2016

PaniniJ: adding the capsule programming abstraction to Java to provide linguistic support for modular reasoning in concurrent program design

Eric Lin
Iowa State University

PaniniJ: Adding the capsule programming abstraction to Java to provide linguistic support for modular reasoning in concurrent program design

by

Eric Lin

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Science

Program of Study Committee:

Hridesh Rajan, Major Professor

Steven Kautz

Andrew Miner

Iowa State University

Ames, Iowa

2016

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

This thesis is done with a lot of help from many people around me. Thanks to my family, my sister Jenny being the nosy big sister she is, and my parents for letting me do what I want to do. Many thanks to my colleagues Mehdi and Ganesha for helping me out on this thesis and other research, and also for giving me moral support from time to time. I'm also thankful for my POS community members Dr. Steve Kautz and Dr. Andrew Miner for advice on my work.

Finally, special thanks to my adviser Dr. Hridesh Rajan for tolerating me for years and spending countless hours to help me out in all different ways.

Some of this work was done collaboratively with many individuals including Hridesh Rajan, Steven M. Kautz, Sarah Kabala, Ganesha Upadhyaya, Yuheng Long, Rex Fernando and Loránd Szakács when working on a technical report on Panini [61]. Especially chapters 1, 2 and 7 in this thesis are based on that work.

This work was supported in part by the NSF under grants CCF-14-23370, CCF-11-17937, and CCF-08-46059.

ABSTRACT

Increasing the speed of single-core processors has been facing practical challenges. Instead, multi-core architecture has been ascending for the past decade as the dominant architecture. To gain full advantage of multi-core processors, it is unavoidable for programmers to write concurrent programs. However, writing and reasoning about concurrent programs is often difficult for programmers. One reason for the difficulty stems from the hurdle of dealing with concurrency abstractions, the other reason is the difficulty in getting rid of concurrency related bugs. To address these problems, a new abstraction for concurrent programming has been proposed called *capsule*. Capsules are inspired by the long-standing ideas explored in the context of message-passing concurrency (MPC) abstractions and other similar models. Although the jury is still out on MPC abstractions as the de facto abstraction for concurrency, their wide availability and advantages combine to warrant research on the use of this model for concurrency. Capsules explore a new point in this design space to balance flexibility and analyzability in the MPC programming models. Unlike common avatars of the MPC model, a capsule is statically deployed and configured, confines its local states, permits only a single activity within itself, and communicates with other capsules in a logically synchronous manner. This thesis focuses on the realization, applicability and performance of this new abstraction. A major contribution of this work is the realization of capsules. We have implemented capsules as an extension of `javac`, the industrial strength standard Java compiler. The implementation shows that it is feasible to extend an object-oriented language with capsules. The default compilation strategy of capsules is based on threads. In this work, we also show alternative compilation strategies to improve the flexibility and adaptability of capsules. This shows that the capsule abstraction can be decoupled from concrete strategies for processing capsule messages and different underlying message processing mechanisms can be deployed without changing the user facing source code. This work also shows the strategy used to retain meta-information about capsules after compilation, so that capsule-oriented programs enjoy the property of separate compilation.

CHAPTER 1. INTRODUCTION

The growing popularity of multicore platforms makes it important to learn how best to design concurrent programs. Yet concurrent programming stubbornly remains difficult and error-prone. Message passing concurrency (MPC) abstractions such as processes [37], actors [36], active objects [45, 41, 52], ambients [51] were designed to address these difficulties. One of the great strengths of the MPC model is its ability to hide some details of concurrency and allow programmers to focus on the program logic. The model enables implicit concurrency at the boundaries of the MPC abstraction. Recently many languages, e.g. Erlang [6], Scala [32], Habanero [14], and AmbientTalk [51], have adopted the MPC model as a key abstraction for concurrency, which speaks for its importance.

1.1 The Problems and their Importance

Foundational work on the MPC model posits that processes have two basic properties: processes can dynamically create other processes, and processes can send identity of processes in messages. All of the aforementioned approaches have these properties [48]. One way to increase the success rate of modern (non-expert) concurrent programmers is to provide them with automated compile-time analyses to check and warn against concurrency hazards. These two properties contribute toward flexibility, but can complicate the design of automated algorithms, especially for analyzing concurrency safety.

For example, a desired property for concurrency is isolation, i.e. distinct processes access separate, mutable memory locations (one region per process [71]). This property simplifies reasoning about concurrent processes in a shared memory setting. However, reasoning about dynamically created, nested processes requires reasoning about nested regions, which is known to be a difficult problem [11, 76]. For the same reason, garbage collection of MPC programs is challenging [39, 19].

Another important property is sequential consistency. According to Lamport, "*[A multiprocessor*

system is sequentially consistent if] the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program. [40]" The possibility of sequential inconsistency in MPC models arises when two successive communications in the code for one source process can reach, via two separate paths, a single destination process where it leads to side-effects. This property is especially important for beginners who are just making a transition from sequential programming to concurrent programming using MPC abstractions. Precisely detecting and warning against such inconsistencies is hard in an MPC program because topology and communication patterns can change dynamically. An process can send the identity of one process to another, thus creating a new communication path between two processes. In fact, de Boer *et al.* show that process reachability in this model is undecidable [20].

In the early 1970's, inventors of the MPC abstractions may have desired the two properties in the model for conceptual unity and in order to model data and control structures using processes [36, p.235]. However, in recent adoptions of the MPC abstractions, e.g. in Java or Scala, processes aren't the only primitive available in the programming model for encoding dynamic data structures. These languages also support the notion of classes and objects. If the primary use of processes is for handling shared-memory concurrency, it is natural to wonder whether giving up these two dynamic properties of processes, and as a result gaining the ability to more effectively analyze programs, would be a good tradeoff.

1.2 Capsule: A Novel MPC Abstraction

This work contributes the design and implementation of a new abstraction called a *capsule* to examine this tradeoff in the programming language design space [8, 59, 61, 62]. Capsules were designed to support safe, implicit concurrency in a shared memory program as a direct result of decomposing a program into capsules. This work was done collaboratively with many individuals including Hridesh Rajan, Steven M. Kautz, Sarah Kabala, Ganesha Upadhyaya, Yuheng Long, Rex Fernando and Loránd Szakács, but focuses on my contributions. Consequently, chapters 1, 2 and 7 in this thesis are based on that work.

Like standard MPC abstractions [2, 36], a capsule is an encapsulated independent activity that confines its local states. Unlike standard MPC abstractions, capsules are statically declared and configured. Each capsule explicitly declares its requirements from the environment, as in ML modules [49] or nesC [29]. Unlike standard MPC models, where processes can exchange identities of other processes, a capsule may not communicate with any other capsule in the program other than those declared explicitly. These design decisions were made to simplify reasoning about concurrent capsules in a shared memory setting. A capsule can also include a *design declaration* that provides facilities for declaring instances of a capsule and for *wiring* them together so that explicitly declared requirements for each capsule instance are satisfied. By analyzing the requirements of each capsule in the program and its design declaration it is fairly easy to construct a static graph representing the topology of a program, where capsule instances are nodes and their interconnections are edges. This simplifies many language implementation considerations.

A capsule defines a set of public procedures that immediately return promises when invoked. The receiver proceeds to complete its work, running concurrently with the client of the capsule. If the client needs results, it blocks until the capsule's work is complete. This design decision to support logically synchronous communication was made to decrease the impedance mismatch between the sequential and the implicitly concurrent code written using MPC abstractions that could be hard for a sequentially trained programmer to overcome. These programmers have traditionally relied upon the sequence of operations to reason about their programs. The possibility of asynchrony and reordering of messages between processes breaks that assumption because it leads to inversion of control.

All arguments passed to a capsule procedure are owned by the capsule that is invoked, and all values returned from capsule procedures are owned by clients. This design decision to implement stricter confinement rules in the semantics itself (versus other implementation of MPC abstractions, e.g. forms of Scala [32] that use an additional type-system [33]), was also made as an experiment to determine how easy or hard could it be to live with stricter rules. Our own experiences implementing over 134,000 lines (noted below) shows that it is not draconian.

Capsules are realized in an extension of the Java programming language and implemented by extending the standard OpenJDK `javac` compiler. This compiler produces standard JVM bytecode. It is available for download and is being actively used worldwide.

We have also gained some experience with this programming model in the process of translating already vetted benchmark suite JavaGrande (JG) [70] to use capsules as the primary mechanism for concurrency. This experiment has involved over 134,000 lines of code. It has also helped to test the robustness of our compiler infrastructure and provided some informal insights into the effect of our design decisions on productivity of student programmers producing this code.

1.3 Comparison with Related Work

There is a vast body of research on concurrency and parallelism so being completely inclusive isn't feasible. However, from this work's vantage point, broadly speaking, there are three kinds of approaches: *explicit*, where programmers manually create concurrent tasks using mechanisms such as threads or fork-join pools and manage synchronization between these tasks, *implicit*, where programmers provide some hints and guidance to the compiler, often in the form of annotations or language features, and *automatic*, where the compiler is on its own to expose concurrency in a program. The capsules approach is an implicit one.

Among implicit approaches, the design of capsules was influenced by, and closely related to, the actor model [2, 36], process calculi like CSP [37], and actor-like language features, e.g. active objects [45, 41, 52] or ambients [51]. Like abstractions in these models, capsules are also an independent activity. However, capsules extend these models in several ways: capsules have confined semantics and thus avoid the need for integrating a separate type system with annotations [33, 17] or to make all data immutable [6]. Capsules are carefully designed to support static checking of sequential inconsistency [40]. Capsules provide in-order and transitive in-order delivery and processing semantics. Finally, capsules naturally supports a simple and intuitive method of resource collection that has been a challenge [39]. None of the existing work provides all of these properties.

The rest of the thesis is organized as follows. In the next chapter, we present the programming model via an example following with higher level descriptions of the model.

1.4 Contributions

The research on capsule-oriented programming [8, 59, 61, 62] is a collaborative effort. Below I present specific contributions of my research leading up to this thesis.

1.4.1 Realization

A major contribution of this work is the realization of capsules. We have implemented capsules as an extension of `javac`, the industrial strength standard Java compiler.

Some of the challenges of realization capsules were:

- Parsing capsules and translating capsules into Java byte code. To do this inside the `javac` compiler, the parser and code generation of the compiler is heavily modified so that capsules defined by programmers are translated into Java classes, while providing the properties a capsule should have.

- To realize asynchronous message passing system in an object-oriented environment. One unique feature Panini provides is to make message sending between capsules to give out the feel of a normal method call between objects. To the programmer, a message send in Panini is simply invoking a method call on a capsule instance, and getting an object of the type of the called method as a returned value. To realize this, we designed and implemented a message passing mechanism where messages can be sent by method calls, and an implicit future mechanism that can return futures for method calls.

- Essential constructs to allow the programmers to easily use capsules as they want. Extra constructs are needed for capsule related declarations such as declaring capsule topologies and instantializing capsule instances. The compiler will have to also be able to parse and compile these constructs into valid JVM bytecode.

All of the above realization have to be integrated carefully with the Java language to be consistent with the style of Java.

The following properties are also important priorities for the implementation:

Correctness The compiler takes the responsibility of managing concurrency from the programmer, and it is important that the compiler does not produce code containing errors. The implementation is carefully done to make sure it is free of errors. Many precautions to ensure correctness are shown in this thesis.

Efficiency A compiler that produces concurrent code that performs worse than sequential code defeats one of the key purpose of introducing concurrency. It is essential for the compiler to produce code as efficient as possible. One of the key to achieve this is the reduction of objects being created for operations of the program. Efficiency is kept in mind through out the implementation, and means to achieve efficiency is introduced in this thesis.

Modularization The implementation detail of capsules are hidden behind interfaces in our realization. This makes the concurrency transparent to the programmers, and also improves the maintainability of the implementation of capsules.

The implementation shows that it is feasible to extend an object-oriented language with capsules.

1.4.2 Compilation Strategies

The default compilation strategy of capsules is based on threads. In this work, we also show alternative compilation strategies that may improve the flexibility and adaptability of capsules.

The main challenge was: we want to be able to switch to different compilation strategies without having the users to recompile the capsule. As we will see later in Chapter 4, the switching between compilation strategies should be as simple as a keyword before capsule declarations. The compile should not have to change the already compiled capsules to be able to make the switch.

Another challenge is that introducing different compilation strategies to the compiler may trigger changes in different parts of the code generation component. Doing so can make the compiler overly complicated, and make it hard to maintain or to add more strategies to the compiler. Thus, the code generation strategy is designed so that most of the code generation logic can be reused between strategy.

The demonstration of these alternative compilation strategies shows that the capsule abstraction can be decoupled from concrete strategies for processing capsule messages and different underlying

message processing mechanisms can be deployed without changing the user facing source code. These implementation varaitions have been used in Upadhyaya and Rajan's work [72, 73] to select appropriate strategies for capsules where Upadhyaya and Rajan report significant performance improvement, which further illustrates their usefulness.

1.4.3 Separate Compilation

This work also shows the strategy used to retain meta-information about capsules after compilation, so that capsule-oriented programs enjoy the property of separate compilation.

Main challenges in enabling separate compilation for capsules were:

- Adding extra information to the produced bytecode without changing the bytecodes or the bytecode generation process of the compiler. We used Java annotation to solve this problem.
- Adding extra information to the produced bytecode without substantially increasing the size of the generated bytecode.
- Decoding of the encoded information must not contain errors or loss of information so that the translations and analyses relying on the encoded information can be done correctly.

1.5 Outline

Chapter 2 describes the semantic of Panini, and explains the idea of capsules in detail. Chapter 3 describes the realization of capsules for Java, and explains the model in detail. Chapter 4 describes the alternative implementations of capsules. Chapter 5 shows the implementation details that allow capsules to be compiled separately. In Chapter 6 we show the performance evaluation of the Panini programs and the usability study we have performed. Finally, we conclude the work of this thesis and future directions of this work in Chapter 7.

CHAPTER 2. THE PANINI LANGUAGE AND ITS SEMANTICS

The goal of the Panini language is to reconcile modularity and concurrency [58, 63, 46, 61]. The Panini programmer specifies a program as a collection of *capsules* and ordinary object-oriented (OO) classes. A *capsule* is an information-hiding module [54] for decomposing a system into its parts that admits implicit concurrency at its interface and a *design* is a mechanism for composing capsules together.[1,2] A capsule defines a set of public operations, hides the implementation details, and could serve as a work assignment for a developer or a team of developers. Beyond these standard responsibilities, a capsule also serves as a confined memory region [31, 18, 17] for some set of standard object instances and behaves as an independent logical process [37, 2, 36]. Inter-capsule calls look like ordinary method calls to the programmer. The OO features are standard, but there are no explicit threads or synchronization locks in Panini.

2.1 Capsules: A Decomposition Mechanism

A capsule declaration consists of the keyword 'capsule', the name of the capsule, zero or more formal parameters representing dependencies on other capsules, and zero or more signatures representing services that the capsule provides, followed by the capsule's implementation. This is similar to the syntax of 'modules' in both the Mesa [50] and nesC [29] languages. Each procedure declaration in every signature implemented by the capsule must match with exactly one capsule procedure. Panini does not have capsule inheritance but does have class inheritance. The primary mechanism for reuse of capsules is composition. The example in Figure 2.1 contains four capsule declarations.

At first glance a capsule declaration may look similar to a class declaration, thus naturally raising the question as to why a new syntactic category is essential, and why class declarations may not be

[1] The namesake notation 'capsule' in UML-RT and ROOM [67] is different and does not provide properties of MPC model such as confinement.

[2] Unrelated to CAPSULE [47] a stream processing framework.

enhanced with the additional capabilities that capsules provide, namely, confinement (as in Erlang [6]) and an activity thread (as in previous work on concurrent OO languages [16, 75, 69]).

There are three main reasons for this design decision in Panini. First, we believe based on previous experiences that objects may be too fine-grained to think of each one as a potentially independent activity [53]. Second, we wanted to specify a program as a set of related capsules with a fixed topology, in order to make it feasible to perform static analysis of the capsule graphs described in Chapter 3; this implies that capsules should not be first-class values. Third, there is a large body of OO code that is written without any regard to confinement. Changing the semantics of classes would have made reusing this vast set of libraries difficult, if not impossible. In the current design of Panini, since syntactic categories are different, sequential OO code can be reused within the boundary of a capsule without needing any modification.

A major difference between capsules and other MPC abstractions is that the capsules make their external dependence explicit. The capsule `Buyer` requires a `Seller` capsule instance, a `Trustee` capsule instance, and a number budget on line 1. Similarly, the capsule `Seller` declares that it requires an instance of `Trustee` capsule on line 8. After a capsule instance is correctly initialized, expressions inside a capsule instance may access these imported capsule instances using their names, e.g., `s`. Like ML modules, Panini capsule instances are not first-class values so capsule instances may not be passed as arguments to methods or stored in capsule states or object fields.

A capsule is considered *closed* if it does not require access to external capsule instances. In our example, `TrustModel` is a closed capsule, whereas `Buyer` is not. A closed capsule is a complete Panini program, and if it declares autonomous behavior it will exhibit that behavior when executed.

A capsule implementation consists of zero or more *state declarations*, zero or more *capsule procedures*, zero or more internal class declarations, an *initializer*, and a *design declaration*. A state declaration has a class type, a name, and optionally an initialization expression, so in that sense it is similar to a field in traditional class declarations, except that a capsule instance controls all accesses to the object graph reachable from its states, i.e., a capsule instance acts as a dominator for the graph [18]. All state declarations are private to a capsule, therefore, no visibility modifiers are necessary. An example appear on line 15 in Figure 2.1. This is one difference between capsules and extant implementations of MPC abstractions that do not, by default, provide confinement [32, 33, 17].

```
1  capsule Buyer(Seller s, Trustee t, int budget) {
2    void buy(){
3      int amount = (int)(Math.random()*budget+1);
4      Deed deed = t.createTrust(amount);
5      s.notify(deed);
6    }
7  }
8  capsule Seller(Trustee t) {
9    void notify(Deed deed){
10     int amount = deed.getAmount(); /* Blocks if necessary */
11     if (t.verifyTrust(deed, amount)) {...}
12   }
13 }
14 capsule Trustee {
15   DeedDB d = new DeedDB(..); /* A capsule state */
16   Deed createTrust(int amount){ /* A capsule procedure */
17     int id = helper(amount);
18     return new Deed(id, amount);
19   }
20   private void helper(int amount){ /* A helper procedure */
21     int id = d.nextRecord();
22     d.record(id,amount);
23     return id;
24   }
25   boolean verifyTrust(Deed deed, int amount){
26     /* Compare deed with information in DeedDB d */
27   }
28 }
29 capsule TrustModel {
30   design { /* A design declaration */
31     Buyer b; Seller s; Trustee t; /* Capsule instances */
32     b(s, t, 10000); s(t); /* Capsule interconnection */
33   }
34   void run(){ /* Autonomous behavior */
35     b.buy(); /* Capsule procedure call */
36   }
37 }
```

Figure 2.1 Program for a simplified trust model.

A capsule procedure has a return type, a name, zero or more arguments, and a body. Helper procedures can be declared by qualifying them with a modifier `private`. All capsule procedures, except helper procedures, constitute the interface of the capsule. There is one designated optional capsule procedure `run` representing that the capsule can start computation without external stimuli.

A capsule can also contain standard object-oriented class declarations. These class declarations are considered internal to the capsule and are not visible outside the capsule. A class declaration that is used across two or more capsules should be declared outside a capsule, as is usual in object-oriented languages.

A capsule initializer, if present, runs immediately after a capsule instance is created. Its main

purpose is to allocate and initialize the state declarations for that capsule instance.

A signature declaration in Panini contains one or more abstract procedure signatures. Each procedure signature has a return type, a name, and zero or more formal parameters. This syntax is similar to interfaces in Java, except that for simplicity we do not allow signature inheritance.

2.2 Design Declaration: A Composition Mechanism

Another major difference between extant MPC abstractions and capsules is that a capsule statically declares and configures its components by providing a *design declaration*. A design declaration is a declarative static specification of the topology of capsule instances that would be part of a capsule. The design declaration for the `TrustModel` capsule appears on lines 30-33 in Figure 2.1; line 31 specifies the capsule instances that are internal to this capsule, e.g., an instance of `Buyer`; and line 32 specifies how these instances are connected, e.g., the `Buyer` instance `b` is connected to the `Seller` instance `s` and `Trustee` instance `t`, the `Seller` instance `s` is connected to `Trustee` instance `t`. Arrays of capsule instances, whose length can be statically determined, can also be declared.

All capsule instances declared in a design are internal to that capsule instance. These instances can be accessed using their declared names, e.g. `s`, in procedures. They may not be accessed outside that instance. To provide other capsules access to these internal instances, a container capsule instance can use delegation.

The topology of internal capsules is fixed for a capsule and does not change dynamically. This is another key difference between capsules and MPC models that allow processes to be dynamically created, and treated as first-class values.

2.3 Semantics of Capsules

The Panini program in Figure 2.1 is an implicitly concurrent program. The execution of this program begins by allocating memory for all capsule instances, and connecting them together as specified in the design declaration on lines 30-33. Recall that capsule parameters define the other capsule instances required for a capsule to function. A capsule listed in another capsule's parameter list can be sent messages from that capsule. Design declarations allow a programmer to define the connections be-

tween individual capsule instances. These connections are established before execution of any capsule instance begins.

2.3.1 Execution model: FIFO Queue Served by One Activity

A capsule instance is a potential *execution domain* as well as a *confined heap region* for its state. An important feature of Panini is that the specification of capsules by the programmer is decoupled from the decisions about how to map each capsule's activities to OS threads. Logically, each capsule is an independent process-like entity, and the invocation of a capsule procedure triggers creation of a request object that is placed on a FIFO queue (the message ring buffer shown in Figure 2.2).

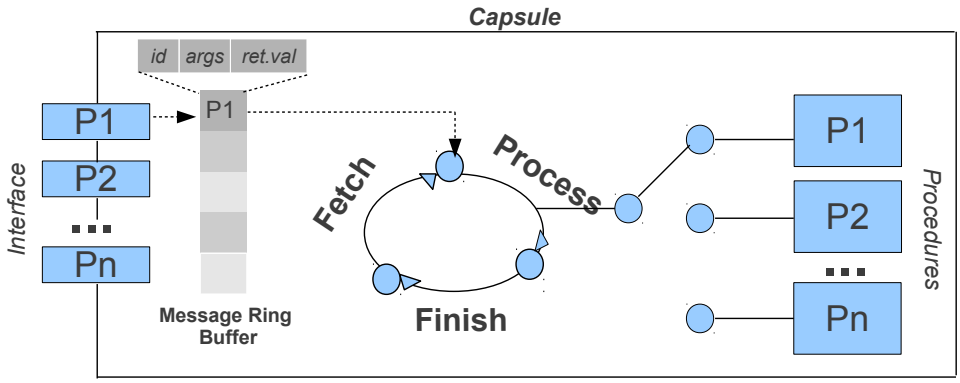

Figure 2.2 The Logical Execution Model of a Capsule.

These requests are executed in FIFO order by a single activity associated with the capsule instance that runs until terminated.

Data race freedom (due to shared data) is primarily ensured by confining the object graphs rooted at each capsule's states. Only a single activity has access to the states of a capsule. Since capsule states are by-design confined, this activity can access them without any synchronization. In inter-capsular calls, built-in and immutable types are passed and returned by value, whereas reference types are passed by linear transfer of ownership for the object graph rooted at the parameter object or result object.

By ensuring that only a single activity can access their states, capsules ensure locality of data, freedom from data races due to shared data, and memory consistency within a capsule instance.

Calling a capsule procedure may have side-effects on the state of the capsule instance, e.g., reading or writing state, and may also lead to external calls to other capsule procedures. For two consecutive

procedure calls on the same capsule instance, the side-effects of the first procedure call are always visible to the second procedure call to provide sequential consistency within a capsule.

2.3.2 Implicit Concurrency and Synchronization

From the caller's point of view, invoking a capsule procedure looks like an ordinary method call. The call returns immediately and the caller receives a *duck future* [44, 63] as a proxy for the actual return value (void return values are allowed). There is a significant body of work on using futures, e.g. [56]. The main difference between previous approaches and duck futures is that the duck future has the same type as the original result object, and it can be used wherever the original result object is expected. Therefore no refactoring of the client code is necessary. If the value is not used immediately, the caller can continue execution. An attempt by the caller to invoke a method on the returned future will cause the caller to block until the callee has finished executing the procedure, automatically providing a synchronization point. Detail of duck futures will be discussed in §3.1.3.

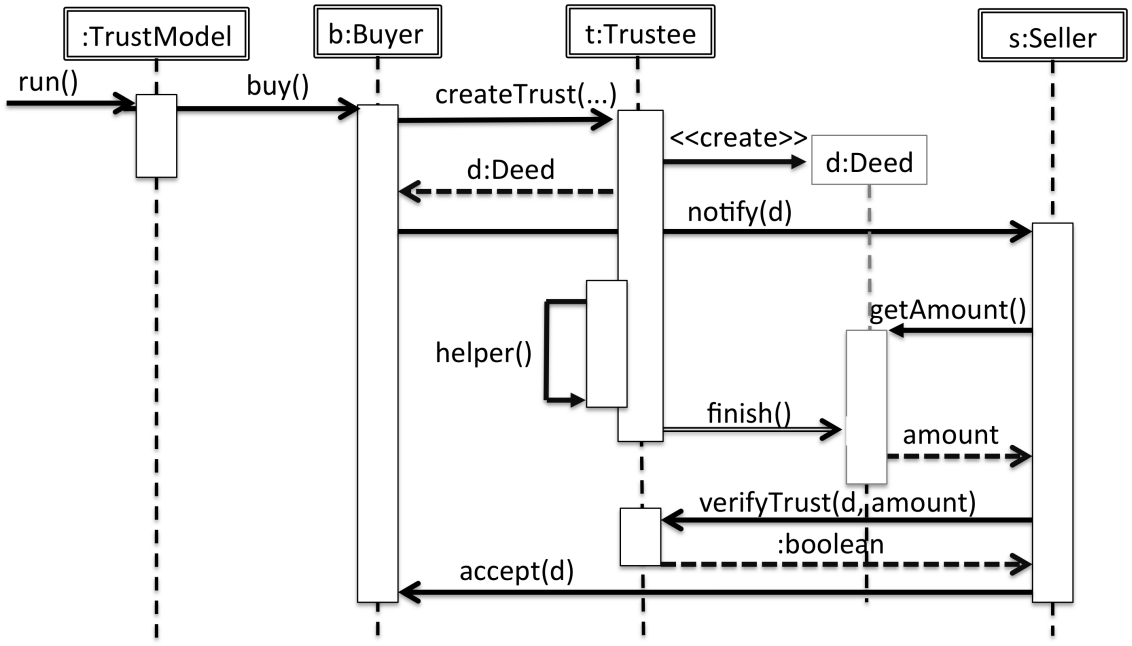

Figure 2.3 Execution sequence for the program in Figure 2.1

Figure 2.3 shows the execution sequence of the program in Figure 2.1. As soon as the initialization

and interconnection of all capsule instances is complete, this program has four independent activities (top of the figure). Next, any capsule with a `run` procedure begins executing independently, and other capsule instances wait to receive calls. For example, an instance of capsule `TrustModel` will run code on lines 34-36. This code will make an inter-capsule procedure call to instance `b` of capsule `Buyer`. In response to this call, instance `b` will run the code on lines 2-6. On line 4 instance `b` will call procedure `createTrust` of instance `t` of capsule `Trustee`. Since this is a non-void procedure call, callee `t` immediately returns a duck future `d` of type `Deed`. The callee then proceeds to create the actual `Deed` object. The instance `b` will continue execution and call procedure `notify` of instance `s` of capsule `Seller`. It also passes along the duck future `d` as a parameter for this call. Notice that the instance `b` did not block since it didn't need to use the actual result. The instance `s` uses the duck future `d` by calling a method `getAmount` on line 10. Figure 2.3 shows one interleaving where `createTrust` procedure of instance `t` has not yet completed, so execution of `s` blocks until the actual result is available. When the callee `t` completes, the duck future is finished, denoted by finish() arrow in Figure 2.3. This causes the instance `s` to resume execution, which verifies the deed and amount and accepts or rejects sellers proposal.

This example illustrates some of the key advantages of capsules for programmers. First, they don't need to create explicit threads or specify whether a given capsule needs its own thread of execution. Second, they don't need to recognize or reason about potential data races due to shared data. Third, they work within a familiar method-call style interface with a reasonable expectation of sequential consistency, and fourth, all synchronization-related details are abstracted away and are fully transparent to them.

2.4 Termination Model for Capsules

Capsules are independent activities that either declare autonomous behavior, e.g., `run` in capsule `TrustModel`, or process procedure calls in FIFO order. For the first kind of capsules, the termination condition is simple: when the `run` procedure terminates the capsule instance terminates. The second kind of capsules are similar to processes, and for this class of processes, sound determination of termination condition has been a challenge [39, 19]. The main reason for this challenge is that a process can

send a message containing its address to the set of reachable processes, thereby making itself reachable. Once a process is reachable, and if it is expected to fulfill requests from other processes, so it cannot be soundly terminated. Efficiently determining termination conditions has remained a challenge and as a result many MPC programs use a sentinel message to signal other processes that it is time to terminate.

This is one of the places where the tradeoff in programming language design that we are exploring in this work is beneficial. Due to our design decisions, detection of termination condition is significantly simplified. Since the topology of a Panini program is statically determinable, the possibility of an unforeseen connection does not arise. We thus define a '*dead*' capsule as follows.

A capsule instance is considered 'dead' if and only if (1) it is not performing any autonomous computation, e.g. in its `run` method as on line 34 in Figure 2.1, (2) it does not have any remaining work in its FIFO queue, and (3) no other capsule instances that can call procedures of this capsule are *alive*. Otherwise, that capsule instance is considered 'alive'.

Using these criteria, the termination detection for capsules works as follows. First, we determines the initial in-degree of each capsule instance from the design declarations. This information is made available to each capsule instance upon initialization. At runtime, in case of both normal and abnormal termination, a capsule instance invokes an inbuilt procedure `disconnect` on all other capsule instances that it is connected to. The semantics of processing a `disconnect` call is as follows. First, update the current capsule instance's in-degree by decrementing it. Second, if the in-degree becomes zero as a result of the previous operation, call `disconnect` on all connected capsule instances. If the current in-degree for a capsule instance is zero, that instance is asked to *shutdown*. A shutdown causes the capsule instance to process remaining procedures in its FIFO queue and then terminate. We anticipate drawing inspirations from the rich body of work on garbage collection to improve this algorithm further.

2.5 Sequential Consistency Model for Capsules

A programmer writing two statements normally expects them to be executed sequentially. (In reality, the compiler or memory subsystem may reorder the actual execution order of the statements, but the observable effects are as if executed sequentially.)

Within a capsule the effects of any two statements are seen in the order written, which is guaranteed by confinement and the execution model. If two statements involve inter-capsule procedure calls, the presence of autonomous capsules introduces the possibility that the procedures may execute out of order. If two calls are made to the same capsule instance (target), since capsules confine their states and process incoming calls in FIFO order, the effects of calls within the target will occur in order.

Likewise, if two calls reach the same target, but one is a direct call and the other is an indirect call, the semantics of capsules ensures that the effects of calls within the target will be seen in order. This property is called *transitive in-order delivery*. The design decision to support statically computable topology and FIFO ordering together enable implementation of this property, as it is feasible to determine that the same capsule instance will be reached using both a direct and an indirect path.

It is also possible that two calls in one capsule ultimately have indirect, non-commutative effects within the same target. In such cases the compiler issues a warning as described in the next section.

2.6 Summary

Capsules are exploring a new point in the design space of MPC abstractions to balance competing needs of flexibility and analyzability. Many of the ideas incorporated here have been explored previously, but their synthesis in a cohesive programming language design and robust, efficient implementation is also a novel contribution of the research on capsule-oriented programming.

CHAPTER 3. REALIZATION OF CAPSULE

To show the feasibility of realizing capsules in an industrial-strength compiler, we created the Panini compiler by extending the OpenJDK Java compiler (javac) with support for capsules. This compiler is available from http://paninij.org. We have considered both a library and an annotation-based approach instead of extending syntax, but the notation burden of these alternatives was significant enough to hamper usability, so we decided to implement a compilation-based strategy by modifying the Java compiler.

In this chapter, we show how the code generation is performed after parsing a capsule declaration written by the programmer. There were several challenges to generate code for capsules:

- Creating a Java class that represents the capsule. The representation of capsules have to provide all the procedures the capsule it represent has. However, only having to let calls made to this representative class to behave as asynchronous message passing instead of Object-oriented procedure invocations, the class has to be modified greatly to achieve the expectations of a capsule.

- Implementing an asynchronous message passing mechanism. Asynchronous message passing style is an important feature of Panini. Inter-capsule procedure calls in a Panini program triggers cause a message to be sent to the callee. This means we need to design a mechanism that can store procedure calls, along with the parameters being sent with the call. Also, the callee capsule will have to be able to automatically fetch the messages in the FIFO order, execute the corresponding procedure correctly, and perform post-processing actions.

- Implicit future mechanism. Being able to store and process messages is not enough. Since the message sending is done in an object-oriented procedure invocation style, the call often expects a returning object from the receiver. However, the asynchronous behavior means that, the returning object the caller receives can not possibly be available until after the message is processed and

the procedure is executed. Before that happens, any attempt to "use" the returned object will have to be stopped and delayed until the execution is completed. On the other hand, the receiver side needs a way to trace back to the caller and the returned object, and signal a completion of processing the message. To allow all this process to be done automatically without requiring any explicit action from the programmer, the returning mechanism has to be carefully designed so that the compiler can provide implicit futures as return objects.

- Providing a mechanism for the programmer to declare capsule instances inside a system, and the communication topology of the capsules. Capsules are designed to have modularized composition. They declare a set of capsule types they connect to, and the programmer can decide how they initialize all the capsule instances and connect them to each other to form a complete system. To let the programmer to be able to do that, we need a way for programmers to be able to express the capsules initialization and connection. The compiler then will have to be able to parse and connect the topology correctly.

- Enhance programmability of Panini. As a programming language, the ease of programming for the programmer is important for us. To make programming in Panini easier, we added several features to make some of the common code patterns easier to program. e.g. constructs to declare common topology patterns, and loops to invoke same procedures on an array of capsules and gathering results in an array.

All the different parts of the realization mentioned above depend on each other, so the code generation has to be carefully designed such that the implementation of the above mechanisms can work together. On top of that, an important priority of the implementation is that any overhead introduced should be as minimal as possible. Our design of Duck futures discussed in §3.1.3 which combines messages, futures and procedure arguments together to reduce the number of objects being created significantly is one example of optimization made for the code generation.

In §3.1, we show and describe the implementation that addresses these challenges. In the following section, §3.1.6, extra constructs for programmability such as syntactic sugars provided by the language is introduced.

3.1 Implementation of Capsules

To illustrate the how the code for capsules are generated, we will be using the following Console capsule as an example throughout this section:

```
1 capsule Console () { // Capsule declaration
2   int a = 0 ;
3   void write(String s) {  // Procedure declaration
4     System.out.println(s);
5     a = 100;
6   }
7 }
```

The Console capsule has a single procedure write, and a single state a. Based on the capsule, we can easily create an interface for the capsule as shown in Figure 3.1. This interface has the same set of procedures of the original capsule, and matches the Console type other capsules expect, therefore the other capsules can invoke the set of procedures they expect Console has.

```
1 interface Console() implements PaniniCapsule {
2     void write(String s);
3 }
```

Figure 3.1 Interface converted from the Console class.

There are two purposes of having this interface. The first purpose is to abstract away the underlying implementation of a capsule, such as the message passing mechanism. Capsules only communicate with each other by procedures, and thus the procedure signatures are all other capsules need to know about. The second purpose is to make it possible to provide more than one compilation strategies for capsules. Depending on the setup of the program, we may want a different set of implementation to optimize the performance of a program. To do that, we can implement the Console interface differently and instantiate the desired implementation when the program starts up. More details of alternative compilation strategies are discussed in Chapter 4.

The interface of Console shows that it implements the PaniniCapsule interface. The PaniniCapsule interface includes the operations we want for a capsule. These operation includes both basic thread operations such as starting up and shutting down capsule threads, and the capability of receiving incoming messages.

The PaniniCapsule interface is shown in Figure 3.2.

```
1 public interface PaniniCapsule{
2   public void panini$push(Object o);
3   public void shutdown();
4   public void exit () ;
5   public void yield(long millis ) ;
6   public void start () ;
7   public void join () throws java.lang.InterruptedException;
8 }
```

Figure 3.2 The PaniniCapsule interface.

The operations a capsule provides are as following:

- panini$push pushes a message o to the end of the message queue of the capsule. This operation is used both for other capsules to deliver messages and for message order management when needed.
- shutdown causes a termination call to be sent to the capsule. Upon receiving a termination call, a capsule disconnects with its parents. The automatic garbage collector kills a capsule if there are no living parent capsules.
- exit immediately terminates a capsule regardless if the queue is empty or not. This operation is allowed only if the client capsule has permission to modify this capsule. The Thread.checkAccess method is used inside this method and may result in a SecurityException.
- yield causes the capsule to sleep for the specified length of duration.
- start causes the capsule to begin execution and calls the run method of this capsule.
- join blocks and waits for this capsule to terminate.

3.1.1 Capsule Interface Implementation

In this section, we show the default implementation of PaniniCapsule shown previously. Recall, for each capsule we generate a concrete class as shown below.

```
1 final public abstract class PaniniCapsuleThread extends Thread implements PaniniCapsule {
2   ...
3 }
```

The PaniniCapsuleThread class extends the Java Thread class to run the capsules as threads. As discussed earlier, capsule operations include message operations that take care of receiving messages and managing message queues, and thread operations of the capsule. Since thread operations make use of the message mechanism of capsules, we show the message operation implementations first in the following section.

3.1.1.1 Message operations

Message queues of capsules are represented as an Object array as shown below.

```
1 final public abstract class PaniniCapsuleThread extends Thread implements PaniniCapsule {
2   protected volatile Object[] panini$capsule$objects;
3   protected volatile int panini$capsule$head, panini$capsule$tail, panini$capsule$size;
```

The message container panini$capsule$objects is shown on line 2, and all the index pointers of the queue to be used for different operations shown on line 3. The two **int** fields panini$capsule$head and panini$capsule$tail points to the position of the head and tail of the queue respectively, and the number of objects in the queue is stored in panini$capsule$size. All the queue related fields are marked with the **volatile** keyword, to make sure all capsules have a consistent view of the queue.

In Figure 3.2, in the public operations of the capsules, panini$push is the only message queue related operations of a capsule. panini$push(Object o) takes an message object and inserts the message at the end of the message array. The concrete implementation of panini$push is as follow:

```
1   public final synchronized void panini$push(Object o) {
2     panini$ensureSpace(1);
3     panini$capsule$size = panini$capsule$size + 1;
4     panini$capsule$objects[panini$capsule$tail++] = o;
5     if (panini$capsule$tail >= panini$capsule$objects.length)
6       panini$capsule$tail = 0;
7     if (panini$capsule$size == 1)
8       notifyAll () ;
9   }
```

On lines 1-9 panini$push is a **synchronized** method to make sure the message push operation is done atomically. The operation ensures there are enough space left in the queue using panini$ensureSpace(1) , and will expand the size of the array if needs be. After that, the size value of the array is incremented, the message is placed, and then the pointer to the tail of the queue is adjusted. The notifyAll method invocation on line 8 is to wake up capsules that may be waiting for a message to come in.

Now that we have the operation to place messages inside the queue, the next operation we need is to fetch the messages. Unlike panini$push, only the owner of the queue needs to get messages from the queue, therefore the method to fetch messages is a **protected** method:

```
1  protected final synchronized Panini$Duck get$Next$Duck() {
2    if (this.panini$capsule$size <= 0)
3      panini$blockCapsule();
4    panini$capsule$size--;
5    Panini$Duck d = (Panini$Duck) panini$capsule$objects[panini$capsule$head++];
6    if (panini$capsule$head >= panini$capsule$objects.length)
7      panini$capsule$head = 0;
8    return d;
9  }
```

Here we briefly discuss the message mechanism implemented by Panini. The returned Panini$Duck being returned by the message fetching method get$Next$Duck() is an object we created to act as message and also as a reply of the message. We will discuss this issue in greater detail in §3.1.3, but for now Panini$Duck can simply be viewed as a message object.

To fetch a message from the queue, the get$Next$Duck() operation first checks if there is a message inside the queue. If the message queue is currently empty, the thread is blocked on line 3. The call panini$blockCapsule() lets the capsule thread to wait until notifyAll () is invoked by another thread invoking panini$push and pushing a message into the queue. If the queue is not empty, the message Object at the head of the queue is returned, and the head pointer and size value of the queue is adjusted.

Next we take a look at the thread operations of the capsules.

3.1.1.2 Thread operations

Figure 3.3 shows the implementation of the rest of the operations of PaniniCapsuleThread.

The procedures shutdown on lines 7-12 and exit on lines 14-18 are the procedures that can be called explicitly by the programmer to shut down a capsule instance, or called implicitly when the program terminates. The shut down mechanism utilizes the message queue of capsules. Two constants are defined on line 4 and line 5 as a special message ids. The two procedure each pushes a message with the corresponding message id into the queue of the capsule instance, to signal the capsule instance to terminate.

The PaniniCapsuleThread class provides the essential operations of a capsule, which is to manage the thread the capsule is being run on, and to manage the message queue of each capsule. Every capsule

```
1  final public abstract class PaniniCapsuleThread extends Thread implements PaniniCapsule {
2    ...
3    public volatile int panini$ref$count;
4    public static final int PANINI$SHUTDOWN = -1;
5    public static final int PANINI$EXIT = -2;

7    public final synchronized void shutdown(){
8      panini$ref$count--;
9      if (panini$ref$count == 0)
10       panini$push(new org.paninij.runtime.types.Panini$Duck$Void(
11           PANINI$SHUTDOWN));
12   }

14   public final void exit () {
15     this.checkAccess();
16     panini$push(new org.paninij.runtime.types.Panini$Duck$Void(
17         PANINI$EXIT));
18   }

20   public void yield(long millis ) {
21     if ( millis < 0) throw new IllegalArgumentException();
22     try {
23       Thread.sleep(millis );
24     } catch (InterruptedException e) { e.printStackTrace(); }
25   }
26 }
```

Figure 3.3 The thread operations of the PaniniCapsuleThread implementation.

class extends PaniniCapsuleThread so that they can have these common operations. The next section talks about how the user defined procedures of capsules are being handled by using these common operations.

3.1.2 Using the Capsule Interface

In §3.1, the Console capsule is shown to be converted to an interface with the public procedures provided to other capsules. In this section, we show the implementation of the Console interface Console$thread, which also extends the PaniniCapsuleThread class shown in §3.1. This implementation example is the default thread-based implementation of capsules.

There are two different parts included in an implementation of a capsule: first is the procedures that are written by the programmer and the message receiving procedures generated for each of the public procedures. The second part is the code that makes the capsule runs. For Console, because it is a *passive capsule*, i.e. a capsule without user defined run methods, as oppose to their counterpart *active*

capsules, which will be introduced in §3.1.5, we have to generate the message processing mechanism for the capsule to fetch and execute the received messages.

The Console class has a single public procedure **void** write(String s). For this procedure two procedure are generated in Console$thread as shown below.

```
1  final class Console$thread extends PaniniCapsuleThread implements Console {
2   public static final int panini$methodConst$write$String =0;
3     int a = 0;

5     public final void write$Original(String s) {
6         System.out.println(s);
7         a = 100;
8     }

10    public final void write(String s) {
11          Panini$Duck$void$Console$thread panini$duck$future = null;
12          try {
13              panini$duck$future = new Panini$Duck$void$Console$thread(
                    panini$methodConst$write$String, s);
14          } catch (Exception e) { throw new DuckException(e); }
15          panini$push(panini$duck$future);
16      }
17       ...
18  }
```

The write method on lines 10-16 is rewritten such that it now it now constructs a message on line 13, using a constant panini$methodConst$write$String and the parameter s passed in by the caller, and then pushes the call message panini$duck$future into the queue. The message identifier constant (panini$methodConst$write$String) is an unique identifier assigned to every procedure in a capsule, and is used to identify the procedure being called. The passed in value s is stored inside panini$duck$future for execution of the procedure later on. If the function requires a return value, panini$duck$future is returned to the caller, as shown in Figure 3.4.

The write$Original procedure on lines 5-8 is a copy of the original programmer defined procedure, and will be invoked later by the message processing mechanism.

To process the messages that are being sent to a capsule instance, Console$thread provides a run method that is implicitly executed when the capsule instance starts. The run method is as shown:

```
1    ...
2   public final void run() {
3           panini$wire$sys();
4           panini$capsule$init();
5           boolean panini$terminate = false;
6           while (!panini$terminate) {
7               Panini$Duck panini$duck$future = get$Next$Duck();
8               boolean panini$check$when = false;
```

```
9                switch (panini$duck$future.panini$message$id()) {
10               case panini$methodConst$write$String:
11                   String var0 = (String)((Panini$Duck$void$Console$thread)panini$duck$future).
                         String$s$write$String
12                   this.write$Original(var0);
13                   panini$duck$future.panini$finish(null);
14                   break;

16               case -1:
17                   if (this.panini$capsule$size > 0) {
18                       panini$push(panini$duck$future);
19                       panini$check$when = false;
20                       break;
21                   }

23               case -2:
24                   panini$terminate = true;
25                   panini$check$when = false;
26                   break;
27               }
28           }
29       }
30   }
```

```
1    public final Bool isTrue(Bool b) {
2            Panini$Duck$Bool$Compare$thread panini$duck$future = null;
3            try {
4                panini$duck$future = new Panini$Duck$Bool$Compare$thread(
                     panini$methodConst$isTrue$Bool, b);
5            } catch (Exception e) {
6                throw new DuckException(e);
7            }
8            panini$push(panini$duck$future);
9            return panini$duck$future;
10   }
```

Figure 3.4 Generated code for the isTrue procedure that returns a Bool.

Both panini$wire$sys and panini$capsule$init are invoked first in run to initialize and startup capsules declared in design blocks if there is any. Examples for design declarations will be shown in §3.1.6.1 and §3.1.4.

The boolean value panini$terminate on line 5 is a termination flag that is triggered when a termination message is received. The loop on lines 6-28 is repeatedly executed until a termination message is received.

The method call get$next$duck on line 7 attempts to fetch the next message (duck future) inside the queue. Once a message is fetched, the method takes action depending on the message identifier

(panini$message$id) stored inside the duck future. A **boolean** flag (panini$check$when) is generated to handle the **when** feature. The usage of **when** construct and the complete code generation strategy is described in §3.1.6.2.

On lines 10-14, the logic for handling different type of messages is shown. Messages can be either a call to procedures of the capsule, or special messages for the capsule, e.g. a shutdown message.

On line 11, the message processing logic extracts the parameters of the procedure from the duck future, and invokes the original code written by the programmer. The method write$Original contains the body of the original Console.write() written by the programmer as shown on lines 5-8. Finally, panini$finish is invoked on the duck future, which notifies any threads that are waiting for the result of the future. If the invoked procedure has a non-void return value, the returned value of the procedure is passed to panini$finish, which will insert the result inside the duck future. Code generated on line 16 and line 23 handles two special cases for message processing. The "-1" case handles shutdown messages. When a shutdown message is received and the message queue is not empty at this moment, the shutdown message is pushed to the end of the queue and panini$check$when is cleared to prevent the shutdown message to trigger the **when** condition check. The shutdown message will circulate inside the queue, until the shutdown message is processed and the message queue is empty. When such conditions are met such that the message queue is empty and a shutdown is signaled, the switch case fallthrough to the "-2" case, where the termination flag (panini$terminate) is set to stop all message processing. The when flag (panini$check$when) is cleared to prevent the shutdown message from triggering the **when** condition check.

3.1.3 Duck Future

Duck futures serve both the purpose of being a message and a future, and are a key component of Panini. As a message, a duck future contains the identifier of the procedure being called and the parameters passed in. As a future, a duck future imitates the original return type, method invocations on the duck future redirect the invocations to the actual returned result, letting programmers to avoid any future management.

In this section, the implementation of duck futures are discussed in detail. For this section, we use a simple Bool class to demonstrate the implementation.

```
class Bool {
  private boolean v;
  public boolean value() { return v; }
  public Bool (boolean v) {this.v = v;}
}
```

Let us suppose that the Bool class is used as a return type of a procedure of a capsule Compare as shown below:

```
capsule Compare{
  Bool isEqual(Bool a, Bool b){
    return new Bool(a.value() == b.value());
  }
}
```

For the duck future mechanism to work, for each unique return type of public procedure a corresponding duck future class must be generated. In this example, a Bool type is returned by isEqual, and a duck future class is generated for Bool.

The duck future generated for Bool is shown below:

```
final class Panini$Duck$Bool$Compare$thread extends Bool implements Panini$Duck<Bool> {
    private Bool panini$wrapped = null;
    private final int panini$message$id;
    private boolean panini$redeemed = false;

  public Panini$Duck$Bool$Compare$thread(int panini$message$id, Bool a, Bool b)
        super(false);
        this.panini$message$id = panini$message$id;
        this.Bool$a$isEqual$Bool$Bool = a;
        this.Bool$b$isEqual$Bool$Bool = b;
    }

    public Bool Bool$a$isEqual$Bool$Bool;
    public Bool Bool$b$isEqual$Bool$Bool;

    public final void panini$finish(Bool t) {
        synchronized (this) {
            panini$wrapped = t;
            panini$redeemed = true;
             notifyAll();
        }
        this.Bool$a$isEqual$Bool$Bool = null;
        this.Bool$b$isEqual$Bool$Bool = null;
    }

    public final Bool panini$get() {
        while (panini$redeemed == false) try {
            synchronized (this) {
                while (panini$redeemed == false) wait();
            }
        } catch (InterruptedException e) { }
        return panini$wrapped;
    }
```

```
35    public final boolean value() {
36          if (panini$redeemed == false) this.panini$get();
37          return panini$wrapped.value();
38      }
39  }
```

The Panini$Duck$Bool$Compare$thread class extends Bool and match the type the caller expects. The Bool panini$wrapped on line 2 is used to store the actual result once the duck future is finished. The field panini$message$id contains the identifier that indicates the procedure being called, and panini$redeemed is a flag that indicates if the duck future is finished or not. The constructor of the duck future is on lines 6-11. The duck future is initialized with the message identifier which indicates the procedure it is calling, and the parameters passed in from the caller. In this example, two fields Bool$a$isEqual$Bool$Bool and Bool$b$isEqual$Bool$Bool in the duck future are used to store the parameters Bool a, Bool b. The names are constructed by concatenating the type of the parameters (Bool), the name of the parameters (a), the name of the procedure (isEqual), and parameter types of the procedure (Bool, Bool, separated by $ signs). The naming convention is such that so the field name is guaranteed to be unique. The field name is also used by client capsules to find the parameter.

After a message is processed, the result is passed into panini$finish, and then result is assigned to panini$wrapped. After switching the panini$redeemed flag to **true**, other threads that is waiting for the duck future to be finished may access the future. Unneeded references are then assigned to **null**, to allow Java garbage collectors to free up the memory.

To understand the rest of the duck future and how it can be claimed implicitly, we illustrate by an use case. Suppose a capsule made a call to isEqual and received a duck future as a result of the call. Because the duck future is transparent to the capsule, it is treated as a normal Bool object, and the user invokes value on the duck future in attempt to 'use' it. The call reaches the method in the duck future on line 35. Inside value, if the finished flag panini$redeemed is **true**, the method delegates the call to the actual returned Bool object panini$wrapped and return it. If the duck future hasn't been finished, the method then calls panini$get on line 26. The get method (panini$get) invokes wait and wait until the duck future is finished, i.e. panini$finish is called by the message processing logic in the run method. The call is then delegated to the actual result object and returned. This completes the implicit claim of a duck future.

3.1.3.1 Limitations of Duck Futures

For duck futures to be legally accepted by the caller as the expected return type, duck futures make use of the Java subtyping mechanism; i.e., if the caller is expecting a class C, the duck future has to extend C. However this is not always legal in Java, e.g. final classes can not be subtyped. This limitation also applies to primitive types and arrays. Duck future also overrides the methods of its supertype to provide a mechanism to wait until the future is finished to delegate calls to the actual object. Methods that are declared as final, however, can not be overridden. Because of the need of subtyping and overriding methods, a class to represent duck futures can not be generated for these classes.

For procedures with return types that cannot be subtyped, we instead block the calls until the result is finished effectively making these calls synchronous. Notice that this is not the same as an object-oreinted call, the caller capsule is not allowed to directly call the procedure, which may let the caller to access states of another capsule. Instead, a duck future is constructed and pushed into the queue for the client capsule to be able to process the procedure call. To illustrate the code generation strategy for duck futures for final classes, consider the example in Figure 3.5.

```
1 final  class C{}
2 capsule FinalClassExample{
3         C proc(){return new C();}
4 }
```

Figure 3.5 Example of a **capsule** with a procedure returning a final class.

The procedure proc has a return type C which is a final class, and a duck future has to be generated for C. The generated duck future Panini$Duck$C$FinalClassExample$thread is shown in Figure 3.6.

The Panini$Duck$C$FinalClassExample$thread class only contains a field panini$wrapped for the actual result object. However instead of implementing Panini$Duck, it now subclasses Panini$Duck$Final , which is shown in Figure 3.7.

The Panini$Duck$Final class contains the finishing and block mechanism we have seen in standard duck futures along with an extra method finalValue on line 28. finalValue simply blocks and wait until the duck future is finished and returns the actual result. Now lets look back to the generated

```
final class Panini$Duck$C$FinalClassExample$thread extends Panini$Duck$Final<C> {
    private C panini$wrapped = null;
    private final int panini$message$id;
    private boolean panini$redeemed = false;

    public Panini$Duck$C$FinalClassExample$thread(int panini$message$id) {
        super();
        this.panini$message$id = panini$message$id;
    }
}
```

Figure 3.6 Duck future generated for the final class C in Figure 3.5.

```
public class Panini$Duck$Final<T> implements Panini$Duck<T>{
  private T panini$wrapped = null;
    private final int panini$message$id = 0;
    private boolean panini$redeemed = false;

    public final void panini$finish(T t) {
        synchronized (this) {
            panini$wrapped = t;
            panini$redeemed = true;
             notifyAll();
        }
    }

    public final int panini$message$id() {
        return this.panini$message$id;
    }

    public final T panini$get() {
        while (panini$redeemed == false) try {
            synchronized (this) {
                while (panini$redeemed == false) wait();
            }
        } catch (InterruptedException e) {
        }
        return panini$wrapped;
    }

    public T finalValue() {
      if (panini$redeemed == false) panini$get();
        return panini$wrapped;
    }
}
```

Figure 3.7 The Panini$Duck$Final class declaration.

FinalClassExample capsule in Figure 3.8.

The generated proc procedure is very similar to the code generation seen before, except that instead

```
1  capsule FinalClassExample$thread() extends PaniniCapsuleThread implements
       FinalClassExample {
2      public final C proc() {
3          Panini$Duck$C$FinalClassExample$thread panini$duck$future = null;
4          try {
5              panini$duck$future = new Panini$Duck$C$FinalClassExample$thread(
                   panini$methodConst$proc);
6          } catch (Exception e) {
7              throw new DuckException(e);
8          }
9          panini$push(panini$duck$future);
10         return (C)panini$duck$future.finalValue();
11     }
12     /* ... */
13 }
```

Figure 3.8 Translated FinalClassExample capsule from Figure 3.5.

of returning the duck future, panini$duck$future.finalValue() is returned. As seen before, finalValue waits until the duck future is finished and returns the actual result. To conclude, a procedure invocation to proc blocks and returns the actual result back to the caller instead of a duck future, and the procedure execution is still performed on the client capsule.

Arrays and primitive types are handled similarly. However this is only a temporary work-around solution. The greatest disadvantage of this solution is that the blocking behavior lets procedure calls become essentially synchronous, which decreases available implicit concurrency. A different solution that can solve this problem without giving up concurrency is one of our future goals for Panini.

3.1.4 Design

The Panini programmer uses the **design** construct to declare capsule instances, and the communication topology between these instances. In Figure 3.9, an example design declaration is shown to illustration.

The **design** declared on line 2 contains four statements. The capsule declaration on line 3 declares a Capsule instance cap. The capsule array declaration on line 4 declares a Capsule array capArray with 5 capsules in it. The wiring statement on line 5 shows an example connection between the capsule instance cap, and the first capsule in capArray. A default value of 0 is assigned to the capsule's state i as its initial value. Non-primitive types may not be used in capsule wiring declarations. This is to

```
1 capsule DesignExample {
2   design{
3     Capsule cap; //Capsule declaration
4     Capsule capArray[5]; //Capsule array declaration
5     cap(capArray[0], 0); Wiring declaration
6     wireall(capArray, cap, 10); // Wireall  Topology operator
7   }
8 }
9 capsule Capsule(Capsule caps, int i){}
```

Figure 3.9 A **capsule** with a **design** block to illustrate the design construct.

prevent capsules from sharing reference of objects. On line 6 the **wireall** operator is shown, one of the several topology operators provided by Panini. These topology operators allow programmers to wire a number of capsules together in certain patterns commonly used. The **wireall** operator connects all the capsules in a capsule array to the same set of capsules and initialization values. In the example, every capsule in capArray is connected with cap and receives an initial value of 10 for their state variable i. The topology operators provided by Panini are :

- The **wireall**(capsuleArray, arg0, arg1, ...) declaration connects every capsule in capsuleArray with the following list of arguments arg0, arg1, ...
- The **ring**(capArray, arg0, arg1, ...) declaration connects a capsule array in a ring fashion such that the ith capsule is connected to the i+1th capsule, and the last capsule in the array is connected to the first capsule.
- The **assoc**(capArray, i, capArray2, l, arg0, arg1, ...) declaration connects the capsules of capArray1 with the capsules in the same index of capArray2 starting from index i for l capsules. The declaration **assoc**(capArray1, 2, capArray2, 2, 0) is equivalent to saying capArray[2](capArray[2], 0) and capArray[3](capArray[3], 0).

The code generated from a **design** block can be separated by its functionality to initializing the capsule instances, connecting the capsule instances and assigning initial values to capsule instances, starting up the threads, and stopping the threads. The converted DesignExample capsule is shown in Figure 3.10. Again, two calls on line 27 and line 28 are made inside run before the message processing logic. However, because the DesignExample capsule has a **design** declaration, the two methods panini$wire$sys

```
1  capsule DesignExample$thread() extends PaniniCapsuleThread implements DesignExample {
2      private final Capsule cap;
3      private final Capsule capArray;

5      void panini$wire$sys(){
6          cap = new Capsule$thread();
7          capArray = new Capsule$thread[5];
8          cap.caps = capArray[0];
9          cap.i = 0;
10         for(int i=0;i<5;i++){
11           capArray[i].caps = cap;
12           capArray[i].i = 10;
13           ((Capsule$thread)cap).panini$ref$count += 1;
14         }
15         ((Capsule$thread)cap).panini$ref$count += 5;
16     }

18     void panini$capsule$init(){
19         cap.start();
20         for(int i=0;i<5;i++){
21           capArray[i].start();
22         }
23     }

25     public final void run() {
26         try {
27             panini$wire$sys();
28             panini$capsule$init();
29             boolean panini$terminate = false;
30             while (!panini$terminate) {
31               /* ... */
32             }
33         } finally {
34             ((PaniniCapsule)cap).shutdown();
35             for(int i=0;i<5;i++){
36               ((PaniniCapsule)capArray[i]).shutdown();
37           }
38         }
39     }
40 }
```

Figure 3.10 The translated DesignExample capsule from Figure 3.9

and panini$capsule$init are now inserted at lines 5-16 and lines 18-23. The panini$wire$sys method initializes the capsules, and assigns the capsule instances the connected capsule instances and initiate state values. It also sets the names of the threads with Thread.setname, and the reference count of capsules used for garbage collection are initialized. The capsule declaration Capsule cap in Figure 3.9 generates the code on line 6 to initialize a Capsule instance. The capsule array declaration CapsulecapArray[5] generates the code on line 7. The wiring statement cap(capArray[0], 0) generates the code on lines 8-9.

The **wireall** statement **wireall**(capArray, cap, 10) generates the unrolled code on lines 10-14. Other topology operators generate code in a similar fashion. The panini$capsule$init method on line 18 starts up all the capsules that have been declared. Code to shutdown the capsules are added inside the **finally** clause on line 33. This makes sure if the current capsule terminates or if any exceptions were thrown the started capsule instances will be terminated.

3.1.5 Active Capsule

Active capsules are capsules that contains an user-defined run method. They essentially define the operations of the program and triggers the program to start running. Similar to main methods in Java, active capsules act as entry points of a program. Every active capsule in a Panini program executes their run method implicitly when a capsule is initialized and started. Active capsules devote their thread to execute the code declared in run Therefore, active capsules can not accept procedure calls.

An example of an active capsule and the code generated from it is shown in Figure 3.11.

The original body of the run procedure in HelloWorld capsule is inside the run method of the HelloWorld$thread class on line 38. Two method calls to panini$wire$sys and panini$capsule$init are inserted before the original run body. As discussed before in §3.1.4, these two methods declared on line 18 and line 29 initialize the capsules and start up the capsule threads.

3.1.6 Other Features of Panini

3.1.6.1 Initializers

Initializers are blocks of code that can be declared inside a capsule, which will be executed first after a capsule has started. As oppose to constructors, the execution of initializers are done inside the local thread and not in the thread that initialized the capsule. Since executing the initializers of different capsules is done concurrently, using initializers may improve performance of programs. An example of a capsule with an initializer is shown in Figure 3.12. In the example, an initializer is used to initialize the **int** array in the capsule.

The thread-based code generation of the Init capsule is shown in Figure 3.13. The body of the initializer has been copied into the panini$capsule$init method, which is invoked by run. If there are

```
1  capsule HelloWorld() {
2    design {
3      Console c;
4      Greeter g;
5      g(c);
6    }
7    void run()  {
8      g.greet();
9    }
10 }

12 capsule Greeter(Console s){ /* ... */ }

14 class HelloWorld$thread() extends PaniniCapsuleThread implements HelloWorld {
15     private final Console c;
16     private final Greeter g;

18     void panini$wire$sys() {
19         c = new Console$thread();
20         ((Console$thread)c).setName("c:Console");
21         g = new Greeter$thread();
22         ((Greeter$thread)g).setName("g:Greeter");
23         ((Greeter$thread)g).s = c;
24         ((Greeter$thread)g).panini$ref$count += 1;
25         this.panini$ref$count += 0;
26         ((Console$thread)c).panini$ref$count += 2;
27     }

29     void panini$capsule$init()  {
30         g.start();
31         c.start();
32     }

34     public final void run()  {
35         try  {
36             panini$wire$sys();
37             panini$capsule$init();
38             g.greet();
39         }  finally  {
40             ((PaniniCapsule)c).shutdown();
41             ((PaniniCapsule)g).shutdown();
42         }
43     }
44 }
```

Figure 3.11 Active capsule HelloWorld and its thread converted class HelloWorld$thread.

multiple initializers, the code will be copied into panini$capsule$init in the order in which they are declared. There is no message processing for this capsule because there is no public procedures.

```
1  capsule Init {
2          int [] elems;
3          => {
4            elems = new int[50];
5            for(int i = 0; i < 50; i++){ elems[i] = i; }
6          }
7  }
```

Figure 3.12 A example capsule with a initializer declaration.

```
1  capsule Init$thread() extends PaniniCapsuleThread implements InitBug {
2      int [] elem;
3      protected void panini$capsule$init() {
4              elems = new int[50];
5              for(int i = 0; i < 50; i++){ elems[i] = i; }
6      }

8      public final void run() {
9          try {
10             panini$wire$sys();
11             panini$capsule$init();
12         } finally {
13         }
14     }
15 }
```

Figure 3.13 Translated Init capsule from Figure 3.12

3.1.6.2 When Clause

The **when** clause is a syntactic sugar of the Panini language to help programmers. A **when** clause is equivalent to adding code to the end of every procedure to check for a certain condition and execute a piece of code if the condition is met. This is useful for programs that periodically checks for a condition to meet to trigger a particular event for the capsule. To illustrate the desugaring of the **when** construct, consider an alternative version of the Console capsule.

In Figure 3.14, the Console capsule now contains a **when** declaration. Whenever a procedure has been processed, executed and the future has been finished, the capsule now checks whether the state a a is equal to a hundred, and executes the code on lines 8-10 if so. i.e. the capsule now prints out a message every time write has been invoked a hundred times. The generated code for the Console$thread class is shown in Figure 3.1.6.2.

The original **when** in Console has been transformed into a private method (panini$when$0). The

```
1  capsule Console () {
2          int a = 0;
3          void write(String s) {
4                  System.out.println(s);
5                  a += 1;
6          }

8          when(a==100) {
9                  System.out.println("100 writes reached!");
10                 a = 0;
11         }
12 }
```

Figure 3.14 An alternative Console capsule with a with declaration.

postfix, here $0, is given using the order in which the **when** clause appears in the capsule. If there are multiple **when** clauses declared, the second clause will be named as panini$when$1 and so on. The panini$check$when value is initialized as **true** on line 18, and a new if clause is inserted on line 38. After a procedure message is being processed by the switch, each of the conditions of the **when** declarations are checked, and executed if the condition meets.

3.1.6.3 Foreach Statement

The **foreach** statement is another syntactic sugar in Panini. A foreach loop invokes the same procedure on every capsule in a capsule array and stores the result inside an array of the same type as the return type of that procedure.

To illustrate the desugaring of **foreach** constructs, consider the capsule in Figure 3.16. The Reader capsule has a process method that gathers the result from the capsule array buckets by calling getCount on each of the capsule in the array. This code is equivalent to inserting elements in an array in a loop as shown in Figure 3.17.

The generated code for the Reader capsule is shown in Figure 3.18. Code unrelated to the **foreach** mechanism is omitted from the figure.

The **foreach** expression is replaced on line 13 to invoke the helper method (foreachHelper$0). For every foreach expression in a capsule, a helper method containing the desugared code of the foreach declaration is generated. In the helper method, here foreachHelper$0, a getCount call is made to every capsule in buckets, returned results are gathered in an array and returned to the original caller.

```
1  final capsule Console$thread() extends PaniniCapsuleThread implements Console {
2      public static final int panini$methodConst$write$String = 0;
3      int a = 0;
4      private final void panini$when$0() {
5          System.out.println("100 writes reached!");
6          a = 0;
7      }
9      ...
11     public final void run() {
12         try {
13             panini$wire$sys();
14             panini$capsule$init();
15             boolean panini$terminate = false;
16             while (!panini$terminate) {
17                 Panini$Duck panini$duck$future = get$Next$Duck();
18                 boolean panini$check$when = true;
19                 switch (panini$duck$future.panini$message$id()) {
20                 case panini$methodConst$write$String:
21                     String var0 = (String)((Panini$Duck$void$Console$thread)panini$duck$future).
                           String$s$write$String;
22                     this.write$Original(var0);
23                     panini$duck$future.panini$finish(null);
24                     break;
26                 case -1:
27                     if (this.panini$capsule$size > 0) {
28                         panini$push(panini$duck$future);
29                         panini$check$when = false;
30                         break;
31                     }
33                 case -2:
34                     panini$terminate = true;
35                     panini$check$when = false;
36                     break;
37                 }
38                 if (panini$check$when) {
39                     if (a == 100) panini$when$0();
40                 }
41                 panini$check$when = true;
42             }
43         }
44     }
45 }
```

Figure 3.15 Code generated from the alternative Console capsule containing a **when** declaration.

```
1  capsule Reader(Bucket[] buckets) {
2    void process() {
3      Long[] results = foreach(Bucket b : buckets) b.getCount();
4      for(int i = 0; i < results.length; i++)
5        p.print("" + i + ":" + results[i].value());
6      System.out.println("READER: work complete.");
7    }
8  }
```

Figure 3.16 A Reader capsule declaration with a foreach expression on line 3.

```
1  capsule Reader(Bucket[] buckets) {
2    void process() {
3      Long[] results = new Long[buckets.size()];
4      for(int i = 0; i<buckets.size(); i++){
5        results[i] = b[i].getCount();
6      }
7      p.print("" + i + ":" + results[i].value());
8      System.out.println("READER: work complete.");
9    }
10 }
```

Figure 3.17 A Reader capsule declaration without using a foreach expression.

```
1  final class Reader$thread extends PaniniCapsuleThread implements Reader {
2      /*synthetic*/ private static Long[] foreachHelper$0(Bucket[] capsules$) {
3          int len$ = capsules$.length;
4          Bucket[] carr$ = capsules$;
5          Long[] result$ = new Long[len$];
6          for (int index$ = 0; index$ < len$; ++index$) {
7              Bucket placeHolder$ = carr$[index$];
8              result$[index$] = placeHolder$.getCount();
9          }
10         return result$;
11     }
12     public void process$original(String fileName) {
13         Long[] results = Reader$thread.foreachHelper$0(buckets);
14         for (int i = 0; i < results.length; i++) p.print("" + i + ":" + results[i].value());
15         System.out.println("READER: work complete.");
16     }

18     /* ... */
19 }
```

Figure 3.18 Reader$thread translated from the Reader capsule of Figure 3.16.

3.2 Summary

In this chapter, we described an implementation for a thread-based realization of capsules. For this implementation, every capsule runs in a dedicated thread. Capsules communicate with each other only using the interface of other capsules, and do not modify states of other capsules directly. Since modification of states of a capsule is always done by the local thread, capsules provide encapsulation of their own states. Communicating only using the interface allows us to hide the implementation details of capsules, which can be implemented differently as will be discussed in Chapter 4.

For capsules to communicate between each other, we introduced a construct called "duck futures". Duck futures act as the message sent to other capsules and at the same time act as implicit futures. This avoid creating separate objects for both a message and a future, which reduces the number of objects being created for message passing. A mechanism for capsules to push message to queues of other capsules is shown in the chapter, and the logic to process these messages is discussed.

This chapter also shows mechanisms for implicitly finishing and claiming duck futures. The implicit future claim provides implicit concurrency so that caller capsules can execute other operations between sending out messages, i.e. making procedure calls, and using the returned value.The caller is oblivious to the future mechanism and can benefit from the performance gain by the increase of parallelism.

We also discussed the challenges of duck futures. Duck future relies on subtyping the type of the object it represents. This is not legal for final classes. Classes with final methods also conflict with the claiming mechanism which is done by overriding methods. We showed the work around which has a the drawback of rendering procedure calls synchronous and reducing concurrency. This is a challenge we aim to solve in the future.

To compose capsules together to construct a program, the **design** construct is introduced. We also showed the code generation strategy for designs.The **design** block contains capsule declaration and the wiring of capsules, i.e. the communication topology of capsules. This centralizes the construction of capsules and the system's topology in a single declaration, allowing programmers to easily understand the system without looking through implementation details for code that create capsules. For wiring capsules Panini provides operations for several common topologies.

Finally, additional features provided by Panini are introduced. Initializers provide a way to declare

initialization code that is executed in the capsule thread and not the thread that initialized the capsules, gaining concurrency. The **when** and **foreach** constructs are syntactic sugars provided by Panini to further reduce lines of code needed for some commonly seen operations. We showed the implementation and code generation strategies for these constructs.

CHAPTER 4. ALTERNATIVE COMPILATION STRATEGIES

4.1 Motivation

In Chapter 3, we described our implementation of capsules that is based on Java threads. The thread-based compilation strategy is the default compilation strategy of capsules, but is not the only possible strategy for implementing capsules. The thread-based compilation strategy creates a dedicated thread for every single capsule instance in the system, and in some systems, this may not be desirable. When using the thread-based compilation strategy, programmers may end up declaring too many capsules in an attempt to modularize their program, and as a result, an excessive amount of threads would be created that would adversely affect the performance of the system.

For programs where an all-thread strategy is not ideal, Panini provides alternative compilation strategies to programmers. Selecting a compilation strategy for a capsule instance can be done by the programmer simply by using different modifiers when declaring the capsule instance. In the Panini implementation discussed in this thesis, the programmer needs to determine the optimal configuration themselves; however, Upadhyaya and Rajan have done some followup work to enable the compiler to automatically suggest appropriate compilation strategies for capsule instances [72, 73].

In this chapter, we describe three alternative compilation strategies of capsules provided by the Panini compiler as of this writing, the task-based and two sequential compilation strategies.

4.2 Implementing Alternative Compilation Strategies

In Chapter 3, we have shown that a capsule interface with procedure signatures is created for every declared capsule. Capsules communicate with other capsules through this interface by calling the visible procedures, while the underlying implementation is hidden. By implementing this interface differently, alternative realization of the same capsule can be provided.

4.2.1 Task-Based Compilation Strategy

For the task-based compilation strategy introduced in this section, capsule instances are separated into *pools*. A single thread is assigned to each pool, which is responsible for processing the messages of all the capsule instances in the pool. The capsule instances in the same pool do not share the same message queue and receive messages independently. The pool thread processes one message in the queue of a capsule instance before moving on to the next capsule instance. The task pool pattern is especially useful when tasks being sent require a smaller amount of computation time, happens frequently, and are independent to each other.

4.2.1.1 Overview

For this section, we use the same Console capsule used in Chapter 3 as an example to demonstrate the translation strategy of task-based capsules. In the last chapter, we made a PaniniCapsuleThread class that include the thread operation and message managing operation of thread-based capsules, and a Console$thread class that implements Console to receive messages and run the capsule. Similarly, we implemented a PaniniCapsuleTask class for task-based capsules and a Console$task extending PaniniCapsuleTask that implements Console.

4.2.1.2 Implementation

The task-based implementation of the Console capsule is shown below:

```
1  final class Console$task extends PaniniCapsuleTask implements Console {
2      ...
3      public final boolean run() {
4          Panini$Duck panini$duck$future = get$Next$Duck();
5          switch (panini$duck$future.panini$message$id()) {
6          case panini$methodConst$write$String:
7              String var0 = (String)((Panini$Duck$void$Console$thread)panini$duck$future).
                   String$s$write$String
8              this.write$Original(var0);
9              panini$duck$future.panini$finish(null);
10             break;

12         case -1:
13             if (this.panini$capsule$size > 0) {
14                 panini$push(panini$duck$future);
15                 break;
16             }

18         case -2:
```

```
19              return true;
20          }
21          return false;
22      }
23  }
```

The class generated for the task-based compilation strategy of Console (Console$task) is similar to the class generated for the thread-based compilation strategy (Console$thread shown in §3.1.2) and shares most of the translation logic. The major difference is in the run method being generated. Although most of the code generation is similar to that of the thread based-version, the message processing inside run only runs once and returns **false** on line 21 immediately after the message is processed, or return **true** if the capsule instance is terminated. Unlike the thread version, where run() implements Thread.run() and is executed when the thread starts, the run for the task-based compilation strategy simply processes one message once it is invoked.

Instead of extending PaniniCapsuleThread as the thread-based compilation strategy do, the task-based strategy extends PaniniCapsuleTask. The PaniniCapsuleTask class is shown below.

```
1  final public abstract class PaniniCapsuleTask implements PaniniCapsule {
2    /* ... */

4    PaniniCapsuleTask panini$capsule$next;
5    PaniniTaskPool panini$containingPool = null;

7    public final void start() {
8      panini$containingPool = PaniniTaskPool.add(this);
9    }

11   /* ... */
12 }
```

The message queue logic of PaniniCapsuleTask is the same as the thread-based version and is omitted from the figure.

A key difference is that a PaniniTaskPool object panini$containingPool field is declared on line 5. The panini$containingPool field is the thread pool the capsule instance is assigned to. The assignment happens when the capsule instance starts, as seen on line 8. Recall that because the thread-based version extends the Java Thread class, the capsule thread is started by calling start() on the capsule instance. For the task-based compilation strategy, the capsule instance is also started by calling start(). This let us able to start up a capsule instance the same way regardless of the concrete compilation strategy.

The PaniniTaskPool class is shown in Figure 4.1.

When there is at least 1 task-based capsule instance that exists, init on line 4 is invoked to initialize an array of PaniniTaskPools. The number of pools is 1 by default but can be specified by the programmer at run time.

Every PaniniTaskPool instance runs in its own thread and maintain their own list of task-based capsule instances in a linked-list. When add is invoked by the PaniniCapsuleTask class, the PaniniCapsuleTask passed in is added to one of the PaniniTaskPool in the array in a round-robin fashion by calling _add as seen on line 20. The _add method is declared on line 27 and adds the PaniniCapsuleTask to the maintained linked list of capsule instances. Finally, the method panini$capsule$init is invoked right after to execute existing initializers. The head pointer of the linked list _headNode is declared on line 3. After a capsule instance is added to its list, the PaniniTaskPool thread is started as seen on line 22. The run method of PaniniTaskPool which is executed after the thread has started is shown on lines 38-51. Inside run, starting from the head of the list, run is being invoked on each PaniniCapsuleTask in the linked list one-by-one in a ring fashion as shown on line 42. If a **true** value is returned by the call, the capsule instance has terminated and is removed from the list. For the task pool strategy, the message processing logic is not inside a loop, and every single invocation on run will process at most one message inside the queue.

An example of a **design** block is shown below:

```
1 @Parallelism(4)
2 design {
3   task Console c;
4   Greeter g;
5   g(c);
6 }
```

The example **design** declares a Console capsule instance with the **task** keyword on line 3. This lets the capsule instance c to be initialized using the task-based compilation strategy. In the example, a parallelism annotation is provided right before the design block on line 1, defining the number of pools to be initialized to be 4.

```
1  final class PaniniTaskPool extends Thread {
2      private static boolean initiated = false;
3      private PaniniCapsuleTask _headNode = null;
4      static final synchronized void init(int size) throws Exception {
5        if ( initiated )
6          throw new Exception("Target already initialized");
7        poolSize = size;
8        _getInstance = new PaniniTaskPool[size];
9        for(int i=0;i<_getInstance.length;i++){
10         _getInstance[i] = new PaniniTaskPool();
11       }
12        initiated = true;
13     }

14     static final synchronized PaniniTaskPool add(PaniniCapsuleTask t) {
15       int currentPool = nextPool;
16       if (nextPool>=poolSize-1)
17         nextPool = 0;
18       else
19         nextPool++;
20       _getInstance[currentPool]._add(t);
21       if (!_getInstance[currentPool].isAlive()) {
22         _getInstance[currentPool].start();
23       }
24       return _getInstance[currentPool];
25     }

27     private final synchronized void _add(PaniniCapsuleTask t) {
28       if (_headNode==null){
29         _headNode = t;
30         t.panini$capsule$next = t;
31       }else{
32         t.panini$capsule$next = _headNode.panini$capsule$next;
33         _headNode.panini$capsule$next = t;
34       }
35       t.panini$capsule$init();
36     }

38     public void run() {
39       PaniniCapsuleTask current = _headNode;
40       while(true){
41         if (current.panini$capsule$size!=0) {
42           if (current.run() == true)
43             remove(this, current);
44           if (_headNode == null)
45             break;
46         }
47         synchronized(this) {
48           current = current.panini$capsule$next;
49         }
50       }
51     }

53     static final synchronized void remove(PaniniTaskPool pool, PaniniCapsuleTask
54     t) { /* ... */}

56     private static PaniniTaskPool[] _getInstance = new PaniniTaskPool[1];
57     private PaniniTaskPool(){}
58     private static int poolSize = 1;
59     private static int nextPool = 0;
60 }
```

Figure 4.1 The PaniniTaskPool Class.

The translated **design** block is shown in Figure 4.2. The number of pools are initialized as 4 on line 3 as specified by the programmer, and Console capsule c is initialized as a Console$task on line 5. Code generated for alternative versions of capsules are very similar to the thread-based version so that most of the code generation can be reused for each version, and reduces the modification needed in the compiler.

```
1  void panini$wire$sys() {
2    try {
3      PaniniCapsuleTask.panini$init(4);
4    } catch (Exception e) { }
5    c = new Console$task();
6    g = new Greeter$thread();
7    ((Greeter$thread)g).setName("g:Greeter");
8    ((Greeter$thread)g).s = c;
9    ((Console$task)c).panini$ref$count += 2;
10   ((Greeter$thread)g).panini$ref$count += 1;
11   this.panini$ref$count += 0;
12 }

14 void panini$capsule$init() {
15   g.start();
16   c.start();
17 }
```

Figure 4.2 Code generated from **design** block.

Finally, for the task-based compilation strategy of capsules, the specified number of PaniniTaskPool threads are first created, and then task-based capsules are assigned to one of the pools. When the PaniniTaskPool thread starts, it take turns to process a single message of each of the containing capsule instances. For this strategy, multiple capsule instances are assigned to a single thread, while not losing the isolation property of capsules.

4.2.1.3 Restrictions

In the task-based strategy, several task-based capsules share the same thread. The thread processes a message, which typically executes a procedure of one of the capsules before moving to the next one. Consider the situation where the procedure being executed sends a call to another task-based capsule in the same pool, and waits for the result. Because the sent message will never be processed, as the pool thread will never switch to the client capsule until the current message execution is completed, this

would create a deadlock. The same program, if under a thread-all configuration, would not have the same deadlock. This implies that two programs are not equivalent under different compilation strategies of capsules. To solve this problem, one strategy could be to assign capsule instances with dependencies into separate thread pools. A solution to automatically detect and remove such deadlocks is planned for the future.

4.2.2 Sequential Implementations

Another possible implementation strategy is to make capsules to run sequentially. For capsules with smaller procedures, this can be beneficial because of the reduction in of message sending and processing overhead. For capsules that frequently receive calls from different capsules at the same time, this can reduce waiting between threads. This can also be useful for capsules where procedures that callers typically need the result in a short time, i.e. there is little implicit concurrency between callee and caller. For capsules that are more suitable to run sequentially, Panini provides two alternative implementation strategies **monitor** and **sequential**, where capsules behave more like Java objects and receive messages as procedure invocations.

4.2.2.1 Implementation

The **monitor** implementation of Console is shown in Figure 4.3. The translation of the **monitor** version of Console is simply modifying the procedures with the **synchronized** keyword. This effectively protects states of the capsule from simultaneous modification. The capsule extends paniniCapsuleSequential which also implements the PaniniCapsule interface. The **sequential** version of capsules do not modify procedures with the **synchronized** keyword. Sequential do not provide thread safety whatsoever and should be used if and only if the capsule is being accessed by exactly one other capsule. Similar to the task-based implementation, capsules can be declared with **monitor** or **sequential** modifiers to use these sequential compilation strategies.

```
1  capsule Console$monitor() extends PaniniCapsuleSequential implements Console {
2      private int a = 0;

4      public final synchronized void write(String s) {
5          System.out.println(s);
6          a = 100;
7      }

9      public final synchronized void shutdown() {
10         panini$ref$count--;
11     }
12 }
```

Figure 4.3 Monitor based implementation of Console.

4.3 Summary

The thread-based implementation shown in Chapter 3 is the default compilation strategy for a capsule. However, it is not always ideal to use the thread-based implementation for every capsules in a program. In this chapter, three alternative compilation strategies are shown: the task-based strategy and two sequential implementations.

The task-based strategy splits the capsule instances into multiple pools. Each pool of capsule instance uses a single thread. The thread switches between capsule instances to execute one message in their queues at a time. This is useful for capsules that execute smaller operations that are independent of other capsules.

A challenge discussed in this chapter is about how using the task-based implementation may introduce deadlocks that are not present when using the thread-based approach. Capsule instances in the same pool with operations dependent to each other may deadlock. The assignment of capsule instances to pools is a challenge that can be addressed in the future. Other future plans include load balancing of pools as well for further optimization.

Two sequential compilation strategies were discussed in this chapter. These strategies are useful for capsules with operations that do not benefit from asynchronous execution. Basic thread safety is provided by the **monitor** implementation by making procedures synchronized, which protects the capsule from being accessed by two different threads simultaneously. The **sequential** implementation does not provide protection and should only be used when only a single other capsule is connected to

that capsule instance.

These alternative compilation strategies can be manually configured by the programmer when declaring capsule instances in the **design** block by adding modifiers to the instance declaration. In the future, it is our goal to automatically suggest these modifiers.

CHAPTER 5. SEPARATE COMPILATION

An important feature of the Java compiler is the integration of separate compilation [3, 4]. Separate compilation enables modules of a program to be compiled into separate objects, and linked together to form an executable program [43, 13, 5]. Advantages of separation compilation include reusability of modules, and the capability of editing and compiling modules separately without recompiling the entire program [60, 68, 12, 74, 15, 1, 42].

Separate compilation produces a link-time representation of modules. In this representation implementation details are hidden, e.g. the method bodies are abstracted away. However, this loss of information becomes a challenge for separate compilation of capsules, because link-time representation (Java bytecode) does not contain critical semantic information about capsules. For example, the information about the interconnections of capsules is lost during compilation. Even distinguishing a capsule from a Java class can become difficult.

To enable separate compilation in the Panini compiler, we need to retain the critical semantic information about capsules in the link-time representation. To solve this challenge, we make use of the Java annotations. Java annotations are a feature of the Java language that allow programmers to provide metadata of a program that is not directly a part of the program itself.

During code generation, the Panini compiler inserts annotations in the generated code of capsules and procedures that contain the semantic information so that it can be obtained from the .class files. An advantage of this solution is that it avoids modification in the bytecode generation process and remains compatible with the existing Java virtual machine. To avoid unnecessary bloat in the generated code, the semantic information of capsules and procedures is encoded to reduce the size overhead added to the generated code.

In this chapter, we show the process of annotation generation, and the encoding and decoding of the link-time representation.

5.1 Identifying Information Needed for Separate Compilation

The first thing we need to do is to identify the information that is needed by the compiler from the link-time representation of a capsule:

- *Capsule interconnection:* The wiring of capsules include the type that a capsule is connected to, and the identifier that is assigned to these capsules. To typecheck a capsule wiring declared in a *design*, we need the type of the capsules each capsules are wired to. As shown in §3.1.4, the code generation from the **design** connects capsules together by assigning capsule instances to local fields of the connecting capsule. To do so, we also assumed that the identifiers of wired capsules named by the programmers are available. Another importance of the wiring information is to allow the compiler to construct a system graph that shows the topology of the capsules and the communication pattern in the system, i.e. the call graph of capsules. The graph is used for verifying properties of the system such as potential sequential inconsistency.

- *Effect information of procedures:* To construct the system graph mentioned above, the call effects and the read/write effects of procedures are also needed. For example, from a procedure call `c.proc()` in a procedure, we know that a call path exists from the capsule to the exact capsule instance `c` is wired to, which we know from the capsule topology. Knowing messages being sent is often not enough for precise analyses. For example, to know if two call paths leading to the same capsule is an instance where the arrival order of message matters, we need to know exactly what the read/write effects of the procedures are. Therefore, the read/write effects of capsules are also included in the link-time representation.

To summarize, the information we add to the link-time representation includes: the capsule wiring information and side effects of the procedures. We include the former in an annotation added to the capsules, and the latter in annotations added to each procedure.

5.2 Annotation of Capsules

5.2.1 Capsule Annotations

In this section we show the capsule annotation we generate to be added to the link-time representation of capsules.

Four separate annotations are used for the four different compilation strategies Panini provides, thread, task, monitor and sequential. This is so that we can distinguish between classes generated for these different compilation strategies if we need to. However, except for the names, the implementations of these annotations are exactly identical. In this section we use the thread-based capsule annotation to explain the implementation.

The thread-based capsule annotation PaniniCapsuleDeclThread is shown in Figure 5.1.

```
1 @Retention(RetentionPolicy.RUNTIME)
2 @Target(ElementType.TYPE)
3 @Inherited()
4 public @interface PaniniCapsuleDeclThread {
5   String  params();
6   boolean definedRun();
7 }
```

Figure 5.1 The PaniniCapsuleDeclThread annotation.

The capsule annotation (PaniniCapsuleDeclThread) has a retention policy of RetentionPolicy.RUNTIME so that it will be retained in the link-time representation, and a ElementType.TYPE target type to specify it to be annotated on classes. The annotation has two fields: the params field is a string that represents the types and identifiers of the capsules the capsule is connected to, and the definedRun boolean denotes whether the capsule has an user-defined run method or not.

To further illustrate, consider the Greeter capsule shown in Figure 5.2. The Greeter capsule is connected to a single Console capsule instance with the identifier of cons, and there are no run methods declared inside the Greeter capsule. Based on these information, we generate the class Greeter$thread with the annotation:

@PaniniCapsuleDeclThread(params = "Console cons", definedRun = **false**).

```
capsule Greeter (Console cons) {
        String  message = "Hello World!";
        int  a=0;
        void greet(){
                cons.write("Panini:  " + message);
                proc();
        }

        private void proc(){  a = 100; }
}
```

Figure 5.2 A Greeter capsule declaration.

5.2.2 Procedure Annotations

Next we show the annotation generated for capsule procedures. As discussed, we want to know about the calls the procedures make, and read/write of fields by the procedure and use these metadata information for the compiler to perform static analyses. To do so, we generate @Effects annotation for procedures. The interface for @Effects annotations is shown in Figure 5.3. The @Effects annotation can be applied to methods and constructors, and is retained until after compile time using RetentionPolicy .RUNTIME. Inside the annotation, effects of the procedure are represented as a String array. This is because a Java annotation can only have fields of primitive types or arrays of primitive types.

```
@Documented
@Target({ElementType.METHOD, ElementType.CONSTRUCTOR})
@Retention(RetentionPolicy.RUNTIME)
public @interface Effects {
   String []  effects () ;
}
```

Figure 5.3 The @Effects annotation.

To show how effects of a procedure are encoded as a String, consider the greet procedure Greeter shown in Figure 5.2. In the example, greet reads from message, and then makes a call cons.write(" Panini: " + message). After that it calls a private method proc, which writes to the state a. The method greet have three effects. It reads from message, makes a call to cons.write while passing in a String, and then it writes to the local state a. These three effects are encoded into strings as follows:

- The read from message is encoded as RFGreeter$thread message. The RF prefix of the string

indicates a read effect of a field. The string is then concantenated with the capsule name Greeter$thread and the field name message.

- The call to cons.write is encoded as CGreeter$thread cons Consule write java.lang.String 1601 51 10. The C prefix of the string indicates a call effect. After the prefix, the string denotes the caller Greeter$thread, the callee type Console and name cons, the procedure name write, the parameters of the procedure (java.lang.String). The numbers following the string indicates the position of the call in the source code, and is used to construct detailed warning messages for the programmer.

- The write to a is encoded as WFGreeter$thread a. The prefix WF indicates a write effect on a field, followed by the name of the capsule Greeted$thread and field identifier a.

The three effects are then joined to annotate the greet procedure with the following generated @Effects annotation:

```
@Effects(effects =
    "RFGreeter$thread message",
    "CGreeter$thread s Stream write java.lang.String 1601 51 10",
    "WFGreeter$thread a").
```

5.2.3 Reading Annotations

During compile time, classes loaded from .class files are checked for Panini specific annotations. From the annotations, we are able to compute the interconnections of the capsules. The capsule annotations also helps differentiate capsules from classes. The @Effects annotations of capsules procedures can then be loaded, and we can decode the annotation back to effects of procedures, so that they can be used for static analyses.

5.3 Summary

Separate compilation is a property of a compiler that allows different modules, e.g. .java files, of a program to be compiled separately. The compilation process of the Java programming language and

its implementation in standard Java compilers is designed to provide separate compilation for object-oriented classes, but it is insufficient to support separate compilation of Panini capsules. This is primarily because some critical semantic information about capsules is not exposed in the meta-level of representation of classes in the Java bytecode.

In this chapter, we described our approach of adding semantic information about capsules in the generated bytecode. The information is added by leveraging Java annotations. The interconnection detail of capsules are needed for system graph construction and type checking, while the effects of procedures are needed for static analyses that require call paths and effects of procedures. Two different kind of annotations for capsules and procedures are generated by the compiler to add these information to link-time representations. By the use of annotations, capsules are able to be compiled separately in a way that the compiler can typecheck the wiring of capsules and obtain the information needed to perform static analyses of programs.

CHAPTER 6. EVALUATION

In this chapter, we investigate the viability of Panini by measuring the performance of Panini programs.

6.1 Benchmarks

For our evaluation we systematically rewrote multithreaded JavaGrande [70], and actor programs from Jetlang [66] and Habanero [38], into Panini programs. In our rewriting, multithreaded concurrent classes and actors are directly mapped to capsules; occasionally extra capsules are introduced for encapsulating shared data.

6.2 Performance

Panini enables modular reasoning while being comparatively performant. To substantiate this claim we measure and compare performances of Panini and original versions of multithreaded, streaming and actor benchmarks. Figure 6.1 shows these performance measurements.

Setup Performance measurements are for *steady-state performance* using one VM invocation and multiple benchmark iterations, where each benchmark is repeated within the VM until its performance reaches steady-state. Iteration time measurements are taken after steady-state is reached. For our experiments, steady-state performance is reached when the coefficient of variation of the most recent three iterations of a benchmark fall below 0.02 [30]. Average of steady-state performances of 30 runs are reported for each benchmark. Our experiments were run on a machine with 24 GB of memory and 24 cores clocked at 400 MHz, running on Linux 3.5.6-1.fc17 kernel with OpenJDK 64-Bit Server VM build 23.2-b09. Panini compiler latest version 0.9.3 was used to compile Panini benchmarks. JavaGrande programs were run for 4 threads and standard input size A.

				Performance (steady)		
	benchmark	***pattern***	***loc***	***original***	***Panini***	***ratio***
JavaGrande	SOR	MW	837	183	1192	6.51
	Sparse	MW	708	243	247	1.02
	Series	MW	750	3326	3278	0.99
	Crypt	MW	1184	102	91	0.89
	LUFact	MW	1152	60	735	12.75
	MonteCarlo	MW	3103	1018	941	0.92
	RayTracer	MW	1387	707	590	0.83
	MolDyn	MW	1137	566	1144	2.02
Jetlang	ThreadRing	EC	34	1630	1376	0.96
	PingPong	EC	46	4573	4410	0.84
	Download	FL	68	673	1170	1.73
Habanero	Prime	PL	65	1104	202	0.18
	Piprec	MW	192	861	180	0.21
	Suduko	MW	373	36270	40160	1.11

Figure 6.1 *(1)* Performance comparison of original and Panini versions of benchmarks. *(2)* Various concurrent programming patterns including master/worker (MW), pipeline (PL), loop parallelism (FL) and event-based coordination (EC). Performance numbers are in miliseconds.

Observations Our findings, in Figure 6.1, show that in most cases Panini's performance is comparable with corresponding original benchmarks. For multithreaded JavaGrande applications with heavy sharing, such as SOR, LUFact and Sparse, Panini versions are slower, at most 1.92 times slower for SOR, because of their confinement requirements, however, for larger benchmarks with different data sharing patterns confinement overhead may become negligible [55]. For other JavaGrande applications, Panini versions do almost the same or better, at most 5.55 times faster for Prime, because of thread and cache locality of capsules. Actor benchmarks that create higher number of messages Panini message creation strategy, discussed in §6.2.1, creates less number of objects per message and thus Panini versions do better.

6.2.1 Observation

To analyze the performance of Panini when compared to Multi-threaded Java, Jetlang and Habanero Scala, we profile the program execution and collect following three metrics: #context switches (voluntary), %cpu consumption and %cache misses. We believe that the overheads of message communication, mailbox contention and data copying are captured by these three metrics.

Data sharing is disallowed due to Panini's confinement model. We implement data sharing in the benchmarks via a shared store (a capsule). The overheads due to heavy sharing is captured by the high message communication and high mailbox contention at the shared store. The three components of performance overhead are:

- message communication overhead (effects messaging throughput and latency)
- mailbox contention overhead
- data sharing overhead

1) Low communication, low contention and low sharing Both Panini and Jetlang versions of ThreadRing benchmark program shows similar performance. ThreadRing program has low message communications, low mailbox contentions which is reflected in our #context switches metric values.

The Panini versions of Pi, PiPre and Sudoku Habanero Scala programs shows better performance although these programs have low message communications, low mailbox contentions. Similar values of #context switches, %cpu consumption and %cache misses between the two programs could not help us reason about the good performance of Panini versions for these benchmarks. We further profiled #context switches (involuntary: context switches due to time slice expiry) which reveals the reason for this. Panini versions have lower values of #context switches (involuntary) when compared to Scala versions. This is due to the diffference in the message processing logic. In Scala, the thread processing the messages of an actor performs busy-waiting when there are no messages in the queue. This leads to more time slice expiry when compared to voluntary-wait mechanism of Panini where the thread processing the messages gives up CPU when there are no messages in the queue instead of busy-waiting.

The performance of Panini programs Crypt, SparseMatMult, MonteCarlo, RayTracer and Series are close to the original Multi-threaded Java version. These programs are data parallel applications

implented using master/worker pattern. The low communication between entities in these programs leads to low mailbox contention and negligible data sharing overheads. This behavior is shown in Figure 6.2 which plots #context switches, %cpu consumption and %cache misses.

2) High communication, high contention and heavy sharing The Panini versions of JavaGrande SOR and MolDyn multi-threaded benchmark programs shows poor performance. This behavior is explained as follows. These benchmarks perform heavy data sharing and that leads to high message communication and high mailbox contentions at the shared stores (capsules). Figure 6.2 shows high values for #context switches and %cache misses which supports our reasoning for the poor performance of Panini versions of SOR and MolDyn programs.

3) High communication, high contention and low sharing The Panini version of Download benchmark shows poor performance against the Jetlang version using Jetlang ThreadFibers. The difference in behavior between the two versions both using threads are revealed by investigating the implementation of Jetlang ThreadFibers. ThreadFibers internally use batch processing of messages, which results in a better performance because of less context switches.

4) High communication, low contention and low sharing The Panini version of PingPong benchmark shows a similar performance compared to the Jetlang version. Batch message processing utilized by Jetlang does not have any performance advantages for this program, because only at most one message may be pending in the mailbox.

	Crypt	LUFact	Matmult	MonteCarlo	RayTracer	Series	SOR	Moldyn	Download	ThreadRing	PingPong	Pi	PiPre	Sudoku
Panini	91	735	247	941	590	3278	1192	1144	1170	1376	4410	139	180	4016
Original	102	60	243	1018	707	3326	183	566	673	1630	4573	717	861	3627

	Crypt	LUFact	Matmult	MonteCarlo	RayTracer	Series	SOR	Moldyn	Download	ThreadRing	PingPong	Pi	PiPre	Sudoku
Panini	482	9251	164	1513	1486	1264	999	681	2077	352	932	37	61	18127
Original	592	2096	271	1518	739	2172	468	3121	1856	153	800	128	2012	25352

	Crypt	LUFact	Matmult	MonteCarlo	RayTracer	Series	SOR	Moldyn	Download	ThreadRing	PingPong	Pi	PiPre	Sudoku
Panini	549	51408	323	577	792	584	22091	10837	31741	251	291067	293	726	8638
Original	495	3519	480	781	750	558	502	541	800	584	200916	482	984	53893

	Crypt	LUFact	Matmult	MonteCarlo	RayTracer	Series	SOR	Moldyn	Download	ThreadRing	PingPong	Pi	PiPre	Sudoku
Panini	266%	248%	234%	398%	389%	293%	348%	298%	252%	208%	180%	105%	181%	393%
Original	288%	361%	263%	466%	387%	385%	307%	406%	410%	238%	135%	123%	162%	423%

	Crypt	LUFact	Matmult	MonteCarlo	RayTracer	Series	SOR	Moldyn	Download	ThreadRing	PingPong	Pi	PiPre	Sudoku
Panini	23.5	11.29	21.31	12.83	16.54	20.22	23.06	6.19	9.02	56.04	28.46	17.21	15.33	31.28
Original	28.27	9.59	21.24	13.62	17.49	18.8	8.62	2.23	10.47	49.93	27.77	7.09	9.85	33.76

Figure 6.2 Panini benchmarks against their original versions: performance, (in)voluntary context switches, CPU usage and cache misses.

6.3 Evaluation: Programmability

To understand the programmability of Panini we have conducted an experiment to see how programmers perform when programming in Panini in comparison to programming in Java.

6.3.1 Experiment Setup

8 developers were asked to independently work on a program, which simulates activity of a stock market. The participants of the experiment were divided into 2 groups, one group were asked to use Java threads to complete the implementation, and the others were asked to implement using Panini. Unfortunately, one of the participants in the Panini group had to drop out of the experiment due to personal reasons, thus the final result only has 3 samples for the Panini group.

Each of the participants were given an identical copy of the specifications of the program they are to work on, an empty repository, and an spreadsheet for keeping record of the time spent on the project. Each of the revisions of the repository is a representation of different progress stage throughout the development process, and analyses was done on each revision to see how the program performs at each stages.

6.3.1.1 Participants

All the participants are trained in and familiar with multi-threaded programming. The participants range between undergraduate students, graduate students majored in computer science and faculty.

6.3.2 Research questions

RQ1: *Does Panini reduces the number of data race errors created by the developers?* We believe the isolation property of capsules make Panini programs to be less prone to data race errors caused by data sharing. We want to know if this is the case, and how often do data race errors happen in Java thread programs.

RQ2: *Does Panini programs require less code for the same application?* Less code may indicate the program is easier to implement, while more code might mean that more effort was needed to complete

the implementation. We would like to know if the implicit concurrency mechanism of Panini reduces the amount of code required from the programmer.

6.3.3 The problem description

The program that the participants were asked to implement is briefly described below. There are two primary entities in the simulation: the market, and the agents. There can be different kinds of agents in the simulations. Each agent as well as the market acts independently and do not operate in lockstep. The market runs indefinitely. A portfolio with a constant number of stocks are maintained by the market. The number of shares and price of each stock are updated by the market after processing each purchase/sale order from the agents. The market maintains its own capital, and keeps track of the purchase/sale orders done and report by standard output after a set numbers of order processing. Agents can independently place purchase/sale requests for the market. A local portfolio consisting of the stocks an agent hold is maintained by the agents. The purchase/sale decisions are based on a random factor and a type of behavior pre-assigned to each of agents. All the participants are required to implement their program so that the market and every agent executes on its own thread.

6.3.4 Evaluation Approach

To measure data races in the programs, we deployed a dynamic data race checker Roadrunner [27] on all of the revisions that can compile and run for each of the repositories. The check detects for simultaneous read write on the same field at run-time, and gives the count of unique occurrences. For tool reported data races, we have also manually checked the code to verify the results.

We use results to understand which repository revision where data races start to manifest in the programs, and how effective the prevention measures applied by the programmers has been. For commits where data races seem to have reduced, we manually inspect the code to understand what the programmers have done.

For RQ2, we simply count the line of code of the completed version of each of the programs.

6.3.5 Experiment Results

6.3.5.1 Data Race Occurrence

To measure the number of data races in each of the programs, Roadrunner was deployed on every revision that can be compiled and run. The measurement of data races of only the final completed version is shown in Figure 6.3.

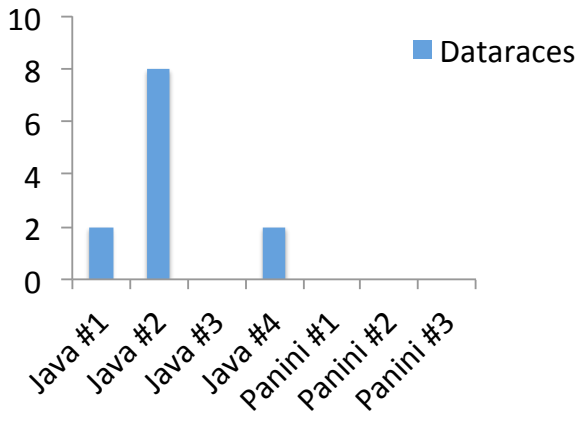

Figure 6.3 Data races measured on the final versions of each of the programs.

The results show a stark difference between Java and Panini versions. No data race occurrences were found in any of the Panini versions. However for the majority of the Java programs, data races appears to exist. The number of data races reported is of distinct fields or objects of the program, i.e. multiple occurrence of data race on the same object is only counted once. The reported data races are then manually inspected to confirm whether they are harmful or benign. The number of data races throughout the development timeline is shown in Figure 6.4.

The number of agents were set to 8 for all the programs. The programmers add different number of extra threads to the program in different revisions, and the final number of threads are detected by the data race checker. Even though some of the threads are only for testing purpose, we can observe when concurrency is being introduced to the program. From the chart, the occurrence of data races shows to have a positive correlation with the introduction of concurrency. When the programmers starts to create threads in their programs, data races start to occur. Prevention measures are then performed in the code in an attempt to remove the data races. The removal of data races have different degree of success for

different programmers, but for most of the programs, some data races remained.

Manual inspection of the code was also performed to verify the observations. The programmers mainly uses the Java **synchronized** keyword for mutual exclusion. All of the data races detected by the tool are manually confirmed to be simultaneous writing of number values by different threads.

Among the 4 Java programs, Java #3 appears to be an outlier and has no data races detected. Manually inspection of the code shows that the programmer has taken extreme measures by using the **synchronized** keyword excessively throughout the program. Although there was no noticeable performance impact in this experiment, excessive use of **synchronized** is known to be harmful, prone to deadlock, and may not be ideal in practice.

The results show that for this experimentation, the isolation property of Panini is able to successfully prevent programmers from introducing data race in their programs. On the other hand, data race appears to be a real issue for Java programmers in this experiment, and programmers struggled to completely eliminate data race errors from their programs.

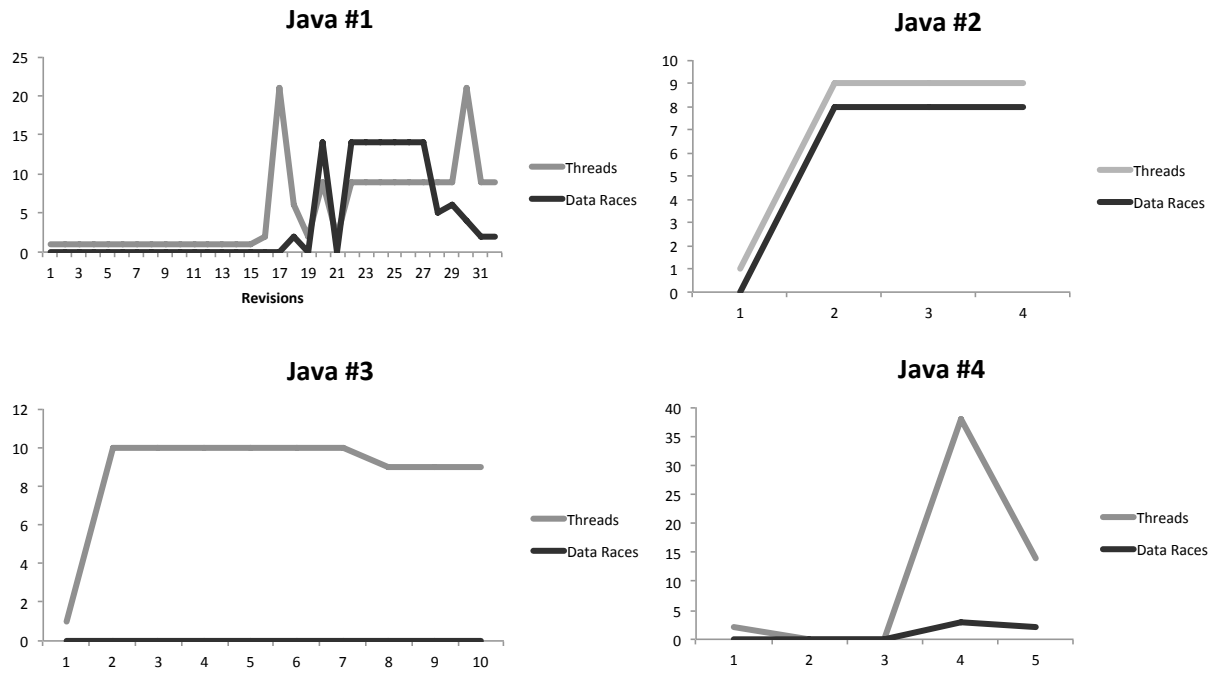

Figure 6.4 Number of data races and number of threads throughout the development.

6.3.5.2 Lines of Code Measurement

The number of lines of code is measured for each of the programs. Blank lines and comment lines are not counted. The result is shown in Figure 6.5. The Java programs range between 329 to 858 lines of code with an average of 630. The Panini programs range between 433 to 510 lines with an average of 461. We manually inspected the code to count the number of concurrency related code of each program, including Thread related operations such as *start* and *join*, and synchronized keywords for Java, and **design** block declarations for Panini. For the Java programs, between 6 to 24 lines of code that is concurrency related is found, while for Panini versions there is between 11-13 lines.

For this experiment, the result shows that the Panini programmers required less code for their implementation. This result however, is an early indication and more work is needed on this topic.

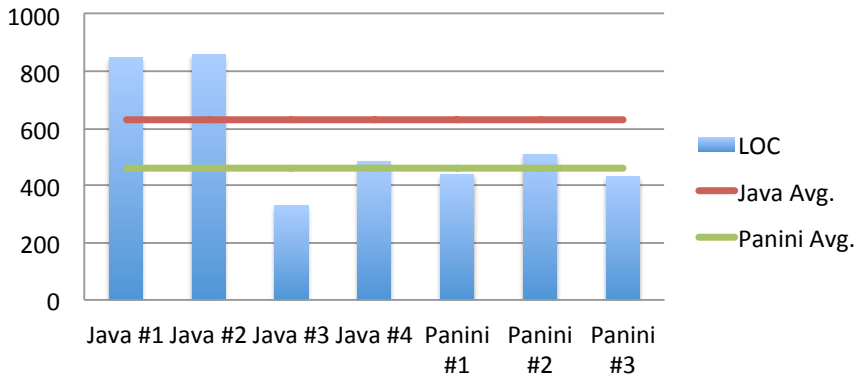

Figure 6.5 Lines of code for each of the programs.

6.4 summary

In this chapter, we evaluated the Panini compiler by two experiments. In the first experiment, we rewrote 14 multi-threaded benchmark programs from other languages such as Java, Jetlang, and Habanero. We then evaluated the performance of the Panini version of these programs and compared the performance to the original versions. In these programs, we have observed that although Panini shows a performance overhead for programs with heavy data sharing or high mailbox contention, the performance of Panini is comparable to other languages for other programs.

For the second experiment we performed an user study by letting two groups of developers to write programs with the same specification using Panini and Java. We then compared the development process by inspecting the different revisions of the programs. In this study, we have shown that Panini developers successfully implemented the programs without introducing data races. In comparison, most of the Java programs have data races in the final version.

CHAPTER 7. CONCLUSION AND FUTURE WORK

The need for concurrency was already apparent due to distributed, event-driven nature of modern software systems. The advent of multicore platforms is bringing it more to the forefront. Despite these pressing needs, concurrent programming remains hard and error prone. Researchers and practitioners have repeatedly observed that concurrent programming using explicit concurrency features like the threads and locks remains difficult to master for a sequentially trained programmer. In response to these observations, there has been significant interest in message passing abstractions as a solution to issues posed by explicit concurrency features. Contrary to common perception, use of the MPC abstractions, as implemented in the state-of-the-practice, does not by default, rules out concurrency hazards. In fact, it turns out that analyzing the full power of the MPC model to rule out concurrency hazards is a computationally hard problem. We explored a different point in the design space of MPC-based programming models that, when combined with an object-oriented model, has all of the important benefits of the full MPC model, but restricts its power only slightly so as to enable compile-time detection of some concurrency hazards in resulting programs. The main new feature proposed is capsule, a new MPC abstraction for concurrency. Capsules are realized in an extension of the Java programming language and implemented by extending the standard OpenJDK javac compiler. This work describes the implementation of the compiler in detail, to show that the capsule abstraction can seamlessly work with object-oriented languages. Evaluation of the performance of compiled Panini programs and a user study to understand the usability of the language are also shown in this work with initial success. Within the boundaries of a capsule, programmers can continue to think sequentially, and reuse legacy sequential code.

Future work The research on capsules is ongoing as of this writing. Plans to explore and expand the capabilities of capsules are still in progress at this moment. In the near future, there are a few research directions that can be explored:

- *Integration with event-driven features.* The design of capsules proposed in this thesis provides passive procedures, i.e. a capsule can only perform actions requested by other capsules or work autonomously. Modern software increasing require reactive concurrent components. To address these needs, it would be sensible to integrate event-driven features such as quantified, typed events [65, 64] and explore reasoning mechanisms [9, 10, 7] for this combination.

- *Relaxing constraints on the design feature.* The current design feature imposes many constraints on the topology of capsules. These constraints are to enable easier verification of design expressions, so certain potential problems such as sequential inconsistency can be identified easily. However this also creates a practical constraint for the user. A promising direction to relax the constraints on the design feature of a capsule is to treat a capsule design as a parameterized system and use recently proposed automated techniques [34, 35], to compute smaller system instance for verification purposes. This allows us to relax constraints on design while being able to verify desired properties of the system.

- *Optimizing capsule compilation strategies.* Panini provides a set of different compilation strategies for capsules. However, figuring out the optimal configuration is left for the programmer. Related works that studies the mapping actors to threads show that mappings can be done statically by simple heuristics [72, 73]. We may be able to integrate these techniques to suggest capsule implementation and pool size configurations to the user, and also automatically perform pool load-balancing for better CPU utilization.

- *Large-scale empirical study.* Dyer *et. al.*'s work [25] uses the Boa infrastructure [21, 22] to mine source code features [26]. To understand the applicability of capsule abstractions, we can use the same infrastructure in a similar manner. This can allow us to know the capabilities of capsules. If we can identify places where capsules comes short, it also makes it possible to work on solutions to overcome what we are lacking. Rajan [57] and then Rajan *et. al.* [63] have also used GOF design pattern [28] to evaluate languages, which may also be a viable evaluation strategy for capsules. Yet another study is performed by Dyer *et. al.* on the design impact of language features [23, 24], where we can use a similar strategy to evaluate Panini.

BIBLIOGRAPHY

[1] Rolf Adams, Walter Tichy, and Annette Weinert. The cost of selective recompilation and environment processing. *ACM Trans. Softw. Eng. Methodol.*, 3(1):3–28, 1994.

[2] Gul Agha and Carl Hewitt. Concurrent programming using actors: Exploiting large-scale parallelism. In *Foundations of Software Technology and Theoretical Computer Science*, pages 19–41. Springer, 1985.

[3] D. Ancona, G. Lagorio, and E. Zucca. True separate compilation of java classes. In *PPDP '02: Proceedings of the 4th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 189–200, New York, NY, USA, 2002. ACM Press.

[4] Davide Ancona, Giovanni Lagorio, and Elena Zucca. A formal framework for java separate compilation. In *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 609–636, London, UK, 2002. Springer-Verlag.

[5] Andrew W. Appel and David B. MacQueen. Separate compilation for standard ml. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 13–23, New York, NY, USA, 1994. ACM Press.

[6] J. Armstrong, R. Williams, M. Virding, and C. Wikstroem. *Concurrent Programming in ERLANG*. Prentice-Hal, 1996.

[7] Mehdi Bagherzadeh, Robert Dyer, Rex D. Fernando, Jose Sanchez, and Hridesh Rajan. Modular reasoning in the presence of event subtyping. In *Modularity'15: 14th International Conference on Modularity*, March 2015.

[8] 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.

[9] Mehdi Bagherzadeh, Hridesh Rajan, and Ali Darvish. On exceptions, events and observer chains. In *AOSD '13: 12th International Conference on Aspect-Oriented Software Development*, March 2013.

[10] Mehdi Bagherzadeh, Hridesh Rajan, Gary T. Leavens, and Sean Mooney. Translucid contracts: Expressive specification and modular verification for aspect-oriented interfaces. In *AOSD '11: 10th International Conference on Aspect-Oriented Software Development*, March 2011.

[11] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von neumann machines via region representation inference. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 171–183, New York, NY, USA, 1996. ACM.

[12] Michael Burke and Linda Torczon. Interprocedural optimization: eliminating unnecessary recompilation. *ACM Trans. Program. Lang. Syst.*, 15(3):367–399, 1993.

[13] Luca Cardelli. Program fragments, linking, and modularization. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 266–277, New York, NY, USA, 1997. ACM Press.

[14] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. Habanero-Java: The new adventures of old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 51–61, New York, NY, USA, 2011. ACM.

[15] Craig Chambers, Jeffrey Dean, and David Grove. A framework for selective recompilation in the presence of complex intermodule dependencies. In *ICSE '95: Proceedings of the 17th international conference on Software engineering*, pages 221–230, New York, NY, USA, 1995. ACM Press.

[16] Rohit Chandra, Anoop Gupta, and John L. Hennessy. Cool: An object-based language for parallel programming. *Computer*, 27(8):13–26, August 1994.

[17] 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.

[18] 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.

[19] Sylvan Clebsch and Sophia Drossopoulou. Fully concurrent garbage collection of actors on many-core machines. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 553–570, New York, NY, USA, 2013. ACM.

[20] Frank S. de Boer, Mahdi M. Jaghoori, Cosimo Laneve, and Gianluigi Zavattaro. Decidability problems for actor systems. In *Proceedings of the 23rd International Conference on Concurrency Theory*, CONCUR'12, pages 562–577, Berlin, Heidelberg, 2012. Springer-Verlag.

[21] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *35th International Conference on Software Engineering*, ICSE'13, pages 422–431, May 2013.

[22] 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.

[23] 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.

[24] Robert Dyer, Hridesh Rajan, and Yuanfang Cai. Transactions on aspect-oriented software development x. chapter Language Features for Software Evolution and Aspect-oriented Interfaces: An Exploratory Study, pages 148–183. Springer-Verlag, Berlin, Heidelberg, 2013.

[25] Robert Dyer, Hridesh Rajan, Hoan Anh Nguyen, and Tien N. Nguyen. Mining billions of AST nodes to study actual and potential usage of java language features. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 779–790, New York, NY, USA, 2014. ACM.

[26] 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, pages 23–32, 2013.

[27] Cormac Flanagan and Stephen N. Freund. The roadrunner dynamic analysis framework for concurrent programs. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '10, pages 1–8, New York, NY, USA, 2010. ACM.

[28] 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.

[29] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 1–11, New York, NY, USA, 2003. ACM.

[30] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 57–76, New York, NY, USA, 2007. ACM.

[31] Christian Grothoff, Jens Palsberg, and Jan Vitek. Encapsulating objects with confined types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(6):32, 2007.

[32] Philipp Haller and Martin Odersky. Scala Actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, February 2009.

[33] Philipp Haller and Martin Odersky. Capabilities for uniqueness and borrowing. In *Proceedings of the 24th European Conference on Object-oriented Programming*, ECOOP'10, pages 354–378, Berlin, Heidelberg, 2010. Springer-Verlag.

[34] Youssef Hanna, Samik Basu, and Hridesh Rajan. Behavioral automata composition for automatic topology independent verification of parameterized systems. In *The 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 09)*, Amsterdam, The Netherlands, August 2009.

[35] Youssef Hanna, David Samuelson, Samik Basu, and Hridesh Rajan. Automating cut-off for multi-parameterized systems. In *The 12th International Conference on Formal Engineering Methods (ICFEM 2010)*, Shanghai, China, November 2010.

[36] 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.

[37] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 26(1):100–106, January 1983.

[38] 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.

[39] Dennis Kafura, Doug Washabaugh, and Jeff Nelson. Garbage collection of Actors. In *Proceedings of the European Conference on Object-oriented Programming on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA/ECOOP '90, pages 126–134, New York, NY, USA, 1990. ACM.

[40] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979.

[41] R Greg Lavender and Douglas C Schmidt. Active object: an object behavioral pattern for concurrent programming. In *Pattern languages of program design 2*, pages 483–499. Addison-Wesley Longman Publishing Co., Inc., 1996.

[42] Xavier Leroy. Manifest types, modules, and separate compilation. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 109–122, New York, NY, USA, 1994. ACM Press.

[43] Michael R. Levy. Type checking, separate compilation and reusability. In *SIGPLAN '84: Proceedings of the 1984 SIGPLAN symposium on Compiler construction*, pages 285–289, New York, NY, USA, 1984. ACM Press.

[44] Eric Lin, Ganesha Upadhyaya, Sean L Mooney, and Hridesh Rajan. Duck futures: A generative approach to transparent futures. Technical report, Dept. of Computer Science, Iowa State University, June 2015.

[45] 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.

[46] Yuheng Long, Sean L. Mooney, Tyler Sondag, and Hridesh Rajan. Implicit invocation meets safe, implicit concurrency. In *GPCE*, pages 63–72. ACM, 2010.

[47] Giuliano Losa, Vibhore Kumar, Henrique Andrade, Buğra Gedik, Martin Hirzel, Robert Soulé, and Kun-Lung Wu. CAPSULE: Language and system support for efficient state sharing in distributed stream processing systems. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, DEBS '12, pages 268–277, New York, NY, USA, 2012. ACM.

[48] Robin Milner. *Communicating and Mobile Systems: The pi-calculus*. Cambridge University Press, New York, NY, USA, 1999.

[49] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990.

[50] James G. Mitchell, William Maybury, and Richard Sweet. Mesa language manual. Technical Report CSL-79-3, Xerox Palo Alto Research Center, 1979.

[51] Stijn Mostinckx, Tom Van Cutsem, Jessie Dedecker, Wolfgang De Meuter, and Theo D'Hondt. Ambient-oriented programming in ambienttalk. In *Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 92–93, New York, NY, USA, 2005. ACM.

[52] 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.

[53] Michael Papathomas. Object-oriented software composition. chapter Concurrency in Object-oriented Programming Languages, pages 31–68. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1995.

[54] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, December 1972.

[55] Alex Potanin, Monique Damitio, and James Noble. Are your incoming aliases really necessary? counting the cost of object ownership. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 742–751, Piscataway, NJ, USA, 2013. IEEE Press.

[56] Polyvios Pratikakis, Jaime Spacco, and Michael Hicks. Transparent proxies for Java futures. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 206–223, New York, NY, USA, 2004. ACM.

[57] Hridesh Rajan. Design pattern implementations in eos. In *Proceedings of the 14th Conference on Pattern Languages of Programs*, PLOP '07, pages 9:1–9:11, New York, NY, USA, 2007. ACM.

[58] 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.

[59] 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.

[60] Hridesh Rajan, Robert Dyer, Youssef Hanna, and Harish Narayanappa. Preserving separation of concerns through compilation. In Lodewijk Bergmans, Johan Brichau, and Erik Ernst, editors, *Software Engineering Properties of Languages and Aspect Technologies (SPLAT 06), A workshop affiliated with AOSD 2006*, March 2006.

[61] Hridesh Rajan, Steven M. Kautz, Eric Lin, Sarah Kabala, Ganesha Upadhyaya, Yuheng Long, Rex Fernando, and Loránd Szakács. Capsule-oriented programming. Technical Report 13-01, Iowa State U., Computer Sc., 2013.

[62] 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.

[63] 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.

[64] Hridesh Rajan and Gary T. Leavens. Quantified, Typed Events for Improved Separation of Concerns. Technical Report TR07-14d, 2007.

[65] Hridesh Rajan and Gary T. Leavens. Ptolemy: A language with quantified, typed events. In *ECOOP '08: 22nd European Conference on Object-Oriented Programming*, July 2008.

[66] Mike Rettig. Jetlang. http://code.google.com/p/jetlang/, 2008.

[67] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-time Object-oriented Modeling*. John Wiley & Sons, Inc., New York, NY, USA, 1994.

[68] Zhong Shao and Andrew W. Appel. Smartest recompilation. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 439–450, New York, NY, USA, 1993. ACM Press.

[69] Bruce Shriver and Peter Wegner, editors. *Research Directions in Object-oriented Programming*. MIT Press, Cambridge, MA, USA, 1987.

[70] L. A. Smith, J. M. Bull, and J. Obdrzálek. A parallel Java Grande benchmark suite. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*, SC '01, pages 8–8, New York, NY, USA, 2001. ACM.

[71] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and computation*, 132(2):109–176, February 1997.

[72] Ganesha Upadhyaya and Hridesh Rajan. An automatic actors to threads mapping technique for JVM-based actor frameworks. In *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control*, AGERE! '14, pages 29–41, New York, NY, USA, 2014. ACM.

[73] 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.

[74] Alexander L. Wolf, Lori A. Clarke, and Jack C. Wileden. Interface control and incremental development in the pic environment. In *ICSE '85: Proceedings of the 8th international conference on Software engineering*, pages 75–82, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press.

[75] A Yonezawa and M Tokoro, editors. *Object-oriented Concurrent Programming*. MIT Press, Cambridge, MA, USA, 1986.

[76] Jisheng Zhao, Roberto Lublinerman, Zoran Budimlić, Swarat Chaudhuri, and Vivek Sarkar. Isolation for nested task parallelism. In *Proceedings of the 2013 ACM SIGPLAN International Confer-*

ence on Object Oriented Programming Systems Languages & Applications, OOPSLA '13, pages 571–588, New York, NY, USA, 2013. ACM.