2015

A unified design of capsules

Sean Lawrence Mooney
Iowa State University

A unified design of capsules

by

Sean Lawrence Mooney

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

David M. Weiss

Iowa State University

Ames, Iowa

2015

DEDICATION

To all those whose kept me sane – especially my wife.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

This thesis is the product of several years worth of work and its completion would not have been possible with the help of many others. First, thanks go to Ganesha Upadhyaya for adapting of the SteamIt benchmarks to Panini 0.9.1 and for helping me adapt them again to the unified design for the case studies in Chapter 3 and Chapter 4. Second, I would like to acknowledge the US National Science Foundation (NSF) grants CCF-14-23370, CCF-13-49153, CCF-11-17937, CCF-10-17334, and CCF-08-46059 which partially supported my work. Finally, I would like to thank my major professor, Hridesh Rajan, for his advice and insights over the years.

MY OTHER RESEARCH CONTRIBUTIONS

During my research career at Iowa State University, I have also actively contributed to several other projects. Below I list some of these. More details about these work can be found in referenced papers.

1. *Translucid contracts*: In collaboration with Mehdi Bagherzadeh, Hridesh Rajan and Gary T. Leavens, I helped developed the notion of translucid contracts [6, 7] that are a greybox specification mechanism for reasoning about implicit invocation systems [29, 44]. We showed that translucid contracts enabled modular reasoning about these kinds of systems, which was thought to be a challenge. In subsequent work, Bagherzadeh and others have further extended the model [12, 5, 3], however, basic ideas were proposed in our collaborative work. My work helped realize these ideas in the compiler of the Ptolemy programming language [38, 37, 39]. Ptolemy is a unified event-driven programming languages [40, 41, 42, 31, 43] that combines beneficial properties of both aspect-oriented and implicit invocation languages. I also helped bootstrap some of the initial work on studying benefits of the Ptolemy approach [10, 11] due to my expertise with the Ptolemy compiler.

2. *Asynchronous event types*: In collaboration with Yuheng Long, Tyler Sondag, and Hridesh Rajan, I helped develop the notion of asynchronous event types [25, 26]. Asynchronous event types are a programming abstraction that decouple modules that announce events (subjects) and modules that respond to events (observers). One of the interesting aspects of this new abstraction is that they enable concurrency between observers. This work on asynchronous event types was along the lines of other ongoing work aimed at reconciling modularity and concurrency goals [32, 36] that I also contributed to [27].

In summary, I have made several significant research contributions. However, in this thesis I focus on describing my contribution towards unifying systems and capsules in capsule-oriented programming.

ABSTRACT

The process of reading, writing, and reasoning about concurrent programs benefits from better abstractions for concurrency than what many common languages, such as Java, offer. Capsule-oriented programming and the Panini language utilize the idea of combining state and control within a linguistic mechanism along with asynchronous message passing to provide sequentially trained programmers with an actor-like language that preserves the expected sequential semantics. The initial design of the Panini language splits the world into two distinct elements – capsules and systems. A capsule acts as the unit of both modularity and concurrency in the program. A system acts as the sole point of composition for capsule instances. The problem is that the dichotomy between systems and capsules leads to uncomposable and non-modular programs. The connections between capsule instances in a system declaration are fixed at exactly one point and all capsules instances in program must be declared and connected to each other at a single block of code. This thesis will explore the implications on modularity and reuse of systems when a basic design decision – separating capsules and systems – is relaxed to allow a capsule to declare an internal composition of other capsule instances.

CHAPTER 1. INTRODUCTION

1.1 The need for concurrency

Writing concurrent code is easy, but writing correct concurrent code is much more difficult. It has been argued that one of the problems with concurrent programming is the absence of sufficiently high level abstractions to manage the distinct and separate concerns which must go into any concurrent program [20]. Often times, the framework and logic to manage any concurrency must be tangled into the main logic of the program. The inherent close coupling of the concurrency framework and the program logic prevents programmers from understanding and implementing either piece in isolation, which leads to more complicated and error prone programs [36, 32, 25].

One successful abstraction for managing concurrency is the actor programming model [1]. Actors represent independent entities within a program and generally communicate by sending 'messages' to each other, as opposed to 'calling' functions, as happens in traditional programming models. A message generally implies that the recipient is free to choose when to act on the message. The message passing paradigm can be further divided into two groups – messages which contain shared data and messages which are completely immutable, by virtue of being a copy of the original data. Erlang [2] falls in to the later category while Scala [30] and Akka [15] fall into the former.

Although the actor-oriented paradigm provides a better abstraction between the concurrency mechanism and the program logic, an impedance mismatch still remains between expectations of sequentially trained programmers and the requirements of writing concurrent programs with actors. Specifically, the interaction between entities must still be considered and the expected serial semantics are not always preserved.

1.2 Overview of the Panini language

The Panini language is an actor style extension to the Java language. Its goal is to allow programmers to do what they are good at – decompose a problem into a collection of related modules and then automatically enable safe and correct concurrency at the module boundaries [35, 34]. Panini introduces a concept called *capsule-oriented* programming [35, 34, 33, 4]. This style of programming strives to reconcile modularity concerns with concurrency concerns by automatically enabling concurrency via the module mechanism of the language. A capsule serves as the unit of modularity within the program. Capsule instances are connected to other capsule instances in a fixed manner and the connection topology between capsule instances is not allowed to change during run-time. Capsule-oriented Panini programs use a combination of compile-time static analysis and a specifically designed runtime to ensure the runtime behavior remains consistent with the expected serial semantics.

The initial version of Panini, 0.9.1, introduced two new type declarations, *capsule*s and *system*s, to the Java language. A capsule is used to define the computation of a program. A system is used to declare both the instances of each capsule type and how those capsule instances are connected to each other. There is only one system in a program.

Panini 0.9.1 strongly separated the role of a capsule from the role of a system in a Panini program. Small example programs worked well with the single system approach in the first version of Panini. However, the separation between capsules and systems proved to be too limiting in non-toy programs to make the language viable for implementing larger capsule-oriented programs. For example, a set of complex connections between many capsule instances cannot be abstracted out behind a type name, nor can they be reused. The unification of systems and capsules to enable a hierarchical composition of capsules in Panini version 0.9.2 is the primary subject of this thesis.

Typographic Conventions: The presentation of Panini's syntax will use the following typographic conventions: keywords are in **bold-face**, optional elements are enclosed by square braces ([]), and a Kleene star (*) indicates an arbitrary number of elements.

capsule-decl ::= **capsule** *c* [**implements** *s**] capsule-param* capsule-body	"Capsule Declaration"
signature-decl ::= **signature** *s* (*t p* formal-arg*)*	"Signature Declaration"
capsule-param ::= formal-arg	"Capsule Parameter"
capsule-body ::= state-decl* proc-decl*	"Capsule Body"
state-decl ::= *t n**	"Capsule State"
proc-decl ::= *t p* formal-arg* proc-body	"Capsule Procedure"
formal-arg ::= *t n*	"Formal Argument"
proc-body ::= java-stmt*	"Procedure Body"

where

$c,s \in \mathbb{C}$, $\mathbb{C}$ is the set of valid capsule names
$n \in \mathbb{N}$, $\mathbb{N}$ is the set of valid state names
$p \in \mathbb{P}$, $\mathbb{P}$ is the set of valid procedure names
$t \in \mathbb{T}$, $\mathbb{T}$ is the set of valid type names

Figure 1.1 Panini 0.9.1 capsule syntax

1.2.1 Capsule declarations

The abstract syntax for a capsule declaration in Panini 0.9.1 is shown in Figure 1.1. A capsule is similar to a class in the standard object-oriented style of programming, in that there may be multiple, independent instances within a single program. A capsule's definition is composed of a name for the capsule, an optional list of implemented signatures, a list of *wiring parameters*, a list of *state* declarations, and list of capsule *procedures*. Like classes in standard object oriented programming, a capsule declaration is the 'blue print' for the individual and independent capsule instances within a Panini program.

Capsule instances are connected to other capsule instances via the capsule's *parameters*. The parameter list forms the 'wiring' interface between capsule instances and gives the programmer a 'local' name to use to refer to the connected capsule instances inside capsule's procedures. The specific place or places where capsule instances can be declared and connected is an important design decision for a language implementing a capsule-oriented style of programming. The topology must be statically determinable at certain points to keep several analyses in the compiler tractable [35, 34].

Capsule *state* is similar to an instance field in a class, with the additional restriction that a reference to a piece of state must stay within the capsule. Confining state inside of a capsule instance allows

each instance to act as a memory region and prevents unfederated access by another capsule instance. Preventing outside mutation of a capsule's state is important to preserve the serial semantics of the capsule.

Capsules communicate with each other via *procedure calls*. The syntax for both declaring and calling a procedure is intentionally identical to a 'regular' Java method. A procedure represents a point for implicit concurrency within the program. Procedure calls return immediately to the caller and place a message in the receiver's message queue, which is then processed on the receiver's thread. The immediate return from the called procedure is handled by returning a *future* for the procedure's value instead of waiting for the actual value to be computed [17, 51]. The future wrapper is automatically generated by the compiler and managed by the Panini runtime. Each capsule instance has a private message queue and services messages in a first-in, first-out (FIFO) order. The FIFO message queue preserves the sequential semantics of the program with respect to the capsule instance.

A capsule is either *active* or *non-active*. Active capsules contain a programmer defined `run` method and are similar to Java classes that implement the `Runnable` interface. An active capsule executes its `run` method as soon as all the capsule instances are connected together at run time and then shuts down. Non-active capsules listen for messages from other capsules instances and respond to those messages. They continue to listen until receiving a shutdown message, at which point they terminate. Panini uses a reference-counting style garbage collection algorithm to ensure non-active capsules do not shutdown until all of the active capsules they are connected to have shutdown.

Finally, a capsule may implement one or more *signatures*. A signature is similar to a Java *interface* and defines a set of procedures which the capsule must implement.

1.2.2 System declarations

The abstract syntax for a system declaration is show in Figure 1.2. A system declaration is composed of a system name, an optional system argument, a list of capsule instance declarations and a list of wiring statements. The optional system argument allows the system to access the command line arguments when a Panini program is executed.

Each capsule instance statement creates a new instance of a capsule in the system. Unlike regular classes, capsule instances do not need to be manually instantiated with the `new` keyword. The compiler

sys-decl ::= **system** *s* sys-body	"System Declaration"
sys-body ::= cap-inst-decl* wiring-stmt*	"System Body"
cap-inst-decl ::= *c n**	"Capsule Instance"
wiring-stmt ::= *n*(*n**)	"Wiring statement"

where

$c \in \mathbb{C}$, $\mathbb{C}$ is the set of valid capsule names
$n \in \mathbb{N}$, $\mathbb{N}$ is the set of valid capsule instance names
$s \in \mathbb{S}$, $\mathbb{S}$ is the set of valid system names

Figure 1.2 Panini 0.9.1 system syntax

takes care of generating the bytecode to ensure each capsule instance is properly constructed at runtime. Wiring statements define the connections between capsules instances. A wiring statement is composed of the name, *n* of the capsule instance to wire, followed by a list of other capsule instance names to connect to *n*. The types of the names in the wiring statement must match the declared types of the parameters in *n*'s definition.

In Panini 0.9.1 the system declaration serves as the program entry point, the only point of capsule instance creation and the only place where capsule instances may be connected together via wiring statements. A system declaration requires a programmer to define *all* of the capsule instances in the program and declaratively state the connections between the capsule instances. A system also provides the compiler with a concrete point to construct a graph of all the potential communication channels within a Panini program. This graph is used to check several properties, including sequential consistency, about the system.

1.2.3 Lifecycle of a Panini program

The lifecycle of a Panini program has two basic phases. Program execution begins in the program's system. The system instantiates each declared capsule instance and connects those instances together, according to the system's wiring statements. Once each capsule instance is wired, it is ready to run as an independent entity. Each instance is started by the system and the Panini program begins the second phase of execution. Each instance is now running and, if capsule does not have a programmer defined `run()` method will simply process its message queue until it gets a shutdown message. If the capsule

does have a defined `run()` method, it will execute that method and terminate. The program continues to run until all capsule instances have terminated.

1.2.4 Searching an address book in Panini

Let us now consider an example which presents both the basic structure of a Panini program, as well as exposing the weaknesses of the system declaration. The program in Figure 1.3 implements a basic address book search in Panini. The goal of the program is to search multiple address books for a series of names and to print out the result of each search.

```
1  signature Book {
2      // return The address if the name found, null if the
           name is not found.
3      Address search(String name);
4  }
5  capsule CSVBook(File file) implements Book { ... }
6  capsule ISUBook implements Book { ... }
7  capsule Search(Book[] books) {
8    List<Address> search(String name) {
9      Address[] addrs = new Address[books.length];
10     // Execute all searches concurrently
11     for(int i = 0; i < books.length; i++) {
12       addrs[i] = books[i].search(name);
13     }
14     ArrayList<Address> results
15                         = new ArrayList<Address>();
16     // Filter out 'null' addresses
17     for(int i = 0; i < addrs.length; i++ {
18       if(addrs[i] != null) { results.add(addrs[i]); }
19     }
20     return res;
21   }
22 }
23 capsule AddressBook(Search search) {
24   String[] names = ... ;
25   void run () {
26     for(String n : names) {
27       // Look for the name and print out each address
             found
28       for(Address l : search(n))
29         System.out.println(l);
30     }
31   }
32   List<Address> search(String name) {
33     return search.search(name);
34   }
35 }
36 system Main {
37   CSVBook csv;
38   ISUBook isu;
39   Search search;
40   AddressBook driver;
41   //Wire the csv book to its data file
42   csv(new File("path/to/file"));
43   //Wire the books into the search capsule
44   search(new Book[]{csv, isu});
45   //Wire the search element into the address book
46   driver(search);
47 }
```

Figure 1.3 Example search program in Panini 0.9.1

The parts of the program are as follows: The signature `Book`, lines 1-4, defines a procedure `search` which any `Book` must implement. Two capsules, `CSVBook` and `ISUBook`, implement the signature and each one provides search functionality from a different data source – a comma separated list of names and the ISU directory, respectively. The `Search` capsule, lines 7-22, has a parameter called `books` of type `Book[]` which it will use to run the search. The `AddressBook` capsule, lines 23-35, is wired to a `Search` capsule which is used to search for a list on names on line 33. Finally, the system `Main`, lines 36-47, declares all of the capsules instances and connects those in-

stances together. Instances of `CSVBook` and `ISUBook` books are wired to the `Search` instance `search` (line 44) and `search` is wired to the 'driver' `AddressBook` capsule instance `driver` (line 46). The `AddressBook` contains a `run` procedure which automatically starts after the wiring is completed when the program executes.

Program execution starts in the system `Main`. First, the capsule instances declared in `Main` are instantiated and then those components are connected together according to the wiring statements. Next, the `run` procedure in the `AddressBook` capsule is automatically started. Capsules which define a `run` procedure are considered *active* and are started as soon as the system's capsules are instantiated and wired together. The `run` procedure in `AddressBook` drives the program by iterating over the list of names and searching for each one by calling the `search` procedure on the parameter called `search`. The `Search` capsule calls the `search` procedure on each `Book` in its list of `Books`. All procedure calls are asynchronous and immediately return a *future* for the value of the procedure call. This technique allows the `search` procedure for each book to execute in parallel because the values of the returned futures are not claimed until later in the procedure at line 18. Finally, when the `run` procedure in `AddressBook` terminates, the system terminates the other non-active capsule instances and the program exits.

1.3 Limitations of system declarations in Panini

The initial design decision to have distinct capsules and systems in the Panini language has been more restrictive than anticipated. The primary problems with the distinct system declarations center around reuse and encapsulation of capsule instance wiring. First, the capsule instances and wiring connections cannot be reused. For example, the connections created for the search program in the `Main` system cannot be used in any other system that also needs to search for names in the address book. Rather, the new system must explicitly recreate the capsule instances and connections in `Main`. The inability to reuse systems leads to the second major problem. Namely, groups of capsule declarations and wiring statements cannot be encapsulated behind a system type. *All* of the capsule instances and wiring statements in the entire Panini program must be contained within a single system. In small programs, these restriction are not an issue. But as the number of capsule instances and the number of

wiring statements increases, a single system becomes a large, unmaintainable proposition. Specifically, a single system declaration does not allow for systems to be composed together in a hierachical fashion and has the following problems:

1. Limits on the ability to keep system declarations small. *All* capsule instances in a program must be declared and connected within a single system.

2. Prevents reuse of existing systems and the capsule connections within a system. Systems cannot include or be connected to other systems.

3. Creates an unusual dichotomy between the top level types of the language. Capsules are used to construct a system but systems cannot be used by capsules.

This thesis explores removing these limitations by unifying capsules and systems in capsule-oriented programming.

1.4 Outline

The remainder of this thesis is organized as follows: Chapter 2 presents the unification of capsules and systems in Panini 0.9.2. Chapters 3 and 4 present two case studies which demonstrate the improvements in modularity and reuse of the unified approach. Chapter 5 discusses related work and Chapter 6 concludes the thesis.

CHAPTER 2. UNIFICATION

2.1 Requirements for unifying systems with capsules

One of the goals of the Panini project is to investigate relationship between modularity and implicit concurrency [36, 35, 25]. We take a module to be a piece of code which can be reasoned about independently based only on the definitions in the module and the external symbols imported into the module. Typically we consider a module to be class or a capsule.

In the Panini language, capsules are the unit of modularity. Systems compose capsule instances together but do not enable any modularity at the system level. One cannot employ composition or encapsulation principles within a system because *all* capsule instances must be declared and wired in a single system. Furthermore, if a system contains multiple copies of a 'sub-system' which has the same wiring connections but uses different capsule instances, then the wiring must be repeated for each 'sub-system' in the main system. The single system problem may lead to large and unmaintainable systems. In short, systems are not reusable and systems cannot be composed of other systems. Finally, as a result of the design decision in Panini 0.9.1 to restrict a program to a single system declaration, programs cannot be designed as a collection of modules, where each module is represented by an independent system connected to other systems.

Returning to the example in the introduction, we can see where the single system concept imposes several limitations on the design of the program. Conceptually, there are three logical elements in the example. First, represented by the `CSVBook` and the `ISUBook` capsules, are the assets to search. Second, represented by the `Search` capsule, are the ways to search, given a set of assets. Finally, represented by the `AddressBook` capsule, are the decisions of which assets to search and what names need to be searched for. Each component in the program is independent from the other components and should be reusable with very little modifications in other programs. However, because the system

`Main` is responsible for declaring and wiring all three facets of the program, the three pieces of the program are coupled together too tightly. Similarly, as was discussed in §1.3 another program cannot reuse the `AddressBook` without recreating all of the capsule instances and wiring statements in the system. These 'copies' of the system also have to be independently maintained. If the original version is changed, e.g. a bug fix is applied, each copy needs to have that change applied manually. A better approach would be to create an `AddressBook` capsule which has a search procedure and internally is able to define how it searches. This design would enable reuse of the `AddressBook` without recreating the wiring, as well as encapsulate the search specifics inside of the `AddressBook`.

The observations about the desire for hierarchical composition and reuse lead to the following requirements for unifying systems and capsules.

1. Allow for hierarchical composition of systems.
2. Provide 'instance' level systems inside of a capsule declaration.
3. Allow capsule procedures to access the capsule instances from the internal system but protect those instances from unrestricted access by external capsules.

Finally, a positive aspect of the idea of a single system in the program design is that it provides both a clear semantics of a complete program as well as the program entry point. A complete program is a system which has correctly wired all of its declared capsule instances and a system is always the entry point for the program when it is executed. Unifying the functionality of capsules and systems leads to two research questions: what does it mean to have a complete capsule-oriented program and where is the entry point of such a program.

2.2 A unified design of capsules

The main contribution of this thesis is a unified design of capsules. Initially, consider the short example of a unified capsule declaration in Figure 2.1. The example in the figure introduces a new element called a *design declaration* on lines 3-7. The capsule design declares a capsule instance named `c3` and wires it to the parameter `other`. Then on line 9 the capsule uses the internal capsule instance,`c3`, by invoking its procedure `proc2` and assigning the result to capsule state `i`.

```
1  capsule C(C2 other) {
2    int i = 0;
3    design {
4      String s;
5      C3 c3;
6      c3(s, other);
7    }
8    void proc() {
9      i = c3.proc2();
10   }
11 }
12 capsule C2() {
13   /* definition elided */
14 }
15 capsule C3(String s, C2 c2) {
16   /* definition elided */
17 }
```

Figure 2.1 A unified capsule declaration

The capsule declaration retains all of the features from the original capsule syntax. It still has a parameter declaration, a state declaration and procedure declaration. The new design declaration takes the place of a system and allows a capsule to declare a set of capsule instances and wire them together to create an 'internal' system. A design is completely encapsulated inside of its capsule declaration and outside capsule instances are not allowed to directly interact with the internal capsule instances of the design declaration. Each instance of a capsule type has its own instance of the design; creating a new capsule instance automatically creates a new copy of the design. Adding the capsule design declaration to the language addresses both the encapsulation and reusability concerns from version 0.9.1 of the Panini language.

Figure 2.2 shows the abstract syntax for unified capsules in Panini. The new syntax has changes to note. First, it no longer supports system declarations. Second, in addition to state and procedure declarations, a capsule can contain an optional element called a *design declaration*. The design declaration allows a capsule to define a set of capsules instances and wiring statements. This 'internal system' provides a capsule with a way to create and connect helper capsule instances. For example, line 9 in Figure 2.1 uses the capsule instance `c3` from the design to assign a new value to `i`. The capsule instances declared in a design declaration *are not* accessible outside of the capsule but they are accessible to any procedure inside of the capsule. The capsule's parameters are accessible in a design declaration, which allows the internal design of a capsule to create wiring statements referencing the parameters. This feature allows a design to wire parameters into the internal system and creates designs which are parameterized on the capsule parameters.

Names in a design declaration have two scopes. Any name with a capsule type is exported outside of the design declaration and is accessible in the other procedures of the capsule. Names which do not have a capsule type are not exported and are only visible inside of the design declaration. For

capsule-decl ::= **capsule** *c* [**implements** signature] capsule-param* capsule-body	"Capsule Declaration"
signature-decl ::= **signature** *s* (*t p* form-arg*)*	"Signature Declaration"
capsule-param ::= formal-arg	"Capsule Parameter"
capsule-body ::= [design-decl] state-decl* proc-decl*	"Capsule Body"
state-decl ::= *t n**	"Capsule State"
proc-decl ::= *t p* form-arg* proc-body	"Capsule Procedure"
formal-arg ::= *t n*	"Formal Argument"
proc-body ::= java-stmt*	"Procedure Body"
design-decl ::= cap-inst-decl* wiring-stmt*	"Design Declaration"
cap-inst-decl ::= *c n**	"Capsule Instance"
wiring-stmt ::= *n*(n*)	"Wiring statement"

where

$c, s \in \mathbb{C}$, $\mathbb{C}$ is the set of valid capsule names
$n \in \mathbb{N}$, $\mathbb{N}$ is the set of valid state names
$p \in \mathbb{P}$, $\mathbb{P}$ is the set of valid procedure names
$t \in \mathbb{T}$, $\mathbb{T}$ is the set of valid type names

Figure 2.2 Unified capsule syntax

example, in Figure 2.1 the name `c3` is accessible in other capsule procedures, but the name `s` is not. The slightly unintuitive set of scoping rules are a design trade-off to keep programmers from confusing state declarations with internal capsule instances and not pollute the namespace inside a capsule with a set of names only useful inside of a design declaration.

2.3 Search revisited

Let us consider how the search example from §1.2.4 can be rewritten in a way that is both easier to understand and reusable. The primary benefit in this decomposition is that each capsule is now a self-contained unit and is dependent only in its declared wiring parameters to function.

Figure 2.3 shows the updated search example using design declarations in both the `AddressBook` and `Main` capsules. The `AddressBook`'s design, lines 18-22 declares what `Books` it will use for the search, as well as a `Search` capsule, to which the `Books` are wired. The `search` procedure now delegates the functionality to the internal `Search` capsule instance instead of requiring a `Search` instance be wired in via a capsule parameter. This change allows the implementation details for the search

```
capsule Search(Book[] books) {
  List<Address> search(String name) {
    Address[] addrs = new Address[books.length];
    // Execute all searches concurrently
    for(int i = 0; i < books.length; i++) {
      addrs[i] = books[i].search(name);
    }
    ArrayList<Address> results
          = new ArrayList<Address>();
    // Filter out 'null' addresses
    for(int i = 0; i < addrs.length; i++) {
      if(addrs[i] != null) { results.add(addrs[i]); }
    }
    return res;
  }
}
```

```
capsule AddressBook() {
  design {
    CSVBook csv; ISUBook isu;
    csv(new File("path/to/file"));
    Search search(new Book[]{csv, isu});
  }
  List<Address> search(String name) {
      return search.search(name);
  }
}
```

```
capsule Main() {
  String[] names = ...; //value elided
  design {
    AddressBook addresses;
  }
  void run() {
    for(String n : names) {
      // Look for the name and print
      // out each address found
      for(Address a : addresses.search(n)
        System.out.println(a);
    }
  }
}
```

Figure 2.3 Search program in Panini 0.9.2

functionality to be completely encapsulated inside of the `AddressBook` declaration. Similarly, the `Main` capsule now only declares an instance of `AddressBook` and just uses it to search for the names. In both cases, the capsule instances declared in the design declarations are limited to the additional capsules needed for that capsule to function.

When the program executes, each capsule's design is evaluated after the capsule instance is initialized but before the capsule's `run` procedure or message handling procedure begins. Execution begins with the `Main` capsule, because it does not have any parameters and defines a `run` method. The `run` procedure in the `AddressBook` capsule from Figure 1.3 is no longer needed; its `run` procedure moved into `Main`. Adding a `run` procedure to the `AddressBook` capsule in Figure 1.3 was an artifact of systems not being able to define procedures or methods and needing at least one of the capsules to be active in order for the program to run. The remaining execution proceeds much as before, with the `search` procedure in the `Search` capsule searching each book concurrently and returning a list of

Addresses when the all of the searches are finished. The AddressBook searches for each name by delegating to the now internal Search capsule instance and returning the result to its caller. Finally, each address is printed out after it has been found.

As an example of how design declarations enable simple reuse of capsule functionality, suppose a new program which generates an address label for each name needs to be written. Reusing the AddressBook to search for each name and the new program only needs to implement the specifics of formatting the address. However, in Panini 0.9.1, reusing the AddressBook wiring *was not possible*. Instead of reusing the wirings declared in the AddressBook, a new system, LabelsMain, is defined. The LabelsMain system must recreate, that is copy, all of the capsule declarations and wiring statements from the AddressBook have the same search capabilities of the AddressBook (Figure 2.4). Conversely, in Figure 2.5 the MailingLabel capsule is able to simply include an AddressBook in its design and use that capsule instance to find the address for the mailing label.

```
1  system MainLabels {
2    CSVBook csv;
3    ISUBook isu;
4    Search search;
5    AddressBook addresses;
6    MailingLabels mailer;
7    csv(new File("path/to/ file ") ;
8    search(new Book[]{csv, isu});
9    address(search);
10   mailer(addresses);
11 }
12 capsule MailingLabels(AddressBook addressBook) {
13   void run() {
14     for (String n : names) {
15       // Assume the first entry is the correct one.
16       List<Address> addrs = addressBook.search(n);
17       makeLabel(addrs.get(0));
18     }
19   }
20   private void makeLabel(Address addr) { ... }
21   }
22 }
```

Figure 2.4 Recreating the AddressBook wirings

This section has presented an example of how the unified capsule design leads to more understandable and reusable Panini programs. The search example from §1.2.4 is simpler to understand because each capsule declaration is able to encapsulate its internal helper capsule instances and wiring statements. Reusability is a natural consequence of the unified design and encapsulated wiring statements.

```
1  capsule MailingLabels {
2    String [] names = ...;
3    design {
4      AddressBook addressBook;
5    }
6    void run() {
7      for (String n : names) {
8        // Assume the first entry is the correct one.
9        List<Address> addrs = addressBook.search(n);
10       makeLabel(addrs.get(0));
11     }
12   }
13   private void makeLabel(Address addr) { ... }
14 }
```

Figure 2.5 Reusing the `AddressBook` capsule

2.4 Compiling the unified design

The compiler for the Panini language is implemented as an extension to the standard Java compiler. Each Panini element has a translation strategy to a corresponding Java element which is then further compiled into bytecode. The approach enables the Panini compiler to extend the Java language and its semantics, while taking advantage of the existing logic to compile Java elements into bytecode. The unified design for capsules in Panini version 0.9.2 required several changes to the original translation strategy. This section presents the original strategy, discusses the changes needed for the unified design and demonstrates the new translation strategy.

2.4.1 Translating Panini 0.9.1 to Java

Translating both capsules and systems into regular Java requires an approach which converts the Panini elements into standard Java compilation units, classes and interfaces. Figure 2.6 shows a simple capsule declaration, its corresponding interface, and the thread based concrete implementation of the capsule.

Translating a capsule declaration. Translating a capsule begins by extracting an interface for the capsule type from the public procedures. The interface is given the same name as the capsule and it is used to represent the capsule type to Java. Each capsule type then has multiple concrete implementations, based on the concurrency model (either serial, task-based, or thread-based) needed for each instance of the capsule. Each version implements the interface for its capsule type.

```
capsule C(C1 p1, C2 p2) {
  R1 f1() { ... }
  R2 f2(int i) { ... }
}
```

```
interface C{
  R1 f1();
  R2 f2(int i);
}
class C$Thread extends Panini$Thread implements C {
  C1 p1; C2 p2;
  R1 f1() { push(new R1$f1$Duck()); }
  R2 f2(int i) { push(new R2$f2$Duck(i); }
  R1 f1$original(R1$f1$Duck d) {
    R1 value = /* compute value */
    d.finish(value);
  }
  R2 f2$original(R2$f2$Duck d, int i) {
    R2 value = /* compute value */
    d.finish(value);
  }
  public void run() {
    //get next message from queue
    //switch on message id to wrapper function
    //continue processing messages until shutdown
  }
}
```

Figure 2.6 A Capsule and its translation to Java

If the capsule is not active, that is if it does not define a `run` procedure, a `run` procedure which processes the capsule's message queue is installed in the translated capsule's concrete implementations. The message queue is processed by removing a message object from the queue, using the stored originating method id to chose which method to call and calling the translated version of the original method with the message object as the argument. Otherwise, if the capsule is active, the defined `run` procedure from the capsule definition is copied into the concrete implementation. Active capsules do not process a message queue and should not have asynchronous messages sent to them.

Wiring parameters are translated by creating a field in each class representing a capsule. The field has the same type and name as the wiring parameter.

Capsule procedures are translated by creating two separate versions of the procedure. The version with the same signature as the capsule's interface has logic installed to create a new message object, push that object into the capsule's message queue and return the object to the caller immediately. The message object serves multiple purposes. First, it records which method it was produced by. This allows the messaging processing loop to know which method needs to be called to compute the real value of the method. Second, it captures the values of the method's arguments for later access when the message is processed. Finally, it acts as the future-style value, which allows the actual value of the procedure to be communicated back to the caller after the message is processed.

```
class C {
  C2 m(int i) { ... }
}
```

```
interface C { C2 m2(int i); }
class C$Duck extends DuckFuture implements C {
    C value; int msgid;
    C$Duck(int msgid) {
      this.msgid = msgid;
    }
    public void finish(C value) {
      this.value = value;
      // signal finished
    }
    public C2 m(int i) {
        blockUntilFinished();
        return value.m(i);
    }
}
```

Figure 2.7 A simple class and its corresponding duck future

The original implementation of the method is copied into a new method named for the old method and appended with “$Original". The new copy of the original procedure implementation has a single argument, which is the message object. The method first extracts the arguments to local variables and then proceeds to to execute the original method logic and compute the return value for the method. Once the return value is computed, it is stored in the message and the caller is able to access the return value for the method through the message object.

Duck Futures. Panini capsule instances communicate by sending messages to each other. When a capsule receives a message, it immediately pushes a new object into its message queue which represents both the method to call when the message is processed, as well as a future for the return value of the message. The reference to the future object is returned to caller immediately. We call this concept a ‘Duck Future’. The term ‘Duck Future’ is a play on the concept of duck-typing. A complete discussion of duck futures is beyond the scope of thesis, but it is presented by Lin *et al.* [22].

The compiler automatically creates a duck future class for each capsule procedure’s return type by first extracting an interface from the public methods for the return type and then generating a new class which implements the interface and extends the `DuckFuture` class from the Panini runtime. Because the new class implements the same interface as the expected return type, the client remains completely oblivious to the changes. Figure 2.7 shows a simple class and its corresponding duck future.

Translating a system to Java Translating a system to Java is much simpler than translating a capsule to Java. A system is the sole entry point for a Panini program and it is translated into a single class with the `public static void main(String[])` method needed to make the class executable in a JVM. Figure 2.8 shows a basic system and its corresponding translation into Java.

```
1  capsule C1 (C2 param1) { ... }
2  capsule C2 { ... }
3  system S {
4    C1 c1; C2 c2;
5    c1(c2);
6  }
```

```
1  class S {
2    public static void main(String[] args) {
3      C1 c1 = new C1();
4      C2 c2 = new C2();
5      // Wire the capsule parameter in C1 called param1 to
           c2
6      c1.param1 = c2;
7      c2.run();
8      c1.start();
9      c2.join();
10   }
11 }
```

Figure 2.8 A system and its corresponding class.

Specifically, each capsule declaration in the system is translated to a variable declaration and instantiation in the main method. Capsules always have a 0 argument constructor, which means any capsule type `C` can be instantiated with '`new C();`' All capsule variable declarations and instantiations are added to the main method before the wiring statements. Wiring statements become field assignments on the wired capsule instance. The wiring parameters define the names of the fields assigned in the wired capsule. For example, the wiring statement `c(w1, w2);` becomes `c.p1 = w1; c.p2 = w2;`, assuming that the type of `c` is a capsule with parameters called `p1` and `p2`.

If a system has n capsule declarations, then the first $n - 1$ instances will have their `run` method called, allowing each capsule instance to begin executing in its own thread. The last capsule instance has its `start` method called, which has the effect of reusing the main original program thread for one of the capsule instance threads. In order to keep the system from terminating when a capsule is still running, the first $n - 1$ capsule threads are joined. The system will terminate when all the threads terminate.

2.4.2 Compiling unified capsules

The translation strategy for the unified design must account for the responsibilities which were previously incorporated into the system translation strategy. The first issue which must be addressed

is when are the capsule instances declared in design declaration instantiated and wired. The second issue is what capsules are executable, that is, need a `public static void main(String[])` method installed in the capsule body.

Translating design declarations Translating a design declaration has two parts. First, each capsule instance declaration inside the design is converted into a field declaration of the translated class. This approach allows us to export the names with a capsule type outside of design declaration and makes them available inside the other capsule procedures. Next, the design declaration is converted into a method called `panini$wirecapsule()`. This method instantiates each capsule instance in the design declaration, performs the required wiring operations and starts the capsule instances declared inside of the design declaration.

The second issue translating a design declaration must address is when to call the method that wires capsules together. In order for the capsule to function correctly, the method must be called before the body of the capsule's run method is executed. However, since the capsule parameters may be wired into the capsule instances defined in the design declaration, the method cannot be called before the capsule itself has been instantiated. For these reasons, the call to the wiring method is placed at the top of the capsule's `run` method (either the generated version, or installed into the programmer defined `run` method.) Figure 2.9 shows a basic capsule definition with a design declaration and its translation to a pure Java class and Figure 2.10 shows a capsule with a programmer defined `run` method in its translation.

Panini 0.9.2 also introduced a garbage collection algorithm to automatically shutdown capsule instances. The idea of the algorithm is that at compile time, the number of times a capsule instance is wired into another capsule instance is assigned. When a run method terminates, it calls a 'disconnect' method for each wired capsule instance. Finally, when a capsule's connected count reaches 0, the capsule instance shutdown and calls the disconnect method for each of its wired capsule instances. The garbage collection algorithm and implementation is under active research as of this writing.

Which capsules are executable? In Panini 0.9.1, any system declaration is runnable. A system is translated into a regular class with all of the capsule instance declarations and wiring statements in

```
capsule C(C2 c2) {
  design {
    C3 c3;
    int i = 5;
    c3(c2, 5);
  }
}
```

```
class C extends Panini$Thread{
  C2 c2;
  C3 other;
  private void panini$wirecapsule() {
    other = new C3;
    int i = 5;
    c3.c2 = c2;
    c3.i = 5;
    c3.start();
  }
  public void run() {
    panini$wirecapsule();
    try {
      // message processing
    } finally {
      c2.panini$disconnect();
      c3.panini$disconnect();
    }
  }
}
```

Figure 2.9 Translating a unified capsule to Java

the system are moved in the `main` method of the class. Without the clear entry point that a system declaration provides, semantics must be defined which determine what capsules are runnable.

In Java, any class with a method called `main` that is both `public` and `static` and which has a single argument of type `String[]` is a valid program entry point. Similarly, any executable capsule must be able to have a `main` method automatically installed. The language does not support static methods inside of a capsule declaration, nor does it allow programmers to create capsule instances. The element which makes a class executable is a well defined entry point, which at most relies on unknown values from the command line arguments. The idea is that executable classes do not rely on 'external' values.

The definition of which capsules should be runnable begins with the definition of a *closed* capsule. We say a capsule is *closed* if its parameter list either empty or has a single parameter of type `String[]`. Next, we need to consider what else is needed for a capsule to be runnable.

Capsules may optionally define a `run` procedure. `run` procedure has the same semantics of the Java method of same name in the `Thread` class. If a capsule does not define a `run` procedure, the Panini compiler generates one automatically for the capsule. All capsules begin executing their `run` procedure once they are initialized.

Automatically generated `run` methods are used to process the message queue for the capsule instance; once the capsule instance finishes its initialization, the `run` procedure will block indefinitely,

```
capsule C(C2 c2) {
  design {
    C3 c3;
    int i = 5;
    c3(c2, 5);
  }
  void run() {
    c3.foo();
  }
}
```

```
class C extends Panini$Thread{
  C2 c2;
  C3 other;
  private void panini$wirecapsule() {
    other = new C3;
    int i = 5;
    c3.c2 = c2;
    c3.i = 5;
    c3.start();
  }
  public void run() {
    panini$wirecapsule();
    try {
      c3.foo();
    } finally {
      c2.panini$disconnect();
      c3.panini$disconnect();
    }
  }
}
```

Figure 2.10 Translating an active unified capsule to Java

waiting for a message to process. This implementation detail means some care must be given to 'when' a closed capsule is executable. Specifically, if a closed capsule does not define its own run procedure, and is not connected to any of its internal capsule instances, such that capsules can send the current capsule a message, it does not make sense to allow that closed capsule to run. Such a capsule will ALWAYS appear to hang as it waits to process a message which will never arrive.

A runnable capsule is a closed capsule which has either a user defined run method, or which wires a reference to itself into at least of its internal capsule instances. In the translation strategy for unified capsules, any runnable capsule has a `main` method installed which can be used to execute the capsule type. Figure 2.11 shows an example of basic closed capsule and its corresponding Java class definition.

```
capsule C(String[] args) {
  void run() {...}
}
```

```
class C extends Panini$Thread{
  String[] args; // The 'args' parameter
  public void run() {...}
  public static void main(String[] args) {
    C c = new C();
    c.args = args;
    c.run();
  }
}
```

Figure 2.11 A closed capsule and its translation

2.5 Graph generation

The last step in the compilation process is constructing a complete program call-graph. The call-graph represents which capsule instances are able to send each other messages. In Panini 0.9.1 construction the graph has two distinct phases. When capsule declarations are compiled, a *template* for just the calls made within the capsule declaration is created. Copies of the templates will be connected together to form the complete graph when a system is compiled. The unified design in Panini 0.9.2 makes graph construction more complicated. We will still create a template for each capsule declaration, but create a final graph requires the ability to connect nested capsule instance creating and wiring §2.5.1 discusses how system-graphs were created in the old version of Panini and §2.5.2 discusses the necessary changes to maintain modular graph construction with the unified design.

2.5.1 System graph construction

The system-graph for Panini 0.9.1 is computed in two parts. First, a call-graph *template* for the capsule declaration is created when it is compiled . The graph is considered to be a template because it will be copied and further manipulated when a system declaration is compiled. The capsule's call-graph template is associated with the capsule's type symbol, allowing later stages of the compiler to look up the template anywhere the capsule type is available. Second, when a system is processed, a copy of the template for each capsule instance is created. The graph instances are connected together according to the wiring statements in the system. This gives us a complete call-graph for the messages sent between all of the capsule instances in the system.

Constructing capsule declaration templates. We will use the simple program in Figure 2.12 as a working example for constructing both the capsule declaration templates and for connecting those templates together in a system. The program defines three simple capsule types, `C`, `D` and `E` and a system which connects instances of the types together.

The vertices in the graph for the capsule declaration template represent the procedures in the capsule declaration. Each vertex is named for the procedure it represents. The names of vertices for procedures accessed via a wiring parameters are prefixed with the name of the wiring parameter. This naming strategy gives us a way to connect inter-capsule procedure calls when we construct the full system-

```
capsule C(D cw) {
  void c1(){
    c2();
  }
  void c2() {
    cw.d1();
  }
}
```

```
capsule D(E dw) {
  void d1(){
    dw.e1()
  }
}
```

```
capsule E() {
  void e1() {
    // Empty for simplicity
  }
}
```

```
system S() {
  C c; D d; E e;
  c(d);
  d(e);
}
```

Figure 2.12 A simple program to demonstrate system-graph construction

graph. Each directed edge in the graph represents a call from the source procedure to the destination procedure. Constructing the graph template for each capsule declaration is *modular*, since we can construct the templates with just the capsule declaration and the types of the wiring parameters.

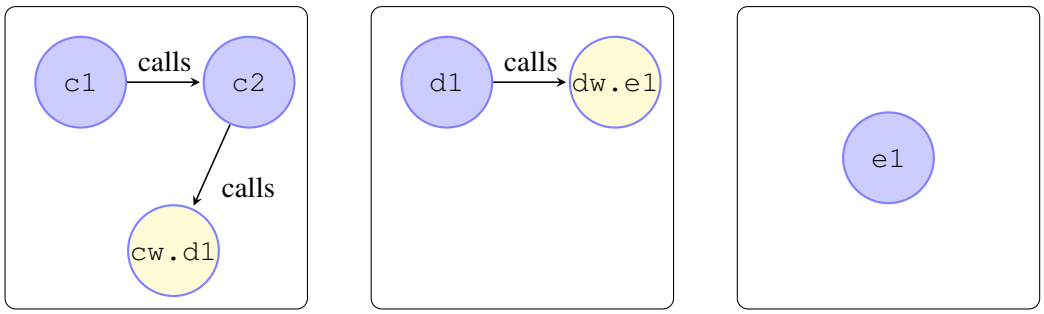


Figure 2.13 Capsule declaration graph templates

Figure 2.13 shows the graph templates for the capsule declarations in Figure 2.12. The vertex for a procedure defined in the capsule declaration is filled with blue. The vertex for a procedure defined in one of the capsule declaration's wiring parameters is filled with yellow. For example, `C`'s template contains the vertices `c1`, `c2`, `cw.d1`, and edges `c1->c2`, `c2->cw.d1`. The template for `E` contains a single vertex and no edges, since `E` does not have any wiring parameters and does not call any procedures.

Connecting the graph templates in a system declaration. The system-graph is simple to construct in Panini 0.9.1. The *single* system declaration allows us to the construct the complete system

graph after all of the capsule instances for the entire program are known. The final system-graph is similar to the template for each capsule declaration. However, instead of representing the call-graph, we now represent which capsule instances are connected to each other, and thus capable of sending each other messages.

Recall from Figure 1.2 that there are two types of statements in a system declaration. Capsule instance declarations create new instances of a capsule type. Wiring statements connect the capsule instances together. With respect to the system-graph, each capsule instance declaration creates a new vertex in the graph. Each wiring statement creates an edge in the graph from the capsule instance being wired to the capsule instances that are the wiring statement arguments. For example, the wiring statement `c(d);` creates an edge from vertex `c` to vertex `d`. The basic system-graph for Figure 2.12 is shown in Figure 2.14.

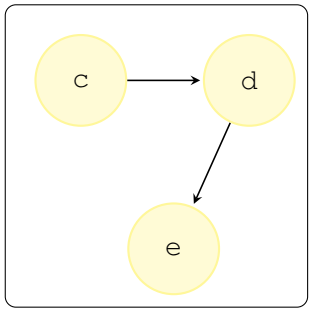


Figure 2.14 Basic system-graph for Figure 2.12 (Panini 0.9.1)

Further, each capsule instance in the graph includes a copy of the call graph template from the capsule instance's type. Once the basic system-graph is in place, we can add an instance of each call graph template to the graph. At this point, the vertices representing calls to wired capsule instances are still in terms of the names used in the capsule declaration's wiring parameters. Substituting the names of capsule instance from the wiring statements into the call graphs allows us to get the call graphs in terms of the *actual* capsule instance names in the system. Finally, we create edges from the wiring-parameter vertices to the vertices representing the procedure that will be executed when the program runs. Figure 2.15 shows the final system-graph for our example. Notice vertex `cw.d1` has been renamed to `d.d1` and `dw.e1` is now `e.e1`. The renamings are the result of substituting the names of the wired capsule instances into the call graphs.

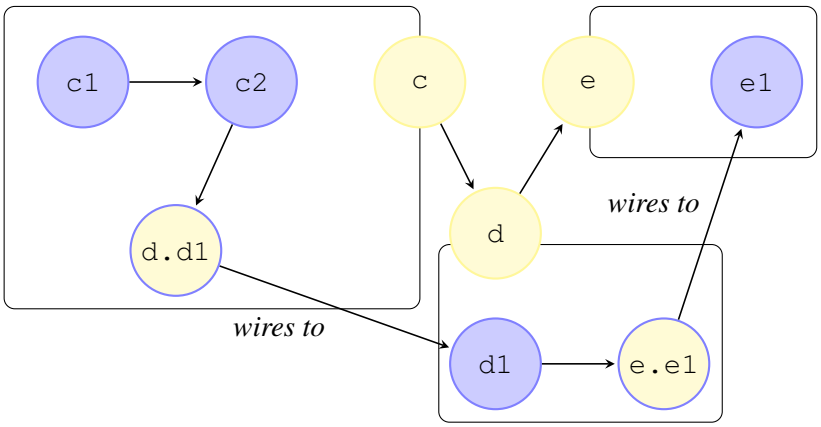


Figure 2.15 System graph for Figure 2.12

Constructing a complete system-graph for Panini 0.9.1 programs is a relatively simple process. The template for each capsule type can be constructed from just the capsule's declaration and the types of the wiring parameters. The *single* system means a complete graph is only ever constructed after all the capsule instances for the program are known, which in turn means we know exactly which capsule instances are wired to other capsule instances, which then gives us the ability to statically and accurately know what the target of any procedure call will be.

2.5.2 System graphs for the unified design

The unified design introduces several challenges for constructing a full system graph. First, all of the capsule instances and wiring statements are not in a single place. Rather they are scattered throughout all the capsule declarations. Second, a capsule instance may 'flow' through many other capsule instances via the wiring parameters. Finally, we no longer have a single system. Instead, any closed capsule may be the entry point for a Panini program.

The single system in Panini 0.9.1 makes for a simple algorithm to construct the complete system graph, because all the capsule instances and wiring statements are known when the system is processed. The nested nature of design blocks and wiring statements in the unified design requires a recursive graph construction approach. The updated algorithm for constructing a system graph is also a two step process (constructing a template for each capsule declaration and later connecting instances of those templates together) from Panini 0.9.1. However, creating the final system graph is not as simple. Instead of

simply ‘connecting’ all of the capsule instances via the wiring statements, the process must ‘interpret’ the capsule instances to create the final graph.

2.5.2.1 Design templates

The capsule declaration template from §2.5.1 must be extended to record the capsule wiring parameters, the capsule instance declarations, and the wiring statements. The extra information will be used later to construct a full system graph for a program. We will call this extended capsule declaration template the *design template*.

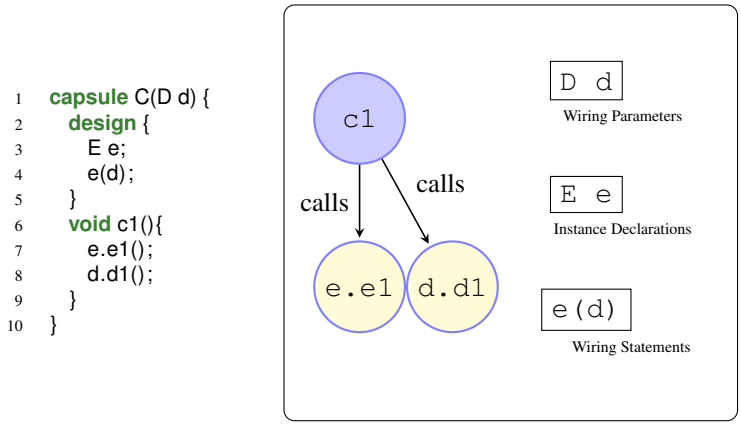


Figure 2.16 A capsule declaration and its design template

Figure 2.16 shows a basic capsule declaration and its corresponding design template. The design template contains the call graph, as well as sections recording the wiring parameter `d`, the instance declaration `e` and the wiring statement `e(d)`. Note the internal instance declaration is wired to an *external* wiring parameter. Connecting an internal instance to a wiring parameter is valid, and must be supported by the updated system graph generation.

Constructing a design template remains a modular process. As with the graph templates in §2.5.1 the process only uses the names and types that are referenced in the capsule declaration. This step does not attempt to instantiate the design templates for the capsule instances created in the design. Attempting to connect the current design template’s call graph to instances of internal capsule instance design graphs would break the modularity. As an example, consider the design template for capsule `C` in Figure 2.16. Suppose we wanted to connect to the call graph from `C`’s design template to the instance of `e`’s design template. The design template is accessible from `e`’s type `E`. But creating an instance of

`E`'s template would also require instances of the design templates for any capsule instances declared in `E`. Instead, a second step will instantiate and connect all of the designs templates.

2.5.2.2 Instance Graphs

The second phase in the system graph construction for the unified design is to construct a 'complete' graph, with respect to some closed capsule type `T`. An instance of `T` will serve as the entry point for the program. `T` acts like a system from Panini 0.9.1. The main difference is that `T` does not explicitly define all the capsule instances in the program. An instance graph is derived from the design graph of a capsule instance's type. Instance graphs have three types of vertices. The vertices represent either *procedures*, *wiring parameters* or other *instance graphs*.

The algorithm for constructing the system graph for the unified design *interprets* the design template for each capsule instance to eventually create a complete graph. Each element in the design template (call graph, wiring parameters, instance declarations and wiring statements) has a specific interpretation and the elements are interpreted in a specific order. Interpretation starts with the closed capsule `T`. Interpretation requires the design templates for *all* the capsule types in the program, and is not modular. However, since the system graph is supposed to enable a whole program analysis of how messages will flow in the program, this is not a drawback. The approach mirrors the way the Java [13] compiler and Java runtime interact. The compiler supports modular and incremental compilation, but the runtime must have *all* the compiled classes available to execute the complete program. The rules to interpret a design template are given in Table 2.1.

Table 2.1 Design template interpretation rules

Call Graph	Copy all vertices and edges into the instance graph
Wiring Parameter	Add a new vertex for the wiring parameter
Instance Declaration	Add the instance graph for capsule instance's type
Wiring Statement	Add an edge from the 'wired' instance graph wiring parameter vertex to the corresponding vertex for the wiring argument

We will continue the example from §2.5.2.1 to illustrate constructing a complete system graph for the unified design. Figure 2.17 shows the capsule declarations for our example. Capsule `P` is the entry point for program execution. The example adds simple capsule declarations for types `D` and `E`.

```
1  capsule C(D d) {
2    design {
3      E e;
4      e(d);
5    }
6    void c1(){
7      e.e1();
8      d.d1();
9    }
10 }
```

```
1  capsule D(C c) {
2    void d1() {
3      c.c1();
4    }
5  }
6  capsule E(D d) {
7    void e1() {
8      d.d1();
9    }
10 }
```

```
1  capsule P() {
2      design {
3        C pc;
4        D pd;
5        pc(pd);
6        pd(pc);
7      }
8      void run() {
9        pc.c1();
10     }
11 }
```

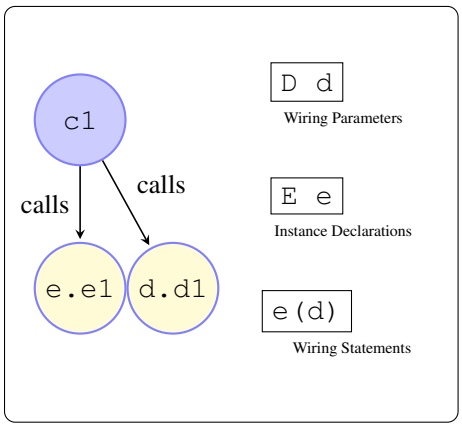
Figure 2.17 Capsule declarations for the design graph construction example

The closed capsule `P` creates an instances of `C` and `D`, wires them together and calls procedure `c1` in its run method. The corresponding design templates for the capsule declarations in Figure 2.17 are show in Figure 2.18. These design templates will be the basis of our interpretation.

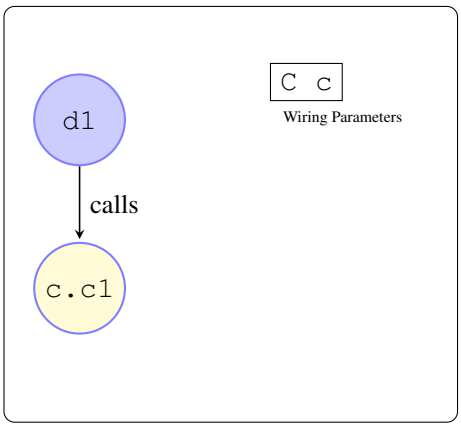
Since the rules for interpreting a design template are recursive, interpreting a design template dt into an instance graph implies the interpretation of the design templates into instance graphs for any for any capsule instance declarations contained within dt. Thus, if we begin the interpretation with the design template for `P`, when we have finished interpreting `P`'s design template we will have a complete graph for the entire program.

Interpreting `P`'s design template into an instance graph has three steps. First we copy the call graph from the design template into the instance graph. Then we create instance graphs for the capsule instances `pc` and `pd`. Finally, we add an edge from the node representing the wiring parameter `d` in `pc`'s instance graph to the instance graph for `pd` and an edge from wiring parameter `c` in `pd`'s instance graph to the instance graph for `pc`.

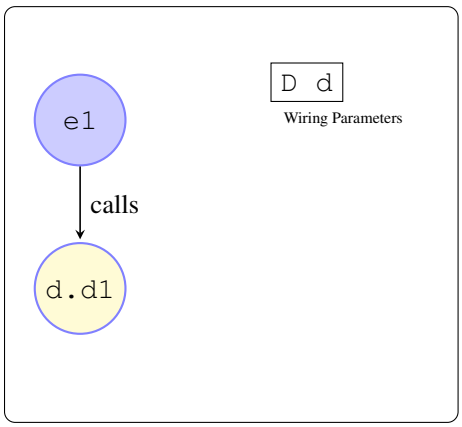
Figure 2.19 shows the final instance graph for our example program. The graph for `P` contains an instance graph for both capsule instances `pc` and `pd`. The instance graph for `pc` then contains an instance graph for `e`, since `C`'s capsule declaration creates a new instance of `E` inside its design. Finally, the capsule instances are connected through the wiring parameters and wiring statements. The wiring parameter `d` in `e`'s instance graph connects to the wiring parameter `d` in `pc`'s instance graph, which in turn connects to the instance graph for `pd`. Following the connection through the wiring parameters allows us to determine which capsule instance will receive the procedure call. For example, the call `d.d1()` in the `e` instance graph will resolve to capsule instance `pd` created in the design of `P`.



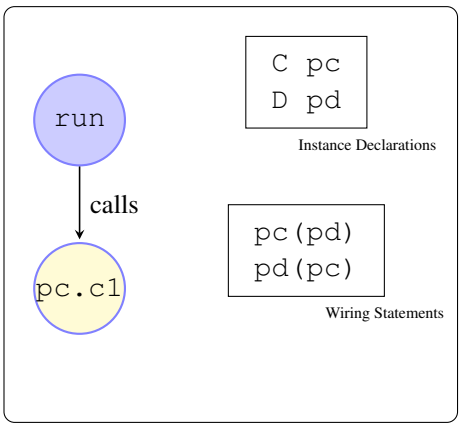
C's design template



D's design template



E's design template



P's design template

Figure 2.18 Design templates for Figure 2.17

The updated algorithm for creating system graphs for the unified design preserves the characteristics of algorithm from Panini 0.9.1. First, it remains modular. The design template for a capsule declaration is constructed using only the procedure names declared in the capsule and the other capsule types that declaration depends on. Next, it is capable of constructing a full graph when a single closed capsule is chosen as the starting point. Choosing a different closed capsule as the entry point will result in a different graph, but it will also result in a different program. Finally, the algorithm will terminate, as long as there no recursive designs. This is not a problem, since it will also lead to programs that never fully create all of their capsule instances. A simple extension to the design template interpretation would be able to detect the loop and warn a programmer of the situation.

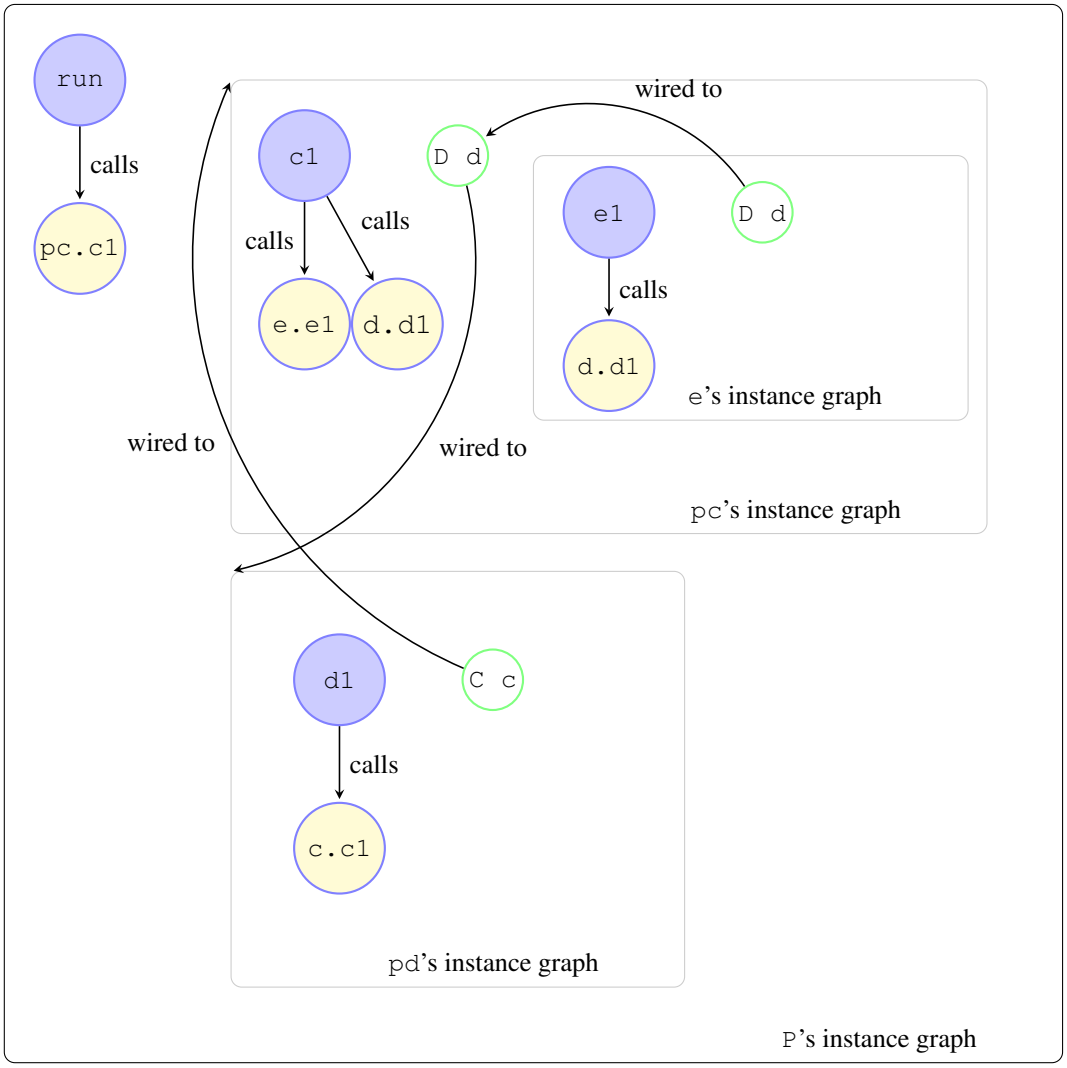


Figure 2.19 Complete instance graph for Figure 2.17

2.6 Conclusion

This section has described the results of unifying system and capsule declarations by adding a design declaration to the capsule declaration. Adding a design enables capsule declarations to be composed together in a hierarchical fashion, allows small 'sub-systems' to be reused and generally makes a Panini program easier to understand.

CHAPTER 3. CASE STUDY: BEAMFORMER

The next two chapters will each present an example of how the unified capsule design leads to more modular, reusable, and easier to understand programs. To that end, we first present an overview of the StreamIt benchmarks [45, 47, 14] and the general strategy for adapting their programs to a capsule oriented programming style. The case studies will follow a common format: First, the description and logical view of the benchmark will be presented, followed by a Panini 0.9.1 implementation and a discussion of the problems the implementation has because of the separation between systems and capsules. Each study will conclude with the unified capsule implementation and a discussion of how the changes in Panini 0.9.2 affected the design of the program and simplified the overall implementation. The discussion will focus on how the connections between capsule instances are created and generally will omit the implementations of the capsule bodies. Both of the examples are adapted from the StreamIt [45, 47, 14] benchmarks.

3.1 StreamIt language overview

The StreamIt language is designed to exploit the natural parallelism of stream processing applications. These applications exhibit a high degree of data parallelism and can split input across multiple processing units which operate independently until their output in merged together. Programs are generally composed of groups of connected stages. Common elements in stream type programs include splitters, which split an input stream to other elements, filters, which transform a stream, and joiners which merge multiple streams back into a single stream. StreamIt takes advantage of the regular and repetitious nature of these type of programs to define specific constructs at the language level which, when combined with specific compilation techniques, produce highly parallel code. Examples of this type of application include digital signal processing and cryptography.

Investigating how stream processing style applications fits into the capsule-oriented model is currently ongoing research. Many of the challenges encountered while adapting the StreamIt benchmarks [14] to Panini served as the initial motivation for the unified capsule design.

The basic approach for converting a StreamIt program to Panini, shown in Figure 3.1, is to first create a `Stage` signature which represents a single stage in any processing pipeline. The `Stage` defines a single procedure, `consume`, which causes the `Stage` to consume its input. Output is passed onto to the next `Stage` by callings its consume method. The signature `Stage`, lines 1-3, defines a processing stage. A capsule type to be linked into the stream must implement this signature. The `S1` capsule, lines 4-12, defines a capsule which is connected to another `Stage` called `next`. `S1` consumes its input at line 5 and starts the next stage of the stream at by calling next.consume(output); at line 9.

```
1  signature Stage {
2      void consume(int[] input);
3  }
4  capsule S1 (Stage next) implements Stage {
5      void consume(int[] input) {
6          //process input to output
7          int [] output = transform(input);
8          // tell next to consume the output as input
9          next.consume(output);
10     }
11     private int [] transform(int [] input){ ... }
12 }
```

Figure 3.1 A basic pipeline in Panini

3.2 Beamformer overview

The first case study we present is an example of digital signal processing (DSP) called 'BeamForming'. BeamForming uses an array of sensors to provide spatial filtering, such as detecting where a signal originates [50]. The BeamFormer example is composed of two primary stages. The first stage uses a group of 'Finite Impulse Response'(FIR) filters to delay each input channel by a different amount. The FIR stage is fed by a splitter which replicates the initial input stream to each FIR filter. There are 12 individual FIR filters. The second stage collects the FIR inputs into a set of 'beams' which are used to detect any location where the signal exceeds a given threshold.

Figure 3.2 gives a visual representation of the program. The initial input is passed through a splitter which splits the input out to 12 'Stage1' FIR filter chains. Each FIR chain is connected to a 'joiner'

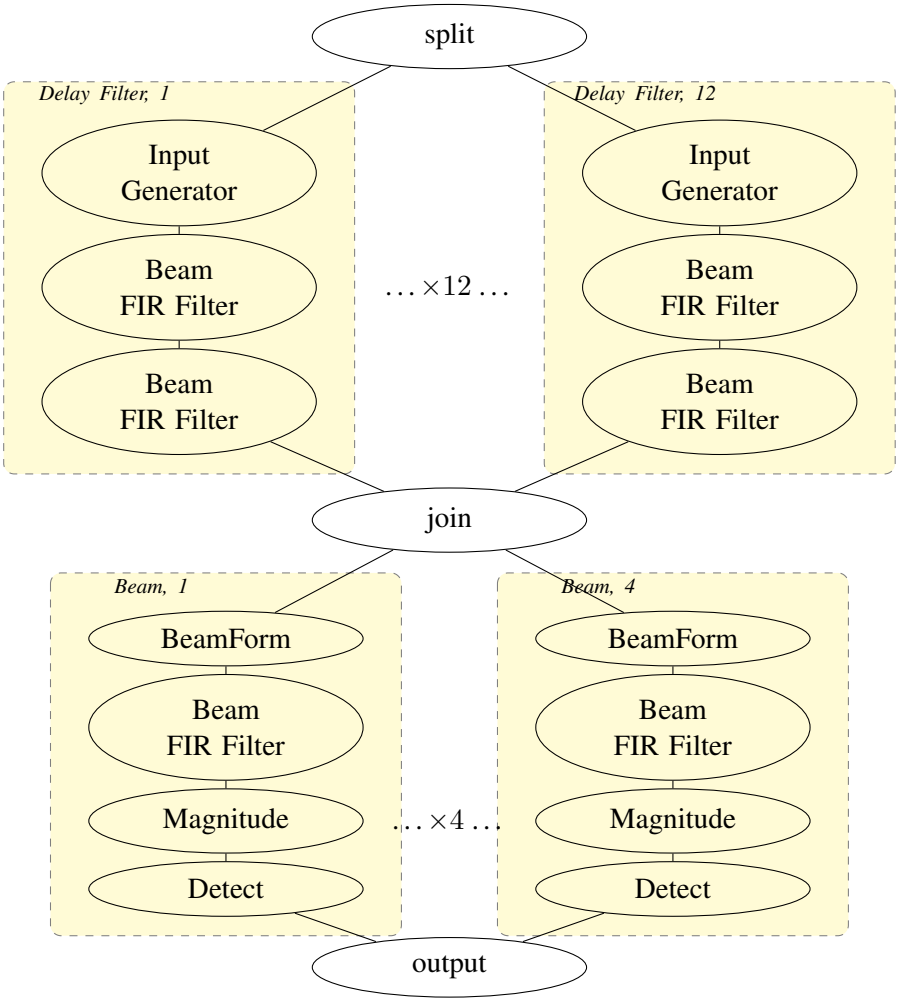


Figure 3.2 BeamFormer stages

which combines the FIR outputs before splitting them across four 'Stage2' detector chains. At the end of the pipeline, the detector outputs are joined together and passed to output filter which stores the output for later use.

3.3 Initial design

Figure 3.3 shows the construction of the BeamFormer program with a single system in Panini 0.9.1. Each filter element is modeled as a capsule implementing the `Stage` interface. All capsule instances are defined in the system `BeamTransformer`. FIR filter elements are declared on lines 5-8 and wired together on lines 19-21; detector elements are declared on lines 11-15 and wired together on lines 25-28.

Two new operators, **wireall** and **associate** are syntactic sugars to simplify the presentation. The **wireall** operator wires each index of the first argument to the remaining arguments. **wireall**(cs, a1, a2); is equivalent to **for**(**int** i = 0; i<cs.length; i++){cs[i](a1, s2);} . The **associate** operator is used to wire each element in one capsule array to another element in a second capsule array. The statement **associate**(cs, start1, ds, start2, length); is equivalent to **for**(**int** i = 0; i<length; i++) {cs[i+start1](ds[i+start2])}. The **associate** operator is syntactically the same as the Java method System.arraycopy.

Legend Stage1 FIR Stage2 Detector

```
1  /** The BeamFormer filter, implemented as a single
       system. */
2  system BeamFormer{
3    // Capsule Instance Declarations
4    Splitter splitter ;
5    FIRGroup firGroup[12];
6    InputGenerate inputGen[12];
7    CoarseBeamFirFilter coarseBeamFirFilter[12];
8    BeamFirFilter beamFirFilter1[12];
9    Joiner joiner ;
10   Splitter2 splitter2 ;
11   DetectorGroup detectorGroup[4];
12   BeamForm beamForm[4];
13   BeamFirFilter beamFirFilter2[4];
14   Magnitude magnitude[4];
15   Detector detector[4];
16   FileWriter fileWriter ;
17   // Wiring statements
18   splitter (firGroup);
19   associate(firGroup, inputGen);
20   associate(inputGen, 0, coarseBeamFirFilter, 0, 12);
21   associate(coarseBeamFilter, 0 beamFirFilter, 0, 12);
22   wireall(beamFirFilter, joiner);
23   joiner( splitter2 );
24   splitter2 (numBeams, detectorGroup);
25   associate(detectorGroup, 0, beamForm, 0, 4);
26   associate(beamForm, 0, beamFirFilter2, 0, 4);
27   associate(beamFirFilter2, 0, magnitude, 0, 4);
28   associate(magnitude, 0, detector, 0, 4);
30   wireall(detector, fileWriter );
31 } // End BeamFormer System
```

Figure 3.3 BeamFormer, with a single system

The logical design of the BeamFormer benchmark uses twelve FIR filter chains to process the initial input and four detector filters chains to finish the processing. Intermediate filters take care of splitting and joining the data streams.

3.4 Limitations

There are two primary, and related, complications with implementing the BeamFilter as a single Panini 0.9.1 system declaration. First, without the benefit of the coloring in Figure 3.3, it is difficult to correctly determine which declarations belong to which filter chain. The internal components of the FIR and detector filters are exposed in the `BeamFormer` system. It is as difficult to even notice there are two distinct filter chains – the FIR chains and the Detector chains – in the system. One must understand how all the capsule instances are connected before being able to identify the FIR and Detector filters.

The second problem with this program is that a 'template' for a single composite filter cannot be defined and instantiated multiple times. Each copy of the filter must be manually wired. Remember, in Panini 0.9.1, there is a *single* system, and *all* capsule instances must be declared and wired in that system.

Both of the implementation problems stem from the inability to define the logical filter chains as a single, composite capsule type. The capsule definition should be able to internally define its capsule instances and how those instances are connected to each other. For example, there are four distinct capsule types which must be connected in order to create an FIR filter. Also, since the program needs twelve separate FIR filters, the `BeamFormer` system must create an array of each capsule type in the FIR filter and wire the correct indexes together. Fortunately, the wiring pattern is regular and the system is able to take advantage of the **associate** operator to simplify the wiring process.

In this example, the common indexing (e.g. inputGen[n] wires to coarseBeamFirFilter[n]) allows us to use the **associate** operator to simplify the task. A more complex wiring scheme would require many more wiring statements. Like the FIR filter, the detector has multiple copies and the individual detectors are created by associating arrays elements of the capsule array instances from the individual filter types which make up the detector. In both cases, the system cannot take advantage of the fact that the program is composed of multiple copies of the same sub-system and each copy must be declared and wired.

3.5 Redesign

Figure 3.4 shows how the BeamFormer program can be redesigned modularly in Panini 0.9.2 to take advantage of the unified design. The `BeamFormer` capsule on lines 1-11 only defines instances for the Input elements, the FIR and Detector filter groups and the final output element. The capsule `FIRFilter` and the capsule `DetectorChain` define a 'single' FIR and detector filter chain, respectively. The individual filter chains are grouped by the `FIRGroup` and `DetectorChainGroup` capsules.

Legend Stage1 FIR Stage2 Detector

```
1  capsule BeamFormer {
2    design {
3      InputGeneration inputs[12];
4      FIRGroup firs;
5      DetectorChainGroup detects;
6      FileWriter fileWriter ;
7      firs (inputs, detects);
8      detects( fileWriter );
9    }
10    ...
11 }
32 capsule FIRFilter(InputGeneration inputGen, Stage next)
33     implements Stage {
34   design {
35     CoarseBeamFirFilter coarseBeamFirFilter;
36     BeamFirFilter beamFirFilter;
37     inputGen(coarseBeamFirFilter);
38     coarseBeamFirFilter(beamFirFilter);
39     beamFirFilter(next);
40   }
41    ...
42 }
43 capsule FIRGroup(InputGeneration[] inputGen, Stage next)
44     implements Stage {
45   design {
46     FIRFilter firs [12];
47     for (int id=0; id < 12; id++) {
48         firs [id](inputGen[i], next);
49     }
50   }
51   ...
52 }
```

```
53 capsule DetectorChain(Stage writer)
54     implements Stage {
55   design {
56     BeamForm beamForm;
57     BeamFirFilter beamFirFilter;
58     Magnitude magnitude;
59     Detector detector;
61     beamForm(beamFirFilter);
62     beamFirFilter(magnitude);
63     magnitude(detector);
64     detector( writer );
65   }
66    ...
67 }
68 capsule DetectorChainGroup(Stage writer)
69     implements Stage {
70   design {
71     DetectorChain detects[4];
72     for (int id=0; id < 4; id++) {
73         detects[id]( writer );
74     }
75   }
76   ...
77 }
```

Figure 3.4 BeamFormer, with multiple design declarations

3.6 Discussion

The unified design declaration enables several improvements over the asymmetric design from Panini 0.9.1. Specifically, the new version improves upon the original in the following ways:

1. The implementation closely follows the logical model. Individual filter components can be connected to form a filter chain and filter chains can be connected in parallel when necessary.

2. Each filter chain has a clear point, the design declaration, where the internal connections are made. A single filter chain's connections can be understood in terms of internal design of the capsule.

3. Clients of the a filter chain do not need to know how the filter chain is connected internally. Composite filters can be re-used without having to re-connect each component. In contrast, the system

in Panini 0.9.1 requires all of the elements of the filter chains to be connected in one place. For example, if we wanted to reuse an `DetectorChainGroup` in another program in Panini 0.9.1, we would be required to know how to wire those connections in the new system. With the design declaration in Panini 0.9.2 we can simply declare and instance of the `DetectorChainGroup` and connect it to the desired output `FileWriter` instance.

In summary, the unified design enables both better modularity and more understandable code. The individual composite filters, FIR and Detector are now encapsulated inside of a single capsule type, `FIRFilter` and `DetectorChain`. The aggregation of the individual filters into groups is handled by the `FIRGroup` and `DetectorChainGroup`. Finally, `BeamFormer`, the main capsule, now only declares the `FIRGroup`, `DetectorChainGroup`, and the output capsule `fileWriter`, making the connections much clearer than in Figure 3.3 version.

CHAPTER 4. CASE STUDY: SERPENT ENCRYPTION ALGORITHM

4.1 Serpent encryption algorithm overview

The ‘serpent’ encryption algorithm [8] is a symmetric block key encryption algorithm, designed for efficient implementation in software. It was one of contenders for the ‘Advanced Encryption Standard’. As the author is by no means a cryptographer, nor is it expected that the reader have a background in cryptography, the discussion in this chapter will focus on the ‘high level’ parts of the algorithm and how the implementation is simplified and made more modular with the unified design in Panini 0.9.2.

The basic serpent algorithm consists of three primary phases. There is an initial permutation, followed by multiple rounds of key-mixing, a pass through an element called an ‘Sbox’ and a linear transformation. The last round replaces the linear transformation with an additional key-mixing operation.

Figure 4.1 shows the basic design of a single round in serpent. Data is permuted in the SlowKeySchedule block and then combined with the original data via an Xor operation before going through another Sbox operation and a linear transformation. The result is passed onto the next round of the algorithm. The SlowKeySchedule passes data through a BitSlice operation, an Sbox operation, a second BitSlice, and a permutation. Again, the purpose of this discussion is not focused on the specifics of the serpent algorithm or the implementation specifics of any phase, but rather the connections and data flow through the algorithm.

4.2 Initial design

Figure 4.2 shows the basic set of capsule declarations and wiring statements for the Panini 0.9.1 serpent implementation. The figure focuses on the capsule instance declarations and the wiring statements connecting the instances together. There are eight stages in the serpent encryption algorithm. Since the wiring for each stage is identical, only the wiring statements for the first stage are included.

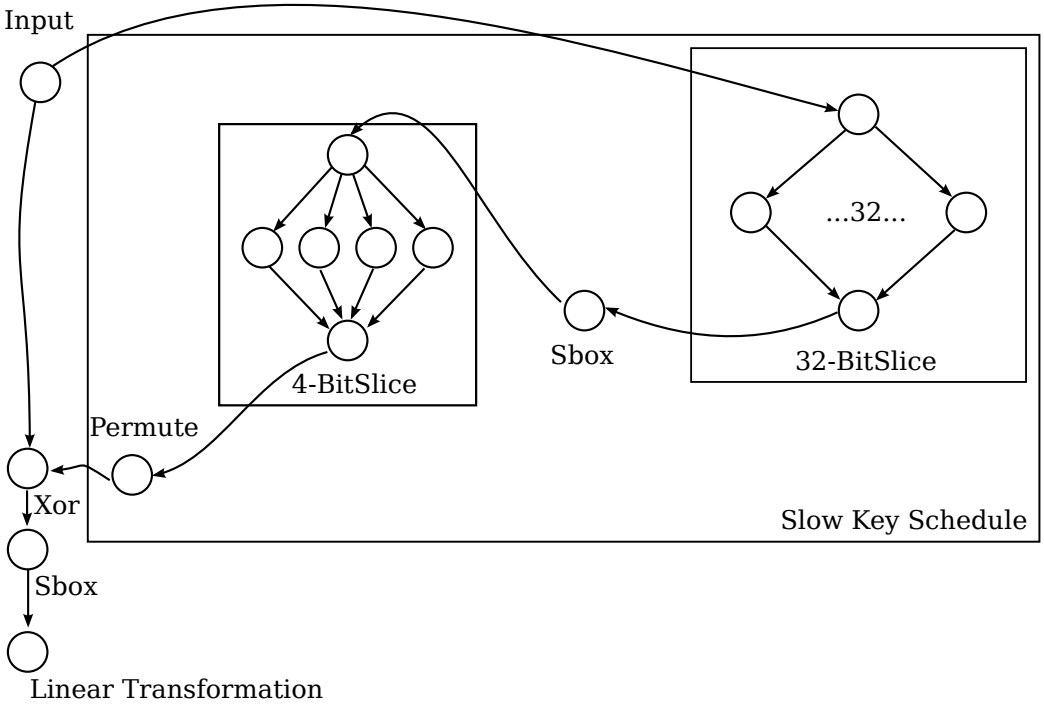


Figure 4.1 One round of the serpent encryption algorithm

4.3 Limitations

All of the problems discussed in Chapter 3 are present in the the system version of the serpent algorithm. The large and highly repetitive system has approximately 40 individual capsules, capsule arrays, or instance declarations and requires around 120 wiring statements. There are eight stages to the algorithm, all of which must be individually wired. The **wireall** keyword is especially helpful in wiring the large 32-bit `Identity` array components to their respective `BitSlice` capsules.

Additionally, the heavy use of arrays and the common pattern of repeating a name with a number (e.g. `smallbitSliceidentity0`, ..., `smallbitSliceidentity7`) are symptomatic of uncomposable systems in Panini 0.9.1. We want to model a single round of the algorithm as a capsule, be able to instantiate multiple copies, and connect them together to create a program which executes multiple rounds the alogorithm. Instead, we are forced to replicate the work multiple times in the system to create the desired program.

```
1  system Serpent {
2    Input input;
3    SerpentEncoder encoder;
4    Permute ip_permute;
5    R rs[8];
6    AnonFilter_a8 anonFilter_a8[8];
7     Identity identity [8];
8    slowKeySchedule slowKeySch[8];
9    AnonFilter_a4 anonFilter_a4[8];
10    IntoBits intoBits [8];
11    BitSlice largebitSlice [8];
12    Identity largebitsliceidentity0 [32];
13    Identity largebitsliceidentity1 [32];
14    Identity largebitsliceidentity2 [32];
15    Identity largebitsliceidentity3 [32];
16    Identity largebitsliceidentity4 [32];
17    Identity largebitsliceidentity5 [32];
18    Identity largebitsliceidentity6 [32];
19    Identity largebitsliceidentity7 [32];
20    Sbox innersbox[8];
21    BitSlice smallbitSlice [8];
22    Identity smallbitSliceidentity0 [4];
23    Identity smallbitSliceidentity1 [4];
24    Identity smallbitSliceidentity2 [4];
25    Identity smallbitSliceidentity3 [4];
26    Identity smallbitSliceidentity4 [4];
27    Identity smallbitSliceidentity5 [4];
28    Identity smallbitSliceidentity6 [4];
29    Identity smallbitSliceidentity7 [4];
30    Perms slowKeySchpermute[8];
31    Xor xor[9];
32    Sbox sbox[8];
33    rawL rawl[7];
34    AnonFilter_a9 anonFilter_a9;
35     Identity identity1 ;
36    KeySchedule keySchedule;
37    AnonFilter_a0 anonFilter_a0;
38    Permute fp_permute;
39    Output output;
40    reader(inputFileName,encoder);
41    encoder(ip_permute);
42    ip_permute(0, NBITS, rs[0]);

44     //wire one stage of the algorithm
45    rs[0](anonFilter_a8[0]);
46    anonFilter_a8[0](identity [0], slowKeySch[0]);
47    slowKeySch[0](anonFilter_a4[0]);
48    anonFilter_a4[0](0, intoBits [0], BITS_PER_WORD,PHI,
          USERKEY_LENGTH,vector);
49     intoBits [0](largebitSlice [0]);
50     largebitSlice [0](largebitsliceidentity0 );
51    wireall(largebitsliceidentity0 [0], innersbox[0]));
52    innersbox[0](0, 0, smallbitSlice [0]);
53     smallbitSlice [0](smallbitSliceidentity0 );
54    wireall(smallbitSliceidentity0 , slowKeySchpermute[0]);
55    slowKeySchpermute[0](N,xor[0]);
56     identity [0](xor[0]);
57    xor[0](sbox[0]);
58    sbox[0](1, 0, rawl[0]);
59    rawl[0](rs[1]);

61     //Repeat 7 more times.
62     //Wire the last copy of the stage to the output capsule
          instance

64  }
```

Figure 4.2 Serpent encryption, Panini 0.9.1

4.4 Redesign

The design of the serpent algorithm shows several primary modules, which are connected together to form a single stage of the encryption algorithm. Multiple stages are ‘stacked’ together form a complete algorithm.

```
capsule Bitstream(Indentity[] idents, Stage sink)
      implements Stage {
    void consume(...) {
        // processing idents elems to results
        sink.consume(results);
    }
}
capsule Bitstream4(Stage next) implements Stage {
    design {
        Indentity idents[4];
        wireall(idents, next);
    }
    ...
}
capsule Bitstream32(Stage next) implements Stage {
    design {
        Indentity idents[32];
        wireall(idents, next);
    }
    ...
}
capsule SlowKeySchedule(Stage next) implements Stage
     {
    design {
        IntoBits intoBits;
        Bitstream32 b32;
        Sbox sbox;
        Bitstream4 b4;
        Permute perm;
        // wiring
        intoBits(b32);
        b32(sbox);
        sbox(b4);
        b4(perm);
        perm(next);
    }
    ...
}
capsule AnonfilterA8(Stage next) implements Stage {
    design {
        SlowKeySchedule sks;
        Identity ident;
        sks(next);
        ident(next);
    }
    ...
}
capsule R(Stage next) implements Stage {
    design {
        AnonFilterA8 a8;
        Xor xor;
        Sbox sbox;
        Rawl rawl;

        a8(xor);
        xor(sbox);
        sbox(rawl);
        rawl(next);
    }
    ...
}
capsule RLast(Stage next) implements Stage {
    design {
        AnonFilterA8 a8;
        Xor xor1, xor2;;
        Sbox sbox;
        AnonFilter_a9 a9; // definition elided

        a8(xor1);
        xor1(sbox);
        sbox(a9);
        a9(xor2);
        xor2(next);
    }
    ...
}
capsule SerpentEncoder(Output output) implements
     Stage {
    design {
        R r[7];
        RLast rLast;
        Input input;
        Permute perm1, perm2;

        perm1(r);
        for(int i = 0; i < 6; r++) {
            r[i](r[i+1]);
        }
        r[6](rLast);
        rLast(perm2);
        perm2(output);
    }
    void consume(...) {
        ...
        output.consume(...);
    }
}

capsule Serpent(String[] args) {
    design {
        Input in;
        Output out;
        SerpentEncoder serpent;

        in(serpent);
        serpent(out);
    }
    void run() {
        in.consume();
    }
}
```

Figure 4.3 Serpent, modularized design in Panini 0.9.2

The major components of each stage are the slow key schedule, sbox, permute, bitslice and identity. Many of the components in a single stage of serpent are created by connecting to several subcomponents together. For example, the BitSlice component is composed of N 'Identity' elements. The SlowKeySchedule uses both a 32 bit and a 4 bit BitSlice. The Sbox component is used in both the SlowKeySchedule and the main part of the serpent stage.

The version of the serpent algorithm using the unified design is shown in Figure 4.3. It is much simpler and easier to understand the system in Figure 4.2. Notably, we can understand the connections of each stage in terms of that stage. For example, we can see that the design of the 'SlowKeySchedule' on lines 22-34 is composed of the `IntoBits`, `Bitstream32`, `Sbox`, `BitStream4` and `Permute` capsule instance. Unlike the system example, which tangled together both the creation of a single key schedule component with need to have eight key schedules (e.g. using capsule arrays, and wiring specific indices together), the new version clearly shows how a key schedule is formed. The IntoBits capsule connects to the 32-bit BitSlicer, which connects to an Sbox and the to the 4-bit BitSlicer. Similarly, using a key schedule is now as simple as declaring a SlowKeySchedule capsule instance and connecting it to the appropriate next stage. Similarly, it easy to see and understand how the entire `SerpentEncoder`, lines 75-94 is formed. As stated in the description, there are eight rounds of the algorithm. The first seven rounds, represented by the `R` type capsule array are the same. The last round, represented by the `RLast` type capsule, replaces the linear transformation, the `Rawl` capsule with a second key mixing operation, implemented by the `Anon_9` type capsule. Overall, the entire implementation is much easier to see and understand what the parts are and how they fit together.

4.5 Discussion

The problems we identified in Chapter 3 are now much more pronounced. The large number of capsule instances and the large number of wiring connections required to connect all of the components in each stage in the program makes the single system approach completely unmanageable. Furthermore, by not being able to abstract away the details of a single component behind a capsule type interface, the programmer is faced with seeing every single detail of each component in one place. Consequently, the improved modularity (§4.4) of the serpent program in Panini 0.9.2 is that much more important. The

biggest improvements from the unified design are that the structure of the algorithm is not obscured by the restrictions imposed by the single system and that the connections needed for a single stage only need to be defined once.

CHAPTER 5. RELATED WORK

5.1 Actor oriented programming

Many different proposals have explored the idea of enabling concurrency at module boundaries. Some of these ideas include actors [1], guardians [23] and active objects [21]. The Scala language [16] includes a built-in actor library, and Akka [15] extends to concept to be much more fault tolerant.

The notion of actors [1, 2], is that certain entities should operate as independent elements in the system, which communicate with each other. The approach helps encapsulate which thread of control the message is handled on, as well as ensures that multiple threads do not mutate the actors internal state.

Active objects [21, 28, 9] follow a similar design approach by separating the thread from the message which requests an operation from the thread which carries out said operation.

5.1.1 Scala

Scala actors communicate through channels which contain messages [16]. Both actors and channels are first class values and may be passed in as arguments in the message. This allows for flexibly, but linguistically uncontrollable communication topology between actor instances. This approach is starkly different from Panini, which requires the communication topology to be fixed when a capsule is compiled and does not allow that topology to change at runtime.

5.1.2 Akka

Akka actors, which supplant the Scala actor framework, are designed to be highly fault tolerant, and incorporate many ideas from Erlang [15]. Akka actors are more strongly encapsulated than Scala actors. In Scala, one actor is permitted to directly call a method on another actor instead of sending that

actor a message. In contrast, Akka actors expose a much smaller interface which limits the potential for breaking the encapsulation model.

Capsules are more strongly typed compared to Akka actors. Specifically, the external type of a capsule's external connections are known and are not allowed to change dynamically.

5.1.3 Kilim

Kilim [46] provides an actor library for Java library which focuses on providing a very light-weight implementation and efficiently enabling many actor instances in a system. Like most actor based systems, Kilim actors communicate via 'mailboxes'. A mailbox is not owned by any specific entity in the system and a mailbox can be passed around to other actors. Like Scala and Akka, the topology in a Kilim program can be highly dynamic.

The main difference between the Panini language and many of the other actor oriented frameworks is the static nature of the communication topology. Generally, actor languages provide some mechanism to change which actors are capable of communicating at runtime. The Panini language removes this ability in favor of a fixed communication graph which is simpler to understand and easier to statically analyze.

5.2 Other work on capsule-oriented programming

This work is related to other ongoing research on capsule-oriented programming. Bagherzadeh and Rajan have investigated the modular reasoning properties of capsules [4], whereas we explored unification in the design of capsules. Lin *et al.* have investigated the notion of duck futures [22]. Upadhyaya and Rajan explored an automatic strategy for mapping capsules to JVM threads [48, 49]. Johnston, Mills, Erenberger and Rajan are exploring an annotation-based implementation of capsule-oriented programming, which is utilizing the unified design proposed in this thesis.

CHAPTER 6. CONCLUSION

This thesis has discussed the capsule-oriented programming model, the Panini programming language, and the changes introduced in Panini 0.9.2 to facilitate better reuse of capsule instance connections via the introduction of a design declaration block in a capsule's definition.

One key feature of capsule-oriented programming is a compile-time fixed topology between capsule instances in the program. Removing the ability to dynamically connect actor instances and change the run-time topology of the communication graph between actor instances, enables compile time detection of key properties such as ensuring sequential consistency within a capsule.

Initially, in Panini 0.9.1, the static topology requirement was accomplished by creating a special top-level type called a system, The system served as the sole point of capsule instance declaration and wiring. The benefits of the single system approach were a clear place to begin the capsule communication graph, required to enforce several properties of the language, as well as a clear place to begin program execution.

This single system approach, however, turned out to be too limiting to allow for the creation of modular, reusable programs. A single point of capsule instance creation and wiring does not present a problem with small example programs. However, as the size and complexity of the program grows, the single point of creation and wiring only becomes a bottle neck. Not only does the number of capsule instance declarations in the system become unmanageable, but the scheme used to wire those instances together may be repeated to create several copies of the same 'sub-system' in the program. Examples of both the complexity and of the potential for system reuse in larger Panini programs were shown in the case studies in Chapter 3 and Chapter 4.

The unified design of Panini 0.9.2 solves the system modularity problem by combining the capsule and system declarations to allow a capsule to contain a single design declaration. A capsule design is like a system, in that it creates a place to declare capsule instances and connect those instances

together. However, because the design is encapsulated inside of a capsule instance, and not a single top-level entity in the program, design declarations are reusable. Each new capsule instance automatically creates a new copy of its design. The unified design solves the problems of a system only being created once, multiple systems cannot be connected together, and the need to include every capsule instance declaration and wiring statement in one place.

Future Work The Panini project continues to investigate capsule-oriented programming. For example, Upadhyaya is looking into modular analysis to determine mapping from capsules to threads [48], Bagherzadeh is looking into modular reasoning properties of Capsules [4], Lin is investigating whether certain bounds in capsule-oriented designs can be relaxed without compromising the ability to statically analyze capsules and their interaction [18, 19], and Long is looking into sequential consistency properties of capsules [24].

BIBLIOGRAPHY

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

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

[3] 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.

[4] Mehdi Bagherzadeh and Hridesh Rajan. Panini: A concurrent programming model for solving pervasive and oblivious interference. In *Modularity'15: 14th International Conference on Modularity*, March 2015.

[5] 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.

[6] 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.

[7] Mehdi Bagherzadeh, Hridesh Rajan, Gary T. Leavens, and Sean L. Mooney. Translucid contracts for modular reasoning about aspect-oriented programs. In *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, SPLASH/OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, pages 245–246, 2010.

[8] Eli Biham, Ross Anderson, and Lars Knudsen. Serpent: A new block cipher proposal. In *In Fast Software Encryption '98*, pages 222–238. Springer-Verlag, 1998.

[9] Dave Clarke, Tobias Wrigstad, Johan Östlund, and Einar Broch Johnsen. Minimal ownership for Active Objects. In *APLAS '08*.

[10] Robert Dyer, Hridesh Rajan, and Yuanfang Cai. An exploratory study of the design impact of language features for aspect-oriented interfaces. In *AOSD '12: 11th International Conference on Aspect-Oriented Software Development*, March 2012.

[11] Robert Dyer, Hridesh Rajan, and Yuanfang Cai. Language features for software evolution and aspect-oriented interfaces: An exploratory study. *Transactions on Aspect-Oriented Software Development (TAOSD): Special issue, best papers of AOSD 2012*, 10:148–183, 2013.

[12] Rex Fernando, Robert Dyer, and Hridesh Rajan. Event type polymorhphism. In *FOAL '12: Workshop on Foundations of Aspect-Oriented Languages workshop*, March 2012.

[13] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[14] Jayanth Gummaraju and Mendel Rosenblum. Stream programming on general-purpose processors. In *Microarchitecture, 2005. MICRO-38. Proceedings. 38th Annual IEEE/ACM International Symposium on*, pages 12–pp. IEEE, 2005.

[15] Philipp Haller. On the integration of the actor model in mainstream technologies: the scala perspective. In *Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions*, pages 1–6. ACM, 2012.

[16] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci. '09*, 410.

[17] Robert H. Halstead. Multilisp: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7:501–538, 1985.

[18] 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.

[19] 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.

[20] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic parallelism requires abstractions. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 211–222, 2007.

[21] R. Lavender and D. Schmidt. Active object – an object behavioral pattern for concurrent programming. In *Pattern languages of program design 2*, 1996.

[22] 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.

[23] Barbara Liskov and Robert Scheifler. Guardians and Actions: Linguistic support for robust, distributed programs. *TOPLAS '83*, 5.

[24] Yuheng Long, Mehdi Bagherzadeh, Eric Lin, Ganesha Upadhyaya, and Hridesh Rajan. Quantification of sequential consistency in actor-like systems: An exploratory study. Technical Report 14-03, Iowa State University, 2014.

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

[26] Yuheng Long, Hridesh Rajan, and Sean L. Mooney. Reconciling concurrency and modularity with panini's asynchronous typed events. In *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, SPLASH/OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, pages 243–244, 2010.

[27] Sean L. Mooney, Hridesh Rajan, Steven M. Kautz, and Wayne Rowcliffe. Almost free concurrency! (using GOF patterns). In *Companion to the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, SPLASH/OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, pages 249–250, 2010.

[28] O. M. Nierstrasz. Active objects in Hybrid. In *OOPSLA '87*.

[29] David Notkin, David Garlan, William G. Griswold, and Kevin J. Sullivan. Adding implicit invocation to languages: Three approaches. In *JSSST International Symposium on Object Technologies for Advanced Software*, pages 489–510, 1993.

[30] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Incorporation, USA, 1st edition, 2008.

[31] Hridesh Rajan. Design pattern implementations in eos. In *PLoP '07, Conference on Pattern Languages of Programs*, September 2007.

[32] Hridesh Rajan. Building scalable software systems in the multicore era. In *2010 FSE/SDP Workshop on the Future of Software Engineering*, Nov. 2010.

[33] Hridesh Rajan. Capsule-oriented programming. In *ICSE'15: The 37th International Conference on Software Engineering: NIER Track*, May 2015.

[34] 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.

[35] 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.

[36] Hridesh Rajan, Steven M. Kautz, and Wayne Rowcliffe. Concurrency by modularity: Design patterns, a case in point. In *2010 Onward! Conference*, October 2010.

[37] Hridesh Rajan and Gary T. Leavens. Quantified, typed events for improved separation of concerns. Technical Report 07-14c, Iowa State University, Dept. of Computer Sc., October 2007.

[38] 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.

[39] Hridesh Rajan, Sean L. Mooney, Gary T. Leavens, Robert Dyer, Rex D. Fernando, Mohammad Ali Darvish Darab, and Bryan Welter. Modularizing crosscutting concerns with ptolemy. In *Companion to the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2011, part of SPLASH 2011, Portland, OR, USA, October 22 - 27, 2011*, pages 31–32, 2011.

[40] Hridesh Rajan and Kevin Sullivan. Eos: instance-level aspects for integrated system design. In *the European software engineering conference and international symposium on Foundations of software engineering (ESEC/FSE)*, pages 297–306, 2003.

[41] Hridesh Rajan and Kevin Sullivan. Aspect language features for concern coverage profiling. In *the international conference on Aspect-oriented software development (AOSD)*, pages 181–191, 2005.

[42] Hridesh Rajan and Kevin J. Sullivan. Classpects: unifying aspect- and object-oriented language design. In *the international conference on Software engineering (ICSE)*, pages 59–68, 2005.

[43] Hridesh Rajan and Kevin J. Sullivan. Unifying aspect- and object-oriented design. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 19(1), August 2009.

[44] Steven P. Reiss. Connecting tools using message passing in the Field environment. *IEEE Softw.*, 7(4):57–66, 1990.

[45] Saman Amarsinghe *et al.* StreamIt. http://groups.csail.mit.edu/cag/streamit/. Accessed: 2013-08-05.

[46] Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-typed actors for java. In *ECOOP 2008–Object-Oriented Programming*, pages 104–128. Springer, 2008.

[47] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A language for streaming applications. In *Compiler Construction*, pages 49–84. Springer, 2002.

[48] Ganesha Upadhyaya and Hridesh Rajan. An automatic actors to threads mapping technique for JVM-based actor frameworks. In *AGERE!'14*.

[49] Ganesha Upadhyaya and Hridesh Rajan. Effectively mapping linguistic abstractions for message-passing concurrency to threads on the Java virtual machine. In *OOPSLA'15*, October 2015.

[50] Barry D Van Veen and Kevin M Buckley. Beamforming: A versatile approach to spatial filtering. *ASSP Magazine*, 5(2):4–24, 1988.

[51] Adam Welc, Suresh Jagannathan, and Antony Hosking. Safe Futures for Java. In *OOPSLA '05*.