not supported in our programming model and logic. To address this challenge we anticipate integrating concurrent behavioral subtyping and separation logic using permissions.

6.4.3 Interference Closure and Interference Behavior

Another avenue of future work is to study the granularity of interference closure and its behavior. One challenge is that the interference closure including all procedure of a module is coarse. To address this challenge we anticipate the use of order types to know which procedures of the module may not be invoked by other modules and therefore excluded from its interference closure. However, this could lead to a reasoning in which to understand a module one need to look at the interface of all modules that invoke the procedures of the module which may not be as desirable. Another challenge is the need for an oracle that knows about the behavior of an interference closure. To address this challenge we anticipate

to use techniques that automatically infer loop invariants using functional behaviors of procedures [70] or ask programmers to specify the behavior of them interference closure.

6.4.4 Applicability

The last avenue of future work is to further study the integration of our proposed programming model with programming languages and models for mobile programming such as Android, message passing programming such as Message Passing Interface (MPI) [1] and actor programming such as Akka [8]. The challenge is that our programming model requires typed messages, sequential process execution and encapsulation. To address this challenge we anticipate to integrate these languages with a light-weight ownership model to guarantee encapsulation and use variations of these models that do not rely on ambiguous interfaces [72]. A module with ambiguous interface accepts a single message and relies on its implementation to process the message differently based on its attributes. Ambiguous interfaces are popular in languages and models with event loops [74]. Another challenge is that the addition of encapsulation may decrease the expressiveness of a programming model. We would like to study this hypothesis by extending our experimental evaluation of encapsulated inheritance using visibility-based encapsulation to include encapsulation in general. In this study we would like to consider semantic features such as aliasing and global variables that may lead to violation of encapsulation.

BIBLIOGRAPHY

[1] Message passing interface. http://www.mpi-forum.org/.

[2] Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. Semantics of transactional memory and automatic mutual exclusion. *ACM Trans. Program. Lang. Syst.*, 33(1):2:1–2:50, January 2011.

[3] Martín Abadi and K Rustan M Leino. A logic of object-oriented programs. In *Verification: Theory and Practice*, pages 11–41. Springer, 2003.

[4] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, Cambridge, MA, USA, 1986.

[5] Gul Agha, Svend Frølund, WooYoung Kim, Rajendra Panwar, Anna Patterson, and Daniel Sturman. Abstraction and modularity mechanisms for concurrent computing. *IEEE Parallel Distrib. Technol.*, 1(2):3–14, May 1993.

[6] Gul Agha and Carl Hewitt. Concurrent programming using actors: Exploiting large-scale parallelism. In *Proceedings of the Fifth Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 19–41, London, UK, 1985. Springer-Verlag.

[7] Gul A. Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. *J. Funct. Program.*, 7(1):1–72, January 1997.

[8] Akka. http://akka.io/.

[9] Pierre America. Inheritance and subtyping in a parallel object-oriented language. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '87, pages 234–242, London, UK, 1987. Springer-Verlag.

[10] Pierre America. Designing an object-oriented programming language with behavioural subtyping. In *Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages*, pages 60–90, London, UK, 1991. Springer-Verlag.

[11] Wladimir Araujo, Lionel Briand, and Yvan Labiche. Concurrent contracts for Java in JML. In *Proceedings of the 2008 19th International Symposium on Software Reliability Engineering*, ISSRE '08, pages 37–46, Washington, DC, USA, 2008. IEEE Computer Society.

[12] Joe Armstrong. Erlang. *Commun. ACM*, 53(9):68–75, September 2010.

[13] Mark Astley. The actor foundry: A Java-based actor programming environment. Technical report, University of Illinois at Urbana-Champaign, 1989.

[14] Mark Astley, Thomas Clausen, and James Waldby. ActorFoundry. http://osl.cs.illinois.edu/software/actor-foundry/.

[15] Mehdi Bagherzadeh, Robert Dyer, Rex D. Fernando, José Sánchez, and Hridesh Rajan. Modular reasoning in the presence of event subtyping. In *Proceedings of the 14th International Conference on Modularity*, MODULARITY 2015, pages 117–132, New York, NY, USA, 2015. ACM.

[16] Mehdi Bagherzadeh, Robert Dyer, Rex D. Fernando, José Sánchez, and Hridesh Rajan. Modular reasoning in the presence of event subtyping. In *Transactions on Modularity and Composition, special edition: Best papers of Modularity'15*, 2016.

[17] Mehdi Bagherzadeh, Gary T. Leavens, and Robert Dyer. Applying translucid contracts for modular reasoning about aspect and object oriented events. In *Proceedings of the 10th International Workshop on Foundations of Aspect-oriented Languages*, FOAL '11, pages 31–35, New York, NY, USA, 2011. ACM.

[18] Mehdi Bagherzadeh and Hridesh Rajan. Panini: A concurrent programming model for solving pervasive and oblivious interference. In *Proceedings of the 14th International Conference on Modularity*, MODULARITY 2015, pages 93–108, New York, NY, USA, 2015. ACM.

[19] Mehdi Bagherzadeh, Hridesh Rajan, and Ali Darvish. On exceptions, events and observer chains. In *Proceedings of the 12th Annual International Conference on Aspect-oriented Software Development*, AOSD '13, pages 185–196, New York, NY, USA, 2013. ACM.

[20] Mehdi Bagherzadeh, Hridesh Rajan, Gary T Leavens, and S Mooney. Translucid contracts for aspect-oriented interfaces. In *the Foundations of Aspect-Oriented Languages workshop (FOAL 2010)*, 2010.

[21] Mehdi Bagherzadeh, Hridesh Rajan, Gary T. Leavens, and Sean Mooney. Translucid contracts: Expressive specification and modular verification for aspect-oriented interfaces. In *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*, AOSD '11, pages 141–152, New York, NY, USA, 2011. ACM.

[22] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *Proceedings of the 2004 International Conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, CASSIS'04, pages 49–69, Berlin, Heidelberg, 2005. Springer-Verlag.

[23] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of Object-oriented Software: The KeY Approach*. Springer-Verlag, Berlin, Heidelberg, 2007.

[24] Bernhard Beckert, Tony Hoare, Reiner Hahnle, Douglas R. Smith, Cordell Green, Silvio Ranise, Cesare Tinelli, Thomas Ball, and Sriram K. Rajamani. Intelligent systems and formal methods in software engineering. *IEEE Intelligent Systems*, 21(6):71–81, November 2006.

[25] Andi Bejleri and Nobuko Yoshida. Synchronous multiparty session types. *Electron. Notes Theor. Comput. Sci.*, 241:3–33, July 2009.

[26] Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. Global progress in dynamically interleaved multiparty sessions. In *International Conference on Concurrency Theory*, pages 418–433. Springer, 2008.

[27] Joshua Bloch. *Effective Java (2nd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2 edition, 2008.

[28] Laura Bocchi, Kohei Honda, Emilio Tuosto, and Nobuko Yoshida. A theory of design-by-contract for distributed multiparty interactions. In *Proceedings of the 21st International Conference on Concurrency Theory*, CONCUR'10, pages 162–176, Berlin, Heidelberg, 2010. Springer-Verlag.

[29] Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A type and effect system for deterministic parallel Java. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 97–116, New York, NY, USA, 2009. ACM.

[30] Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. PACER: Proportional detection of data races. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 255–268, New York, NY, USA, 2010. ACM.

[31] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 259–270, New York, NY, USA, 2005. ACM.

[32] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 211–230, New York, NY, USA, 2002. ACM.

[33] Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free Java programs. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, pages 56–69, New York, NY, USA, 2001. ACM.

[34] John Boyland. Checking interference with fractional permissions. In *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, pages 55–72, Berlin, Heidelberg, 2003. Springer-Verlag.

[35] Denis Caromel, Ludovic Henrio, and Bernard Paul Serpette. Asynchronous and deterministic objects. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 123–134, New York, NY, USA, 2004. ACM.

[36] Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Enhancing modular OO verification with separation logic. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 87–99, New York, NY, USA, 2008. ACM.

[37] Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 292–310, New York, NY, USA, 2002. ACM.

[38] 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.

[39] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '98, pages 48–64, New York, NY, USA, 1998. ACM.

[40] Silvia Crafa. Behavioural types for actor systems. *CoRR*, abs/1206.1687, 2012.

[41] Christoph Csallner, Leonidas Fegaras, and Chengkai Li. New ideas track: Testing Mapreduce-style programs. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 504–507, New York, NY, USA, 2011. ACM.

[42] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Theo D'Hondt, and Wolfgang De Meuter. Ambient-oriented programming in AmbientTalk. In *Proceedings of the 20th European Conference on Object-Oriented Programming*, ECOOP'06, pages 230–254, Berlin, Heidelberg, 2006. Springer-Verlag.

[43] Mariangiola Dezani-Ciancaglini, Dimitris Mostrous, Nobuko Yoshida, and Sophia Drossopoulou. Session types for object-oriented languages. In *European Conference on Object-Oriented Programming*, pages 328–352. Springer Berlin Heidelberg, 2006.

[44] Krishna Kishore Dhara and Gary T. Leavens. Weak behavioral subtyping for types with mutable objects. *Electronic Notes in Theoretical Computer Science*, 1:91 – 113, 1995.

[45] Krishna Kishore Dhara and Gary T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering*, ICSE '96, pages 258–267, Washington, DC, USA, 1996. IEEE Computer Society.

[46] Crystal Chang Din and Olaf Owe. Compositional reasoning about active objects with shared futures. *Form. Asp. Comput.*, 27(3):551–572, May 2015.

[47] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 504–528, Berlin, Heidelberg, 2010. Springer-Verlag.

[48] Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. Deny-guarantee reasoning. In *Proceedings of the 18th European Symposium on Programming Languages and Systems*, ESOP '09, pages 363–377, Berlin, Heidelberg, 2009. Springer-Verlag.

[49] Johan Dovland, Einar Broch Johnsen, Olaf Owe, and Martin Steffen. Lazy behavioral subtyping. In *Proceedings of the 15th International Symposium on Formal Methods*, FM '08, pages 52–67, Berlin, Heidelberg, 2008. Springer-Verlag.

[50] Robert Dyer, Mehdi Bagherzadeh, Hridesh Rajan, and Yuanfang Cai. A preliminary study of quantified, typed events. In *AOSD Workshop Empirical Evaluation of Software Composition Techniques*, 2010.

[51] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Proceedings of the 2013*

International Conference on Software Engineering, ICSE '13, pages 422–431, Piscataway, NJ, USA, 2013. IEEE Press.

[52] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: Ultra-large-scale software repository and source-code mining. *ACM Trans. Softw. Eng. Methodol.*, 25(1):7:1–7:34, December 2015.

[53] Robert Dyer, Hridesh Rajan, and Yuanfang Cai. An exploratory study of the design impact of language features for aspect-oriented interfaces. In *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development*, AOSD '12, pages 143–154, New York, NY, USA, 2012. ACM.

[54] Robert Dyer, Hridesh Rajan, and Yuanfang Cai. Language features for software evolution and aspect-oriented interfaces: An exploratory study. In Gary T. Leavens, Shigeru Chiba, and Éric Tanter, editors, *Transactions on Aspect-Oriented Software Development X*, pages 148–183. Springer-Verlag, Berlin, Heidelberg, 2013.

[55] Robert Dyer, Hridesh Rajan, and Tien N. Nguyen. Declarative visitors to ease fine-grained source code mining with full history on billions of ast nodes. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, GPCE '13, pages 23–32, New York, NY, USA, 2013. ACM.

[56] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. A calculus of atomic actions. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 2–15, New York, NY, USA, 2009. ACM.

[57] 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. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, EuroSys '06, pages 177–190, New York, NY, USA, 2006. ACM.

[58] Azadeh Farzan and P. Madhusudan. Causal atomicity. In *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV'06, pages 315–328, Berlin, Heidelberg, 2006. Springer-Verlag.

[59] Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *Proceedings of the 16th European Symposium on Programming*, ESOP'07, pages 173–188, Berlin, Heidelberg, 2007. Springer-Verlag.

[60] Rex D. Fernando, Robert Dyer, and Hridesh Rajan. Event type polymorphism. In *Proceedings of the Eleventh Workshop on Foundations of Aspect-Oriented Languages*, FOAL '12, pages 33–38, New York, NY, USA, 2012. ACM.

[61] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. Technical report, 2000.

[62] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '02, pages 48–59, New York, NY, USA, 2002. ACM.

[63] Robert Bruce Findler, Mario Latendresse, and Matthias Felleisen. Behavioral contracts and behavioral subtyping. In *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-9, pages 229–236, New York, NY, USA, 2001. ACM.

[64] Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 219–232, New York, NY, USA, 2000. ACM.

[65] Cormac Flanagan and Stephen N. Freund. FastTrack: Efficient and precise dynamic race detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 121–133, New York, NY, USA, 2009. ACM.

[66] Cormac Flanagan, Stephen N. Freund, Marina Lifshin, and Shaz Qadeer. Types for atomicity: Static checking and inference for Java. *ACM Trans. Program. Lang. Syst.*, 30(4):20:1–20:53, August 2008.

[67] Cormac Flanagan, Stephen N. Freund, and Shaz Qadeer. Thread-modular verification for shared-memory programs. In *Proceedings of the 11th European Symposium on Programming Languages and Systems*, ESOP '02, pages 262–277, London, UK, 2002. Springer-Verlag.

[68] Cormac Flanagan, Stephen N. Freund, Shaz Qadeer, and Sanjit A. Seshia. Modular verification of multithreaded programs. *Theor. Comput. Sci.*, 338(1-3):153–183, June 2005.

[69] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 234–245, New York, NY, USA, 2002. ACM.

[70] Carlo Alberto Furia and Bertrand Meyer. Inferring loop invariants using postconditions. In Andreas Blass, Nachum Dershowitz, and Wolfgang Reisig, editors, *Fields of Logic and Computation*, pages 277–300, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[71] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[72] Joshua Garcia, Daniel Popescu, Gholamreza Safi, William G. J. Halfond, and Nenad Medvidovic. Identifying message flow in distributed event-based systems. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 367–377, New York, NY, USA, 2013. ACM.

[73] Simon Gay and Malcolm Hole. Subtyping for session types in the Pi calculus. *Acta Inf.*, 42(2):191–225, November 2005.

[74] Philipp Haller and Martin Odersky. Scala Actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, February 2009.

[75] Per Brinch Hansen. Distributed processes: A concurrent programming concept. In Per Brinch Hansen, editor, *The Origin of Concurrent Programming*, pages 444–463. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[76] 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.

[77] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

[79] Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *Proceedings of the 7th European Symposium on Programming: Programming Languages and Systems*, ESOP '98, pages 122–138, London, UK, 1998. Springer-Verlag.

[80] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 273–284, New York, NY, USA, 2008. ACM.

[81] Raymond Hu, Nobuko Yoshida, and Kohei Honda. Session-based distributed programming in Java. In *Proceedings of the 22nd European Conference on Object-Oriented Programming*, ECOOP '08, pages 516–541, Berlin, Heidelberg, 2008. Springer-Verlag.

[82] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, May 2001.

[83] Shams M. Imam and Vivek Sarkar. Integrating task parallelism with actors. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 753–772, New York, NY, USA, 2012. ACM.

[84] Michael Isard and Andrew Birrell. Automatic mutual exclusion. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems*, HOTOS'07, pages 3:1–3:6, Berkeley, CA, USA, 2007. USENIX Association.

[85] Bart Jacobs, Jan Smans, Frank Piessens, and Wolfram Schulte. A statically verifiable programming model for concurrent object-oriented programs. In *Proceedings of the 8th International Conference on Formal Methods and Software Engineering*, ICFEM'06, pages 420–439, Berlin, Heidelberg, 2006. Springer-Verlag.

[86] Einar Broch Johnsen, Olaf Owe, and Ingrid Chieh Yu. Creol: A type-safe object-oriented model for distributed concurrent systems. *Theor. Comput. Sci.*, 365(1):23–66, November 2006.

[87] Cliff B Jones. Specification and design of (parallel) programs. In *IFIP congress*, volume 83, pages 321–332, 1983.

[88] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 637–650, New York, NY, USA, 2015. ACM.

[89] Günter Kniesel and Dirk Theisen. JAC: Access right based encapsulation for Java. *Softw. Pract. Exper.*, 31(6):555–576, May 2001.

[90] Aditya Kulkarni, Yu David Liu, and Scott F. Smith. Task types for pervasive atomicity. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 671–690, New York, NY, USA, 2010. ACM.

[91] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[92] James Larus and Christos Kozyrakis. Transactional memory. *Commun. ACM*, 51(7):80–88, July 2008.

[93] Steven Lauterburg, Mirco Dotta, Darko Marinov, and Gul Agha. A framework for state-space exploration of Java-based actor programs. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 468–479, Washington, DC, USA, 2009. IEEE Computer Society.

[94] R. Greg Lavender and Douglas C. Schmidt. Pattern languages of program design 2. In John M. Vlissides, James O. Coplien, and Norman L. Kerth, editors, *Active Object: An Object Behavioral Pattern for Concurrent Programming*, pages 483–499. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[95] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, May 2006.

[96] Gary T. Leavens and David A. Naumann. Behavioral subtyping, specification inheritance, and modular reasoning. *ACM Trans. Program. Lang. Syst.*, 37(4):13:1–13:88, August 2015.

[97] Gary T. Leavens and William E. Weihl. Specification and verification of object-oriented programs using supertype abstraction. *Acta Inf.*, 32(8):705–778, August 1995.

[98] Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, May 2006.

[99] Jinjiang Lei and Zongyan Qiu. Modular reasoning for message-passing programs. In *International Colloquium on Theoretical Aspects of Computing*, pages 277–294. Springer, 2014.

[100] Eric Lin. PaniniJ: Adding the capsule programming abstraction to Java to provide linguistic support for modular reasoning in concurrent program design. Master's thesis, Iowa State University, Ames, IA, USA, 2016.

[101] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, December 1975.

[102] Barbara Liskov and Robert Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Trans. Program. Lang. Syst.*, 5(3):381–404, July 1983.

[103] Barbara Liskov and Jeannette M. Wing. Specifications and their use in defining subtypes. In *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '93, pages 16–28, New York, NY, USA, 1993. ACM.

[104] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, November 1994.

[105] Yuheng Long, Mehdi Bagherzadeh, Eric Lin, Ganesha Upadhyaya, and Hridesh Rajan. On ordering problems in message passing software. In *Proceedings of the 15th International Conference on Modularity*, MODULARITY 2016, pages 54–65, New York, NY, USA, 2016. ACM.

[106] Yuheng Long, Sean L. Mooney, Tyler Sondag, and Hridesh Rajan. Implicit invocation meets safe, implicit concurrency. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, GPCE '10, pages 63–72, New York, NY, USA, 2010. ACM.

[107] Yuheng Long and Hridesh Rajan. A type-and-effect system for asynchronous, typed events. In *Proceedings of the 15th International Conference on Modularity*, MODULARITY 2016, pages 42–53, New York, NY, USA, 2016. ACM.

[108] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 329–339, New York, NY, USA, 2008. ACM.

[109] Satoshi Matsuoka and Akinori Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-oriented Programming*, pages 107–150. MIT Press, Cambridge, MA, USA, 1993.

[110] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition, 1988.

[111] Robin Milner. *Communicating and Mobile Systems: The Pi-calculus*. Cambridge University Press, New York, NY, USA, 1999.

[112] Naftaly H. Minsky. Towards alias-free pointers. In *Proceedings of the 10th European Conference on Object-Oriented Programming*, ECOOP '96, pages 189–209, London, UK, 1996. Springer-Verlag.

[113] Sean L. Mooney. A unified design of capsules. Master's thesis, Iowa State University, Ames, IA, USA, 2015.

[114] Carroll Morgan. Procedures, parameters, and abstraction: Separate concerns. *Science of Computer Programming*, 11(1):17 – 27, 1988.

[115] Dimitris Mostrous and Vasco T. Vasconcelos. Session typing for a featherweight Erlang. In *Proceedings of the 13th International Conference on Coordination Models and Languages*, COORDINATION'11, pages 95–109, Berlin, Heidelberg, 2011. Springer-Verlag.

[116] Dimitris Mostrous, Nobuko Yoshida, and Kohei Honda. Global principal typing in partially commutative asynchronous sessions. In *Proceedings of the 18th European Symposium on Programming Languages and Systems*, ESOP '09, pages 316–332, Berlin, Heidelberg, 2009. Springer-Verlag.

[117] Peter Müller. *Modular Specification and Verification of Object-oriented Programs*. Springer-Verlag, Berlin, Heidelberg, 2002.

[118] Stas Negara, Rajesh K. Karmani, and Gul Agha. Inferring ownership transfer for efficient message passing. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 81–90, New York, NY, USA, 2011. ACM.

[119] Rumyana Neykova. Session types go dynamic or how to verify your Python conversations. In *Proceedings 6th Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software, PLACES 2013, Rome, Italy, 23rd March 2013.*, pages 95–102, 2013.

[120] Rumyana Neykova and Nobuko Yoshida. Multiparty session actors. In *International Conference on Coordination Languages and Models*, pages 131–146. Springer, 2014.

[121] Nicholas Ng and Nobuko Yoshida. Static deadlock detection for concurrent go by global session graph synthesis. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 174–184, New York, NY, USA, 2016. ACM.

[122] Hoan Anh Nguyen, Robert Dyer, Tien N. Nguyen, and Hridesh Rajan. Mining preconditions of APIs in large-scale code corpus. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 166–177, New York, NY, USA, 2014. ACM.

[123] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. Type and effect systems. In *Principles of Program Analysis*, pages 283–363. Springer, 1999.

[124] Piotr Nienaltowski, Bertrand Meyer, and Jonathan S. Ostroff. Contracts for concurrency. *Formal Aspects of Computing*, 21(4):305–318, 2009.

[125] O. M. Nierstrasz. Active objects in hybrid. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '87, pages 243–253, New York, NY, USA, 1987. ACM.

[126] Tobias Nipkow and Leonor Prensa Nieto. Owicki/Gries in Isabelle/HOL. In *Proceedings of the Second Internationsl Conference on Fundamental Approaches to Software Engineering*, FASE '99, pages 188–203, London, UK, 1999. Springer-Verlag.

[127] Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, April 2007.

[128] David von Oheimb and Tobias Nipkow. Hoare logic for NanoJava: Auxiliary variables, side effects, and virtual methods revisited. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods - Getting IT Right*, FME '02, pages 89–105, London, UK, UK, 2002. Springer-Verlag.

[129] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319–340, 1976.

[130] Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 247–258, New York, NY, USA, 2005. ACM.

[131] Matthew John Parkinson. *Local reasoning for Java*. PhD thesis, University of Cambridge, 2006.

[132] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, December 1972.

[133] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

[134] Cees Pierik. *Validation techniques for object-oriented proof outlines*. PhD thesis, Utrecht University, 2006.

[135] Hridesh Rajan. Building scalable software systems in the multicore era. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, pages 293–298, New York, NY, USA, 2010. ACM.

[136] Hridesh Rajan. Capsule-oriented programming. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 611–614, Piscataway, NJ, USA, 2015. IEEE Press.

[137] Hridesh Rajan, Steven M. Kautz, Eric Lin, Sean L. Mooney, Yuheng Long, and Ganesha Upadhyaya. Capsule-oriented programming. Technical Report 13-01, Iowa State University, 2013.

[138] Hridesh Rajan, Steven M. Kautz, Eric Lin, Sean L. Mooney, Yuheng Long, and Ganesha Upadhyaya. Capsule-oriented programming in the Panini language. Technical Report 14-08, Iowa State University, 2014.

[139] Hridesh Rajan, Steven M. Kautz, and Wayne Rowcliffe. Concurrency by modularity: Design patterns, a case in point. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 790–805, New York, NY, USA, 2010. ACM.

[140] Hridesh Rajan and Gary T. Leavens. Quantified, typed events for improved separation of concerns. Technical Report 07-14d, Iowa State University, 2007.

[141] Hridesh Rajan and Gary T. Leavens. Ptolemy: A language with quantified, typed events. In *Proceedings of the 22nd European Conference on Object-Oriented Programming*, ECOOP '08, pages 155–179, Berlin, Heidelberg, 2008. Springer-Verlag.

[142] Hridesh Rajan and Gary T. Leavens. Design, semantics and implementation of the Ptolemy programming language: A language with quantified typed events. Technical Report 15-10, Iowa State University, 2015.

[143] Hridesh Rajan, Tien N Nguyen, Robert Dyer, and Hoan Anh Nguyen. Boa website. http://boa.cs.iastate.edu/, 2012.

[144] Paruj Ratanaworabhan, Martin Burtscher, Darko Kirovski, Benjamin Zorn, Rahul Nagpal, and Karthik Pattabiraman. Detecting and tolerating asymmetric races. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 173–184, New York, NY, USA, 2009. ACM.

[145] Henrique Rebêlo, Gary T. Leavens, Mehdi Bagherzadeh, Hridesh Rajan, Ricardo Lima, Daniel M. Zimmerman, Márcio Cornélio, and Thomas Thüm. AspectJML: Modular specification and runtime checking for crosscutting contracts. In *Proceedings of the 13th International Conference on Modularity*, MODULARITY '14, pages 157–168, New York, NY, USA, 2014. ACM.

[146] Henrique Rebêlo, Gary T. Leavens, Mehdi Bagherzadeh, Hridesh Rajan, Ricardo Lima, Daniel M. Zimmerman, Márcio Cornélio, and Thomas Thüm. Modularizing crosscutting contracts with AspectJML. In *Proceedings of the Companion Publication of the 13th International Conference on Modularity*, MODULARITY '14, pages 21–24, New York, NY, USA, 2014. ACM.

[147] Martin Rinard, Alexandru Salcianu, and Suhabe Bugrara. A classification system and analysis for aspect-oriented programs. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, SIGSOFT '04/FSE-12, pages 147–158, New York, NY, USA, 2004. ACM.

[148] Edwin Rodríguez, Matthew Dwyer, Cormac Flanagan, John Hatcliff, Gary T. Leavens, and Robby. Extending JML for modular specification and verification of multi-threaded programs. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, ECOOP'05, pages 551–576, Berlin, Heidelberg, 2005. Springer-Verlag.

[149] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J.Sel. A. Commun.*, 21(1):5–19, September 2006.

[150] Alireza Sadeghi, Hamid Bagheri, and Sam Malek. Analysis of Android inter-app security vulnerabilities using COVERT. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 725–728, Piscataway, NJ, USA, 2015. IEEE Press.

[151] Jan Schäfer and Arnd Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 275–299, Berlin, Heidelberg, 2010. Springer-Verlag.

[152] Christophe Scholliers, Éric Tanter, and Wolfgang De Meuter. Parallel actor monitors: Disentangling task-level parallelism from data partitioning in the actor model. *Sci. Comput. Program.*, 80:52–64, February 2014.

[153] Steve M. Shaner, Gary T. Leavens, and David A. Naumann. Modular verification of higher-order methods with mandatory calls specified by model programs. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 351–368, New York, NY, USA, 2007. ACM.

[154] Yannis Smaragdakis, Anthony Kay, Reimer Behrends, and Michal Young. Transactions with isolation and cooperation. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 191–210, New York, NY, USA, 2007. ACM.

[155] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPLSA '86, pages 38–45, New York, NY, USA, 1986. ACM.

[156] StackOverFlow. http://stackoverflow.com/questions/11966763/java-encapsulation.

[157] Christopher A. Stone, Melissa E. O'Neill, and The OCM Team. Observationally cooperative multithreading. In *Proceedings of the ACM International Conference Companion on Object*

Oriented Programming Systems Languages and Applications Companion, OOPSLA '11, pages 205–206, New York, NY, USA, 2011. ACM.

[158] Herb Sutter and James Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, September 2005.

[159] Kasper Svendsen and Lars Birkedal. Impredicative concurrent abstract predicates. In *European Symposium on Programming Languages and Systems*, pages 149–168. Springer, 2014.

[160] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *Proceedings of the 6th International PARLE Conference on Parallel Architectures and Languages Europe*, PARLE '94, pages 398–413, London, UK, 1994. Springer-Verlag.

[161] Samira Tasharofi, Peter Dinges, and Ralph E. Johnson. Why do Scala developers mix the actor model with other concurrency models? In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 302–326, Berlin, Heidelberg, 2013. Springer-Verlag.

[162] Samira Tasharofi, Rajesh K. Karmani, Steven Lauterburg, Axel Legay, Darko Marinov, and Gul Agha. TransDPOR: A novel dynamic partial-order reduction technique for testing actor programs. In *Proceedings of the 14th Joint IFIP WG 6.1 International Conference and Proceedings of the 32nd IFIP WG 6.1 International Conference on Formal Techniques for Distributed Systems*, FMOODS'12/FORTE'12, pages 219–234, Berlin, Heidelberg, 2012. Springer-Verlag.

[163] Samira Tasharofi, Michael Pradel, Yu Lin, and Ralph Johnson. Bita: Coverage-guided, automatic testing of actor programs. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 114–124, Nov 2013.

[164] Omer Tripp, Marco Pistoia, Stephen J. Fink, Manu Sridharan, and Omri Weisman. TAJ: Effective taint analysis of web applications. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 87–97, New York, NY, USA, 2009. ACM.

[165] Aaron Turon, Derek Dreyer, and Lars Birkedal. Unifying refinement and Hoare-style reasoning in a logic for higher-order concurrency. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 377–390, New York, NY, USA, 2013. ACM.

[166] Typesafe. http://www.typesafe.com/company/casestudies.

[167] Ganesha Upadhyaya. Abstraction and performance, together at last: autotuning message-passing concurrency on the Java virtual machine. Master's thesis, Iowa State University, Ames, IA, USA, 2015.

[168] Ganesha Upadhyaya and Hridesh Rajan. Effectively mapping linguistic abstractions for message-passing concurrency to threads on the Java virtual machine. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 840–859, New York, NY, USA, 2015. ACM.

[169] Antonio Vallecillo, Vasco T. Vasconcelos, and António Ravara. Typing the behavior of software components using session types. *Fundam. Inf.*, 73(4):583–598, September 2006.

[170] Carlos Varela and Gul Agha. Programming dynamically reconfigurable open systems with SALSA. *SIGPLAN Not.*, 36(12):20–34, December 2001.

[171] Mandana Vaziri, Frank Tip, Julian Dolby, Christian Hammer, and Jan Vitek. A type system for data-centric synchronization. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 304–328, Berlin, Heidelberg, 2010. Springer-Verlag.

[172] Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in ERLANG (2nd Ed.)*. Prentice Hall International Ltd., Hertfordshire, UK, 1996.

[173] Philip Wadler and Robert Bruce Findler. Well-typed programs can't be blamed. In *Proceedings of the 18th European Symposium on Programming Languages and Systems*, ESOP '09, pages 1–16, Berlin, Heidelberg, 2009. Springer-Verlag.

[174] Adam Welc, Suresh Jagannathan, and Antony Hosking. Safe futures for Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 439–453, New York, NY, USA, 2005. ACM.

[175] Jaeheon Yi. *Cooperability: A New Property for Multithreading*. PhD thesis, University of California at Santa Cruz, Santa Cruz, CA, USA, 2011. AAI3497948.

[176] Jaeheon Yi, Tim Disney, Stephen N. Freund, and Cormac Flanagan. Cooperative types for controlling thread interference in Java. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 232–242, New York, NY, USA, 2012. ACM.

[177] Jaeheon Yi, Caitlin Sadowski, and Cormac Flanagan. Cooperative reasoning for preemptive execution. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 147–156, New York, NY, USA, 2011. ACM.

[178] Nobuko Yoshida, Pierre-Malo Deniélou, Andi Bejleri, and Raymond Hu. Parameterised multiparty session types. In *International Conference on Foundations of Software Science and Computational Structures*, pages 128–145. Springer, 2010.