A Unified Model for Hardware/Software Codesign

by

Nirav Dave

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2011

Author .....
Department of Electrical Engineering and Computer Science
July 26, 2011

Certified by .....
Arvind
Johnson Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Professor Leslie A. Kolodziejski
Chair, Department Committee on Graduate Students

A Unified Model for Hardware/Software Codesign

by

Nirav Dave

Submitted to the Department of Electrical Engineering and Computer Science
on July 26, 2011, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

Embedded systems are almost always built with parts implemented in both hardware and software. Market forces encourage such systems to be developed with different hardware-software decompositions to meet different points on the price-performance-power curve. Current design methodologies make the exploration of different hardware-software decompositions difficult because such exploration is both expensive and introduces significant delays in time-to-market. This thesis addresses this problem by introducing, Bluespec Codesign Language (BCL), a unified language model based on guarded atomic actions for hardware-software codesign. The model provides an easy way of specifying which parts of the design should be implemented in hardware and which in software without obscuring important design decisions. In addition to describing BCL's operational semantics, we formalize the equivalence of BCL programs and use this to mechanically verify design refinements. We describe the partitioning of a BCL program via computational domains and the compilation of different computational domains into hardware and software, respectively.

Thesis Supervisor: Arvind
Title: Johnson Professor of Electrical Engineering and Computer Science

Acknowledgments

This thesis has been a long time in coming and as one might expect, the list of those who deserve proper acknowledgment are long and hard to list.

Foremost, I would like to my Advisor Professor Arvind for his help and mentorship. It is not an easy thing to find an advisor with a love of both the theoretical aspects of the work and an appreciation for practical application. Arvind never forced me to step away from any vector of interest. This perhaps is part of why my time at MIT was so long, but also so worthwhile.

I would also like to thank Rishiyur Nikhil, Joe Stoy, Jamey Hicks, Kattamuri Ekanadham, Derek Chiou, James Hoe, and John Ankcorn for their advice and discussion on various aspects of this work and its presentation over the years.

To Jacob Schwartz, Jeff Newbern, Ravi Nanavati, and everyone else at Bluespec Inc. who patiently dealt with both my complaints about the compiler and my off the wall ideas.

To my fellow graduate students, Charles O'Donnell, Ryan Newton, Man Cheuk Ng, Jae Lee, Mieszko Lis, Abhinav Agarwal, Asif Khan, Muralidaran Vijayaraghavan, and Kermin Fleming, who put up with my distractions. Special thanks are needed for Michael Pellauer who helped slog through some of the semantic details, Myron King who spent many years working with me on the BCL compiler and Michael Katelman who was instrumental in the verification effort.

To Peter Neumann, for significant English assistance, and convincing me to keep up the editing process even after the thesis was acceptable.

To my parents who were always there to remind me that my thesis need not solve the field and perhaps eight years of work was enough. And last but not least Nick and Larissa who got me out of the office and kept me sane over the years. I do not think I would have made it without you both.

Contents

List of Figures

Chapter 1

Introduction

Market pressures are pushing embedded systems towards both higher performance and greater energy efficiency. As a result, designers are relying more on specialized hardware, both programmable and non-programmable, which can offer orders of magnitude improvements in performance and power over standard software implementations. At the same time, designers cannot implement their designs entirely in hardware, which leaves the remaining parts to be implemented in software for reasons of flexibility and cost. Even fairly autonomous non-programmable hardware blocks are frequently controlled by software device drivers. In this sense all embedded designs involve hardware-software codesign.

In businesses where embedded designs are necessary, first-to-market entries enjoy a substantially higher profit margin than subsequent ones. Thus designers are under great pressure to prevent delays, especially those caused by final integration and testing. For subsequent products, the novelty of new functionalities have a much lower effect, and their value is driven by performance, power, and of course cost. Thus, over the life-cycle of a product class, individual embedded designs frequently have to make the transition from rapidly-designed but good-enough to time-consuming but highly-efficient designs.

Given this shifting set of requirements, engineers would like to be able to start with a design implemented mainly in software using already developed hardware blocks, and gradually refine it to one with more hardware and better power/performance

properties. Isolating such refinements from the rest of the system is important to smooth the testing and integration processes. Unfortunately such flexibility is not easily provided due to the radically different representation of hardware and software. Embedded software is generally represented as low-level imperative code. In contrast, hardware systems are described at the level of registers, gates, and wires operating as a massively parallel finite state machine. The differences between these two idioms are so great that the hardware and software parts of the design are done by entirely separate teams.

The disjointedness of software and hardware teams strongly affects the standard design process. Since time-to-market is of utmost importance, both parts of the design must be specified and implemented in parallel. As a result, the hardware-software decomposition and the associated interface are specified early. Even when the hardware-software decomposition is fairly obvious, specifying the interface for interaction without the design details of the parts is fraught with problems. During the design process the teams may jointly revisit the early decisions to resolve specification errors or deal with resource constraints. Nevertheless, in practice, the implemented interface rarely matches the specification precisely. This is quite understandable as the hardware-software interface must necessarily deal with both semantic models. Frequently during the final stages of integration, the software component must be modified drastically to conform to the actual hardware to make the release date. This may involve dropping useful but non-essential functionalities (*e.g.,* using low-power modes or exploiting concurrency). As a result, designs rarely operate with the best power or performance that they could actually achieve.

The problem of partitioning can be solved if we unify the description of both software and hardware using a single language. An ideal solution would allow designers to give a clear (sequential) description of the algorithm in a commonly-used language, and specify the cost, performance and power requirements of the resulting implementation. The tool would then take this description, automatically determine which parts of the computation should be done in hardware, insert the appropriate hardware-software communication channel, parallelize the hardware computation effi-

ciently, parallelize the software sufficiently to exploit the parallelism in the hardware, and integrate everything without changing the semantic meaning of the original program. In the general case, each of these tasks is difficult and requires the designer's input, making the possibility of this type of design flow infeasible.

This thesis discusses a more modest language-based approach. Instead of trying to solve the immense problem of finding the optimal solution from a single description, our goal is to facilitate the task of exploration by allowing designers to easily experiment with new algorithms and hardware-software partitionings without significant rewriting. The designer's task is to construct not one, but many different hardware-software decompositions, evaluate each one, and select the best for his needs. This approach lends itself to the idea of retargeting, since each design becomes a suite of designs and thus is robust to changes needed for performance, functionality, or cost. This approach also helps in keeping up with the constantly evolving semiconductor technology.

1.1 Desirable Properties for a Hardware-Software Codesign Language

Any hardware-software codesign solution must be able to interoperate with existing software stacks at some level. As a result, a hardware-software codesign language need not be useful for all types of software, and can focus only on the software that needs to interact with hardware or with software that might be potentially implemented in hardware. In such a context, the designer isolates the part of the design that could *potentially* be put into hardware and defines a clean and stable interface with the rest of the software. As we now only consider "possibly hardware" parts of the design, *i.e.,* parts that will be implemented in hardware or as software expressed naturally in a hardware style, the semantic gap between the hardware and software is smaller and it becomes reasonable to represent both in a single unified language.

With a single language, the semantics of communication between hardware and

software are unambiguous, even when the hardware-software partitioning is changed. To be viable, a unified language must have the following properties:

1. *Fine-grain parallelism:* Hardware is inherently parallel, and any codesign language must be flexible enough to express meaningful hardware structures. Low-level software that drives the hardware does so via highly concurrent untimed transactions, which must also be expressible in the language.

2. *Easy specification of partitions:* In complex designs it is important for the designer to retain a measure of control in expressing his insights about the partitioning between hardware and software. Doing so within suitable algorithmic parameters should not require any major changes in code structure. Further the addition of a partition should not affect the semantics of the system; designers should be able to reason about the correctness of a hardware-software design as either a pure hardware or software system.

3. *Generation of high-quality hardware:* Digital hardware designs are usually expressed in RTL (Register-Transfer Level) languages like Verilog from which low-level hardware implementations can be automatically generated using a number of widely available commercial tools. (Even for FPGAs it is practically impossible to completely avoid RTL). The unified language must compile into efficient RTL code.

4. *Generation of efficient sequential code:* Since the source code is likely to contain fine-grained transactions to more clearly expose pipeline parallelism for exploitation in hardware, it is important that software implementations are able to effectively sequence transactions for efficiency without introducing unnecessary stalls when interacting with hardware or other external events.

5. *Shared communication channels:* Often the communication between a hardware device and a processor is accomplished via a shared bus. The high-level concurrency model of the codesign language should permit sharing of such channels without introducing deadlocks.

Given such a language, it should be possible for designers to reason about system changes in a straightforward manner. Not only should it be possible to easily modify a hardware-software design by changing the partitioning, but it should also be possible to reason about the correctness of this system as easily if it had been implemented entirely in software or entirely in hardware.

1.2 Thesis Contributions

This thesis is about the semantic model embodied in the language and the challenges that must be addressed in the implementation of such a language. It is not about the surface syntax, types, or the meta-language features one may wish to include in a design language. The starting point of our design framework is guarded atomic actions (GAAs) and Bluespec SystemVerilog (BSV), a language based on such a framework. BSV is an industrial-strength language for hardware design [20]. Significant work has been done towards a full implementation of BCL, the proposed language. BCL programs are currently running on multiple mixed hardware-software platforms. However, even a preliminary evaluation of BCL's effect requires an significant amount of additional platform; application specific effort has been done, mostly by Myron King and will appear in his PhD thesis.

This thesis makes the following contributions:

- We introduce Bluespec Codesign Language (BCL), a unified hardware-software language and give its operational semantics. BCL is an extension of the semantic model underlying BSV, adding *sequential composition of actions*, *dynamic loops*, and *localization of guards*. Like BSV, the execution of a BCL program must always be understood as a serialized execution of the individual rules. However, as any serialization of rule executions is valid, the BCL programs are naturally parallel; multiple rules can be executed concurrently without committing to a single ordering resolution. The additional capabilities of BCL make it convenient to express low-level software programs in addition to hardware designs. The operational semantics were developed jointly with Michael Pellauer.

(Chapter 2)

- We introduce a notion of equivalence of a BCL program based on state observability. This, for the first time, provides a proper foundation for some well-known properties of BSV, such as derived rules and the correctness of parallel scheduling. (Chapters 2 and 5).

- We use this notion of observability to develop an equivalence checking tool based on satisfiability-modulo-theories (SMT). This tool is particularly useful as a debugging aid as it can automatically verify whether splitting a rule into smaller rules has introduced new behaviors. This is a common step-wise refinement design process used in the development of BCL programs. This tool was developed jointly with Michael Katelman. (Chapters 2 and 8).

- We extend the notion of clock domains [30] to allow for multiple *computational domains* both in hardware and software. We use this type-based mechanism to express the precise partitioning of a BCL program. Domains allow the compiler to figure out the precise communication across hardware-software boundaries. Annotating a BCL program with domains also suggest how to directly implement the resulting communication (Chapter 6).

- Efficient implementations of a BCL program (and thus a BSV program) practically must restrict the choice inherent in the program, *i.e., scheduling* the rules. We provide a representation of this scheduling process via rule composition. This allows the designer to understand the scheduling restrictions programmatically and even express it themselves. It also enables *partial scheduling* where nondeterminism is left in the final implementation, an important property for efficient hardware-software implementations. (Chapter 5).

- The construction of an initial BCL compiler that can partition a BCL design into hardware and software. We introduce methods for compiling the software partition to both Haskell and C++. The former is used in our verification effort while the later is used for implementing embedded systems and makes use of

nontrivial optimizations to remove the need for the non-strictness and dynamic allocation associated with Haskell. The C++ compilation was developed jointly with Myron King who is continuing the development of the BCL compiler for his PhD thesis. (Chapters 3 and 7).

1.3 Thesis Organization

This thesis has many topics and the reader can read the chapters in multiple orders. This section serves to prime the user as to the chapter contents and the relative data dependencies between them.

The remainder of this chapter discusses the various works in different areas that have bearing on the this work. It is included for context and does not greatly impact knowledge transfer.

Chapter 2 introduces BCL and its operational semantics. It also introduces various concepts needed to understand program refinement and implementation. This chapter is necessary preliminaries for all subsequent chapters, though the later sections may be skipped if one is not interested in verification (Chapter 8) or the semantic issues of implementation (Chapter 5).

Chapter 3 discusses the translation of BCL to λ-calculus. This translation is an important step in hardware generation (Chapter 4) and verification efforts (Chapter 8). It also lends insight to software generation (Chapter 7) but is not strictly necessary to understand software generation.

In Chapter 4 we discuss the generation of synchronous hardware from a BCL program, and how BCL module interfaces relate to their FSM implementations. This chapter sheds some light onto choices made by the historical approaches to scheduling of hardware implementations which is discussed Chapter 5, but is not necessary to understand the rest of the thesis.

In Chapter 5 we describe the task of scheduling programs, *i.e.,* reducing choice in an implementation with the goal of improving efficiency. We discuss this in terms of rule composition and derived rules, which enables us to schedule the hardware and

software portions a program independently. This chapter does not have any direct bearing on the understanding of subsequent chapters.

In Chapter 6, we show the partitioning of a BCL program via computational domains. It also discusses how communication channels can be abstracted and the compilation task isolated to each individual substrate. This chapter is not necessary for any subsequent chapters.

We discuss the implementation of BCL in C++ in Chapter 7. This involves overcoming various restrictions due to the imperative nature of the backend language. We also discuss various optimizations that reduce the memory and execution time overhead.

In Chapter 8 we discuss a notion of state-based equivalence of programs and show how this model allows us to consider many non-trivial refinements (*e.g.,* pipelining) as equivalence preserving. It also introduces an algorithm that decomposes the notion of observability into a small number of SMT queries and a tool which embodies it.

Finally, in Chapter 9 we conclude with a summary of the thesis contributions, and a discussion of future work.

1.4 Related Work

The task of dealing with hardware-software systems is well-established and appears in multiple contexts with vastly different goals. However, collectively this literature gives a good intuition about the state of the art. In the remainder of this chapter, we will discuss previous work as it relates to the task of hardware-software codesign.

1.4.1 Software Models of Hardware Description Languages

Hardware implementation is an expensive and time consuming task, and it is impractical to design hardware directly in circuitry in the design process. Even with programmable hardware blocks, such as PLAs [44] or Field Programmable Gate Arrays (FPGAs) [21] it is still useful to execute models of hardware in software.

Fundamentally, digital hardware is represented at the transistor level and we can

model the underlying physics of circuits in systems such as SPICE [63]. Practically this level of detail is too great and designers moved to gate-level schematics which model only the digital aspects of design. This abstraction greatly speeds up simulation of circuits but is still immensely slow to construct and test. Languages at the Register-Transfer Level (RTL), such as Verilog [90] and VHDL [12] allowed designers to go from a graphical description to textual descriptions. This was a significant improvement as it decoupled the description from its physical implementation, a necessary step in moving towards higher-level representations.

As a matter of course, designers wanted to express not only their designs, but also their test bench infrastructure in RTL. As such, certain C-like "behavioral" representations were allowed. As this became more prevalent, approaches were proposed for expressing not only the test bench but parts of the design themselves in a behavioral style [24, 28, 94]. However, it is hard to distinguish the parts of the description meant to be interpreted behaviorally, from those which are meant to serve as static elaboration. As a result such behavioral representations are used in highly stylized fashions. SystemVerilog [3], the successor to Verilog formalized some of these distinctions by introducing the concept of generate blocks to be used explicitly for static elaboration.

The hardware description language Lava [18], an embedded domain-specific language in Haskell, made this distinction precise, by expressing all higher-level functional operators as circuit connections. At some level, Lava can be viewed as a meta-programming layer on top of the previous gate-level representation.

Cycle-accurate simulation of RTL models is not always necessary. Practically, it is often of value to be able to significantly speed up simulation in exchange for small semantic infidelities. Popular commercial products like Verilator [92] and Carbon [1] do just this. However, the resulting performance is still often several orders of magnitude slower than natural software implementations of the same algorithm.

1.4.2 C-based Behavioral Synthesis

Many consider hardware description languages like Verilog and VHDL to be too low-level. One proposed solution to ease the burden of hardware design is to generate

hardware from familiar software languages, *e.g.,* C or Java. These Electronic System Level (ESL) representations generally take the control-data flow graphs and through a series of transformations, optimize the result and directly implement it as a circuit [46, 48, 57, 75, 91]. Systems like CatapultC [62], HandelC [23], Pico Platform [86], or AutoPilot [13] have been effective at generating some forms of hardware from C code. Assuming the appropriate state elements can be extracted, these can be quite efficient in the context of static schedules. However, generating an efficient design in the context of dynamic choice can be very hard, if not impossible [5].

1.4.3 Implementation Agnostic Parallel Models

There are several parallel computation models whose semantics are agnostic to implementation in hardware or software. In principle, any of these can provide a basis for hardware-software codesign. Threads and locks are used extensively in parallel programming and also form the basis of SystemC [59] – a popular language for modeling embedded systems. However, SystemC has the same problem as other C-like language in generating good hardware, in that only highly restrictive idioms are efficiently implementable.

Dynamic Dataflow models, both at macro-levels (Kahn [54]) and fine-grained levels (Dennis [35], Arvind [10]), provide many attractive properties but abstract away important resource-level issues that are required to express efficient hardware and software. Nevertheless dataflow models where the rates at which each node works are specified statically have been used successfully in signal processing applications [58]. However, such systems extend become inefficient in the context of conditional operations. The Liquid Metal Project [53] extends StreamIt [89], one such static dataflow language aimed at parallel software, to make use of hardware. It leverages the type system, an extension of the Java type system, to annotate which aspects can be implemented in hardware and which in software.

In contrast to asynchronous or untimed dataflow models mentioned earlier, synchronous dataflow offers a model of concurrency based on synchronous clocks. It is the basis of for a number of programming languages, *e.g.,* LUSTRE [25], Esterel [17],

Rapide [60], Polysynchrony [87] SyncCharts [6], and SHIM [40]. All of these languages are used in mixed hardware-software designs. Though still synchronous like RTL models, they represent a clear improvement over behavioral synthesis of RTL in their semantics and implementation. Edwards [39, 74] and Berry [16] have presented methods to generate hardware from such descriptions, but these efforts have yet to yield the high-quality hardware, predictability, and descriptional clarity needed to overtake the well understood RTL-based design.

1.4.4 Heterogeneous Simulation Frameworks

There are numerous systems that allow co-simulation of hardware and software modules. Such systems, which often suffer from both low simulation speeds and improperly specified semantics. Additionally they are typically not used for direct hardware or software synthesis.

Ptolemy [22] is a prime example of a heterogeneous modeling framework, which concentrates more on providing an infrastructure for modeling and verification, and less on the generation of efficient software; it does not address the synthesis of hardware at all. Metropolis [14], while related, has a radically different computational model and has been used quite effectively for hardware/software codesign, though primarily for validation and verification rather than to synthesize efficient hardware.

SystemC [59], a C++ class library, is the most popular language to model heterogeneous systems. The libraries provide great flexibility in specifying modules, but SystemC lacks clear compositional semantics, producing unpredictable behaviors when connecting modules. Synthesis of high-quality hardware from SystemC remains a challenge.

Matlab [2] and Simulink [69] generate production code for embedded processors as well as VHDL from a single algorithmic description. Simulink employs a customizable set of block libraries that allows the user to describe an algorithm by specifying the component interactions. Simulink does allow the user to specify modules, though the nature of the Matlab language is such that efficient synthesis of hardware would be susceptible to the same pitfalls as C-based tools. A weakness of any library-based

approach is the difficulty for users to specify new library modules.

1.4.5 Algorithmic Approaches to Design

One possible approach to embedded design is to automatically generate a design from a high-level algorithm. This can be very successful in contexts where the domain is well understood (*e.g.,* SPIRAL [88]). However, it does not generally apply to less well-understood and complex systems. Nurvitadhi et al. [68] have made some progress on a more general design class, taking a synchronous datapath and automatically pipelining it, automatically leveraging user-provided speculation mechanisms.

Another area of interest that can be solved algorithmically is the selection of an appropriate cut between hardware and software. Viewed as an optimization problem, designers can make a selection based on estimates of the cost-performance tradeoff. In certain restricted cases, the choice of implementation of an algorithm is sufficiently constrained that it becomes reasonable for an automated process [7, 27, 42] to be used in selecting the appropriate partitioning. This is especially effective in digital approximations where numeric errors from approximations must be analyzed.

All automated approaches try to avoid implementing and checking all cases by leveraging high-level knowledge to approximate the relevant parts of the exploration process. Such analysis should be used in a complementary manner to the hardware-software codesign as discussed in this thesis.

1.4.6 Application-Specific Programmable Processors

Another approach to the hardware-software codesign problem is to limit hardware efforts to the specialization of programmable processors [4, 49, 82, 93]. These system are a very limited form of hardware-software codesign as they find kernels or CISC instructions which can be accelerated by special-purpose hardware functional units. These units are either user-generated or automatically derived from the kernel implementation. The majority of attention in these approach is spent on processor issues and the associated compilation stack. This approach is attractive as it gives

some of the performance/power benefits of general hardware-software solutions while still appearing to be "standard" software. However, it fundamentally is unable to get the many orders of magnitude that more general-purpose hardware-software codesign solutions are able to achieve.

1.4.7 Single Specification Hardware/Software Approaches

One of the earliest efforts to do total-system design of hardware and software was SRI's Hierarchical Development Methodology (HDM) [64, 76] in the 1970s, with its SPECIfication and Assertion Language (SPECIAL). HDM is aimed at breaking the task of verifying the abstract high-level properties on real production-level system into small managable steps gradually refining from abstract machine to the real design. The expression of the state machines at each level is sufficiently abstract that the change from software to hardware does not change the abstraction methodology, requiring only the necessary limitations to be implementable in the appropriate substrate. HDM and SPECIAL were used in the design of SRI's Provably Secure Operating System and its underlying capability-based hardware [64, 65]. This approach is being revisited in a joint project of SRI and the University of Cambridge [66]. HDM was later extended to EHDM [81] and has led to many of SRI's subsequent formal methods systems, *e.g.,* PVS [70], SAL [15], and Yices [38].

1.4.8 Previous Work on Atomic Actions and Guards

.

BCL and BSV are most closely related to Chandy and Misra's UNITY programming language [26] whose execution model is virtually identical to ours, differing only in a few intra-atomic action constructors. That said, there are many other languages that use guards or atomic actions to describe distributed software systems, *e.g.,* Djisktra's Guarded Command Language [36], Hoare's Communicating Sequential Processes [50], and Lynch's IO Automata [61].

Another set of languages in the context of hardware employ vastly different tech-

niques than their software counterparts. Initially such descriptions were used for the purpose of precise specification of hard-to-verify hardware models such as cache coherence processors, *e.g.,* Dill's Murphi [37] system and Arvind and Shen's TRSs [11]. Such models focused on modeling the protocol, and not on actual synthesis.

Initial aspects of hardware synthesis from guarded atomic actions can be found Staunstrup's Synchronous Transactions [85], Sere's Action Systems [73]. These systems used basic processor pipelines to demonstrate their practicality. Staunstrup was able to demonstrate automatic synthesis; however, this synthesis was unable to reach the level of concurrency required to make the system practical.

In contrast, Arvind and Shen's TRS's [11] focused on more complex and more obviously parallel structures such as reorder buffers and cache-coherence protocols [83], represented using bounded guarded atomic actions. Hoe and Arvind then showed that such descriptions could be synthesized into highly concurrent and relatively efficient structures [52] by attempting to execute each individual rule in parallel each cycle. These were later refined by Esposito [43] and Rosenband [78] to allow more efficient results without compromising understandability.

This idea was commercialized and packaged into the hardware description language Bluespec SystemVerilog [20] and surrounding tool chain and infrastructure [19]. Significant work has gone into making the hardware generation efficient; it has been shown that compiled BSV is as efficient as hard-written RTL [9]. BSV has been used in many contexts to develop large hardware systems such as processor designs [31], video decoders [45], kernel accelerators [33], cache coherence engines [34], hardware-based processor simulators [41, 72], and wireless basebands [67]. These have leveraged Guarded Atomic Actions to reduce code size, increase modularity and design flexibility, and reduce design time. These projects have shown that Guarded Atomic Actions are good abstraction for hardware designs.

The rest of this thesis discusses the Bluespec Codesign Language.

Chapter 2

BCL: A Language of Guarded Atomic Actions

This chapter introduces the semantics and fundamental properties of Bluespec Codesign Language (BCL). BCL can be considered as an extension of Bluespec SystemVerilog (BSV) [20]. BSV is a statically-typed language with many useful language features that are not meaningful after the early part of the compilation (*i.e.,* after static elaboration). To simplify our discussion of the semantics, we consider BCL [32], a language roughly corresponding to BSV programs after type checking and instantiation of modules.

In BCL, behavior is described using guarded atomic actions (GAAs) or *rules* [51]. Each rule specifies a state transition (its *body*) on the state of the system and a predicate (a *guard*) that must be valid before this rule can be executed, *i.e.,* the state transformation can take place. One executes a program by randomly selecting a rule whose predicate is valid and executing its body. Any possible sequencing of rules is valid; the implementation is responsible for determining which rules are selected and executed.

The grammar for BCL is given in Figure 2-1. Most of the grammar is standard; we discuss only the novel parts of the language and explain the language via an illustrative example in Section 2.1. A BCL program consists of a name, a set of a modules, and a set of rules. Each BCL module consists of a set of (sub)modules and

program ::=	**Program** *name* [*m*] [**Rule** *R*: *a*]	// A list of Modules and Rules
m ::=	[**Register** *r* (v_0)]	// Reg with initial values
‖	**Module** *name*	
	[*m*]	// Submodules
	[**ActMeth** *g* = $\lambda x.a$]	//Action method
	[**ValMeth** *f* = $\lambda x.e$]	//Value method
v ::=	*c*	// Constant Value
‖	*t*	// Variable Reference
a ::=	*r* **:=** *e*	// Register update
‖	**if** *e* **then** *a*	// Conditional action
‖	*a* | *a*	// Parallel composition
‖	*a* **;** *a*	// Sequential composition
‖	*a* **when** *e*	// Guarded action
‖	(*t* = *e* **in** *a*)	// Let action
‖	**loop** *e a*	// Loop action
‖	**localGuard** *a*	// Localized guard
‖	*m.g*(*e*)	// Action method call g of m
e ::=	*r*	// Register Read
‖	*c*	// Constant Value
‖	*t*	// Variable Reference
‖	*e op e*	// Primitive Operation
‖	*e* **?** *e* **:** *e*	// Conditional Expression
‖	*e* **when** *e*	// Guarded Expression
‖	(*t* = *e* **in** *e*)	// Let Expression
‖	*m.f*(*e*)	// Value method call f of m
op ::=	&& | || | ...	// Primitive operations

Figure 2-1: Grammar of BCL. For simplicity we will assume all module and method names are unique.

sets of *action methods* and *value methods* which are called by methods of the enclosing module or rules in the top-level program. **Register** is a primitive module with special syntax for calling its read and write methods. Though rules are allowed at all levels of the module hierarchy in BCL, we restrict BCL such that all rules are at the top-level to simplify our discussion of scheduling. This can be done programmatically by repeatedly replacing a rule in a submodule with an action method containing the body of the rule and a rule that calls just that action method.

We refer to the collection of all registers in a BCL program as its *state*. The evaluation of a BCL rule produces a new value for the state and a boolean guard value which specifies if the state change is permitted. Every rule in BCL is deterministic in the sense that the guard value and state change computed when evaluating a rule are a function of the current state. The execution of a BCL program can be described as follows:

1. Choose a rule R to execute.
2. Evaluate the new state and guard value for rule R on the current state.
3. If the guard is true, update the state with the new value.
4. Repeat Step 1.

Since this procedure involves a nondeterministic choice and the choice potentially affects the observed behaviors, our BCL program is more like a specification as opposed to an implementation. To obtain an effective implementation we selectively restrict the model to limit nondeterminism and introduce a notion of fairness in rule selection. As we discuss in Chapter 5, in the case of synchronous hardware we usually want to execute as many rules as possible concurrently, whereas in software we construct long chains of rules to maximize locality.

```
int lpm(IPA ipa){
  int p;
  p = RAM [rootTableBase + ipa[31:16]];
  if (isLeaf(p)){
     return p;
  }
  p = RAM [p + ipa [15:8]];
  if (isLeaf(p)){
     return p;
  }
  p = RAM [p + ipa [7:0]];
  return p; // must be a leaf
}
```

Figure 2-2: Sequential C code for longest prefix match example

2.1 Example: Longest Prefix Match

We use the example of the Longest Prefix Match module in a high-speed router to illustrate the essential concepts in BCL. This module is for determining to which physical output should a particular packet be routed based on its IPv4 destination address. The Longest Prefix Match is based on a routing table that consists of a set of IP address prefixes, each associated with an output port. Since more than one prefix can match an incoming packet, we choose the output corresponding to the longest matching prefix.

Since these routing tables are updated infrequently, it is possible for us to pre-process the prefixes for efficient lookup and update the hardware memories when necessary. The most natural way of expressing this prefix is with a flat table of size 2^{32}. However this is not feasible due to cost and power reasons. To reduce the cost, we exploit the tree-like structure of the table to produce a multi-lookup table. Now to do a lookup we start with a prefix of the IP and get either a result (a leaf) or a pointer back into the table to which we add the next 8-bit part of the IP address to from a new memory address to lookup. In C, this would look as the code in Figure 2-2.

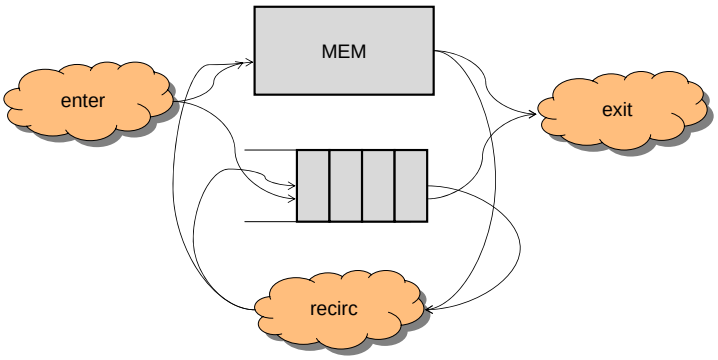
This drastically cuts down on total memory size, but now requires that in the worst case three sequential memory lookups, each of which may take k clock cycles. This

means in the worst case it may take $3k$ cycles to process a request that, given standard hardware components is too slow to meet the required packet processing rate. A way to overcome this is to overlap the computation of multiple packets because these computations are independent of each other. To do so we use a pipelined memory that is capable of handling a new request every cycle, though the latency of results is still k cycles.

We use a circular pipeline organization, shown in Figure 2-3, to handle multiple concurrent requests. We assume that the memory module (`mem`) may take an arbitrary amount of time to produce a result and will internally buffer waiting responses. The program has an input FIFO `inQ` to hold incoming requests, an internal FIFO `fifo` to hold outstanding requests while a memory reference is in progress. Once a lookup request is completed, the message leaves the program via the FIFO `outQ`.

The program has three rules `enter`, `recirc`, and `exit`. The `enter` rule enters a new IP request into the pipeline from `inQ`. In parallel, it enqueues a new memory request and places the IP residual into `fifo`. While there are no explicit guards, this rule is guarded by the guards implicit in the methods it calls. Therefore, the guard of the `enter` rule is simply the conjunction of the guards of the `fifo.enq` and `mem.req` methods. These guards may represent, for example, that the `fifo` or `mem` is full and cannot handle a new request.

The `recirc` rule handles the case when a response from memory results in a pointer, requiring another request. It is guarded by the condition that the memory response is not the final result. In parallel, the body of the rule accepts the result from memory, makes the new memory request and in sequence dequeues the old IP address residual and enqueues the new one. It is very important that this sequencing occurs, and we not use parallel composition. For this rule to execute the guards on both method calls must be valid simultaneously. This means the `fifo` must both have space and not be empty. Note that this is not possible if we have a one-element FIFO. A definition of such a FIFO is given in Figure 2-4. Consequently `fifo.deq | fifo.enq(x)` will never cause a change in the state of one-element FIFO. However, this issue goes away with sequential composition as the `deq` call will make




```
Program IPLookup
    Module mem ...
    Module fifo ...
    Module inQ ...
    Module outQ ...

    Rule enter:
        x = inQ.first() in
        inQ.deq()
        fifo.enq(x) |
        mem.req(addr(x))

    Rule recirc:
        x = mem.res() in
        y = fifo.first() in
        (mem.resAccept()  |
         mem.req(addr(x)) |
         (fifo.deq();
          fifo.enq(f2(x,y)))
             when  !isLeaf(x)

    Rule exit:
        x = mem.res() in
        y = fifo.first() in
        (mem.resAccept() |
         fifo.deq() |
           outQ.enq(f1(x,y)))
           when isLeaf(x)
```

Figure 2-3: Example: A Table-Lookup Program. Other modules implementations are given in Figure 2-4

```
Module fifo
  Register vf0 (false)
  Register  f0 (0)
  ActMeth enq(x) =
      (vf0 := true | f0 := x) when !vf0
  ActMeth deq() =
      (vf0 := false) when vf0
  ValMeth first() =
      f0 when vf0

Module mem
  Register r0 (0)
  Register r1 (0)
  ...
  Register rN (0)
  Module memfifo ...
  ActMeth req(x) =
      if (x = 0) memfifo.enq(r0) |
      if (x = 1) memfifo.enq(r1) |
      ...
      if (x = n) memfifo.enq(rN)
  ActMeth resAccept() =
      memfifo.deq()
  ValMeth res() =
      memfifo.first()
```

Figure 2-4: One-element FIFO and Naïve Memory Modules

the subsequent `enq` call valid to execute.

The `exit` rule deals with removing requests that are fulfilled by the most recent memory response. In parallel it accepts the memory response, dequeues the residual IP in `fifo` and enqueues the final result into the `outQ`. "`x=mem.res()`" and "`y=fifo.first()`" represent pure bindings of values being returned by the methods `mem.res()` and `fifo.first()`. The entire action is guarded by the condition that we found a leaf, `isLeaf(x)`. In addition to this guard are the guards embedded in the method calls themselves. For instance `fifo.deq` is guarded by the condition that there is an element in the queue.

Figure 2-4 includes an implementation of a one-element FIFO. Its interface has two action methods, `enq` and `deq`, and one value method `first`. It has a register `f0` to hold a data value and a one-bit register `vf0` to hold the valid bit for `f0`. The encoding of all methods is self-explanatory, but it is worth pointing out that all methods have guards represented by the **when** clauses. The guards for `first` and `deq` signify that the FIFO is not empty while the guard for `enq` signifies that the FIFO is not full.

We have not shown an implementation of the actual memory module, only a naïve implementation that operates as a flat memory space with the FIFO `memfifo` to hold intermediate requests. All reasoning about the correctness remains the same for this design. The guard of `mem.req` indicates when it can accept a new request. The guards of value method `mem.res` and action method `mem.resAccept` would indicate when `mem` has a result available.

2.2 Semantics of Rule Execution in BCL

Having given some intuition about BCL in the context of our example, we now present the operational semantics of a rule execution using Structured Operational Semantics (SOS) style evaluation rules. The state S of a BCL program is the set of values in its registers. The evaluation of a rule results in a set of state updates U where the empty set represents no updates. In case of an ill-formed rule it is possible that multiple updates to the same register are specified. This causes a double update error

which we represent with the special update value DUE.

Our semantics (described in Figures 2-5, 2-6, and 2-7) builds the effect of a rule execution by composing the effects of its constituent actions and expressions. To do this compositionally, the evaluation of an expression that is not ready due to a failing guard must return a special not-ready result NR in lieu of its expected value. A similar remark applies to the evaluation of actions.

Each action rule specifies a list of register updates given an environment $\langle S, U, B \rangle$ where S represents the values of all the registers before the rule execution; U is a set of register-value pairs representing the state update; and B represents the locally-bound variables in scope in the action or expression. Initially, before we execute a rule, U and B are empty and S contains the value of all registers. The NR value can be stored in a binding, but *cannot* be assigned to a register. To read a rule in our semantics, the part over the bar represents the antecedent derivations, and the part under the bar, the conclusion, and $\Rightarrow$ represent evaluation of both actions and expressions. Thus we read the reg-update rule in Figure 2-6 as "If e returns v, a non-NR value in some context, then the action $r := e$ returns an update of r to the value v".

The semantic machinery is incomplete in the sense that there are cases where no rule may apply, for example, if one of the arguments to the op-rule is NR. Whenever such a situation occurs in the evaluation of an expression, we assume the NR value is returned. This keeps the semantics uncluttered without loss of precision. Similarly when no rule applies for evaluation of actions, the NR value is returned. For rule bodies, a NR value is interpreted as an empty U.

We now discuss some of the more interesting aspects of the BCL language in isolation.

2.2.1 Action Composition

The language provides two ways to compose actions together: *parallel composition* and *sequential composition*.

When two actions $A_1|A_2$ are composed in parallel they both observe the same

reg-read	$\langle S, U, B\rangle \vdash r \twoheadrightarrow (S{+}{+}U)(r)$
const	$\langle S, U, B\rangle \vdash c \Rightarrow \underline{c}$
variable	$\langle S, U, B\rangle \vdash t \Rightarrow B(t)$
op	$\dfrac{\langle S, U, B\rangle \vdash e_1 \Rightarrow v_1,\ v_1 \neq \text{NR} \quad \langle S, U, B\rangle \vdash e_2 \Rightarrow v_2,\ v_2 \neq \text{NR}}{\langle S, U, B\rangle \vdash e_1\ op\ e_2 \Rightarrow v_1\ \underline{op}\ v_2}$
tri-true	$\dfrac{\langle S, U, B\rangle \vdash e_1 \Rightarrow true,\ \langle S, U, B\rangle \vdash e_2 \Rightarrow v}{\langle S, U, B\rangle \vdash e_1\ \mathbf{?}\ e_2\ \mathbf{:}\ e_3 \Rightarrow v}$
tri-false	$\dfrac{\langle S, U, B\rangle \vdash e_1 \Rightarrow false,\ \langle S, U, B\rangle \vdash e_3 \Rightarrow v}{\langle S, U, B\rangle \vdash e_1\ \mathbf{?}\ e_2\ \mathbf{:}\ e_3 \Rightarrow v}$
e-when-true	$\dfrac{\langle S, U, B\rangle \vdash e_2 \Rightarrow true,\ \langle S, U, B\rangle \vdash e_1 \Rightarrow v}{\langle S, U, B\rangle \vdash e_1\ \mathbf{when}\ e_2 \Rightarrow v}$
e-when-false	$\dfrac{\langle S, U, B\rangle \vdash e_2 \Rightarrow false}{\langle S, U, B\rangle \vdash e_1\ \mathbf{when}\ e_2 \Rightarrow \text{NR}}$
e-let-sub	$\dfrac{\langle S, U, B\rangle \vdash e_1 \Rightarrow v_1, \langle S, U, B[v_1/t]\rangle \vdash e_2 \Rightarrow v_2}{\langle S, U, B\rangle \vdash t = e_1\ \mathbf{in}\ e_2 \Rightarrow v_2}$
e-meth-call	$\dfrac{\langle S, U, B\rangle \vdash e \Rightarrow v,\ v \neq \text{NR}, \quad m.f = \langle \lambda t.e_b\rangle,\ \langle S, U, B[v/t]\rangle \vdash e_b \Rightarrow v'}{\langle S, U, B\rangle \vdash m.f(e) \Rightarrow v'}$

Figure 2-5: Operational semantics of a BCL Expressions. When no rule applies the expression evaluates to NR

reg-update	$\dfrac{\langle S,U,B\rangle \vdash e \Rightarrow v,\ v \neq \text{NR}}{\langle S,U,B\rangle \vdash r := e \Rightarrow\ \{r \mapsto v\}}$
if-true	$\dfrac{\langle S,U,B\rangle \vdash e \Rightarrow true,\ \langle S,U,B\rangle \vdash a \Rightarrow U'}{\langle S,U,B\rangle \vdash \textbf{if } e \textbf{ then } a \Rightarrow U'}$
if-false	$\dfrac{\langle S,U,B\rangle \vdash e \Rightarrow false}{\langle S,U,B\rangle \vdash \textbf{if } e \textbf{ then } a \Rightarrow \{\}}$
a-when-true	$\dfrac{\langle S,U,B\rangle \vdash e \Rightarrow true,\ \langle S,U,B\rangle \vdash a \Rightarrow U'}{\langle S,U,B\rangle \vdash a \textbf{ when } e \Rightarrow U'}$
par	$\dfrac{\langle S,U,B\rangle \vdash a_1 \Rightarrow U_1, \langle S,U,B\rangle \vdash a_2 \Rightarrow U_2,\ U_1 \neq \text{NR}, U_2 \neq \text{NR}}{\langle S,U,B\rangle \vdash a_1 \mid a_2 \Rightarrow U_1 \uplus U_2}$
seq-DUE	$\dfrac{\langle S,U,B\rangle \vdash a_1 \Rightarrow \text{DUE},}{\langle S,U,B\rangle \vdash a_1 \textbf{ ; } a_2 \Rightarrow \text{DUE}}$
seq	$\dfrac{\langle S,U,B\rangle \vdash a_1 \Rightarrow U_1, U_1 \neq \text{NR}, U_1 \neq \text{DUE}\ \ \langle S,U{+}{+}U_1,B\rangle \vdash a_2 \Rightarrow U_2, U_2 \neq \text{NR}}{\langle S,U,B\rangle \vdash a_1 \textbf{ ; } a_2 \Rightarrow U_1{+}{+}U_2}$
a-let-sub	$\dfrac{\langle S,U,B\rangle \vdash e \Rightarrow v, \langle S,U,B[v/t]\rangle \vdash a \Rightarrow U'}{\langle S,U,B\rangle \vdash t = e \textbf{ in } a \Rightarrow U'}$
a-meth-call	$\dfrac{\langle S,U,B\rangle \vdash e \Rightarrow v,\ , v \neq \text{NR},\ m.g = \langle \lambda t.a\rangle,\ \langle S,U,B[v/t]\rangle \vdash a \Rightarrow U'}{\langle S,U,B\rangle \vdash m.g(e) \Rightarrow U'}$
a-loop-false	$\dfrac{\langle S,U,B\rangle \vdash e \Rightarrow false}{\langle S,U,B\rangle \vdash \textbf{loop } e\ a \Rightarrow \{\}}$
a-loop-true	$\dfrac{\langle S,U,B\rangle \vdash e \Rightarrow true\ , \langle S,U,B\rangle \vdash a \textbf{ ; loop } e\ a \Rightarrow U'}{\langle S,U,B\rangle \vdash \textbf{loop } e\ a \Rightarrow U'}$
a-localGuard-fail	$\dfrac{\langle S,U,B\rangle \vdash a \Rightarrow \text{NR}}{\langle S,U,B\rangle \vdash \textbf{localGuard } a \Rightarrow \{\}}$
a-localGuard-true	$\dfrac{\langle S,U,B\rangle \vdash a \Rightarrow U', U' \neq \text{NR}}{\langle S,U,B\rangle \vdash \textbf{localGuard } a \Rightarrow U'}$

Figure 2-6: Operational semantics of a BCL Actions. When no rule applies the action evaluates to NR. Rule bodies which evaluate to NR produce no state update.

Merge Functions:

$$
\begin{array}{rl}
L_1{+}{+}(\text{DUE}) & = \text{DUE} \\
L_1{+}{+}(L_2[v/t]) & = (L_1{+}{+}L_2)[v/t] \\
L_1{+}{+}\{\} & = L_1 \\
U_1 \uplus U_2 & = \text{DUE if } U_1 = \text{DUE or } U_2 = \text{DUE} \\
 & = \text{DUE if } \exists r.\{r \mapsto v_1\} \in U_1 \wedge \{r \mapsto v_2\} \in U_2 \\
 & \qquad \text{otherwise } U_1 \cup U_2 \\
\{\}(x) & = \bot \\
S[v/t](x) & = v \text{ if } t = x \text{ otherwise } S(x)
\end{array}
$$

Figure 2-7: Helper Functions for Operational Semantics

initial state and do not observe the effects of each other's actions. This corresponds closely to how two concurrent updates behave. Thus, the action $r_1 := r_2 \mid r_2 := r_1$ swaps the values in registers r_1 and r_2. Since all rules are determinate, there is never any ambiguity due to the order in which subactions complete. Actions composed in parallel should never update the same state; our operational semantics views a double update as a dynamic error. Alternatively we could have treated the double update as a guard failure. Thus, any rule that would cause a double update would result in no state change.

Sequential composition is more in line with other languages with atomic actions. Here, $A_1; A_2$ is the execution of A_1 followed by A_2. A_2 observes the full effect of A_1 but no other action can observe A_1's updates without also observing A_2's updates.

In BSV, only parallel composition is allowed because sequential composition severely complicates the hardware compilation. BSV provides several workarounds in the form of primitive state (*e.g.,* RWires) with internal combinational paths between its methods to overcome this lack of sequential composition. In BCL we include both types of action composition.

2.2.2 Conditional versus Guarded Actions

BCL has both conditional actions (**if**s) as well as guarded actions (**when**s). These are similar as they both restrict the evaluation of an action based on some condition. The difference is their scope: conditional actions have only a local effect whereas

A.1	$(a_1$ **when** $p) \mid a_2$	$\equiv$	$(a_1 \mid a_2)$ **when** p
A.2	$a_1 \mid (a_2$ **when** $p)$	$\equiv$	$(a_1 \mid a_2)$ **when** p
A.3	$(a_1$ **when** $p)\ ;\ a_2$	$\equiv$	$(a_1\ ;\ a_2)$ **when** p
A.4	**if** $(e$ **when** $p)$ **then** a	$\equiv$	(**if** e **then** a) **when** p
A.5	**if** e **then** $(a$ **when** $p)$	$\equiv$	(**if** e **then** a) **when** $(p \vee \neg e)$
A.6	$(a$ **when** $p)$ **when** q	$\equiv$	a **when** $(p \wedge q)$
A.7	$r := (e$ **when** $p)$	$\equiv$	$(r := e)$ **when** p
A.8	$m.f(e$ **when** $p)$	$\equiv$	$m.f(e)$ **when** p
A.9	$m.g(e$ **when** $p)$	$\equiv$	$m.g(e)$ **when** p
A.10	**localGuard** $(a$ **when** $p)$	$\equiv$	**if** p **then localGuard** (a)
	p has no internal guards		
A.11	**Rule** n **if** p **then** a	$\equiv$	**Rule** n $(a$ **when** $p)$

Figure 2-8: When-Related Axioms on Actions

guarded actions have an effect on the entire action in which it is used. If an **if**'s predicate evaluates to false, then that action doesn't happen (produces no updates). If a **when**'s predicate is false, the subaction (and as a result the whole atomic action) is invalid. If we view a rule as a function from the original state to the new state, then **when**s characterize the partial applicability of the function. One of the best ways to understand the differences between **when**s and **if**s is to examine the axioms in Figure 2-8.

Axioms A.1 and A.2 collectively say that a guard on one action in a parallel composition affects all the other actions. Axioms A.3 says similar things about guards in a sequential composition. Axioms A.4 and A.5 state that guards in conditional actions are reflected only when the condition is true, but guards in the predicate of a condition are always evaluated. Axiom A.6 deals with merging **when**-clauses. A.7, A.8 and A.9 state that arguments of methods are used strictly and thus the value of the arguments must always be ready in a method call. Axiom A.10 tells us that we can convert a **when** to an **if** in the context of a **localGuard**. Axiom A.11 states that top-level **when**s in a rule can be treated as an **if** and vice versa.

In our operational semantics, we see the difference between **if** and **when** being manifested in the production of special value NR. Consider the rule e-when-false. When the guard fails, the entire expression results in NR. All rules explicitly forbid the use of the NR value. As such if an expression or action ever needs to make use of

NR it gets "stuck" and fails, evaluating to NR itself. When a rule body is evaluated as NR, the rule produces no state update.

In BSV, the axioms of Figure 2-8 are used by the compiler to lift all **when**s to the top of a rule. This results in the compilation of more efficient hardware. Unfortunately with the sequential connective, this lifting cannot be done in general. We need an axiom of the following sort:

$$a_1 \ ; \ (a_2 \ \textbf{when} \ p) \quad \equiv \quad (a_1 \ ; \ a_2) \ \textbf{when} \ p' \\ \text{where } (p' \text{ is } p \text{ after } a_1)$$

The problem with the above axiom is that in general there is no way to evaluate p' statically. As a result BCL must deal with **when**s interspersed throughout the rules.

2.2.3 Looping Constructs

The **loop** action operates as a "while" loop in a standard imperative language. We execute the loop body, repeated in sequence until the loop predicate returns false. We can always safely unroll a **loop** action according to the rule:

```
loop e a = if e then (a ; loop e a)
```

The BSV compiler allows only those loops that can be unrolled at compile time. Compilation fails if the unrolling procedure does not terminate. In BCL, loops are unrolled dynamically.

Suppose the unrolling of **loop** e a produces the sequence of actions $a_1; a_2; ...; a_n$. According to the semantics of sequential composition, this sequence of actions implies that a guard failure in any action a_i causes the entire action to fail. This is the precise meaning of **loop** e a.

2.2.4 Local Guard

In practice, it is often useful to be able to bound the scope of guards to a fixed action, especially in the expression of schedules as discussed in Chapter 5. In this semantics,

localGuard a operates exactly like a in the abscence of guard failure. If a would fail, then **localGuard** a causes no state updates.

2.2.5 Method Calls

The semantics of method calls have a large impact on efficiency of programs and on the impact of modularization. For instance, the semantically simplest solution is for methods to have inlining semantics. This means that adding or removing a module boundary has no impact on the behavior of the program. However, this notably increases the cost of hardware implementation as a combinational path from the calling rule's logic to the method's implementation and back before execution can happen. It is extremely complicated to implement this semantic choice in the context of a limited number of methods, where different rules must share the same implementation.

As such, we may want to increase the strictness of guards to ease the implementation burden. This will necessarily make the addition/removal of a modular boundary change the behavior; this is not an unreasonable decision as modules represent some notion of resources, and changing resources should change the behavior. Our choice of restrictions will have to balance the efficiency and implementation concerns against the semantic cleanliness.

There are two restrictions we can make:

1. The guards of method arguments are applied strictly. To understand this, consider the following action:

   Action 1: `m.g(e when False)`
   where `m.g` is λ `x.if p then r := (x)`

   and the result of inlining the definition of `m.g`:

   Action 2: `if p then r := (x when False)`

   Under this restriction Action 1 is never valid. However, Action 2 is valid if p is false. This restriction corresponds to a call-by-value execution. Such semantic

changes due to language-level abstractions (*e.g.,* functions) are very common in software language (*e.g.,* C, Java, and SML).

2. The argument of a method does not affect the guard. Under this semantic interpretation guards can be evaluated regardless of which particular rule is calling them drastically simplifying the implementation. This restriction is used by the BSV compiler for just this reason. However, this causes unintuitive restrictions. Consider the method:

   ActMeth g = λ `(p,x).`**if** `p` **then** `fifo1.enq(x)` |
   if `!p` **then** `fifo2.enq(x))`

   This method is a switch which enqueues requests into the appropriate FIFO. The enqueue method for each FIFO is guarded by the fact that there is space to place the new item. Given this restriction that we cannot use any of the input for the guard value, this method's guard is the intersection of both the guards of `fifo1` and `fifo2` which means that if *either* FIFO is full this method cannot be called, even if we wish to enqueue into the other FIFO. This is likely not what the designer is expecting.

We choose to keep the first restriction and not the second restriction in BCL, as the second restriction's unintuitiveness is highly exacerbated in the context of sequential composition and loops. In contrast, most arguments are used unconditionally in methods and as such the first restriction has small practical downside while still giving notable implementation simplification.

2.2.6 Notation of Rule Execution

Let $\mathcal{R}_P = \{R_1, R_2, ..., R_N\}$ be the set of rules of Program P. We write $s \xrightarrow{R} s'$, where $R \in \mathcal{R}$, to denote that application of rule R transforms the state s to s'.

An execution σ of a program is a sequence of such rule applications and is written as:

$$s_0 \overset{R_{i_1}}{\rightarrow} s_1 \overset{R_{i_2}}{\rightarrow} s_2 ... \overset{R_{i_n}}{\rightarrow} s_n$$

or simply as: $s \overset{\sigma}{\twoheadrightarrow} s'$.

A program is characterized by the set of its executions from a valid initial state.

2.3 A BCL Program as a State Transition System

So far we have described how a rule transforms the state of a BCL program. To give complete operational semantics, we also need to specify how a rule is chosen at each step of the execution. Since the choice is nondeterministic, the same BCL program can have more than one outcome or behavior. In order to characterize the range of permitted behaviors, we associate a state transition system with each program:

Definition 1 (**State Transition System** of Program P)**.** Each BCL program P is modeled by a *state transition system* given by a triple of the form:

$$(S, S_0, \longrightarrow)$$

where S is the set of *states* associated with program P; $S_0 \subseteq S$ is the set of states corresponding to initial configurations of P; and $\longrightarrow_{\mathcal{S}} \subseteq S \times S$ is the transition relation, defined such that $(s, s') \in \longrightarrow_{\mathcal{S}}$ if and only if there exists some rule R in P whose execution takes the state s to s'. In addition, we write $\twoheadrightarrow$ to denote the *reflexive transitive closure* of $\longrightarrow$. ■

Notice that state transition systems are closed; there is no notion of an outside world. We show later that the BCL provides enough flexibility to model any reasonable context in which we would want to place an "open" BCL program. We discuss these notions in the context of closed systems.

The question naturally arises as to what aspects of a closed BCL program is observable. Abstractly, we can think of program P as a box with two buttons: "Start" and "Display". Pressing the "Start" button initializes the internal state and starts the execution. Once started, we can press "Display" at which point the machine pauses execution, and displays the current program state. Between subsequent presses of

"Display" any non-zero number of rule applications can occur. This leads to the following definition of *observation*:

Definition 2 (**Observation** of a Program P)**.** An *observation* of Program P modeled by $(S, S_0, \longrightarrow)$ is a sequence of states $\sigma = s_0, s_1, s_2, ..., s_N$ such that $s_o \in S_0$ and for all $0 \leq i < N$, $s_i \twoheadrightarrow s_{i+1}$. ■

Note that we observe the effect of a rule execution only indirectly by observing the current state. As such, we cannot tell when an executed rule produces no state update (a *degenerate* execution) or how many rules have fired between two observations of the state. This means that it is perfectly acceptable for us to "skip" states in the one-rule-at-a-time understanding of an execution.

From a practical point of view sometimes it is not worth distinguishing between some observations of a program. Consider the following program P_1 with two registers `x` and `y` initialized to 0:

Program P_1

Register `x` (x_0)
Register `y` (y_0)

Rule `incX`: `x := x + 1`
Rule `incY`: `y := y + 1`

In this program there exists an execution corresponding to executing `incX` once which takes the state from $(\mathtt{x}, \mathtt{y}) = (0, 0)$ to $(1, 0)$. Similarly there is an execution corresponding to one execution of `incY` which takes the state from $(0, 0)$ to $(0, 1)$. Clearly these executions are different. However, we can bring them both to $(1, 1)$ by executing the appropriate rule. Whether we select one or the other rule does not matter from the point of view of observations.

In contrast, consider another program P_2 with the same registers `x` and `y` when $x_0 \neq y_0$:

Program P_2

Register `x` (x_0)
Register `y` (y_0)

Rule `copyX2Y: y := x`
Rule `copyY2X: x := y`

The first execution of one of these rules causes a transition to either state (x_0, x_0) or (y_0, y_0). Once such a transition has been made, the system can never leave that state. An implementation produces either (x_0, x_0), or (y_0, y_0), but not both. We characterize these different "behaviors" using the notion of joinability or *confluence*.

Definition 3 (A **Behavior** of Program P)**.** A *behavior* B, written as $\twoheadrightarrow_B$ of program P modeled by $\mathcal{S} = (S, S_0, \longrightarrow_{\mathcal{S}})$ is a *maximal* subset of $\twoheadrightarrow_{\mathcal{S}}$ such that each pair of executions in B are *joinable*. That is, $\forall s_0, s_1, s_2 \in S.(s_0 \twoheadrightarrow_B s_1) \wedge (s_0 \twoheadrightarrow_B s_2) \implies \exists s_3 \in S.(s_1 \twoheadrightarrow_B s_3) \wedge (s_2 \twoheadrightarrow_B s_3)$. B_P is the set of behaviors of a program P. ■

By this definition P_1 has one behavior corresponding the following set of executions:

$\{(0,0) \longrightarrow (0,1) \longrightarrow (1,1), (0,0) \longrightarrow (1,0) \longrightarrow (1,1), ...\}$

and P_2 has two behaviors corresponding to the following executions:

$\{(x_0, y_0) \longrightarrow (x_0, x_0), ...\}$

$\{(x_0, y_0) \longrightarrow (y_0, y_0), ...\}$

Definition 4 (**Live** Behavior)**.** B is a *live* behavior if $s_0 \twoheadrightarrow_B s_1$ implies there exists $s_2 \neq s_1$ such that $s_1 \twoheadrightarrow_B s_2$. ■

Definition 5 (**Terminal** State)**.** A terminal state of transition relation $\twoheadrightarrow$ is a state s such that $s \twoheadrightarrow s' \implies s = s'$. ■

Definition 6 (**Terminating** Behavior)**.** B is a *terminating* behavior if $s_0 \twoheadrightarrow_B s_1$ then there exists s_2 such that $s_1 \twoheadrightarrow s_2$ and s_2 is a terminal state of $\twoheadrightarrow_B$. ■

Lemma 1. *A behavior is either live or terminating, but not both.*

The behavior of program P_1 is *live* while both the behaviors of program P_2 are *terminating*. A program with multiple behaviors may have some of the behaviors which are live and some which are terminating.

Definition 7 (**Deterministic Program**). P is deterministic if it has exactly one behavior. Otherwise P is nondeterministic. ■

According to this definition program P_1 is deterministic while program P_2 is nondeterministic. Nondeterminism is not a sign that a program is incorrect or buggy. In fact, nondeterminism is essential if one wants to represent higher-level requirements of a program. For instance, consider the description of a speculative microprocessor. The choice of whether to speculatively fetch a new instruction or resolve the execution of an outstanding branch instruction is an integral part of the processor specification. One would like to be able to express the idea that the correctness of description does not rely on how many instructions execute on the wrong path as long as the processor resumes executing instructions on the correct path in some finite number of steps.

However, because designers do not see the exhaustive set of executions of a BCL program it is possible that a latent bug will not be found. For instance, in implementations of our speculative processor we may never speculatively fetch an instruction; execution of this implementation gives us no guarantee that the speculation we specified in the BCL program was done correctly. We discuss a more complete guarantee of correctness in Chapter 8.

2.4 Program Equivalence and Ordering

It is important to know if any two programs are equivalent. This concept is necessary both in terms of doing refinements and implementation. In the context of nondeterminism, it is possible for a program P to be "smaller" than another program P' in that it exhibits fewer transitions. We define these concepts using the natural partial ordering induced by the relation $\twoheadrightarrow$.

Definition 8 (Program Ordering). Consider Program P modeled by $(S, S_0, \longrightarrow_{\mathcal{S}})$ and Program P' modeled by $(S', S'_0, \longrightarrow_{\mathcal{S}'})$. P is less than P', *i.e.,* $P \sqsubseteq P'$, if and only if $(S = S') \wedge (S_0 = S'_0) \wedge (\twoheadrightarrow_{\mathcal{S}} \subseteq \twoheadrightarrow_{\mathcal{S}'})$ ■

Definition 9 (Program Equivalence). Program P and P' are *equivalent* when $(P \sqsubseteq P') \wedge$

$(P' \sqsubseteq P)$. ■

It is common to want to restrict the executions allowed in a program. This may be a refinement to remove undesirable behavior or to improve efficiency in implementation.

Definition 10 (Behavioral Restriction)**.** Program P, modeled by $\mathcal{S} = (S, S_0, \longrightarrow_{\mathcal{S}})$ is a *behavioral restriction* of P', modeled by $\mathcal{S}' = (S, S_0, \longrightarrow'_{\mathcal{S}})$ if and only if: $B_P \subseteq B_{P'}$. ■

To understand this consider the following program P_3 in relation to P_2.

Program P_3

Register `x` (x_0)
Register `y` (y_0)

Rule `copyX2Y: y := x`

In this program we have only one behavior which moves from (x_0, y_0) to (x_0, x_0). This new program has only one of the behaviors of P_2.

When implementing a program P, it is common for the desired final result P' to approximate the program in that some transitions may not be implemented, *i.e.,* $P' \sqsubseteq P$. This may remove some behaviors entirely, which is reasonable as we rarely want all behaviors in practice. However, for an approximation to be complete, it should not get stuck unexpectedly; that is we do not have unexpected terminal states.

Definition 11 (Complete Approximation of Program P)**.** Program P' is a *complete approximation* P if and only if: $P' \sqsubseteq P$, and for all states s, if s is a terminal state of P' it is also a terminal state of P. ■

To understand this definition concretely, consider the following complete approximation of P_1, which does two rule executions at a time:

Program P_4

Register `x` (x_0)
Register `y` (y_0)

Rule `incXY`:
```
y := y + 1 |
x := x + 1
```

While P_1 has many executions, allowing us to increment `x` and `y` each by one repeated in any way, allowing us to reach any natural number value for `x` and `y`, this program has only one of those executions:

$\{(0,0) \longrightarrow (1,1) \longrightarrow (2,2) \longrightarrow (3,3) \longrightarrow ...\}$

Essentially, if P' is a complete approximation of P that P' cannot stop (*i.e.,* reach a terminal state) before P does.

2.4.1 Derived Rules

Various interesting equivalence-preserving transformations of a BCL program involve the modification of rules. The following definitions are motivated by these program transformations. We use these definitions in our scheduling discussion in Chapter 5.

Definition 12 (**Null** Rule)**.** A rule R is said to be a null rule if it never changes the state of a program, *i.e.,* $\forall s \in S.\ (s \xrightarrow{R} s)$. ■

Definition 13 (**Restricted** Rule)**.** A rule R' is a *restriction* of rule R if R' can change state only when R can, *i.e.,* $\forall s, s' \in S.\ s \neq s' \wedge (s \xrightarrow{R'} s') \implies (s \xrightarrow{R} s')$. ■

It is useful to know whether two rules are mutually exclusive, that is, there is no state from which both rules produce nondegenerate executions.

Definition 14 (**Mutually Exclusive** Rules)**.** Two rules R_1 and R_2 are mutually exclusive when $\forall s.(s \xrightarrow{R_1} s) \vee (s \xrightarrow{R_2} s)$. ■

It is sometimes useful to partition a rule R into multiple rules that can directly mimic a single execution of R.

Definition 15 (**Partitioned** Rules)**.** Rules R_1 and R_2 are said to be a partition of rule R if they are mutually exclusive and $\forall s \in S.(s \xrightarrow{R} s') \implies ((s \xrightarrow{R_1} s') \vee (s \xrightarrow{R_2} s'))$. ■

A compound rule of R_1 and R_2 is a rule R_{12} that behaves as if R_1 is applied and then R_2 is applied. Choosing to execute R_{12} is akin to making the decision to execute R_1 and then R_2 as a single step.

Definition 16 (**Compound** Rule)**.** Rule R_{12} is said to be a compound rule of rules R_1 and R_2 if:

$\forall s_i, s' \in S.((s \xrightarrow{R_{12}} s') \iff \exists s_t.(s \xrightarrow{R_1} s_t \xrightarrow{R_2} s'))$ ■

Definition 17 (**Derived** Rule of P)**.** A rule R is said to be a derived rule of program P if for every state s, if $s \xrightarrow{R} s'$, then the same transition can be made by a sequence of rule executions in $\mathcal{R}_P - \{R\}$ (*i.e.,* $s \twoheadrightarrow s'$). ■

Intuitively, a derived rule of a program P is a rule that always can be emulated with the rules already in P. Although they may change our model, they do not change the executions that characterize our model.

Lemma 2. *A null rule is a derived rule of program P* □

Lemma 3. *A restriction of rule $R \in \mathcal{R}_P$ is a derived rule of program P* □

Lemma 4. *Each partitioned rule R_1, R_2 of rule $R \in \mathcal{R}_P$ is a derived rule of program P* □

Lemma 5. *A sequenced rule of two rules $R_1, R_2 \in \mathcal{R}_P$ is a derived rule of program P* □

Theorem 1 (Equality of Programs under Derived Rules)**.** *Adding a derived rule R_d to program P_1 results in an equivalent program P_2.*
Corollary: Removing a rule R_d of Program P_2 which is a derived rule of P_1, the program after removing R_d is an equivalent program. □

Chapter 3

A Rule-level Interpreter of BCL

The operational semantics in Chapter 2 specifies how a rule in a BCL program transforms the state of the system described by that program. The semantics includes the possibility that the guard of a rule may evaluate to false, and thus the application of the rule may not change the state of the system. In this chapter, for each rule R we describe a method to compute the function f_R such that:

$$f_R : State \rightarrow (Boolean, State)$$

where $f_R(s)$ produces either (True, s') consistent with the operational semantics presented in Chapter 2, *i.e.,* $s \twoheadrightarrow s'$, or (False, "don't care") in case the operational semantics dictates that rule R is not applicable in state s, *i.e.,* $s \longrightarrow s$.

The set of f_R functions of a program is useful in understanding both the hardware and software compilation of BCL. It also serves as a starting point for the verification of BCL programs. We use normal-order typed λ-calculus with `let` blocks to describe these functions.

3.1 Functionalization of BCL: The GCD Example

Before we discuss the translation of rules into λ-expressions, let us consider the GCD program shown in Figure 3-1.

```
Program GCD
  Rule start: gcd.req(0x17, 0x31)

  Rule subtract: gcd.subtract()

  Rule swap: gcd.swap()

  Module gcd
    Register x (0)
    Register y (0)

    ActMeth req = λ(a,b).
         (x := a | y := b)
              when (x == 0 && y == 0)

    ValMeth resp = λ().
         (x) when (x != 0 && y == 0)

    ActMeth getResp() = λ().
         (x := 0) when (x != 0 && y == 0)

    ActMeth subtract = λ().
       (x := x-y)
          when (x >= y) && (y != 0)

    ActMeth swap = λ().
       (x := y | y := x)))
          when (x < y)
```

Figure 3-1: GCD Program in BCL

The program uses Euclid's algorithm to compute GCD. It repeatedly applies methods `subtract` and `swap` to the values in registers `x` and `y` until the register `y` holds the value 0 and register `x` holds the answer. The computation is started by invoking the method `req`, which sets `x` and `y` to the initial values `a` and `b` respectively. A computation can be started only when both `x` and `y` are zero. The `resp` method lets us read the answer when we have finished computing it (*i.e.,* when `y` is 0). The action method `getResp` resets the GCD module for a new request once the answer has been computed by setting `x` to 0.

The state of this program can be represented in Haskell-like syntax as the following data structure:

```
data State = State{
  x :: Bit 32,
  y :: Bit 32
}
```

We can create a new state, where the register `x` holds the value `0x17` and register `y` holds the value `0x31` by writing `State{x=0x17,y=0x31}`. To extract a field we use the familiar dot notation, where `s.x` refers to the `x` field of `s`.

Often we wish to construct a new state where most of the field values are the same as another state value. In these cases it is convenient to express the new state as an update of the value. For instance, given a state value `s` with fields `x` and `y`, the new state value `s{x=v}` has an `x` field with value `v` and the `y` field with `s`'s `y` field value, `s.y`, *i.e.,* `State{x=v,y=s.y}`.

Functions corresponding to the three rules in our GCD example may be represented as follows:

```
rule_start =     λs.((s.x = 0) ∧ (s.y = 0), State{x=0x17, y=0x31})
rule_subtract =  λs.((s.x ≥ s.y) ∧ (s.y ≠ 0), s{x= s.x - s.y })
rule_swap =      λs.((s.x < s.y), State{x=s.y, y=s.x})
```

Each rule is represented by a function from the state of the system to a tuple containing a boolean guard predicate and a new state. For example, the swap rule is

valid to execute when the value of the register `x` is less than the value of the register `y` in the state that it is passed in as an argument and the new state has the value of `x` as the old value of `y` and the value of `y` is the old value of `x`.

We revisit this example again after explaining how these functions are generated from the original BCL program. As we explain below, we choose a slightly more complicated representation of the state to keep the translation scheme simple.

3.2 Translation of BCL to λ-Calculus

We define the construction of the λ expressions corresponding to our rules in a syntax-directed and compositional way. The main challenge in this arises when we consider the parallel composition of actions.

Consider the translation of actions $x := y$ and $y := x$ that turn into functions `f1 = λs.(True, s{x=s.y})` and `f2 = λs.(True, s{y=s.x})` respectively. We can model the sequential composition by composing the two functions as:

```
λs0.let (g1, s1) = f1 s0 in
      let (g2, s2) = f2 s1 in
       (g1 ∧ g2, s2)
```

In contrast, for parallel composition as we saw in the operational semantics, we have to take the least upper bound of state changes dictated by `f1` and `f2`. One way to model this is by first attaching a boolean (bit value) to each register in our state to indicate whether it has been modified or not, and then building the new state based on which elements have been modified in each function. In case both functions have modified the same state, we need to generate a double-update error. In this regard our state becomes a bounded representation of the update list U value in our operational semantics. It is worth noting that we cannot remove the boolean guard value, as we must distinguish between NR and the empty update {}.

Thus we change the default representation of state so that registers are modeled, not just as a value, but as a structure with a modified bit and a value. Thus the state of our GCD example is now:

```
data State = State{
  x :: Reg (Bit 32)
  y :: Reg (Bit 32)
}
data Reg n = Reg {
  modified :: Bool
  value    :: n
}
```

With this change the three functions from the GCD now look like the following:

```
   rule_start=λs.(s.x.value = 0 ∧ s.y.value = 0,
                  State{x=Reg{modified=True, value=0x17},
                        y=Reg{modified=True, value=0x31}})
rule_subtract=λs.(s.x.value ≥ s.y.value ∧ s.y.value ≠ 0,
                  s{x=Reg{modified=True, value=s.x.value-s.y.value}})
    rule_swap=λs.(s.x.value < s.y.value,
                  State{x=Reg{modified=True, value=s.y.value},
                        y=Reg{modified=True, value=s.x.value})
```

Now our translated rules also mark the bits of the registers that they modify. Notice that because these bits are initially false, we update them to true only; all registers that are not updated are unmentioned, *e.g.,* the register `y` in `rule_subtract`.

We could have added a wrapper function to add modified bits initialized as false at the beginning of a rule and one to the modified bits from the final state. However, this has minor benefit and would clutter the description; as such, we do not clutter our description with this.

With the change of state representation the corresponding representations for $x := y$ and $y := x$ respectively are:

```
f1 = λs.(True, s{x=Reg{modified=True, value=s.y}})
f2 = λs.(True, s{y=Reg{modified=True, value=s.x}})
```

Now we can implement the parallel composition by running the two actions on the initial state to generate two different copies of the state, and merge the resulting states together looking at the modified bits to determine which parts of the state

to select. To keep the invariant that the modified bits correspond to whether we modified the register or not, we need to first reset all the modified bits in our initial state before running the two parallel actions, and then after merging the two resulting states in parallel, merge back the original modified bits to the output state. To finish the translation we need to reintroduce all the missing modified bits from the initial state. The parallel swap action $x := y \mid y := x$ is represented in our translation as:

```
λs.let ns = newState s in
    let (g1, s1) = f1 ns in
     let (g2, s2) = f2 ns in
      let (g3, s12) = parMerge s1 s2
       in ((g1 ∧ g2 ∧ g3), (seqMerge s s12))
```

where `newState` clears the modified bits, `parMerge` selects the appropriate parts of the two states to keep for the parallel composition , and `seqMerge` sequentially copies the new updates back on the old state reintroducing the correct modified bits. Notice that `parMerge` returns a boolean guard, which represents that the parallel merge did not cause a double-update error. In this translation we interpret such errors are guard failures.

A full translation of BCL to the typed lambda calculus is given in Figures 3-2, 3-3, and 3-4. The only complication in understanding this translation is distinguishing the meta-syntax to construct the λ expressions (the *italic* font) from the concrete syntax of the λ expressions being constructed (the `fixed-width` font). It's also helpful to understand the signature of the translation procedure for each major syntactic category. We translate actions to functions taking the initial state and returning a tuple of a valid predicate and a new state. Similarly we translate expressions to functions taking the state and returning a tuple of a valid predicate and the translated expression. Methods are translated into functions that take the argument and return the appropriate translated action or expression; as these are functions themselves, translated methods can be thought of as taking two arguments. The translation of a program results in a definition for each method and rule in the system as well as the three merging functions `newState`, `parMerge`, and `seqMerge`. Notice that each of these functions calculate the value of the part of state corresponding to a register

TA :: BCL-Action →
  (Translated-State → (Bool, Translated-State))
TA ⟦ r := e_0 ⟧ =
 λs.(g, s{r = Reg{modified:True, value:e_1}})
  where (g, e_1) = (**TE** ⟦ e_0 ⟧) s
TA ⟦ **if** (p_0) a_0 ⟧ =
 λs.($g_1 \wedge (\neg p_1 \vee g_2)$, if p_1 then s' else s)
  where (g_1, p_1) = (**TE** ⟦ p_0 ⟧) s
  (g_2, s') = (**TA** ⟦ a_0 ⟧) s
TA ⟦ a_0 **when** e_0 ⟧ =
 λs.($g_1 \wedge g_2 \wedge e_1$, s')
  where (g_1, s') = (**TA** ⟦ a_0 ⟧) s
  (g_2, e_1) = (**TE** ⟦ e_0 ⟧) s
TA ⟦ a_1 | a_2 ⟧ =
 λs.let ns = newState s in
     let (g, s')=parMerge s_1 s_2 in ($g_1 \wedge g_2 \wedge$ g, seqMerge s s')
  where (g_1, s_1) = (**TA** ⟦ a_1 ⟧) (ns)
  (g_2, s_2) = (**TA** ⟦ a_2 ⟧) (ns)
TA ⟦ a1 ; a2 ⟧ =
 λs.($g_1 \wedge g_2$, s'')
  where (g_1, s') = (**TA** ⟦ a_1 ⟧) (newState s)
  (g_2, s'') = (**TA** ⟦ a_2 ⟧) (newState s')
TA ⟦ $t = e_0$ **in** a_0 ⟧ =
 λs.let tg = g_1 in (let tb = e_2 in (g_2, s'))
  where (g_1, e_1) = (**TE** ⟦ e_0 ⟧) s
  (g_2, s') = (**TA** ⟦ $a_0[(tb$ **when** $tg)/t]$ ⟧) s
  tg, tb are fresh names
TA ⟦ **loop** e_0 a_0 ⟧ =
 loopUntil (**TE** ⟦ e_0 ⟧) (**TA** ⟦ a_0 ⟧)
  where loopUntil = λfe.λfa.λs0.
   let (g1, e1) = fe s0 in
    let (g2, s1) = fa s0 in
     let (g3, s2) = loopUntil fe fa s1 in
      (g1 ∧ (¬ e1 ∨ (g2 ∧ g3)), if e1 then s2 else s0)
TA ⟦ **localGuard** a_0 ⟧ =
 λs.(True, if g then s' else s)
  where (g, s') = (**TA** ⟦ a_0 ⟧) s
TA ⟦ m.$g(e_0)$ ⟧ =
 λs.let (g, s') = meth_g e_1 s in ($g_0 \wedge$ g, s')
  where (g_0, e_1) = (**TE** ⟦ e_0 ⟧) s

Figure 3-2: Functionalization of Actions. `Fixed-width` text is concrete syntax of the λ-calculus expression

TE :: BCL-Expr →
  (Translated-State → (Bool, Translated-Expr))
TE ⟦ r ⟧ =λ`s.  (True, s.r.value)`
TE ⟦ c ⟧ =λ`s.  (True, c)`
TE ⟦ t ⟧ =λ`s.  (True, t)`
TE ⟦ e_1 **op** e_2 ⟧ =
 λ`s.(`$g_1 \wedge g_2$`,` fe_1 `op` fe_2`)`
  where (g_1, fe_1) = (**TE** ⟦ e_1 ⟧) `s`
  (g_2, fe_2) = (**TE** ⟦ e_2 ⟧) `s`
TE ⟦ e_b **?** e_t **:** e_f ⟧ =
 λ`s.(`g_b∧ `(if` fe_b `then` g_t `else` g_f`), if` fe_b `then` fe_t `else` fe_f`)`
  where (g_b, fe_b) = (**TE** ⟦ e_b ⟧) `s`
  (g_t, fe_t) = (**TE** ⟦ e_t ⟧) `s`
  (g_f, fe_f) = (**TE** ⟦ e_f ⟧) `s`
TE ⟦ e **when** e_g ⟧ =
 λ`s.(`$g_g \wedge fe_g \wedge g_e$`,` fe`)`
  where (g_e, fe) = (**TE** ⟦ e ⟧) `s`
  (g_g, fe_g) = (**TE** ⟦ e_g ⟧) `s`
TE ⟦ $t = e_1$ **in** e_2 ⟧ =
 λ`s.(let` tg `=` g_1 `in (let` tb `=` fe_1 `in (`g_2`,` fe_2`)))`
  where (g_1, fe_1) = (**TE** ⟦ e_1 ⟧) `s`
  (g_2, fe_2) = (**TE** ⟦ $e_2[(tb$ **when** $tg)/t]$ ⟧) `s`
  tg, tb are fresh names
TE ⟦ m.$f(e_0)$ ⟧ =
 λ`s.let (g,e) = meth_`f fe_0 `s in (`$g_0 \wedge$ `g, e)`
  where (g_0, fe_0) = (**TE** ⟦ e_0 ⟧) `s`

Figure 3-3: Functionalization of BCL Expressions. `Fixed-width` text is concrete syntax of the λ-calculus expression

```
TP :: BCL-Program → (λ-calculus declarations)
TP (Main ms rules) =
  (map TM ms)
  (map TR rules)
  seqMerge = λs0.λs1.
    (foreach Reg r.
     (λs.x{r=if s1.r.modified then s1.r else s0.r})) (True, s0)
  parMerge = λs0.λs1.
    (foreach Reg r.
     (λ(g, s).(g ∧ !(s0.r.modified ∧ s1.r.modified),
               s{r=if s0.r.modified then s0.r else s1.r})) (True, s0)
  newState =λs0.(foreach Reg r. (λs.s{r = s.r{modified = False}})) s0
TM :: BCL-Module → (λ-calculus declarations)
TM ⟦ Module mn ms ameths vmeths ⟧ =
  map TM ms
  map TVM vmeths
  map TAM ameths
TR :: BCL-Rule → (λ-calculus declarations)
TR ⟦ Rule rn a ⟧ =
  rule_rn = λs0.TA ⟦ a ⟧ s0
TVM :: BCL-Value-Method → (λ-calculus declarations)
TVM ⟦ VMeth f λx. e ⟧ =
  meth_f = λx.(λs.(TE ⟦ e ⟧) s)
TAM :: BCL-Action-Method → (λ-calculus declarations)
TAM ⟦ AMeth g λx. a ⟧ =
  meth_g = λx.(λs.(TA ⟦ a ⟧ s))
```

Figure 3-4: Conversion of Top-level BCL Design to top-level definitions. `Fixed-width` text is concrete syntax of the λ-calculus expression.

r using the corresponding parts of the input states. This means that each merge computation is completely parallel. This is important when we discussion generating hardware in Chapter 4.

3.3 The GCD Example Revisited

The full translation of the GCD example after some α-renaming and β-reductions can be found in Figure 3-5. To understand this translation and its relation to the definitions we had earlier, consider the swap rule (`rule_swap`). First we do an η-reduction to get `meth_swap ()`. After inlining the definition of `meth_swap` and doing a β-reduction we get the following expression:

```
λs0.let ns = newState s0
        (g, s1) = parMerge(ns{x=Reg{modified=True, value=ns.y.value}})
                           ns{y=Reg{modified=True, value=ns.x.value}})
    in (s.x.value < s.x.value ∧ g, seqMerge s0 s1)
```

Further inlining and constant propagation results in the following expression:

```
    λs.(s.x.value < s.y.value,
        State{x=Reg{modified=True, value=s.y.value},
              y=Reg{modified=True, value=s.x.value})
```

which is the same as what we showed at the beginning of the previous section.

```
seqMerge =
   λs0.λs1.s0{x=if s1.x.modified then s1.x else s0.x,
              y=if s1.y.modified then s1.y else s0.y}
parMerge =
   λs0.λs1.(¬(s0.x.modified ∧ s1.x.modified) ∧
            ¬(s0.y.modified ∧ s1.y.modified),
            s0{x=if s1.x.modified then s1.x else s0.x,
               y=if s1.y.modified then s1.y else s0.y})
newState = λs0.s0{x=s0.x{modified=False}, y=s0.y{modified=False}}
meth_resp = λe0.λs0.(s0.x.value ≠ 0 ∧ s0.y.value=0, s0.x.value)
 meth_getResp = λe0.λs0.(s0.x.value ≠ 0 ∧ s0.y.value=0,
                         s0{x=Reg{modified=True,value=0}})
meth_req =
   λ(a,b)λs0.let ns = newState s0
                 (g, s1) = parMerge(ns{x=Reg{modified=True,
                                             value=a}})
                                    ns{y=Reg{modified=True,
                                             value=b}})
rule_start = λs0.meth_req (0x17, 0x31) s0
meth_subtract =
   λ().λs0.(s0.x.value ≥ s0.y.value ∧ s0.y.value ≠ 0,
            s0{x=Reg{modified=True,value=s0.x.value − s0.y.value}})
rule_subtract = λs.meth_subtract () s
meth_swap =
   λ(a,b).λs0.let ns = newState s0
                  (g, s1) = parMerge(ns{x=Reg{modified=True,
                                              value=ns.y.value}})
                                     ns{y=Reg{modified=True,
                                              value=ns.x.value}})
               in (s.x.value < s.x.value ∧ g, seqMerge s0 s1)
rule_swap = λs.meth_swap () s
```

Figure 3-5: Functionalized form of GCD program

Chapter 4

Hardware Synthesis

There are many issues in generating a hardware implementation of a BCL program. Most obvious of these issues is how exactly does one implements a rule. Atomicity requires that we construct some notion of shadow state so that we can unwind a computation if we reach a guard failure in the evaluation. These shadows can require a substantial hardware cost if they are implemented via stateful constructs. However, if we can store them ephemerally in wires, they become cheap. In synchronous hardware, we can guarantee that all shadows are implementable in wires if each rule is totally executed in a single clock cycle. This is one of the primary optimizations applied by previous schemes [43, 51, 52] to compile guarded atomic actions into synchronous circuits.

It may not be possible to implement loops in a single cycle as they may require an unbounded amount of work and therefore an unbounded amount of hardware to implement in a single cycle. This is a major reason why dynamic loops are not directly supported by Bluespec SystemVerilog; they must be statically removed by unrolling. Practically, this approximation is not an issue as almost all reasonable loops in a BCL rule have a statically known upper bound. As such we assume the following transformation has already been applied for all discussions in this chapter.

The other significant challenge in hardware generation is how to deal with the choice of executing multiple rules in a single cycle. This issue becomes more complicated if we wish to exploit hardware's inherent parallelism in executing multiple rules

concurrently in a single cycle. As will become clearer in Chapter 5, in the context of synchronous hardware we can reduce the construction of a hardware implementation to the implementation of a single rule as an FSM that executes that rule once a cycle. This drastically simplifies the discussion of hardware generation. The remainder of this chapter deals with the synthesis of program with a single rule in synchronous hardware.

4.1 Implementing a Single Rule

Given a program P with a single rule R, there is a straightforward way of implementing the program as an FSM. Each cycle, the state of the system is read from persistent memory (*e.g.,* flip flops), the next state of the system is calculated, and the resulting state is stored back into persistent memory.

We do this by constructing a direct implementation of f_R, the functionalization of rule R as defined in Chapter 3 as a circuit. To understand this transliteration, first notice that because all non-function expressions have a bounded number of values, it can be represented with a fixed-sized bit-vector. Each basic type (*e.g.,* int or boolean) has a natural representation as fixed-size bit-vectors; more complex types can be represented as the concatenation of the bit-vector of its components. Thus the tuple type `(Bool, Bool)` would be a 2-bit value with the first bit corresponding to the first boolean, and the second bit to the second.

Each typed λ-expression in our translation of lambda term translates to a circuit with a set of input and output wires. To keep from having to deal with functions (and thus circuits) being arguments to other circuits, we restrict our translation to the first-order λ-expression. Our functionalization never abstracts functions as variables save in the case of methods. We assume these definitions are inlined. Our translation is effectively identical to the translation between functional expressions and circuits in Lava [84].

Intuitively each primitive function is represented using a gate-level circuit. For instance a + operator adding to 32-bit integers would correspond to a 32-bit ripple-

carry adder circuit. An `if` expression would translate to a multiplexer circuit of the appropriate size with the boolean condition being the first input, the true clause the second input, and the false clause as the third input. Constant values are just a set of wires tied to power or ground to represent the correct value.

Field selection can be implemented using wires only; it takes as input a set of wires corresponding to the structure data and outputs only the wires corresponding the correct field. Similarly, value updates can be done with wires by constructing a new bundle of wires where the new field value are used in lieu of the wires from the old field value.

Thus the operation of incrementing a register `x`, *i.e.,* `x := x + 1`, results in a circuit that takes the state of the system, separates out the wires associated with the value of `x`, sets that as input to an incrementer circuit, and creates a new bundle of wires representing state where `x`'s value is the new incremented value and the wire associated with `x`'s modified bit is replaced with a wire tied to true. In Verilog this is the following RTL module:

```
module incXCircuit(inputState, outputState);

 // modified bit of x is at bitlocation [hiX]
 // value of x is at bitlocations [hiX-1:loX]
 input  [stateWidth-1:0] inputstate;

 output [stateWidth-1:0] outputstate;

 wire [xRegisterWidth-1:0] oldxvalue = inputstate[hiX]; //
 wire [xRegisterWidth-1:0] oldxvalue = inputstate[hiX-1:loX];
 wire [xRegisterWidth-1:0] newxvalue = oldxvalue + 1;
 outputstate = {inputState[stateWidth-1:hiX+1],
                true,newxvalue,
                 inputState[loX:0]};
endmodule
```

Lambda abstraction adds a new first input to the circuit and connects each instance of the variable in the circuit to that input. Its dual operation, application, connects the first input to the output of the operand function (which must be a circuit with no inputs). Let bindings are just like redexs and correspond to wire

```
λs.if s.x.value > 1 then (λs.(s{x=Reg{modified = True,
                                     value = s.x.value + 1}}) s)
                     else (λs.(s{x=Reg{modified = True,
                                     value = s.x.value − 1}}) s)
```

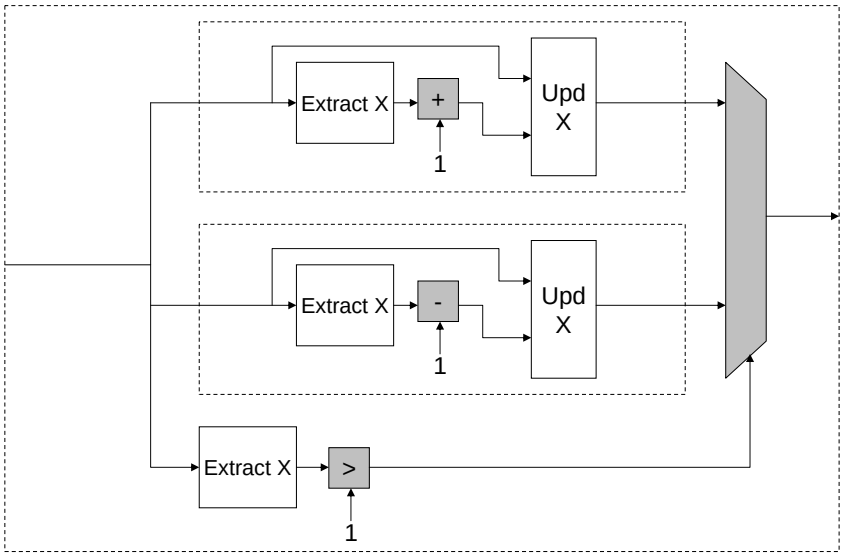


Figure 4-1: Initial Example translation of λ-expressions to Circuits. White boxes can be implemented solely with wires, gray boxes need gates, dotted boxes correspond to λ abstractions.

diagrams; the output of the bound variable is connected to every wire input use of the expression. Notice that β-reduction does not change the circuit that we describe, only removes a circuit boundary. By not doing β-reductions, it is easier to isolate circuitry to the part of the BCL program associated with it; this makes carrying over a notion of modularity from BCL to the FSM circuit.

Also, note that constant propagation on the λ-expression can be understood as circuit-level simplification. To get a feel for this, consider the λ-expression in Figure 4-1 and subsequent simplifications of the expression in Figure 4-2 and Figure 4-3.

4.1.1 Implementing State Merging Functions

It is worthwhile to consider what hardware is generated by the state-merging functions `parMerge`, and `seqMerge`. While these functions have many combinators internally, being described as a sequence of composed functions, they generate parallel sets

```
λs.if s.x.value > 1 then s{x=Reg{modified = True,
                                 value = s.x.value + 1}}
                    else s{x=Reg{modified = True,
                                 value = s.x.value - 1}}
```

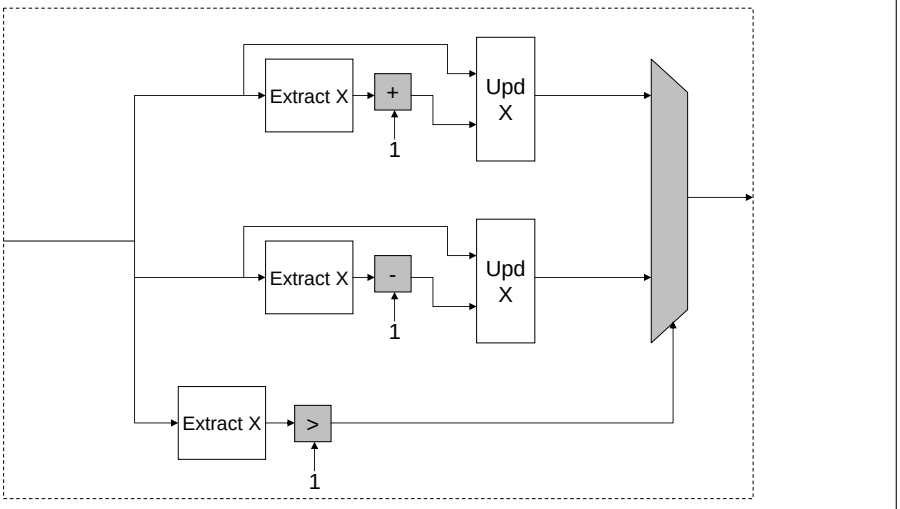


Figure 4-2: Result of β-reduction on expression in Figure 4-1. Notice how the fundamental circuit structures does not change

```
λs.s{s=Reg{modified = True,
           value = if s.x.value > 1 then s.x.value + 1
                                    else s.x.value - 1}}
```

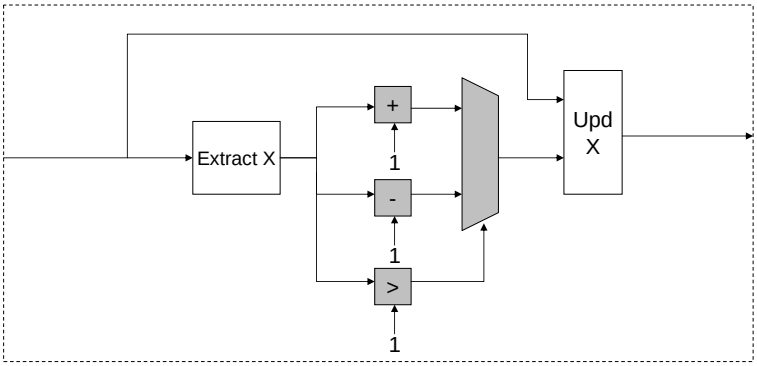


Figure 4-3: Result of Distribution and Constant Propagation on expression in Figure 4-2. Sharing the white box structures (wire structures) do not change the

of multiplexers to merge state on a per-register level only. Further the selection line on these multiplexers are known statically, and as such will be removed either by simple constant propagation in our translation or base-level circuit simplification in the implementing RTL synthesis tool. To see an example of this, consider the `parMerge` generated from the functionalization of action:

$$\texttt{x} := \texttt{y} \mid \texttt{y} := \texttt{x}$$

Assuming only the only state is `x` and `y`, this action translates to the following λ-expression which after some *beta*-reduction to make the term more understandable is:

```
λs.let ns = State{x=Reg{modified = False, value = s.x.value},
                  y=Reg{modified = False, value = s.y.value}} in
    let (g1, s1) = (True, ns{x=Reg{modified = True,
                                   value = ns.y.value) in
     let (g2, s2) = (True, ns{y=Reg{modified = True,
                                    value = ns.x.value) in
      let (g3, s12) = (True ∧ !(s1.x.modified ∧ s2.x.modified)
                            ∧ !(s1.y.modified ∧ s2.y.modified),
                       s1{x=if s1.x.modified then s1.x else s2.x,
                          y=if s1.y.modified then s1.y else s2.y}) in
        (g1 ∧ g2 ∧ g3, (seqMerge s s12))
```

Here we statically know `s1.x.modified` and `s2.y.modified` are true and `s1.y.modified` and `s2.x.modified` are false. As such the multiplexers implied by the if statements can be statically removed and we are left with:

```
λs.let ns = State{x=Reg{modified = False, value = s.x.value},
                  y=Reg{modified = False, value = s.y.value}} in
    let (g1, s1) = (True, ns{x=Reg{modified = True,
                                   value = ns.y.value}) in
     let (g2, s2) = (True, ns{y=Reg{modified = True,
                                    value = ns.x.value}) in
      let (g3, s12) = (True, s1{x = s1.x, y = s2.y}) in
        (g1 ∧ g2 ∧ g3, (seqMerge s s12))
```

If we continue the simplification, we end up with:

```
λs.let ns = State{x=Reg{modified = False, value = s.x.value},
                  y=Reg{modified = False, value = s.y.value}} in
      (True,seqMerge s s{x=Reg{modified = True, value = ns.y.value},
                         y=Reg{modified = True, value = ns.x.value}})
```

At this point all the circuitry associated with `parMerge` has been removed. This is also true of the instance `seqMerge` that we had left symbolic. In general we only generate additional circuitry for merges when we have two writes of the same register where the selection of which one dominates (is in the final result) is dynamically dependent on the state of the rule.

4.1.2 Constructing the FSM

Given the circuit representation of the rule, we only need to connect this to persistent state to finish our FSM. To do so, we construct a hardware register for each register in our program. The input of our circuit representing the rule R consists of a modified bit and a value for each BCL register. We connect the read output of the register to the associated value input and tie the modified bit inputs to the constant false value. As output of our combinational we have modified bits and the new values for each register as well as a boolean guard bit. We connect the new values to the associated next state input for each register and the result of and-ing the guard into and the modified bit to the enable line of the register. This causes us to update only those registers that changed.

To understand this more concretely consider the program in Figure 4-4. After functionalization, some β-reduction, and some constant propagation we are left with the definitions in Figure 4-5. By inlining the top-level definitions doing further simplification we get the final λ expression in Figure 4-6. Translating this to a circuit and adding the appropriate state we get Verilog module in Figure 4-7.

```
Program P

  Rule top:
      v = r1 + 1 in
      if (r1 < 3) (r1 := v | m.add(r1))

  Register r1 (0)

  Module m
      Register r2 (0)

      ActMeth add(x) =
          r2 := r2 + x
```

Figure 4-4: Example Single-Rule Program for Hardware Synthesis

```
parMerge = λs0.λs1.
             ((!s0.r1.modified ∧ !s1.r1.modified) ∧
               (!s0.r2.modified ∧ !s1.r2.modified),
              State{r1 = if s1.r1.modified then s1.r1 else s0.r1,
                    r2 = if s1.r2.modified then s1.r2 else s0.r2})
seqMerge = λs0.λs1.
             State{r1 = if s1.r1.modified then s1.r1 else s0.r1,
                   r2 = if s1.r2.modified then s1.r2 else s0.r2})
newState = λs.State{r1=s.r1{modified = False},
                    r2=s.r2{modified = False}}
meth_add = λx.λs.(s{r2 = Reg{modified = True,
                             value = s.r2.value + x})
     f_R = λs0.let vb = s0.r1.value + 1
                   vg = true
                   ns = newState s0
                   (g,s1) = parMerge (ns{r1 = Reg{modified = True,
                                                  value = vb}})
                                     (meth_add ns.r1.value ns)
               in (g,seqMerge s0 (if s0.r1.value<3 then s1 else s0))
```

Figure 4-5: Simplified λ expressions of functionalization of Figure 4-4

```
f_R = λs0.let vb=s0.r1.value + 1
             ns=State{r1 = s0.r1{modified = False},
                      r2 = s0.r2{modified = False}}
             (g,s1)=(ns{r1 = Reg{modified= True, value=vb},
                        r2 = Reg{modified = True,
                                 value = ns.r2.value + ns.r2.value}
         in (g,if s0.r1.value < 3 then s1 else s0)
```

Figure 4-6: Final functionalization of Program in Figure 4-4

4.2 Modularizing the Translation

Our initial translation is very simple. Given the correctness of the functionalization given in Chapter 3, it is fairly easy to see why it is correct. However, there are a few practical issues with this approach. First, the description of the compilation is monolithic and is not a readily available way to partition the task. Second, no system is truly closed; we must interact with the outside world. In the context that some aspect of our BCL program interacts with hardware, we must have some way of interfacing the two parts. These problems can be addressed if we can keep the modular boundaries of the BCL program meaningful. These modules can be compiled separately, and the characterization will give the outside world some understanding of how to interact with the hardware implementation of the BCL module.

Although this change may appear potentially difficult, conceptually we can achieve most of the desired changes by taking the flat circuit generation, but keeping the modular hierarchy correctly preserved. The top-level modules become the object compiled, and the top-level rule serves only to characterize how the outside world may use the compiled FSMs. Parallelization of the compilation can be added once we can characterize the possible use of a RTL implementation of a BCL module and which usages are valid.

Before we get into the details about the precise meaning of modules, we discuss some issues with our notion of keeping the module boundaries in the circuit. Intuitively we want the wire-level interface of the RTL module corresponding to a par-

```
module program(CLK, RST_N);

 input CLK, RST_N;

 // bit 65 is r1.modified, bits 64:33 is r1.value
 // bit 32 is r2.modified, bits 31:0  is r2.value
 wire [65:0] s0, s1, ns, s2;

 wire [31:0] r1_in r1_out, r1_in, r2_out, vb;
 wire r1_en, r2_en, g, true, false;

 assign true = 1; assign false = 0;

 Register#(.width(32),.init(0))
     r1(.CLK(CLK), .RST_N(RST_N),
        .D(r1_in), .EN(r1_en),
        .Q(r1_out));

 Register#(.width(32),.init(0))
     r2(.CLK(CLK), .RST_N(RST_N),
        .D(r2_in), .EN(r2_en),
        .Q(r2_out));

 assign r1_en = g && s2[65];
 assign r2_en = g && s2[32];

 assign r1_in = s2[64:33];
 assign r2_in = s2[31: 0];

 assign s0 = {false,r1_out,false,r2_out};
 assign ns = {false,s0[64:33],false,s0[31:0]};
 assign s1 = {true,(ns[64:33] + 1),true,(ns[31:0] + ns[64:33])};
 assign s2 = (s0[64:33] < 3) ? s1 : s0;

 assign vb = s0[64:33] + 1;
 assign g  = true;
endmodule
```

Figure 4-7: Example of implemented rule. State structure `s0`, `s1`, `ns`, and final output `s2` have been flattened into a single bit-vector. {} is Verilog bit-vector concatenation.

ticular BCL module to correspond to the BCL module's methods. This means that every RTL port (except the clock and reset signal ports needed to allow synchronous hardware) should correspond to some aspect of an *instance* of a BCL method. As a result, all wires in the circuit associated with the state internal to the module should be internal to the module, the only exposure of the state is through the methods.

Given this, it is fairly easy to mark where the modular boundary would go for any particular module. The interfaces of the circuits associated with the translation of the `meth_`f and `meth_`g definitions form the interface save for those wires associated with the state. Further, the part of the state bit-vector associated with the module needs to be separated from the rest of the state and be internal to the module. Conceptually this is just partitioning the circuit across a state boundary; for instance, `(s0[64:33] < 3) ? s1 : s0` can be turned into {`(s0[64:33] < 3) ? s1[65:33]: s0[65:33], (s0[64:33] < 3) ? s1[32:0]: s0[32:0]`}. This explicitly reflects the data flow in the resulting circuit.

This intuitive module partitioning and interface scheme has one problem; multiplexing of state requires more inputs that this interface implies. To understand this, consider the `ns`, `s0`, `s1`, and `s2` in Figure 4-7. These wires represent the whole state of program P, however only the state associated with `r2` should go into the RTL module implementation of BCL module `m`. We have isolated the state by splitting the wires between `r1` and `r2`, but we must deal with the `if` expression (*i.e.,* the multiplexer in the circuit) in the definition of `s2`. The issue is that the selector line `(s0[64:33] < 3)`, does not relate to `m` directly and so must be calculated outside of RTL module and passed in as an input. Unfortunately it is not clear how this input corresponds to a method of `m`.

Taking a step back, we can see that the issue comes from our use of whole-state multiplexing, generated from `if` and `localGuard` actions. We do not have this issue with our action compositions, as `parMerge` and `seqMerge` operate on a per-register level and can be easily split across module boundaries.

To resolve this problem we modify the functionalization so that all multiplexing done by conditional state updates are done in the state-merging operators. We add

an extra "valid" bit as input (the *enable* bit) to the translated λ term. This bit signifies whether this action is used dynamically in the evaluation. We use this to mark whether any particular state value (*e.g.,* the appropriate subvector of the state bit-vector of `s0`, `s1`, `s2`, or `ns` in our example) is actually used. Essentially we are distributing the boolean predicate from the whole-state multiplexer to each module that it affects.

Figure 4-8 and Figure 4-9 show the modified functionalization. The non-trivial changes occur in the register assignment, conditional action, and local guard rule. Notice that we have split the body (δ) and the guard (π) in the description. This separation does not change the circuit or the result, but makes it more obvious that the guard of an action is *not* affected by the enable bit.

With this characterization our intuitive boundary point works; the unassociated input port is now clearly associated to the method `add`. The interface has ports associated with each method class at the level of given by the functional definitions in Figure 4-10, *e.g.,* `meth_`g. An input port when needed, an output port for value methods, a π guard output port (or *ready* signal) for all methods and an enable bit input for action methods.

As future work, we could exploit this notion of enable to value methods (and guards) as well. The resulting additional hardware serves as an enable for the combinational circuit and could allow dynamic power optimizations to occur.

4.3 Understanding the FSM Modularization

With our new functionalization, modularizing a program becomes quite straightforward. However, we still do not have a view through which to understand exactly what our RTL module interface means. The ports represent different *instances* of the BCL methods of the module, but the real question is how do two method instances relate to each other. If we could note all of these relations, we could completely characterize the RTL implementation; the BCL module it implements explains what each instance does, and the relation explains how they operate in regards to each other.

δ_A :: BCL-Action $\rightarrow$ (Bool $\rightarrow$ Translated-State $\rightarrow$ Translated-State)
δ_A [[r := e_0]] = λp.λs.(s{r = Reg{modified:p, value:δ_E [[e_0]] s}})
δ_A [[**if** (p_0) a_0]] = λp.λs.δ_A [[a_0]] (p $\wedge$ δ_E [[p_0]] s) s
δ_A [[a_0 **when** e_0]] = λp.λs.(δ_A [[a_0]]) p s
δ_A [[a_1 | a_2]] =
 λp.λs.let ns = newState s in
 seqMerge s (parMerge (δ_A [[a_1]] p ns) (δ_A [[a_2]] p ns))
δ_A [[a1 ; a2]] λp.λs.(δ_A [[a_2]] p (δ_A [[a_1]] p s))
δ_A [[t = e_0 **in** a_0]] =
 λp.λs.let tg = π_E [[e_0]] s in let tb = δ_E [[e_0]] s in
 δ_A [[a_0[tb **when** tg/t]]] p s
 tb and tg are a fresh names
δ_A [[**localGuard** $a0$]] = λp.λs.δ_A [[a_0]] (p $\wedge$ (π_A [[a_0]] s)) s
δ_A [[**loop** e_0 a_0]] = λp.λs.δ_A [[**loop** e_0 a_0]]
 (p $\wedge$ δ_E [[e_0]] s) (δ_A [[a_0]] p s)
δ_A [[m.g(e_0)]] = λp.λs.meth_δ_g (δ_E [[e_0]] s) p s

δ_E :: BCL-Expr $\rightarrow$ (Translated-State $\rightarrow$ Translated-Expr)
δ_E [[r]] =λs.(s.r.value)
δ_E [[c]] =λs.c
δ_E [[t]] =λs.t
δ_E [[e_1 **op** e_2]] = λs.(δ_E [[e_1]] s $\wedge$ δ_E [[e_2]] s)
δ_E [[e_b **?** e_t **:** e_f]] = λs.(if δ_E [[e_b]] s then δ_E [[e_t]] s else δ_E [[e_f]] s)
δ_E [[e **when** e_g]] = λs.(δ_E [[e]] s)
δ_E [[t = e_1 **in** e_2]] =
 λs.(let tg = π_E [[e_1]] s in (let tb = δ_E [[e_1]] in
 δ_E [[e_2[tb **when** tg/t]]] s))
 tb and tg are a fresh names
δ_E [[m.f(e_0)]] = λs.meth_δ_f (δ_E [[e_0]] s) s

Figure 4-8: Generation of δ Functions. `Fixed-width` text is concrete syntax of the λ-calculus expression.

π_A :: BCL-Action $\rightarrow$ (Translated-State $\rightarrow$ Bool)
π_A [[r := e_0]] = λs.π_E [[e_0]] s}})
π_A [[**if** (p_0) a_0]] = λs.(if δ_E [[p_0]] then π_A [[a_0]] s else True)
π_A [[a_0 **when** e_0]] = λs.(π_E [[e_0]] s $\wedge$ δ_E [[e_0]] s $\wedge$ π_A [[a_0]] s)
π_A [[a_1 | a_2]] =
 λs.let ns = newState s in π_A [[a_1]] ns $\wedge$ π_A [[a_1]] ns
π_A [[a1 ; a2]] = λs.(π_A [[a_1]] s $\wedge$ π_A [[a_2]] (δ_A [[a_1]] (s)))
π_A [[t = e_0 **in** a_0]] =
 λs.let tg = π_E [[e_0]] s in let tb = δ_E [[e_0]] in π_A [[a_0[tb **when** tg/t]]] s
 tb and tg are fresh names
π_A [[**localGuard** $a0$]] = λs.True
π_A [[**loop** e_0 a_0]] =
 λs.(π_E [[e_0]] s) $\wedge$
  if (δ_E [[e_0]] s) then π_A [[a_0]] s $\wedge$ π_A [[**loop** e_0 a_0]] (δ_A [[a_0]]s)
  else True
π_A [[m.g(e_0)]] = λs.meth_π_g (δ_E [[e_0]]) s) s

π_E :: BCL-Expr $\rightarrow$ (Translated-State $\rightarrow$ Bool)
π_E [[r]] =λs. True
π_E [[c]] =λs. True
π_E [[t]] =λs. True
π_E [[e_1 **op** e_2]] = λs.(π_E [[e_1]] s $\wedge$ π_E [[e_2]] s)
π_E [[e_b **?** e_t **:** e_f]] =
 λs.(π_E [[e_b]] s $\wedge$ (if δ_E [[e_b]] s then π_E [[e_t]] s else π_E [[e_f]] s))
π_E [[e **when** e_g]] = λs.(π_E [[e_g]] s) $\wedge$ (δ_E [[e_g]] s) $\wedge$ (π_E [[e]] s))
π_E [[t = e_1 **in** e_2]] =
 λs.(let tg = π_E [[e_1]] s in (let tb = δ_E [[e_1]] s in
  π_E [[e_2[tb **when** tg/t]]] s))
 tb and tg are fresh names
π_E [[m.f(e_0)]] = λs.meth_π_f (δ_E [[e_0]] s) s

Figure 4-9: Generation of π Functions. `Fixed-width` text is concrete syntax of the λ-calculus expression

To understand this, first consider how actions and expressions interact with the execution context in our operational semantics given in Chapter 2. Each action and expression takes an input notion of state. Actions also results in an output notion of state. These have been split into the original state (S) and the new updates (U), but they represent a concrete view of the state upon which the action occurs and the subsequent state.

If we think about the input and output states for two actions a and b, there are three possible data flow relationships possible: the output state of a may be used directly or indirectly as the input of b ($a < b$), the output state of b may be used directly or indirectly in the input state of a ($b < a$), or neither a nor b observe the outputs state of the other ($a \mid b$).

These three relations represent everything we need to know about two method instances in an RTL implementation. If $a < b$, then the effect of instance a effect is visible to instance b. If $b < a$ or $a|b$, then a effect is not visible. As it is impossible for loops in this observability, $<$ forms a partial ordering on the method instances of an implementation. To better understand this, consider the following program:

```
Program P
 Rule doOperation
    m.wr(m.rd() + 1) ;
    m.wr(m.rd() + 1)
 Module m
        Register r (0)
        ActMeth wr(x) =
            r := x
        ValMeth rd() = r
```

This program sequentially reads and writes a single register many times. Our functionalization results in the following module definition for `m`:

```
module m(CLK, RST_N, RDY_rd0, rd0,
         RDY_wr0, EN_wr0, wr0_IN,
         RDY_rd1, rd1,
         RDY_wr1, EN_wr1, wr1_IN);

 output RDY_rd0, RDY_wr0, RDY_rd1, RDY_wr1;
 output [31:0] rd0, rd1;
 input  EN_wr0, EN_wr1;
 input  [31:0] wr0_IN, wr1_IN;
 wire [31:0] r0, r1, r2;

 Register#(.width(32),.init(0))
    r2(.CLK(CLK), .RST_N(RST_N),
       .D(r0), .EN(1),
       .Q(r2));

 assign RDY_rd0 = true; assign RDY_rd1 = true;
 assign RDY_wr0 = true; assign RDY_wr1 = true;

 assign rd0 = r0;
 assign r1 = (EN_wr0) ? wr0_IN : r0;
 assign rd1 = r1;
 assign r2 = (EN_wr1) ? wr1_IN : r1;

endmodule
```

This module models multiple read-write "cycles" of a single register. Each of the two BCL methods have two instances (which we have labeled by version 0 and 1 to disambiguate them). Both the `rd0` and `wr0` see the same initial state as do `rd1` and `wr1`. This later pair of method instances *does* observe the changes from `wr0` because of the sequential composition. If we enumerate these facts pairwise, we get the following list of relations: {`rd0 | wr0, rd0 | rd1, rd0 | wr1, wr0 < rd1, wr0 < wr1, rd1 | wr1`}. We can denote this concisely by the partial ordering relation: `(rd0, wr0) < (rd1, wr1)`.

Given this annotation and the BCL module it implements, we know how the methods operate when they are used in its context; In a cycle where we use `wr0` (*i.e.,* the enable bit of `wr0` is true), `rd1` observes its update meaning that the module outputs the value just written through `wr0`'s interface. If `wr1` is also enabled it

happens last, and dominates `wr0`'s write; however, it does not affect the results of `rd0` or `rd1`.

If we change our perspective, this annotation becomes a requirement for safe use of this module. The context that uses this module must use all methods in a way that has the same $<$ relation to which we have built the RTL module. Note that practically, it is easy to deal with a method `g` that we do not use in our new context rule; it is akin to adding an action that uses `g` but never dynamically calls it (*e.g.,* **if** `false` **then** `m.g(x)`).

In general each partial ordering of method instances results in a different RTL module. However, in many cases two different orderings result in identical hardware. For instance if two methods operate on completely different states, and if we construct the module associated with calling them in either order, we end up with the same RTL module. In light of this, it would be better to annotate each module, not with one partial ordering, but all valid partial orderings that it represents. This can be represented by enumerating all valid pairwise interpretations between two methods. Any partial ordering where each pairwise relation is acceptable. There are 7 possible subsets of possible interpretations ranging from any three interpretations ($\{<, |, >\}$). Having the full set means that we can always call both methods in a cycle and have it be valid; having only one means that we can only interpret it in one way.

This codification of relations is very similar to Rosenband and Arvind's pairwise rule relations [79]. The difference is that in their construction, it was possible for an invalid parallel composition to occur due to a double-update error. To prohibit this, the notion of conflicting methods (corresponding to the empty set relation) was developed. We need not worry about such errors as we prohibit double update errors as guard failures. Thus the empty pairwise relation $a\,\{\}\,b$ is always the same as $a\,\{|\}\,b$.

T_{VM} :: BCL-Value-Method $\rightarrow$ (λ-calculus declarations)
T_{VM} $[\![$ **VMeth** f λx. e $]\!]$ =
 `meth_`f = λ`x.(`λ`s.(`π_E $[\![$ e $]\!]$ `s,`δ_E $[\![$ e $]\!]$ `s))`
 `meth_`π`_`f = λ`x.(`λ`s.(let (g,s') = meth_f x s in g))`
 `meth_`δ`_`f = λ`x.(`λ`s.(let (g,s') = meth_f x s in s'))`
T_{AM} :: BCL-Action-Method $\rightarrow$ (λ-calculus declarations)
T_{AM} $[\![$ **AMeth** g λx. a $]\!]$ =
 `meth_`g = λ`x.(`λ`p.`λ`s.(`π_A $[\![$ a $]\!]$ `s,`δ_A $[\![$ a $]\!]$ `p s))`
 `meth_`π`_`g = λ`x.(`λ`s.(let (g,s') = meth_g x _ s in g))`
 `meth_`δ`_`g = λ`x.(`λ`p.`λ`s.(let (g,s') = meth_g x p s in s'))`

Figure 4-10: Generation of method functions. Notice that `meth_`π`_`g may be evaluated without knowing the particular value `p`, though it shares the same input argument `x` as `meth_`δ`_`g. `Fixed-width` text is concrete syntax of the λ-calculus expression.

Chapter 5

Scheduling

We presented a nondeterministic procedure in Chapter 2 to describe the semantics of BCL. An essential step to implement a BCL program is to construct a procedure that will do this selection. We call this procedure the *scheduler*. At first blush, one may assume that this is a bounded problem, where at each step we select exactly one rule. However, as we see shortly, there is no reason to make this choice deterministically. Depending on the current dynamic resources of our system, we may want to make a different choice. Further, in implementations it would be often far more efficient to execute multiple rules concurrently. However, it is unclear which executing multiple rules concurrently is valid. One can evaluate an implementation with respect to different metrics such as performance, resource allocation, power consumption, etc. In this chapter, we focus only on the semantic aspects. In this context we can consider an implementation of a program as a program itself.

Before we get to our formal definition of implementation, we revisit our two-button black-box model introduced in Chapter 2. An implementation can be thought of as just such a black box. However, the result of "Display" may not directly show the internal state. Rather, it can display a function of the internal state that mimics the program implemented. In this regard it externally looks just like a program, but can have a completely different internal structure. An implementation is correct if all observations of it can be understood as observations of the specification program.

As we have stated before, for efficiency we may want to have an implementation

of a program execute multiple rules concurrently while still having them logically happen in sequence. This results in us being unable to observe every state change in the implementation as we would in the original program. Such implementations correspond directly to our definition of a complete approximation; in the context of implementations, such a program is called a direct implementation.

Definition 18 (Direct Implementation)**.** P' is a *direct implementation* of P if it is a complete approximation of P. ■

5.1 A Reference Scheduler

We mentioned in Chapter 4 that all synchronous hardware implementations can be represented as a BCL program with exactly one rule. A direct implementation of this form would have a single derived rule. By definition we can understand the execution of this implementation as a deterministic sequence of rules of program P.

It is possible that this derived rule may be degenerate in a state that is not a terminating state of the original program. In this case, when we reach that state, we will necessarily be stuck, as no other rules exist to take us out of this state. This means that the direct implementation terminated prematurely. To prevent this we would like to guarantee that our derived rule always makes progress when possible; one easy way to do this is to consider each rule in sequence.

We could construct a new program P_I with a single compound rule of each rule described in sequence, but this may generate poor hardware as it allows very long critical paths. Another approach would be to create a single rule that executes as the first rule on the first execution, then the second rule on the second cycle, and so on, before returning to the first rule after it has finished.

As the behavior of a rule is a function of the state, this necessarily involves changing the state of the program. We add a `cnt` register that reflects which one of the original rules our single rule will emulate. Our new rule then checks the value of `cnt`, executes the body of the corresponding rule, and increments `cnt` to point to the next rule. To prevent guard failure from preventing `cnt` from incrementing, we must wrap

each body in a `localGuard`. If each rule R_i has body a_i, the new rule is:

```
Rule topRule:
   (if (cnt == 0) then (localGuard(a_0)) |
   (if (cnt == i) then (localGuard(a_i)) |
   ...
   (if (cnt == max) then (localGuard(a_max")) |
   (cnt := (cnt == max) ? 0 : cnt + 1)
```

This program is clearly not a direct implementation since we added `cnt` to the state. However there is a natural way to understand each state in this new program as a state in the original, namely eliding the extra register. In general, we would like P_I to be an implementation of P if we have a function allowing us to understand the states and transitions of P_I as states and transitions of program P. This leads to the following more general definition of an implementation.

Definition 19 (**Implementation** of Program P)**.** An *implementation* I of program P modeled by $(S', S'_0, \longrightarrow_{P'})$ is a pair (P', f) where P' is a program modeled by $(S', S'_0, \longrightarrow_{P'})$ and f is a function of type $S' \longrightarrow S$. I is an implementation of P when the following are true:

- **Total Correspondence:** $S = \{f(s) | s \in S'\}$
- **Initial Correspondence:** $S_0 = \{f(s) | s \in S'_0\}$
- **Faithfulness:** $s \twoheadrightarrow_{P'} s' \implies f(s) \twoheadrightarrow_P f(s')$

■

Notice that the function f is total, meaning that at every point in the execution we retain a notion of the computation performed. In fact we can project all aspects of P' via f to get a new program $P'' \sqsubseteq P$. Practically, f may become quite complicated to allow for various speculative execution approaches. A direct implementation is now just a implementation where $fx = x$.

With a formal notion of an implementation we can define the observations of an implementation thus:

Definition 20 (**Observation** of **Implementation** I)**.** An *observation* of implementation $I = (P', f)$ of program P modeled by $(S, S_0, \longrightarrow_P)$ where P' is modeled by $(S', S'_0, \longrightarrow_{P'})$ is a sequences of states in S, $\sigma = s_0, s_1, s_2, ..., s_N$ such that:

- $s'_0 \longrightarrow_{P'} s'_1 \longrightarrow_{p'} s'_2 \longrightarrow_{P'} ... \longrightarrow_{P'} s'_N$ and $\forall i. s_i = f(s'_i)$.
- $\forall\, 0 \leq i < N.\ s'_i \neq s'_{i+1}$. ■

Lemma 6. *if σ is an observation of implementation I of program P, σ is also an observation of P.*

5.2 Scheduling via Rule Composition

Given our definition of correctness, it is straightforward to verify whether any program P' is a valid implementation of BCL program P. However, this approach is unwieldy when one is exploring possible implementations; having to check not only the performance of an implementation, but also its correctness is impractical. Instead, it would be far better to constrain ourselves so that all possible implementations considered are correct by construction. This would leave only the performance/cost tradeoff about which to reason.

Before we deal with adding state to our implementation, let us consider implementations which have the same notion of state as our program. In this context, each rule in an implementation of Program P when projected is a derived rule of P. It must be the case that we can construct its behaviors by composing the rules of P into larger rules. For instance, if wanted to always execute Rule `R1` with body $a1$ followed by `R2` with body $a2$, we could generate a new rule:

Rule `R1R2: (a1;a2)`

Executing this rule has the same result as executing `R1` then `R2`. Extending this idea, we can create a set of *rule compositions*, functions taking rules as input and returning a new rule, which will let us construct a large set of rules. If we prove that for each composition the output rule can always be understood as a derived rule of the input

rules, we are guaranteed that any rule we get from this starting from the rules of P must be a derived rule of P.

In general, it is not practical to construct a set of compositions that generate all possible derived rules; we would need compositions that do significant inter-rule analysis. For instance consider the two rules:

```
Rule R3: (if  p1 then r1 := 5) ; (if !p2 then r2 := 2)
Rule R4: (if !p1 then r1 := 3) ; (if  p2 then r2 := 8)
```

A valid derived rule would be:

```
Rule R3R4: (r1 := (p1) ? 5 : 3) | (r2 := (p1) ? 2 : 8)
```

This rule behaves as `R3` followed by `R4`. However, this is not simply sequentially composing the bodies; it makes use of the fact that the updates in both rules are mutually exclusive to merge the updates to each register and to parallelize the bodies. A composition function that takes in `R3` and `R4` would need to do nontrivial analysis to be able to get this level of optimization.

However, it is straightforward to generate a rule that emulates any particular derived rule R. For each state in the system there is some sequence of rules that behaves as R. We can compose their bodies in sequence and construct a rule that examines the state of the program to select which of the 2^N bodies to execute. While capturing the dynamics we expect this rule will lead to poor implementation; in hardware this would generate an exponential number of method instances and thus exponentially large multiplexers. To generate efficient hardware (or software), either a better strategy for generating the derived rule should be found, or else we should apply semantics-preserving rule-to-rule transformations to simplify the bodies of the newly composed rules.

In general, working with a different notion of state is complicated and cannot be easily built up in an incremental manner; once the notion of implementation state becomes decoupled from the program's state there is so much choice that it is not clear how one can incrementally build up a design. However, just augmenting state

$$
\begin{array}{rcl}
DR & ::= & R \\
& \| & \text{compose}(DR, DR) \\
& \| & \text{par}(DR, DR) \\
& \| & \text{restrict}(DR, DR) \\
& \| & \text{repeat}(DR)
\end{array}
$$

Figure 5-1: A BCL Scheduling Language

for scheduling can be viewed as just a preprocessing step. We first add the new state and rules that modify only this state, leaving the original state untouched. Our relation function elides this extra state. As such these new rules do not change the program state and appear as degenerate rule executions.

Conceptually, scheduling can be reduced to the following steps:

1. Add state to the program and rules which may read the entire state of the new program but are allowed to modify only the added state.

2. Construct new derived rules via a set of verified rule compositions. When we have sufficient rules, remove rules undesired in the final implementation

3. Use semantics-preserving transformations to change the rules to be more efficient to implement without changing the program meaning.

Practically, most desirable implementations correspond closely to simple compositions. As such the final simplification step is generally not necessary.

5.2.1 Rule Composition Operators

We can define all possible derived rules with four "basic" rule compositions (compose, par, restrict, and repeat) producing the composed derived rule grammar shown in Figure 5-1. These compositions form the basis of a user-defined scheduling language, which can be used to define the desired implementation.

For clarity, we assume that all parameter rules to a composition are in the form a `when` p: they have had all guards lifted to the top by following the axioms presented in Figure 2-8. If a rule does not have loop or sequential composition, *i.e.,* the subset of BCL corresponding to BSV, this lifting is always total, allowing us to lift the guard

to the top. A procedure to do the total lifting of this subset is given in Figure 5-2, Figure 5-3, and Figure 5-4. Lifting is not truly necessary; but doing so makes the compositions easier to understand.

5.2.2 Sequencing Rules: compose

To sequence two rule executions $R1$ and $R2$ together atomically with $R1$ observing $R2$, we introduce the compose operator.

```
compose(Rule R1 a1 when p1, Rule R2 a2 when p2) =
     Rule R1R2 (a1 when p1);(a2 when p2)
```

Note that the composed rule can be enabled only when *both* $R1$ and $R2$ can fire in sequence.

5.2.3 Merging Mutually Exclusive Rules: par

To deal with choice, we introduce the parallel composition operator — which uses the parallel action composition. If the rules are not mutually exclusive, the new rule may exhibit new behaviors. For instance if $R1$ was $r1 := r2$ and $R2$ was $r2 := r1$, then the above action would swap the values, a behavior not expressible via sequential executions of $R1$ and $R2$.

We prevent this situation from occurring by forcing the new rule to be enabled only when exactly one of the rules is ready. The operator is:

```
par(Rule R1 a1 when p1, Rule R2 a2 when p2) =
  Rule R1R2 (if p1 then a1)|(if p2 then a2)
            when (p1 ≠ p2)
```

This rule can be made simpler if we interpret a DUE error as NR; this is exactly the choice we made in the functionalization presented in Chapter 3. With this interpretation we can guarantee the composition is correct by augmenting the actions $a1$ and $a2$ such that they always result in a DUE. For instance, we could introduce a new register r and have both update it as follows:

$$
\begin{array}{rl}
\multicolumn{2}{c}{LW_e :: \text{BCL-Expr} \rightarrow \text{BCL-Expr}} \\
LW_e[\![r]\!] = & (r \textbf{ when } \texttt{true}) \\
LW_e[\![c]\!] = & (c \textbf{ when } \texttt{true}) \\
LW_e[\![t]\!] = & (t \textbf{ when } \texttt{true}) \\
LW_e[\![e_1 \; op \; e_2]\!] = & (e'_1 \; op \; e'_2) \textbf{ when } (e_{1g} \wedge e_{2g}) \\
\text{where} & \quad (e'_1 \textbf{ when } e'_{1g}) = LW_e[\![e_1]\!] \\
& \quad (e'_2 \textbf{ when } e'_{2g}) = LW_e[\![e_2]\!] \\
LW_e[\![e_1]\!] \textbf{ ? } e2 \textbf{ : } e3) = & (e'_1 \textbf{ ? } e'_2 \textbf{ : } e'_3) \textbf{ when } (e_{1g} \wedge (e'_1 \textbf{ ? } e_{2g} \textbf{ : } e_{3g})) \\
\text{where} & \quad (e'_1 \textbf{ when } e_{1g}) = LW_e[\![e_1]\!] \\
& \quad (e'_2 \textbf{ when } e_{2g}) = LW_e[\![e_2]\!] \\
& \quad (e'_3 \textbf{ when } e_{3g}) = LW_e[\![e_3]\!] \\
LW_e[\![e_1 \textbf{ when } e_2]\!] = & e'_1 \textbf{ when } (e'_2 \wedge e_{1g} \wedge e_{2g}]\!] \\
\text{where} & \quad (e'_1 \textbf{ when } e_{1g}) = LW_e[\![e_1]\!] \\
& \quad (e'_2 \textbf{ when } e_{2g}) = LW_e[\![e_2]\!] \\
LW_e[\![t = e_1 \textbf{ in } e_2]\!] = & ((t' = e'_1) \textbf{ ; } e'_2)) \textbf{ when} \\
& ((t' = e'_1) \textbf{ ; } (t_g = e_{1g}) \textbf{ ; } e_{2g}]\!] \\
\text{where} & \quad (e'_1 \textbf{ when } e_{1g}) = LW_e[\![e_1]\!] \\
& \quad e_3 = e_2[(t' \textbf{ when } t_g)/t] \\
& \quad (e'_2 \textbf{ when } e_{2g}) = LW_a[\![e_3]\!] \\
LW_e[\![m.f(e)]\!] = & m.f_b(e') \textbf{ when } e_g \wedge m.f_g(e') \\
\text{where} & \quad (e' \textbf{ when } e_g) = LW_e[\![e]\!]
\end{array}
$$

Figure 5-2: Total guard lifting procedure for BCL Expressions resulting in the guard being isolated from its guard.

$$
\begin{array}{rl}
\multicolumn{2}{c}{LW_a :: \text{BCL-Action} \rightarrow \text{BCL-Action}} \\
LW_a[\![t = e_1 \textbf{ in } e_2]\!] = & ((t' = e_1') \textbf{ ; } e_2')) \textbf{ when } ((t' = e_1') \textbf{ ; } (t_g = e_{1g}) \textbf{ ; } e_{2g}]\!] \\
\text{where} & (e_1' \textbf{ when } e_{1g}) = LW_e[\![e_1]\!] \\
 & (e_2' \textbf{ when } e_{2g}) = LW_a[\![e_2[(t' \textbf{ when } t_g)/t]]\!] \\
LW_e[\![m.f(e)]\!] = & m.f(e') \textbf{ when } e_g \\
\text{where} & (e' \textbf{ when } e_g) = LW_e[\![e]\!] \\
LW_a[\![r := e]\!] = & (r := e') \textbf{ when } e_g \\
\text{where} & (e' \textbf{ when } e_g) = LW_e[\![e]\!] \\
LW_a[\![a \textbf{ when } e]\!] = & a' \textbf{ when } (a_g \wedge e' \wedge e_g) \\
\text{where} & (a' \textbf{ when } a_g) = LW_a[\![a]\!] \\
 & (e' \textbf{ when } e_g) = LW_e[\![e]\!] \\
LW_a[\![\textbf{if } e \textbf{ then } a]\!] = & (\textbf{if } e' \textbf{ then } a') \textbf{ when } (e_g \wedge (a_g \vee \neg e')) \\
\text{where} & (a' \textbf{ when } a_g) = LW_a[\![a]\!] \\
 & (e' \textbf{ when } e_g) = LW_e[\![e]\!] \\
LW_a[\![a_1 \mid a_2]\!] = & (a_1' \mid a_2') \textbf{ when } (a_{1g} \wedge a_{2g}) \\
\text{where} & (a_1' \textbf{ when } a_{1g}) = LW_a[\![a_1]\!] \\
 & (a_2' \textbf{ when } a_{2g}) = LW_a[\![a_2]\!] \\
LW_a[\![t = e \textbf{ in } a]\!] = & ((t' = e') \textbf{ in } a') \textbf{ when } ((t' = e') \textbf{ in } (t_g = e_g) \textbf{ in } a_g) \\
\text{where} & (e' \textbf{ when } e_g) = LW_e[\![e]\!] \\
 & (a' \textbf{ when } a_g) = LW_a[\![a[(t' \textbf{ when } t_g)/t]]\!] \\
LW_a[\![m.g(e)]\!] = & (m.g_b(e') \textbf{ when } e_g \wedge m.g_g(e') \\
\text{where} & (e' \textbf{ when } e_g) = LW_e[\![e]\!]
\end{array}
$$

Figure 5-3: Total guard lifting procedure for restricted subset of BCL Action resulting in the guard being isolated from its guard.

$$
\begin{array}{rl}
\multicolumn{2}{c}{LW_{VM} :: \text{BCL-Value-Method} \rightarrow (\text{BCL-Value-Method}, \text{BCL-Value-Method})} \\
LW_a[\![\textbf{VMeth} f \lambda x.e]\!] = & (\textbf{VMeth } f_b \lambda x.e', \textbf{ VMeth } f_g \lambda x.e_g) \\
\text{where} & (e' \textbf{ when } e_g) = LW_e[\![e]\!] \\
 & \\
\multicolumn{2}{c}{LW_{AM} :: \text{BCL-Value-Method} \rightarrow (\text{BCL-Action-Method}, \text{BCL-Value-Method})} \\
LW_a[\![\textbf{VMeth} f \lambda x.a]\!] = & (\textbf{VMeth } f_b \lambda x.a', \textbf{ VMeth } f_g \lambda x.e_g) \\
\text{where} & (a' \textbf{ when } e_g) = LW_a[\![a]\!]
\end{array}
$$

Figure 5-4: Total guard lifting procedure for restricted subset of BCL Methods. Methods are transformed into two separate methods; one for the body and one for the guard.

```
par(Rule R1 a1 when p1, Rule R2 a2 when p2) =
   Rule R1R2 (localGuard(a1 when p1 | r := dummyvalue) |
              localGuard(a2 when p2 | r := dummyvalue))
```

This allows us to avoid explicitly having to extract the guards. In the context of BSV either implementation is valid.

5.2.4 Choosing from Rules: restrict

Sometimes we wish to restrict a rule so that it is not valid to fire in certain circumstances. For instance, we want to make a rule mutually exclusive with another rule so we can compose them in parallel safely. Restricting a rule to be nondegenerate always results in a derived rule. We can express this as a derived rule where the boolean is extracted from the guard. As new rules may have any guard it is sufficient to just extract the top-level guard; this preserves our "rules as inputs" property. In general, restricting rules are used only in the context of par composition. As such we rarely need to consider the added rules when applying this composition to generate our final implementation.

```
restrict(Rule R2 a1 when p1, Rule R2 a2 when p2) =
      Rule R1R2 a2 when (¬p1 ∧ p2)
```

5.2.5 Repeating Execution: repeat

The previous three compositions gives us enough expressivity to represent any possible bounded deterministic schedulers for a BCL program. However, it is possible for an unboundedly long rule to execute. In principle we could just add a "never terminates" composition, but practically, it is more helpful to introduce a looping combinator that represents "try until failure", a strategy in execution that is very natural in software implementations. For terminating loops this could be more safely done using a loop rerolling transformation onto the fully composed rule, but having this directly is convenient.

```
repeat(Rule R1 a1 when p1) =
       Rule repeatR1
        r := true;
         loop r (r := false ; localGuard(a2 when p1; r := true))
```

5.2.6 Expressing Other Rule Compositions

With these compositions, we can generate a slew of new rule compositions by composing them. These new operators are useful in two ways. First, they can be a useful shorthand when doing compositions. For example, in the context of hardware implementation, it is common practice to execute rule $R2$ unless rule $R1$ could be done, in which case we only do $R1$. Thus we can generate the following operator from restrict and par:

```
pri(R1, R2) = par(R1, restrict(R1, R2))
```

Second, they can serve as a place rule-level optimizations; any optimization we apply within these compositions can be immediately shared when used. Consider a rule composition seq. This composition take rules $R1$ and $R2$ and tries to execute $R1$ and then (whether it was executed or not) attempts to execute $R2$. This could be expressed with our three compositions as:

```
seq(R1, R2) = let R1R2  = compose(R1,R2)
                  R1nR2 = restrict(R1R2,R1)
                  nR1R2 = restrict(R1,R2)
              in par(par(R1R2, R1nR2), nR1R2)
```

This breaks out each non-degenerate case and unions them. However, we could also express this as:

```
seq(Rule R1 a1 when p1, Rule R2 a2 when p2) =
   Rule R1R2 (localGuard(a1 when p1);localGuard(a2 when p2))
```

This representation has much more sharing and results in less circuitry if implemented in hardware or less code if implemented in software.

5.3 Scheduling in the Context of Synchronous Hardware

Consider the resulting FSM from the implementation of a BCL program P in synchronous hardware. As our grammar is complete enough to express arbitrary bounded expressions, we can *always* understand the execution of the FSM over a cycle as the application of a single rule that reads the state of the system, and calculates the new state and produces the new state. This corresponds to the dual transformation as the implementation discussed in Chapter 4.

As such, all synchronous FSM implementations can be understood as a single rule. This gives us an opportunity to better understand various scheduling algorithms. Previously, such algorithms were described via their circuits, which immediately introduced questions of the correctness of concurrency and the multiplexing of logic. In this section we reexamine these algorithms via rule compositions with an eye towards simplifying this arguments. Although we can describe these algorithms with the four compositions given previously, when appropriate we introduce other safe compositions which encapsulate the rule-level transformations that would need to be done to get the appropriate FSM.

All previous algorithms have the same restrictions we made regarding loops — that they must be unrolled completely for implementation. They also had no ability to deal with the sequential connective in the input, though they can support some limited form of sequencing between rules. Also they do not add state for scheduling purposes. As such the schedulers they generate are unfair with respect to rule selection, and the designer may need to coerce the scheduling algorithm to make a different choice for correctness, not just performance.

5.3.1 Exploiting Parallel Composition

Hoe was the first to describe [51] how to implement a rule-based system (an *abstract transition system*) in synchronous hardware. To keep hardware costs understandable

are most one circuit for any particular method was ever created. As only parallel composition is allowed, multiple instances of the same method called in the same rule can be programmatically transformed and shared. For this discussion we assume that all rules and methods are already in this form. If two rules use the same method (*e.g.,* enqueue into the same FIFO), they have a *structural hazard* and can not fire in the same cycle. This restriction does not apply to zero-argument read methods or methods with the same input expression as both can read the same value without additional circuitry.

At a high-level Hoe's scheduling algorithm merges rules using the following merge operation:

```
parCompose(Rule R1 a1 when p1, Rule R2 a2 when p2) =
        Rule R1R2 (localGuard(a1 when p1) |
                   localGuard(a2 when p2))
```

This is repeated until we have a single rule that represents the operation of a single clock cycle. Note that as boolean intersection and parallel composition are commutative, the order of this does not change the meaning of the final rule. Note also that each rule appears exactly once, meaning rule logic is never duplicated. To deal with structural hazards from both rules using the same method call, one rules one rule is replaced with a restricted version of itself (using the *restrict* composition) making the two rules mutually exclusive. This allows intra-rule level optimizations to merge any calls to the same method. While this does duplicate the guard of one of the rules into the other, as they are composed in parallel both rules observe the same state and as such the two instances of the expression can be shared.

This resolves structural hazards but may still introduce an error due to the composition. Consider a program with the following rules:

Program P_1

Rule `R1`:
`x := f(x,y)`

Rule `R2`:
`y := f(x,y)`

Register `x (0)`
Register `y (0)`

These two rules read both `x` and `y`, an operation that we can share between the two, and do not have any necessary structural hazards. However, the result of the parallel composition is not a derived rule. It produces the transition $(x_0, y_0) \longrightarrow (f(x_0, y_0), f(x_0, y_0))$. This is not what we get from executing one rule and then the other, *i.e.,* $(x_0, y_0) \longrightarrow (f(x_0, f(x_0, y_0)), f(x_0, y_0))$ or $(x_0, y_0) \longrightarrow (f(x_0, y_0), f(f(x_0, y_0), y_0))$.

Our intuition for this parallel composition was to execute two rules in parallel. As such it should be the case, that we can understand the execution of the new composed rule as executing the rules in either order. Thus, if Rule $R1$ has body $a1$ and Rule $R2$ has body $a2$, then the composition is only valid if the pair are *conflict-free*, that is:

TA $[\![$ a1 **;** a2 $]\!]$ = **TA** $[\![$ a2 **;** a1 $]\!]$ = **TA** $[\![$ a1 | a2 $]\!]$

We denote this property as $R1 <> R2$. Hoe's scheduler did not directly analyze conflict-free analysis and instead does the following approximation based on the set of read and written states of an Rule A (R_A) and (W_B) respectively:

$$\begin{aligned} A <> B \iff\ & W_A \cap W_B \neq \emptyset \wedge \\ & W_A \cap R_B \neq \emptyset \wedge \\ & W_B \cap R_A \neq \emptyset \end{aligned}$$

To resolve this the algorithm first restricts all rules so that they are all pairwise conflict-free. The choice of which rules to restrict and in what order changes the resulting behavior, but there is no way a priori to evaluate which is better. Thus the

algorithm makes an arbitrary choice. Once all the rules are conflict-free, they can be composed in parallel. As the composition is commutative, the order of this does not matter.

5.3.2 Extending for Sequentiality

Hoe's initial parallel scheduler gets a significant amount of cycle-level parallelism, given our one instance of a rule per cycle and no combinational data passing between rules in a cycle restriction. However, we still fail to get parallelism when two rules are not conflict free but can be executed in parallel in a way that is understandable. Consider the rules:

Program P_2

Rule `R1: x := f(x,y)`

Rule `R2: y := f2(y)`

Register `x (0)`
Register `y (0)`

Here, `R1` and `R2` are not conflict-free, but the parallel composition is correct. However, when both rules execute, we can understand them only as one sequence. We say that Rule $R1$ with body $a1$ is "sequenceable-before" Rule $R2$ with body $a2$ (denoted $R1 < R2$) when:

TA $[\![$ a1 **;** a2 $]\!]$ = **TA** $[\![$ a1 | a2 $]\!]$

As before this is approximated using read and write sets as:

$$A < B \iff W_A \cap W_B \neq \emptyset \wedge W_A \cap R_B \neq \emptyset$$

As only one ordering make sense, the relative order of rules becomes important. Depending on which rules are valid to fire in a cycle, a different relative order will be seen. We must assure that whatever parallel composition we end with can always be understood as a total ordering of the composite rules. There exists not total ordering

of rules that is optimal in that it allows a maximum number of rules to execute in every case. Consider the following program:

Program P_3

Rule `R1: x := y`

Rule `R2: y := z`

Rule `R3: z := x`

Register `x (0)`
Register `y (0)`
Register `z (0)`

Each pair of rules can be executed in parallel as a particular sequence. However there is no total ordering of rules of which each two-rule order can be understood; `moveX` $<$ `moveY`, `moveY` $<$ `moveZ`, and `moveZ` $<$ `moveX`.

It was also noticed that the multiplexing logic to deal with multiple rules writing to the same register is a single cycle is the same as the multiplexing logic in the context that only one rule may write a rule in a single cycle except for the control signal. Thus we can loosen our structural restriction to allow multiple ordered write method instances for registers. To resolve this, we need to fundamentally be aware of the order of rule execution in a cycle.

This idea was formed into a more efficient scheduling algorithm [52], which exploits these two improvements. The fundamental algorithm remains the same. First we restrict the rules so that they may be composed in parallel safely, and we compose them in parallel. The only difference is that we must now consider each of the exponential subsets of rules that may be valid in a cycle to determine if we have to restrict any particular rule. Doing so efficiently requires sharing decisions across multiple similar subsets of rules. The process is fairly involved and does not fundamentally add anything to this discussion. As such we do not discuss it further.

5.3.3 Improving Compilation Speed

This sequencing scheduler removed many of the unnecessary rule conflicts that reduced performance. However, it had a few major faults. First, it was extremely slow as it requires hyper-exponential time to build a final design. Second, the complexity of correctly multiplexing the correct value to each state element may cause huge additional logic costs if the decisions do not directly align. Lastly, because the understanding of what happened in a single state was so compiled, designers had significant issues understanding what was the problem if an unexpected scheduler was generated.

As a solution to this problem, Esposito [43] suggested a greedy scheduling model that established a total temporal ordering of rules. This reduces the exponential considerations to $O(n^2)$. To do this, a directed graph was constructed where each node represents a rule. An edge exists between `R1` and `R2` if `parCompose(R1,R2)` cannot be understood as trying $R2$ then trying $R1$. That is, the edge exists, if this is a required order when both rules happen in parallel. To generate a total order, we go through the nodes in some ascending priority and restrict "less important" rules in case they appear in a cycle in the graph. The resulting graph can now be linearized into our total order, and we have a known order in which to execute and can compose them as before.

5.3.4 Introducing Sequential Composition

Esposito's schedule produces high-quality FSMs with high concurrency quickly. However, it is fundamentally limited to composing rules in parallel. As such there can be no data passing between rules. One way to allow this is to introduce new primitives where the interpretation of parallel composition passes data. For instance, a register where the execution of write is visible to a read in parallel. If we forget the stored value at the end of each cycle, this corresponds to a hardware wire. This gives us the ability to emulate sequential composition without changing from the previous schedule. This approach is exactly what the current version of the BSV compiler

does.

However, this is deeply unsatisfying for many reasons. Most obviously, we no longer have a clear execution model as various parts of a rule may affect other parts of a rule; it is even possible to construct an infinite loop, *e.g.,* `wire.write(wire.read() +1)`. To prevent this we can require that a rule cannot call two methods with such a path. This is still not enough as cycles may appear through a cycle of many rules, like so:

Program P_4

Rule `R1: x := y`

Rule `R2: y := z`

Rule `R3: z := x`

Wire `x (0)`
Wire `y (0)`
Wire `z (0)`

A much greater problem is that the semantic meaning of a rule is now tied to the primitive state implementation and not to the method itself. A much better approach is to actually expose some limited form of sequential composition, but how does one concisely express this? Rosenband introduced one possibility with Performance Guarantees [80]. Performance guarantees form a sketch of the desired total ordering that the user gives to the compiler, expressing his desired concurrent execution.

Consider a program implementing the five-stage DLX processor in [71] with a rule corresponding to each of the five pipeline stages, `Fetch`, `Decode`, `Execute`, `Memory`, and `Writeback`. To allow these to be untimed, we implement each pipeline register as a single element FIFO; each rule dequeues from the source register and enqueues into the next. The expected implementation would allow each rule to happen in sequence, This corresponds to trying `Writeback` then `Memory` then `Execute`, `Decode` and finally `Fetch`. As each FIFO has only one space, this necessarily requires us to pass data, specifically the "full" bit, combinationally. This is represented by the guarantee:
`Writeback` $<$ `Memory` $<$ `Execute` $<$ `Decode` $<$ `Fetch`

The < guarantee is an associative operator accepting two sets of rules. It denotes that all rules on the left happen in a cycle, and then their updates are visible to all rules to the right. In our rule composition parlance, this can be represented by the following composition.

```
perfGurantee(Rule R1 a1 when p1, Rule R2 a2 when p2) =
      Rule R1R2 (localGuard(a1 when p1) ;
                 localGuard(a2 when p2))
```

Rosenband implemented these schedules by way of Ephemeral History Registers (EHRs) [77], which are registers with an arbitrary number of read and write ports in sequence. Effectively it is a normal register with an arbitrarily long chain of multiplexers.

To implement a guarantee, one merely has to number each method instances, to establish the order of methods each rule calls must have to implement. This is recursively repeated until we reach the primitive registers whereupon we drop in the appropriately large EHR and replace the numbered calls with references to the appropriate EHR methods.

5.4 Expressing User-defined Schedules via Rule Compositions

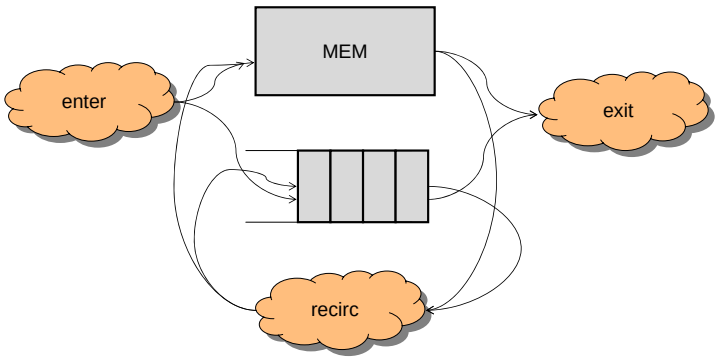
Using the rule composition primitives introduced in this chapter, the user can represent desirable derived rules. This allows the user to guide the final implementation directly and explore the tradeoff in schedules between high-level performance properties (*e.g.,* pipeline throughput) and low-level physical properties (*e.g.,* clock frequency, area, locality) without changing the rules themselves. This scheduling may be very tedious and so it is likely that this choice should be automated for each computational substrate (*e.g.,* synchronous hardware or software). The key advantage is that this description explains precisely how scheduling affects the programs execution, which is of paramount importance to the designer.

We now discuss such an exploration for implementing the circular IP lookup example from Chapter 2. For convenience, we include the code again in Figure 5-5. For this discussion, we assume we are implementing the design in synchronous hardware and as such we want a single rule implementation.

One of the most important efficiency issues in the circular pipeline is whether we are able to enter a packet into the system in the same clock cycle when another one is leaving the system. That is, can the `recirc` rule and the `enter` method execute concurrently. If these actions do not take place in the same cycle, then the system is supposed to contain a *dead cycle*. Is this a serious issue? Suppose our concrete lookup algorithm takes at most 3 lookups for each request. The dead cycle in this case would increase the total number of cycles needed to serve the incoming requests by at least 33%! The user can avoid such a dead cycle by giving an appropriate schedule. However, as we show later, exploiting this level of concurrency may increase the critical combinational path delay. A designer may want to consider the total performance when selecting a schedule.

One important implementation detail is that it is infeasible for a memory of this size to do a single cycle lookup at the frequency we want. As a result our implementation of memory will need to take multiple cycles and for concurrency we want the one-element FIFO to be replaced with a multi-element one; in our experiment we chose a size of six to match the expected latency at the desired clock frequency.

Note also that since all three rules interact with `fifo` and `mem`, scheduling these systems will imply the specific method instances of these modules. Some of these may lead to undesirable hardware. For instance, the `recirc` and `enter` cannot execute together without two memory ports. This necessarily doubles the size of the memory, but removed only one cycle of latency from the processing time of a packet (we can overlap the entering of a new packet with a recirculation). However, `exit` and `enter` use disjoint methods and do not require additional methods instances in the implementation if implemented concurrently.




```
Program IPLookup
    Module mem ...
    Module fifo ...
    Module inQ ...
    Module outQ ...

    Rule enter:
        x = inQ.first() in
        inQ.deq()
        fifo.enq(x) |
        mem.req(addr(x))

    Rule recirc:
        x = mem.res() in
        y = fifo.first() in
        (mem.resAccept()  |
         mem.req(addr(x)) |
         (fifo.deq();
          fifo.enq(f2(x,y)))
             when  !isLeaf(x)

    Rule exit:
        x = mem.res() in
        y = fifo.first() in
        (mem.resAccept() |
         fifo.deq() |
           outQ.enq(f1(x,y)))
           when isLeaf(x)
```

Figure 5-5: The Table-Lookup Program

	Schedule 1	Schedule 2	Schedule 3
Clock Period (ns)	2.0	2.0	2.0
Area (μm^2)	36,320	42,940	43,206
Max Latency (CCs)	15	18	15
Benchmark Perf. (CCs)	28,843	18,927	18,927

Worst-Case Latency refers to the maximum number of clock cycles that an operation can take to complete the pipeline. Although Schedules 2 and 3 have the same performance on our benchmark, the worst-case latencies of Schedule 2 is worse.

Figure 5-6: Implementation Results

5.4.1 The Three Schedules

We consider three schedules: Schedule 1 where `enter` and `exit` do not execute concurrently, Schedules 2 and 3 where they do, but with different priorities.

Schedule 1: `pri(recirc, pri(exit,enter))`

Schedule 2: `pri(recirc,seq(exit, enter))`

Schedule 3: `pri(seq(exit,enter), recirc)`

Schedule 1 executes only one rule per clock cycle, with `recirc` being the highest priority, followed by `exit`, then `enter`. Schedule 2 can execute `exit` and `enter` in the same cycle. It will choose to execute `recirc` over either or both of these. Schedule 3 also allows `exit` and `enter` to execute in the same cycle. However, in this schedule both of these rules take priority over `recirc`.

5.4.2 Performance Results

We evaluated each of these schedules using Bluespec Compiler version 2006.11 and synthesized using Synopsys Design Compiler version Y-2006.06 with TSMC 180 nm libraries. To deal with the variations between Bluespec and BCL, we manually performed the necessary program transformations. The performance and synthesis results are shown in Figure 5-6. To keep both area and timing comparable, we show results within 100 ps of the minimal clock period.

We can see that all schedules are able to meet a 2 ns timing requirement, but schedules 2 and 3 result in significantly larger area than Schedule 1.

Schedule 1 takes 28,803 cycles to complete our synthetic benchmark. In contrast, both schedules 2 and 3 only take 18,927 cycles, an improvement of nearly 35%. This matches our intuition of the cycle-level effect of allowing `exit` and `enter` to execute concurrently.

The same analysis shows that in the worst case an operation under schedule 2 can take 3 more cycles to complete than an operation in schedule 3. This is because when `recirc` has priority it prevents a sixth instruction from entering the memory until something has left, whereas in schedule 3, we will enter new requests until `fifo` is full (i.e. we will have 6 in-process requests). Thus, though our benchmark did not exercise this feature, the design generated from Schedule 2 has better performance in this regard.

A designer considering these three schedules would thus choose either Schedule 1 or 2, depending what mixture of area and performance is more valued.

Chapter 6

Computational Domains

A key aspect of designing in BCL is how one can split a unified BCL design into multiple *partitions*, some of which are implemented in hardware and some in software. There are numerous requirements that must hold for any sort of partitioning framework. Specifically:

1. Partitioning computation between multiple substrates, *e.g.,* hardware or software, is a first-order concern in many designs. There should be no ambiguity as to how a design is to be implemented. An important aspect of this implementation is how communication channels are constructed. Since such channels must be implemented in multiple substrates, it may seem natural to abstract the details of the construction into the language and have the compilation procedure insert communication channels as necessary. However, this would drastically reduce the ability of the designer to reason about the communication itself and hamper attempts to improve efficiency. A proper partitioning scheme must allow enough of the communication structure to be exposed within the language that the designer can reason about it.

2. Partitioning must not impede the modularity of the code. For instance, it would be highly undesirable if the hardware and software parts of a design needed to be segregated into separate disjoint modules. This would mean that we could not use library modules that have both hardware and software components without

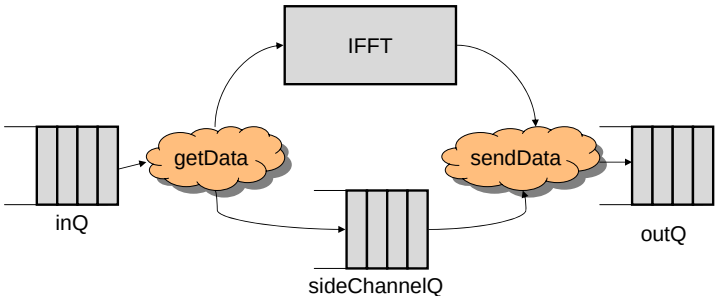
first partitioning them into their individual single-substrate components. Any restriction to our modularity due to our partitioning scheme can cause even a small change in one part of the design to have non-trivial affects on the rest of the design. As a result partitioning should be orthogonal to our notion of modularity.

3. Once partitioned, we should be able to reason about the implementation of single partition without concerns for the other partitions in the system. This isolation does not preclude whole-system reasoning, but rather asserts that low-level optimizations from one partition should not unduly restrict other portions of the design.

4. Partitioning should be not be limited to a single hardware and software partition. Our partitioning notion should allow multiple software partitions to be implemented on separate cores as well as multiple hardware partitions operating at different clocks. Similarly, it should be natural to extend the partitioning task to allow partitions implemented on new substrates like special-purpose DSPs and GPUs.

To represent this partitioning, we introduce the notion of computational *domains* within our design. Each computational domain corresponds to a computational substrate. For instance, a simple embedded system may consist of a single software domain and a single hardware domain, while a more complex system may have multiple hardware domains operating at different clock frequencies and multiple software domains operating on different processor cores.

To understand this more concretely, consider the BCL pipeline in Figure 6-1. This systems represents a simplified packet-based audio decoder pipeline targeted at an embedded mobile platform. We receive packets of data from `inQ`. We then apply a preprocessing phase splitting the data into frequency spectra and additional decoding information. The frequency data is passed to the `ifft` block, which applies an Inverse Fast Fourier Transform to convert the data to the time domain and integrated it with the rest of the data using a post-processing computation.



Program P_1

```
Rule getData:
   x = inQ.first() in
   (nx,ny) = preprocess(x) in
    (inQ.deq() |
     ifft.input(nx); |
     sideChannelQ.enq(ny))
Rule sendData:
   x = ifft.output() in
   y = sideChannelQ.first(); in
   nxy = postprocess(x,y) in
     (ifft.getOutput() |
      outData.enq(nxy) |
      sideChannelQ.deq())
Module inQ ...
Module outQ ...
Module sideChannelQ ...
Module ifft  ...
```

Figure 6-1: Original BCL Pipeline

m ::=	[**Register** r v_0]		// Regs with initial values
‖	**Module** *name*		
	[m]		// Submodules
	[**ActMeth** g = $\lambda x.a$]		//Action method
	[**ValMeth** f = $\lambda x.e$]		//Value method
‖	**PrimSync** *name* t		// Primitive Synchronizer with domain type t

Figure 6-2: Updated Module Grammar of BCL with Primitive Synchronizers

Given the relatively large computational cost of the IFFT, the designer may wish to implement the `ifft` block in hardware. However, as the remainder of the pipeline has relatively low computing requirements, it may not we worth the hardware area cost to implement the remainder of the design in hardware, saving this additional space for a faster IFFT or just reducing the area requirements.

Having decided on this partitioning of the design, we have a good sense of roughly where each particular computation must take place. The `ifft` module is entirely in hardware, the `preprocess` and `postprocess` functions should be entirely in software. What is not clear is where exactly the hand-off from hardware to software and vice versa must take place.

In principle, any possible partitioning of the part of the program that is unclear is valid. However, some partitionings are easier than others. We insist that all rules must be fully implemented in a single partition. This avoids having to coordinate a single atomic action across two or more different physical substrates.

As each rule occurs fully in a single partition, all communication must be done implicitly in modules that have methods in two or more different partitions. We call such modules *synchronizers*. Synchronizers internally must do all the complex cross-domain interactions and as a result will be nontrivial to write efficiently. However, because they fit cleanly in our module abstraction, we can effectively reuse synchronizers as a library across multiple implementations. Synchronizers can have any interface so long as it has methods in different domains. Practically, almost all synchronizers are variations of a FIFO interface – which exposes the inherent decoupling between partitions naturally. However, in some cases synchronizers with different interfaces, *e.g.,*a memory array, are used.

This partitioning approach prohibits some partitionings of a BCL program. Consider the case where the `ifft` module in the previous example is completely in hardware. As such, its methods are also implemented in hardware. This means that both rules given must also be implemented in hardware. As these are the only rules in the system, the entire system must be purely hardware. This is clearly not what we want. In this chapter, we discuss not only how this notion of partition is represented in BCL, but also how designers can modify the program to have this and other partitionings.

6.1 Representing Partitioning in BCL: Computational Domains

We represent each partition in the implementation of a BCL program as a *computational domain*. Abstractly a domain can be viewed as another type abstraction representing the partition that an object will be implemented in. Each rule, method, expression, and action has an associated domain. All objects implemented in hardware are typed with the domain "HW" and all objects implemented in software are labeled "SW". Modules do not have a single domain type; instead their domain type is a conglomeration of the types of its methods. Notice that this representation gives no guarantees about the domains of the internal rules and submodules in a module given its domain; it is possible that all methods may be in the "SW" domain but some internal operation is hardware.

Synchronizers may require extra-lingual capabilities, like interactions with a hardware bus, requiring new primitives to encapsulate the idea. As the original BCL grammar does not allow any primitive save registers, we must modify out grammar to allow new synchronizer modules. This is shown in Figure 6-2. To ensure safe usage, each primitive synchronizer is annotated with its domain type.

To allow the partitioning of a design to be easily changed, we must be able to move BCL objects, *e.g.,* rules and modules, from one domain to another easily. The natural way for this to happen is for the description of an object to be agnostic to

τ	::=	p_i	// Primitive domain (*e.g.*,HW_1, SW_1)
	‖	()	// No domain (the program domain type)
	‖	t	// Domain variable
	‖	**Module** mm	// Module domain representation containing
			// method name to domain map
mm	::=	$\{\}$	Empty Map
	‖	$mm[n \mapsto \tau]$	Map with association of name n to domain type τ

Figure 6-3: Grammar of Domain Annotations.

the partition it is implemented in as much as possible. For instance, a register should be implementable in any domain; we only really care that our notions of reading and writing occur in the same domain.

This is not possible if the description of each module or rule has a concrete domain associated with it. This does not appear directly in the instantiated BCL, but in the definitions of the module *definitions*. An obvious solution is to leverage the notion of parametric polymorphism to allow module definitions to be moved from domain to domain. This polymorphism holds true for constants and other generic expressions. We can construct the value 1 or the + operator in any domain. However, we do not want polymorphism to appear on all objects. Consider the implications of a polymorphic method? Do we implement it in hardware, software, or both? Does it depend on how we call it? We insist that all such polymorphism is restricted to the module definition; after module instantiation, the domains of all internal objects must be fixed.

We can enforce domain safety using the standard typing mechanisms. The grammar of domain annotations is listed in Figure 6-3. We present domain inference rules for BCL are presented in Figures 6-4 and 6-5. As this description is on the instantiated grammar, we do not explicitly need to deal with polymorphism; all polymorphism exists only in pre-elaborated language.

reg-update	$\dfrac{\Sigma \vdash e : \tau, \Sigma \vdash r : (\textbf{Module}\, m_r), m_r(write) = \tau}{\Sigma \vdash r := e : \tau}$
if-action	$\dfrac{\Sigma \vdash e : \tau, \Sigma \vdash a : \tau}{\Sigma \vdash \textbf{if}\ e\ \textbf{then}\ a : \tau}$
when	$\dfrac{\Sigma \vdash a : \tau, \Sigma \vdash e : \tau}{\Sigma \vdash a\ \textbf{when}\ e : \tau}$
par	$\dfrac{\Sigma \vdash a_1 : \tau, \Sigma \vdash a_2 : \tau}{\Sigma \vdash a_1 \mid a_2 : \tau}$
seq	$\dfrac{\Sigma \vdash a_1 : \tau, \Sigma \vdash a_2 : \tau}{\Sigma \vdash a_1\ ;\ a_2 : \tau}$
let	$\dfrac{\Sigma \vdash e : \tau, \Sigma \cup \{e : \tau\} \vdash e : \tau}{\Sigma \vdash t = e\ \textbf{in}\ e : \tau}$
meth-call	$\dfrac{\Sigma \vdash e : \tau, \Sigma \vdash m : \textbf{Module}\, m, m(h) = \tau}{\Sigma \vdash m.h(e) : \tau}$
loop	$\dfrac{\Sigma \vdash e : \tau, \Sigma \vdash a : \tau}{\Sigma \vdash \textbf{loop}\ e\ a : \tau}$
loopGuard	$\dfrac{\Sigma \vdash a : \tau}{\Sigma \vdash \textbf{loopGuard}\ a : \tau}$
reg-read	$\dfrac{\Sigma \vdash r : \textbf{Module}\, m, m(read) : \tau}{\Sigma \vdash r : \tau}$
const	$\vdash c : \tau$
variable	$\Sigma \cup x : \tau \vdash x : \tau$
op	$\dfrac{\Sigma \vdash e_1 : \tau, \Sigma \vdash e_2 : \tau}{\Sigma \vdash e_1\ op\ e_2 : \tau}$
if-expr	$\dfrac{\Sigma \vdash e_1 : \tau, \Sigma \vdash e_2 : \tau, \Sigma \vdash e_3 : \tau}{\Sigma \vdash (e_1\ \textbf{?}\ e_2\ \textbf{:}\ e_3) : \tau}$

Figure 6-4: Domain Inference Rules for Actions and Expression

module-reg	$\vdash (\textbf{Register}\, r\ v) : \textbf{Module}\, (\{\})[read \mapsto \tau, write \mapsto \tau])$
module-primsync	$\vdash (\textbf{PrimSync}\, n\ \tau) : \tau$
module	$\dfrac{\begin{array}{c}\forall(\textbf{ValMeth}f\ \lambda x.e) \in vm.\Sigma \vdash (\textbf{ValMeth}f\ \lambda x.e) : \tau_f, \\ \forall(\textbf{ActMeth}g\ \lambda x.a) \in am.\Sigma \vdash (\textbf{ValMeth}f\ \lambda x.e) : \tau_g, \\ \forall m \in subs.\Sigma \vdash m : \tau_{mn}, \\ mm = \{\}(\text{map}(\lambda(\textbf{ValMeth}f\ \lambda x.e).[f \mapsto \tau_f])vm) \\ (\text{map}(\lambda(\textbf{ActMeth}g\ \lambda x.a).[g \mapsto \tau_g])am),\end{array}}{\Sigma \vdash (\textbf{Module}\, subs\ am\ vm) : \textbf{Module}\, mm}$
rule	$\dfrac{\Sigma \vdash a : \tau}{\Sigma \vdash (\textbf{Rule}\, R\, a) : \tau}$
value-method	$\dfrac{\Sigma[x : \tau] \vdash e : \tau}{\Sigma \vdash (\textbf{ValMeth}\ f\ \lambda x.e) : \tau}$
action-method	$\dfrac{\Sigma[x : \tau] \vdash a : \tau}{\Sigma \vdash (\textbf{ActMeth}\ f\ \lambda x.a) : \tau}$
program	$\dfrac{\begin{array}{c}\forall m \in mods.\Sigma \vdash m : \tau_m, \\ \forall r \in rules.\Sigma \vdash r : \tau_r,\end{array}}{\Sigma \vdash (\textbf{Main}\, mods\ rs) : ()}$

Figure 6-5: Domain Inference Rules for Programs, Modules, Rules, and Methods

6.2 Partitioning with Computational Domains

To partition a BCL program we must introduce synchronizers to separate the design. In the simplest case, this is merely replacing a module with a synchronizer that has the same semantic properties. For instance consider the program:

Program P_2

```
Rule swRule:
   (r1 := f1(r1) |
    fifo.enq(f3(r)))

Rule hwRule:
   x = fifo.first() in
   (fifo.deq() |
    r2 := f2(r2))

Module fifo ...
Register r1 (0)
Register r2 (0)
```

Assuming a standard one-element FIFO implementation with registers, this system has only one domain. To partition it so that `hwRule` is implemented in hardware and `swRule` is implemented in software, we merely have to replace the `fifo` module with a primitive FIFO synchronizer giving us the new program:

Program P_3

```
Rule swRule:
   (r1 := f1(r1) |
    fifo.enq(f3(r)))

Rule hwRule:
   x = fifo.first() in
   (fifo.deq() | r2 := f2(r2))

PrimSync fifo ({}[enq ↦ SW]
                 [deq ↦ HW]
                 [first ↦ HW])
Register r1 (0) // in SW
Register r2 (0) // in HW
```

Assuming the primitive synchronizer acts as a FIFO and can hold the same maximum number of elements as the original `fifo` module this is P_2 has the exact same one-rule-at-a-time execution semantics as P_3; the partitioning only comes into play in the implementation, not in the program semantics.

Practically a synchronizer does not precisely match an obvious register-based equivalents due to important efficiency concerns. For instance, consider an efficient software-to-hardware FIFO synchronizer implemented on a shared bus. To do the communication, the software part of the communication must grab the bus and marshal the data onto the bus, while the hardware part must constantly check for writes that it unmarshals and adds to an internal FIFO. To make sure software does not send data when hardware does not have space, hardware must communicate back to the software over the bus how much space it has available. This is generally done with a token-counting scheme. We do not want to have to wait for the round-trip communication to send or receive a message. Rather it should be able to leave that information at the synchronizer rendezvous to be sent later; this also allows us to improve interpartition communication drastically by sending multiple messages in bursts. The net result of this decision is that the "full" and "empty" guard signals do not propagate atomically with `enq` and `deq` methods, clearly differentiating the synchronizer from the naïve hardware FIFO. This synchronizer is actually equivalent to a module with three FIFOs in sequences with two separate rules which dequeuing data from one FIFO and enqueuing it into the next one; the delays in signal propagation are represented by the choice of rule execution. For the replacement of `fifo` by the synchronizer to be correct, it must also be the case that replacing it with three FIFOs should preserve our high-level correctness guarantee.

6.2.1 Modifying the Program for Partitioning

Sometimes, it is not reasonable to merely replace some state with synchronizers. For instance in our audio pipeline (Program P_1) we could replace some of the internal state of the `ifft` module with synchronizers. However, `ifft` is a well understood computational core; it's likely that we want to use a library module and not have to

modify it so that communication would not have a significant impact on performance. Instead it makes sense to modify the program to add new state elements to the program to serve as our partitioning point. As before, this change possibly changes the structure of the program and thus we must be sure that the new program is correct.

In the case of our audio pipeline we can exploit the fact that the `ifft` operates in a latency insensitive manner to introduce new state; we can split the rules sending and taking data from the `ifft` module into two: one responsible for doing the data transfer to the `ifft` and the other to do the rest of the original rule's work. The resulting new program (P_4) from splitting the rules is described in Figure 6-6. These rules interact with one of the two logical FIFOs that we have added to hold the messages between the two partitions. These FIFOs are implemented as a single module – a bidirectional synchronizer with methods corresponding to each of the methods from both FIFOs. Internally, they both marshal data on the same bus; the unified representation lets us share circuitry and logic between the channels, both of which require bidirectional communication to pass token information.

One could imagine that for each for each possible set of communication channels over a single shared medium, we would want a new primitive synchronizer to properly allow for sharing. Practically, this single bidirectional channel will work for all systems with latency insensitive communication; we can model larger ones by explicitly representing what would have been the internal multiplexing logic as a BCL shim around this primitive, allowing us to ask about the correctness of the new synchronizers as a BCL refinement problem as we discuss in more detail in Chapter 8.

6.3 Isolating Domains for Implementation

This domain-based partitioning allows the user to represent how the computation should be partitioned without affecting the modular decomposition of the program itself. This is extremely important for partition exploration. However, it requires the interactions of partitions to be handled by the compiler. This can in principle be

Program P_4

```
Rule getDataSW:
   x = inQ.first() in
   (nx,ny) = preprocess(x) in
     (inQ.deq() |
      hwswSync.swenq(nx) |
      sideChannelQ.enq(ny))

Rule getDataHW:
   ifft.input(hwswSync.hwfirst()) |
   hwswSync.hwdeq()

Rule sendDataSW:
   x = hwswSync.swfirst() in
   y = sideChannelQ.first(); in
   nxy = postprocess(x,y) in
     (hwswSync.swdeq() |
      outQ.enq(nxy) |
      sideChannelQ.deq())

Rule sendDataHW:
   hwswSync.hwenq(ifft.output()) |
   ifft.getOutput()

Module inQ ...
Module outQ ...
Module sideChannelQ ...
Module ifft  ...
PrimSync hwswSync ({}[hwenq ↦ HW]
                    [hwdeq ↦ HW]
                    [hwfirst ↦ HW]
                    [swenq ↦ HW]
                    [swdeq ↦ HW]
                    [swfirst ↦ HW])
```

Figure 6-6: Pipeline Example with IFFT put in hardware

.

extremely complicated as the compiler may need to reason about every partition at once to correctly orchestrate the computation without violating the execution semantics. However, our domain abstraction requires all such issues be handled completely by the synchronizers. As such, each domain can be isolated into its own subprogram, which can be compiled without worrying about the rest of the design.

Intuitively, we can extract one domain from a program. This is just a matter of removing all methods and rules from the program that are not typed with that particular domain. To deal with primitive synchronizers that are monolithic and cannot be obviously reduced, we replace them with a primitive representing the part of the synchronizer in that domain. This programmatic description corresponds nicely to the actual synchronizer implementation as the synchronizer must have a subcomponent in the appropriate computational substrate.

To better understand how this works, consider the bus-based synchronizer from our audio pipeline example. The synchronizer primitive consists of a software component and a hardware component, each of which has an interface corresponding to the methods of the synchronizer in each domain.

In addition to the user-exposed methods, each component has an additional interface to communicate between them. For the hardware component this is a physical bus interface on which the entire hardware partition sits. Software has an interface to the processor's bus interface as a memory-mapped region in the address space. The software component is able to deal with the hardware-software communication via a series of loads and stores. Hardware can notify software of the need to do work by raising an interrupt. Depending on the desire to exploit bursts and improve latency, this interface may become very complicated.

6.4 A Design Methodology in the Presence of Domains

One of the main goals of partitioning is to allow the designer to be able to partition a design in many different ways. However, as we have discussed in general partitioning a design presents a correctness requirement as we cannot just replace some module with a synchronizer without changing the rules. Thus when designing a BCL program where the partition is not necessarily clear, the designer needs to be "overpartition" the design, *i.e.,* construct many more partitions than would be desired in the final implementation. By assigning the same domain to many communicating domains, we can construct a partitioning with a realistic partitioning granularity. To make this possible we need only to assign the appropriate synchronizers (or normal modules if the domains are the same). Overpartitioning imposes an additional burden on the designer over what is needed had they been designing for one particular partitioning. It is, however, work that fundamentally must be done to explore various partitioning possibilities.

There are multiple ways for a designer to do go about doing this. One effective approach is to limit the interactions between parts of the program that may be implemented in different domains in a latency insensitive fashion or other style which allows us to drastically change the components.

Some of this may be simplified by automatic transformations. An effective example would take a rule, split it in two and add a FIFO module to pass the relevant data, modifying the guards of all rules in the system that are not conflict-free to fail to prevent unexpected behavior. Our hand translation of the audio pipeline did exactly this. The major concern with such approaches is the efficiency of the resulting program. Significant exploration into partitioning must be done before such automation becomes practically desirable.

Chapter 7

Software Generation

While the purpose of this thesis involves primarily the language-level aspects of the solution, significant implementation was necessary to motivate the changes and choices in the language. In this chapter we discuss the compilation process and the obvious optimizations one might wish to do.

With the exceptions of loops and sequential composition, the translation of BCL to an RTL-level hardware description via BSV is relatively straightforward. Since BSV does not support loops or sequential composition, there is currently no support for the compilation of these constructs when they occur in hardware partitions. However, since multiple theses have discussed strategies for their implementation in hardware [55, 78], we consider this a "solved" problem. This chapter presents a compilation strategy for a software implementation in C++ for the full BCL language.

7.1 Rule Canonicalization

The translation of a single rule to a straight-line C++ program is, in most regards, very similar to the functionalization of BCL given in Chapter 3.

The first major difference is due to state construction. For each action, we must construct a new state value. However, as this construction is expensive requiring the copying of the entire state, we would like to simplify modify state values in place if possible, *i.e.,* the input state is only used once. Then we would need to copy only when

explicit duplication is necessary. This sort of optimization is done implicitly within most functional compilers; as such this detail is not reflected in our functionalization. However, it must appear in our translation to C++.

The other substantial difference in the compilation strategy presented here involves the linearization of parallel composition (a result of the fact that the generated C++ is single threaded). Unlike the normal-order λ-calculus, C++ is strict in binding and function arguments. If we implemented the functionalization directly, we would always execute the entire rule body, even in the event of an early guard failure. A more efficient translation would abort immediately upon encountering a failing guard, avoiding the unnecessary work of completing the action. Doing this presents a complication, since bound expressions in BCL have inlining semantics with regards to guard failure; that is, the failures from bound expressions happen only when the associated variable is used. The following expression:

```
x = (() when False)
in 0
```

always evaluates to 0 and never encounters the guard failure. Since C++ evaluates variables strictly, we must make sure that we do not fail prematurely (or in this case: ever). Similar concerns arise for $\perp$ but can be dismissed, since expressions cannot represent infinite computation.

Instead of the naïve translation's approach of keeping guards as pure data, we transform our program so that no bound expression, either from let bindings of value methods definitions, contains a **when** clause. This change not only prevents unnecessary computation in translated rule bodies, but reduces the code to deal with the guard predicates. Intuitively, we "push" all internal **when**s to the top-level by lifting **when**s using the axioms in Figure 2-8. Then we transform these bound top-level guards by splitting the binding into a body part and a guard part and moving the `when`s into the call sites. This can be done mechanically with the following procedure:

1. Replace all value method invocations `m.f(e)` with `m.f`$_{Body}$`(e) when m.f`$_{Guard}$`(e)`. Similarly replace any let-bound variable instances `x` whose value is a when ex-

pression, with the expression of two fresh variables `xb` **when** `xg`.

2. Lift all whens in expression to the top-level using the procedure in Figure 7-1. After this, all guards exist only in top-level expressions (*i.e.,* those which are not directly in subexpressions, only actions and method bindings).

3. Replace all value method definitions `m.f = λ x. e when g` with the two method definitions:

   - $\texttt{m.f}_{Body}$ `= λ x.e`
   - $\texttt{m.f}_{Guard}$ `= λ x.g`

4. Similarly convert all let bindings of variable `x` to define the two fresh variables. This defines the variables we used but not defined in Step 1,.

```
x = eb when eg in e ⇒
      xb = eb in xg = eg in e
```

7.2 Syntax-Directed Compilation

Each BCL module definition is compiled into a C++ class, and each of the module's rules and methods are compiled into a separate class method. To handle guard failures and atomicity issues arising from concurrent rule execution, we take a lazy transactional memory approach and create shadow copies of the object state, which are committed only after an action has completed without guard failures and no data conflicts are detected that would violate the sequential consistency of the execution.

We present a syntax-directed compilation of BCL. For the sake of brevity we take a few notational liberties in describing translation rules. Generated C++ code is represented by the conjunction of three different idioms: literal C++ code (given in the `fixed-width` font), syntactic objects that evaluate to yield C++ code (given in the document font), and environment variables used by the compiler procedures

$$
\begin{array}{rl}
LW_e[\![r]\!] = & (r \textbf{ when } \texttt{true}) \\
\\
LW_e[\![c]\!] = & (c \textbf{ when } \texttt{true}) \\
\\
LW_e[\![t]\!] = & (t \textbf{ when } \texttt{true}) \\
\\
LW_e[\![e_1 \; op \; e_2]\!] = & (e'_1 \; op \; e'_2) \textbf{ when } (e_{1g} \wedge e_{2g}) \\
\text{where} & \quad (e'_1 \textbf{ when } e'_{1g}) = LW_e[\![e_1]\!] \\
& \quad (e'_2 \textbf{ when } e'_{2g}) = LW_e[\![e_2]\!] \\
\\
LW_e[\![e_1]\!] \; \textbf{?} \; e2 \; \textbf{:} \; e3) = & (e'_1 \; \textbf{?} \; e'_2 \; \textbf{:} \; e'_3) \textbf{ when} \\
& \quad (e_{1g} \wedge (e'_1 \; \textbf{?} \; e_{2g} \; \textbf{:} \; e_{3g})) \\
\text{where} & \quad (e'_1 \textbf{ when } e_{1g}) = LW_e[\![e_1]\!] \\
& \quad (e'_2 \textbf{ when } e_{2g}) = LW_e[\![e_2]\!] \\
& \quad (e'_3 \textbf{ when } e_{3g}) = LW_e[\![e_3]\!] \\
\\
LW_e[\![e_1 \textbf{ when } e_2]\!] = & e'_1 \textbf{ when } (e'_2 \wedge e_{1g} \wedge e_{2g}) \\
\text{where} & \quad (e'_1 \textbf{ when } e_{1g}) = LW_e[\![e_1]\!] \\
& \quad (e'_2 \textbf{ when } e_{2g}) = LW_e[\![e_2]\!] \\
\\
LW_e[\![t = e_1 \textbf{ in } e_2]\!] = & ((t' = e'_1) \; ; \; e'_2)) \textbf{ when} \\
& ((t' = e'_1) \; ; \; (t_g = e_{1g}) \; ; \; e_{2g}]\!] \\
\text{where} & \quad (e'_1 \textbf{ when } e_{1g}) = LW_e[\![e_1]\!] \\
& \quad e_3 = e_2[(t' \textbf{ when } t_g)/t] \\
& \quad (e'_2 \textbf{ when } e_{2g}) = LW_a[\![e_3]\!] \\
\\
LW_e[\![m.f(e)]\!] = & m.f(e') \textbf{ when } e_g \\
\text{where} & \quad (e' \textbf{ when } e_g) = LW_e[\![e]\!]
\end{array}
$$

Figure 7-1: Procedure to lift **when** clauses to the top of all expressions of BCL. This is the same as the expression lifting procedure of the restricted language in Figure 5-2. Method calls and bound variables are expected to already be split between body and guard.

(represented as symbols). The names of compiler procedures that generate code fragments are given in **boldface**.

7.2.1 Compiling Expressions

The translation of a BCL expression produces a C++ expression and one or more statements that must be executed before the expression. These statements are responsible for executing parts of the expression that do not have corresponding representations in C++ expressions such as let bindings. The procedure to translate expressions (**TE**) is shown in Figure 7-2. The **when** transformations we performed during rule canonicalization have modified the structure in such a way that a direct translation to the call-by-value semantics of C++ is equivalent to the original call-by-name semantics of BCL. Thus, upon the evaluation of a failed guard, the execution can immediately throw away all speculative work. Since execution occurs in speculative state, we can accomplish this by simply throwing an exception (which is caught by the lexically outermost action enclosing the expression) and fail to commit the speculative state.

7.2.2 Compiling Actions

A rule is composed of actions, which may be guarded. Earlier we explained the meaning of a guarded action by saying that a rule is not eligible to fire (execute) unless its guard evaluates to true. However, due to conditional and sequential composition of actions, in general it is impossible to know if the guards of all the constituent actions of a rule are true before we execute the rule. To circumvent this limitation, we execute a rule in three phases: In the first phase we create a shadow of all the state elements using the `copy` constructor. We then execute all constituent actions, updating the shadow state. Sometimes we need more shadows to support the internal actions, *e.g.,* for parallel composition we operate each of the two composed actions in a separate shadow which we later compose. Finally, if no guard failures are encountered we commit the shadows, that is, atomically update the original state variables with

TE :: **Env** × ⟦ e ⟧ → (CStmt, CExpr)

$$
\begin{aligned}
\textbf{TE}\ \rho\ [\![\ \text{r}\ ]\!] &= (;,\ \rho[\text{r}].\text{read}()) \\
\\
\textbf{TE}\ \rho\ [\![\ \text{c}\ ]\!] &= (;,\ \text{c}) \\
\\
\textbf{TE}\ \rho\ [\![\ \text{t}\ ]\!] &= (;,\ \text{t}) \\
\\
\textbf{TE}\ \rho\ [\![\ \text{e1}\ \textbf{op}\ \text{e2}\ ]\!] &= (\text{s1;s2},\ \text{ce1}\ \textbf{op}\ \text{ce2}) \\
\text{where}\quad (\text{s1},\ \text{ce1}) &= \textbf{TE}\ \rho\ [\![\ \text{e1}\ ]\!] \\
(\text{s2},\ \text{ce2}) &= \textbf{TE}\ \rho\ [\![\ \text{e2}\ ]\!] \\
\\
\textbf{TE}\ \rho\ [\![\ \text{ep}\ \textbf{?}\ \text{et}\ \textbf{:}\ \text{ef}\ ]\!] &= (\text{sp};\ \text{st};\ \text{sf},\ \text{cep}\ \texttt{?}\ \text{cet}\ \texttt{:}\ \text{cef}) \\
\text{where}\quad (\text{sp},\ \text{cep}) &= \textbf{TE}\ \rho\ [\![\ \text{ep}\ ]\!] \\
(\text{st},\ \text{cet}) &= \textbf{TE}\ \rho\ [\![\ \text{et}\ ]\!] \\
(\text{sf},\ \text{cef}) &= \textbf{TE}\ \rho\ [\![\ \text{ef}\ ]\!] \\
\\
\textbf{TE}\ \rho\ [\![\ \text{e}\ \textbf{when}\ \text{ew}\ ]\!] &= (\text{sw};\ \text{mthrow};\ \text{se},\ \text{ce}) \\
\text{where}\quad (\text{se},\ \text{ce}) &= \textbf{TE}\ \rho\ [\![\ \text{e}\ ]\!] \\
(\text{sw},\ \text{cw}) &= \textbf{TE}\ \rho\ [\![\ \text{ew}\ ]\!] \\
\text{mthrow} &= \texttt{if(!cw)\{throw GuardFail;\}} \\
\\
\textbf{TE}\ \rho\ [\![\ \text{t} = \text{et}\ \textbf{in}\ \text{eb}\ ]\!] &= (\text{st};\ \text{t} = \text{ct};\ \text{sb},\ \text{cb}) \\
\text{where}\quad (\text{st},\ \text{ct}) &= \textbf{TE}\ \rho\ [\![\ \text{et}\ ]\!] \\
(\text{sb},\ \text{cb}) &= \textbf{TE}\ \rho\ [\![\ \text{e}\ ]\!] \\
\\
\textbf{TE}\ \rho\ [\![\ \text{m.f(e)}\ ]\!] &= (\text{se},\ \rho[\text{m}]\texttt{.f(ce)}) \\
\text{where}\quad (\text{se},\ \text{ce}) &= \textbf{TE}\ \rho\ [\![\ \text{e}\ ]\!]
\end{aligned}
$$

Figure 7-2: Translation of Expressions to C++ expression and the C++ statement to be evaluated for expression to be meaningful

values of the shadowed state variables. On the other hand if the evaluation encounters a failed guard, it aborts the computation and the original state is not updated.

For perspicuity, the rules present an inefficient but simple translation where shadows of the entire environment are created whenever a shadow may be needed. Figure 7-3 gives the procedure for translating BCL actions (**TA**).

State Assignment ($r := e$): This causes a side-effect in the relevant part of the state of the object that can be extracted from ρ. If e evaluates to bottom, the control would have already been transferred automatically up the call stack via the `throw` in e.

Parallel Composition ($a1 \mid a2$): Both $a1$ and $a2$ observe the same initial state, though they update the state separately. Consider the parallel action $r_1 := r_2 | r_2 := r_1$, which swaps the values of r_1 and r_2. Such semantics are naturally implemented in hardware as swaps can be done with no intermediate state (the values are read in the beginning of a clock cycle and updated at the end of it). However, in software if we update r_1 before executing the second action, the second action will read the new value for r_1 instead of the old one. To avoid this problem, the compiler creates shadow states for each parallel action, which are subsequently merged after both actions have executed without guard failures. In a legal program, the updates of parallel actions must be to disjoint state elements. Violation of this condition can be detected only dynamically, in which case an error is thrown.

The compiler uses several procedures to generate code to be used in implementing parallel composition. The **makeShadow** procedure takes as its argument an environment (ρ) and returns a tuple consisting of a new environment (say $\rho 1$), and C++ statements (say cs1). cs1 is executed to declare and initialize the state elements referenced to in $\rho 1$. The new environments are then used in the translation of each of the actions. The procedure **unifyParShadows** is used to unify $\rho 1$ and $\rho 2$, implicitly checking for consistency. Along with $\rho 3$, which contains the names of the unified state elements, it returns a C++ statement (pm) that actually implements the unification. Lastly, the **commitShadow** procedure generates code (ms) to commit the

TA :: **Env** × ⟦ a ⟧ → CStmt

TA ρ ⟦ r **:=** e ⟧ = se; ρ[r]`.write(`ce`)`;
where (se, ce) = **TE** ρ ⟦ e ⟧

TA ρ ⟦ **if** e **then** a ⟧ = se; `if(`ce`){`**TA** ρ ⟦ a ⟧`}`
where (se, ce) = **TE** ρ ⟦ e ⟧

TA ρ ⟦ a1 | a2 ⟧ = cs1; cs2; (**TA** ρ1 ⟦a1⟧); (**TA** ρ2 ⟦a2⟧); pm; ms;
where (cs1, ρ1) = **makeShadow** ρ
(cs2, ρ2) = **makeShadow** ρ
(pm, ρ3) = **unifyParShadows** ρ1 ρ2
ms = **commitShadow** ρ ρ3

TA ρ ⟦ a1**;**a2 ⟧ = cs; (**TA** ρ1 ⟦a1⟧); (**TA** ρ1 ⟦a2⟧); ms;
where (cs, ρ1) = **makeShadow** ρ
ms = **commitShadow** $\rho\rho$1

TA ρ ⟦ a **when** e ⟧ = se;`if(!`ce`){throw GuardFail;}`;ca
where (se, ce) = **TE** ρ ⟦ e ⟧
ca = **TA** ρ ⟦ a ⟧

TA ρ ⟦ t = e **in** a ⟧ = se; `t = `ce`;` (**TA** ρ ⟦ a ⟧)
where (se, ce) = **TE** ρ ⟦ e ⟧

TA ρ ⟦ m.g(e) ⟧ = se; (ρ[m]`.g(`ce`)`);
where (se, ce) = **TE** ρ ⟦ e ⟧

TA ρ ⟦ **loop** e a ⟧ = cs; `while(true){`se;
`if(!`ce`) break;` ca;`}` ms;
where (cs,ρ1) = **makeShadow** ρ
(se, ce) = **TE** ρ ⟦ e ⟧
ms = **commitShadow** ρ ρ1
ca = **TA** ρ1 ⟦a⟧

TA ρ ⟦ **localGuard** a ⟧ = `try{do{` cs;
ca; ms;`}while(false);}`catch {}
where (cs,ρ1) = **makeShadow** ρ
ms = **commitShadow** ρ ρ1
ca = **TA** ρ1 ⟦a⟧;

Figure 7-3: Translation of Actions to C++ Statements

speculative state held in $\rho 3$ back into the original state ρ.

In order to understand **unifyParShadows**, consider the parallel merge of two primitive Registers found in Figure 7-7. The reader should be able to extrapolate the implementation of other primitive modules that we may wish to have, *i.e.,* a `VectorReg`. The `parMerge` for this module would most likely allow updates to disjoint locations in parallel branches of execution, but throw an error if the same location were written. **unifyParShadows** generates code that recursively invokes `parMerge` pointwise on the two environments, whereas **commitShadow** ρ $\rho 1$ simply performs pointwise updates of objects in ρ from dirty objects in $\rho 1$ by invoking `seqMerge`.

Sequential composition ($a1; a2$): This composition translates very closely to the C++ model. We take our notion of state of the system and apply $a1$ effects and then $a2$ effects. As the sequential rule in Figure 7-3 shows, after creating the shadow $\rho 1$, we pass it to the translation of both $a1$ and $a2$. If no failures occur during that computation, the code block ms commits the resulting state. Notice that this translation requires a new shadow for every sequential composition. However, $a1; a2; a3$ can all be executed using only one shadow. We address this obvious inefficiency in Section 7.4

Guarded Action (a when e): Actions are called only when they are to be used. As such when we run into a failing **when** guard in an action we know that the entire action must fail. We do this by evaluating e and throwing a `guardFail` exception if ce evaluates to false. This will immediately jump us out of the rule body skipping the final shadow commit. If the guard is true, we move on to executing the code for a.

Atomic Loop (loop e a): Loops translate directly into a C++ `while` loop. Much as with the sequential composition, a shadow must be made before the loop to deal with failures midway through the execution.

Protected Loop (loopGuard e a): The difference between the protected loop and the atomic loop is in the termination semantics. Since the protected loop doesn't

throw away the entire rule body on a failure, we must make a shadow of the state for each iteration. These shadows are merged back in after the iteration successfully completes.

7.2.3 Compiling Rules and Methods

Figure 7-4 gives the translation of rules and methods, differentiating between action and value methods. The important thing to note is that rules catch guard failures, whereas methods do not. This is consistent with our implementation of the BCL semantics that specify rules as *top-level* objects that cannot be invoked within the language. Methods, on the other hand, must be called from inside a rule or another method.

7.2.4 Compiling Modules

The C++ class corresponding to a BCL module has five methods in addition to a method for each rule and method in BCL: a default constructor (used to initiate a module instance and instantiate its submodules recursively), a copy constructor (used to generate shadows, also recursive), `ParMerge` the parallel merging operator we've discussed, `SeqMerge` to merge a shadow into the shadow (including the original state) from which it was generated, and `execSchedule` (which tries to run each rule in the module in order). These methods are indirectly used during the compilation of actions through calls of the helper functions given in Figure 7-5. All of these methods are recursively definable by the corresponding methods of their submodules. Figure 7-6 gives their generic description and Figure 7-7 gives the full definition of the register primitive.

7.3 The Runtime: Constructing `main()`

After compilation of each module definition, we can instantiate our system to its initial state and enumerate the individual rules from each module instance. Now

```
genRule ⟦ (Rule nm a) ⟧ =
    void nm(){
     try{
       TA *this ⟦ a ⟧
     } catch{//guard failure };

genAMeth ρ  ⟦ (AMeth nm v a) ⟧ =
    void nm(t v){
     TA *this ⟦ a ⟧
    }

genVMeth ρ  ⟦ (VMeth nm v e) ⟧ =
    let (se,ce) = TE *this ⟦ e ⟧
    in void nm(t v){
       se;
       return ce;
     }
```

Figure 7-4: Translation of Rules and Methods. The initial state is the current object which is the "real" state in the context that we are calling. Thus if we call a method or rule on a shadow state, we will execute do its execution in that state.

```
Env :: BCLExpr → CExpr

makeShadow :: Env → ([CStmt],Env)
makeShadow ρ  = (copy_stmts, new_mapping)
    where   sh ⟦ e↦n ⟧ = (new t = n.copy(),[e↦t])
            (copy_stmts, new_mapping) = unzip (map sh ρ)

commitShadow :: Env × Env → [CStmt]
commitShadow ρ1 ρ2 =
    map (λ(⟦ e↦n ⟧). e.SeqMerge(ρ2[n])) ρ1

unifyParShadows :: Env × Env → ([CStmt],Env)
unifyParShadows ρ1 ρ2 = (merge_stmts, new_mapping)
    where   sh [n↦n] = (new t = n.ParMerge(ρ2[e]), [e↦t])
            (merge_stmts, new_mapping) = unzip (map sh ρ1)
```

Figure 7-5: Helper Functions used in Action compilation

```
TM :: ⟦ m ⟧ → CClassDef

TM ⟦ (ModuleDef name args insts rules ameths vmeths) ⟧ =
class name {
  public:
    map (λ⟦ (Inst mn n _) ⟧. mn* n;) insts

    name(){
      map(λ⟦ (Inst mn n v) ⟧. n=new mn(v);) insts
    }

    name* copy(){
      name rv = new name;
      map (λ⟦ (Inst mn n _) ⟧. rv.n = n.copy();) insts
      return rv;
    }

    ~name(){map (λ⟦ (Inst _ n _) ⟧.delete n;) insts}

    void ParMerge(name shadow){
      map (λ⟦ (Inst mn n _) ⟧.  n.ParMerge(shadow.n)) insts
    }

    void SeqMerge(name shadow){
      map (λ⟦ (Inst mn n _) ⟧.  n.ModuleMerge(shadow.n)) insts
    }

    map genRule rules
    map genAMeth ameths
    map genVMeth vmeths
    void execSchedule(){
      map (λ⟦ (Rule n a) ⟧.n();) rules
      map (λ⟦ (Inst mn n _) ⟧.n.execSchedule();) insts} }
```

Figure 7-6: Translation of Modules Definitions to C++ Class Definition

```
template<typename T> class Reg{
  public:
     bool   modified;
     T         value;
     Reg<T>  *parent;

     inline  T& read(){
       return (modified) ? value : parent->read();
     }
     inline void write(const T& new_val){
       modified = true;
       value = new_val;
     }
     Reg<T>(T def): //constructor
       modified(true),
       value(def),
       parent(NULL)
       { }
     Reg<T>(class Reg<T>& ref): //copy constructor
       modified(false), // copied shadows are not
modified
       parent(&ref)
       { }
     ~Reg<T>() {//destructor
       value.~T();
     }
     inline void ModuleMerge(class Reg<T>& a){
       modified |= a.modified;
       if (a.modified){value = a.value; }
     }
     inline void ParMerge(class Reg<T>& a){
       if(a.modified){
         if(modified){throw ParMergeFail;}
         value = a.value;
         modified = true;
       }
     }
     inline void execSchedule(){}
   };
```

Figure 7-7: Implementation for C++ Register class. This primitive is templated to hold any type. A point to the parent is made to avoid unnecessary duplication of large values.

```
TP :: [[ mds ]] × ModInst → CProgram
TP [[ ms ]] (md mn v) =   map TM ms;
                          int main(void){
                            se;
                            mn topState = new mn(ce);
                            while(true){
                              topState.execSchedule();
                            }
                          }
```

Figure 7-8: Simple Top-Level Runtime Driver

all that is left is the driving runtime loop. We present in the simplest scheduler in Figure 7-8, which tries each rule in sequence repeatedly. This matches exactly a fair interpretation of the execution semantics. With the addition of this driver we now have a complete C++ implementation.

7.4 Software Optimizations

Having implemented a naïve translation, the runtime speed of a BCL translated to software is roughly on par with the speed of the BCL program translated to RTL via Bluespec and run through an efficient RTL simulator. This is orders of magnitude off from hand-written RTL. Some of this can be addressed with standard optimizations such as inlining and loop unrolling. In this section we discuss a set of nonstandard optimizations that drastically improve the performance of software implementation.

7.4.1 Shadow Minimization

A large portion of the memory and computational overhead for our initial translation stems from copying and initialization associated with shadows. However, much of this can be removed by static analysis of the computation structure. We use two minimization techniques to reduce the shadow generation.

Context-Aware Shadow Generation

Once we have a shadow for a particular state in a rule, we need not construct a new shadow to prevent premature writes; we can reuse the current shadow. This can save significant overhead, especially when dealing with sequences of sequential actions. We devise a new translation scheme that makes use of two state maps represented as a set of shadow maps, an "initial" state, which represents the place where updates are finally committed and the "active" state (which represents the current shadow copy). These form our new ρ "state" representation in our translation of expressions, actions, rules, and methods presented in Figures 7-9, 7-10, and 7-11; the translation of module definitions remains the same as before. The translation remains fairly direct, with the notable additional concern that we must guarantee that new shadow states are always in scope for their eventual merges. The helper functions in Figure 7-12 reflect this.

TE :: **Env** × ⟦ e ⟧ → (CStmt, CExpr, **Env**)

TE ρ ⟦ r ⟧ = (merges, ρ'[r].read(), ρ')
where (merges, ρ') = getReadState(r, ρ)

TE ρ ⟦ c ⟧ = (;, c, ρ)

TE ρ ⟦ t ⟧ = (;, t, ρ)

TE ρ ⟦ e1 **op** e2 ⟧ = (s1;s2;, ce1 **op** ce2, ρ")
where (s1, ce1, ρ') = **TE** ρ ⟦ e1 ⟧
(s2, ce2, ρ") = **TE** ρ' ⟦ e2 ⟧

TE ρ ⟦ ep **?** et **:** ef ⟧ = (sp;st;sf;, cep `?` cet `:` cef, ρ"')
where (ms, ρ0) = **guaranteeReadState** ρ
(ReadState et) ∪ (ReadState ef)
where (sp, cep, ρ') = **TE** ρ ⟦ ep ⟧
(st, cet, ρ") = **TE** ρ' ⟦ et ⟧
(sf, cef, ρ"') = **TE** ρ" ⟦ ef ⟧

TE ρ ⟦ e **when** ew ⟧ = (sw; se; mthrow;, ce, ρ")
where (sw, cw, ρ') = **TE** ρ ⟦ ew ⟧
(se, ce, ρ") = **TE** ρ' ⟦ e ⟧
mthrow = `if(!cw){throw GuardFail;}`

TE ρ ⟦ t = et **in** eb ⟧ = (st; t = ct; sb, cb, ρ")
where (st, ct, ρ') = **TE** ρ ⟦ et ⟧
(sb, cb, ρ") = **TE** ρ' ⟦ eb ⟧

TE ρ ⟦ m.f(e) ⟧ = (merges;se;, ρ'[m]`.f(ce)`, ρ')
where (merges, ρ') = getReadState(m, ρ)
(se, ce, ρ") = **TE** ρ' ⟦ e ⟧

Figure 7-9: Shadow Minimized Translation of BCL's Expressions

TA :: **Env** × ⟦ a ⟧ → (CStmt, **Env**)

TA ρ ⟦ r := e ⟧ = (se; gen; s.`write(ce);`,ρ")
where (se, ce, ρ') = **TE** ρ ⟦ e ⟧
(gen, ρ", s) = **getActiveState** ρ' r

TA ρ ⟦ **if** e **then** a ⟧ = (se;gens;`if(ce){ca};`,ρ"')
where (se, ce, ρ') = **TE** ρ ⟦ e ⟧
(gens,ρ") = **guaranteeShadowState** ρ' (WriteState a)
(ca, ρ"') = **TA** ρ" ⟦ a ⟧

TA ρ ⟦ a1 | a2 ⟧ = (ca1;ca2;merges;, ρ')
where ρ1 = newStateMap ρ
ρ2 = newStateMap ρ
(ca1, ρ1') = **TA** ρ1 ⟦ a1 ⟧
(ca2, ρ2') = **TA** ρ2 ⟦ a2 ⟧
(merges,ρ') = unifyParMerges ρ1' ρ2'

TA ρ ⟦ a1 ; a2 ⟧ = (cs; ca1; ca2,ρ")
where (ca1, ρ') = **TA** ρ ⟦ a1 ⟧
(ca2, ρ") = **TA** ρ' ⟦ a2 ⟧

TA ρ ⟦ a **when** e ⟧ = (se;`if(!ce){throw GuardFail};`ca;, ρ")
where (se, ce, ρ') = **TE** ρ ⟦ e ⟧
(ca, ρ") = **TA** ρ' ⟦ a ⟧

TA ρ ⟦ t = e **in** a ⟧ = (se;`t = ce;` ca, ρ")
where (se, ce, ρ') = **TE** ρ ⟦ e ⟧
(ca, ρ") = **TA** ρ' ⟦ a ⟧

TA ρ ⟦ m.g(e) ⟧ = (se; gen; s.`g(ce);`,ρ")
where (se, ce, ρ') = **TE** ρ ⟦ e ⟧
(gen, ρ", s) = getActiveState ρ' m

TA ρ ⟦ **loop** e a ⟧= (cs; `while(true){`
`se;if(!ce) break;` ca;`}`)
where (cs,ρ') = **guaranteeShadowState** ρ (WriteState a)
(se, ce) = **TE** ρ' ⟦ e ⟧
(ca,ρ")= **TA** ρ' ⟦ a ⟧

TA ρ ⟦ **loopGuard** e a ⟧ = `cs;try{while(true){ca;ms;}} catch{};`
where (cs, ρ') = **guaranteeShadowState** ρ (WriteState a)
ρ" = **newStateMap** ρ'
ms = commitShadow ρ' ρ"

Figure 7-10: Shadow Minimized Translation of BCL's Actions

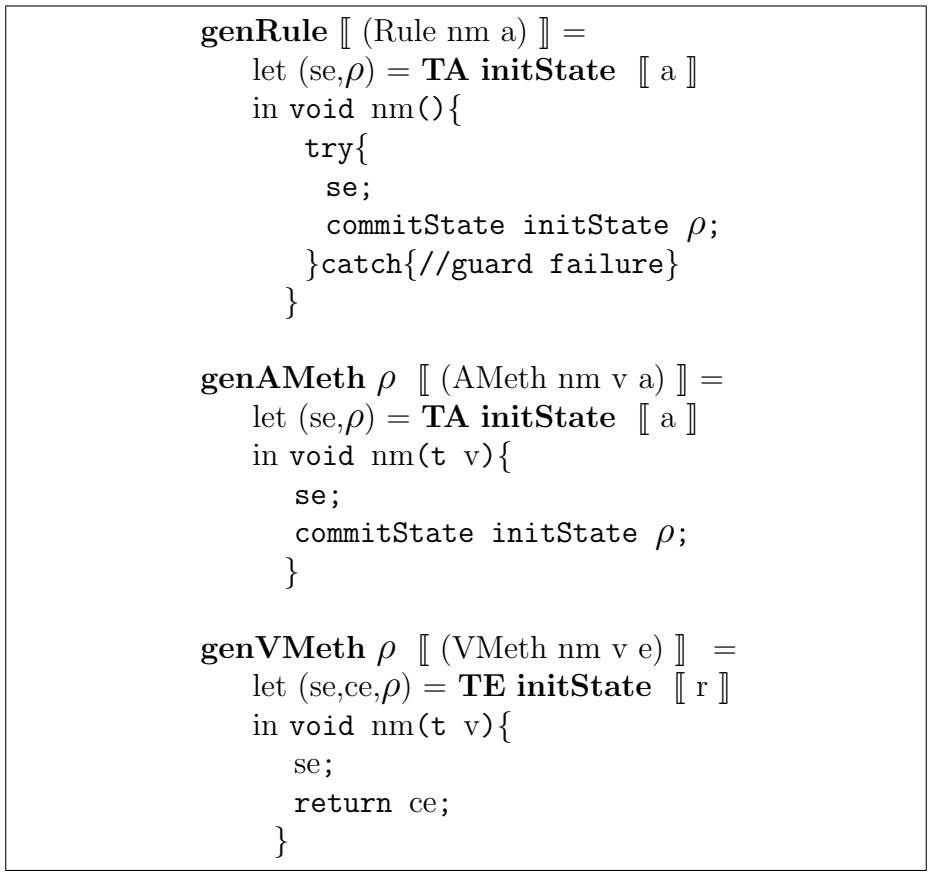


Figure 7-11: Shadow minimized translation of Rules and Methods.

```
data Env = Env{  validActiveState :: Bool,
                 activeState :: CExpr,
                 initState :: CExpr}
    guaranteeReadState ρ  =  (;, ρ)
        getReadState m ρ  =  if (validActiveMap ρ) then activeState ρ
                                                   else initState ρ
guaranteeShadowState m ρ  =
  if (validActiveMap ρ)  then (;, activeState ρ)
                         else (Mod v = new Mod(initState ρ);),
                              ρ{validActiveMap = True, activeState = v}
           newStateMap ρ  =
  if !(validActiveMap ρ) then  ρ{validActiveMap = False,
                                 initState = activeState ρ}
                         else  ρ
     unifyParMerges ρ1 ρ2 =
       if (validActiveMap ρ1)  then if (validActiveMap ρ2)
                                    then (ρ1.parMerge(ρ2), ρ1)
                                    else (;, ρ1)
                               else (;, ρ2)
       getActiveState m ρ  =  let (ms, ρ') = guaranteeShadowState m ρ
                              in (ms, ρ', ρ'[m])
```

Figure 7-12: HelperFunctions for Shadow minimized translation

Increasing the Granularity of Shadowing

Having removed completely unnecessary shadows with the previous optimization, much of the remaining overhead from shadowing comes from the fact that each shadow is a complete duplication of the entire state. This is far more than necessary, as most actions occur on a local part of the state. As such, it makes sense to construct shadow states not of the whole system but only the state that are possibly updated.

This translates to extending our notion of state to a set of shadow variables, each associated with the point in the module hierarchy that they represent. We keep the invariant that the longest matching path represents the most up-to-date data. Thus if we have a shadow `s` of module `m` and shadow `s'` of module `m.m'`, the full shadow state of module `m` is the merged state `s{m'=s'}`.

We can modify our new translation to make use of this, affecting only the helper functions. For the most part, this is a mundane translation. It requires only that, when we get a shadow variable, that we must merge all partial shadows of that state together. The only complexity comes from dealing with parallel actions. When generating a new notion of state, our scheme merges the active state with the initial state. Later, if we need a copy of a module whose most recent state is held only partially in the speculative state, we must not merge the states together — but rather make a new shadow copy. As this is simple but tedious, we omit the new helper functions.

7.4.2 Action Sequentialization

Frequently in BCL's code originally aimed at hardware generation, there are many parallel compositions. These pose a major cost to the software runtime due to the need for shadowing. Conversely, sequential compositions can reuse the same shadow data, and thus be much more efficient for software implementations.

Thus, we can reduce shadow overhead by automatically transforming parallel compositions into sequential ones when they are provably equivalent. We can conservatively approximate when this is safe by observing which modules are read (via

expression methods) and written (via action methods) as follows:

The action $a_1|a_2$ can be rewritten to `a1;a2` when:

$$\texttt{WriteState}(a_1) \cap \texttt{ReadState}(a_2) = \emptyset \wedge$$

$$\texttt{WriteState}(a_1) \cap \texttt{WriteState}(a_2) = \emptyset$$

Appropriate inlining of method definitions and lifting of let bindings out of actions allows us to isolate most parallel actions to the scope of the updates to a single sequential action.

7.4.3 When Lifting

As part of our translation, we first had to lift out all `when`s from expressions. This allowed us to efficiently implement `when`s via execution of control flow instructions instead of as data-level values. A side benefit of this change is that guards bubble up to the top of expressions. Thus, if the expression would eventually fail, we will fail before the evaluation of the expression result is done. A logical extension of this is to extend the lifting process to actions as well. Then we should be able to check for the validity of a rule before we execute any actions. Like expression lifting, this requires separating action methods into body and guard methods. This transformation may not be easy, *e.g.,* moving a guard buried in the second action of a sequential composition requires we "lift" the expression to emulate the changes due to the action that it had once observed. This only gets more complicated when we consider loops; in general there is no way to transform a guard in (or after) a loop into one before the loop execution. As such, we cannot necessarily lift all `when` guards to the top of a rule. Instead we leave guards in loop bodies and sequentialized actions. The procedure to do this is exactly the same as the one described in Section 7.1, save that the operations on methods applies to action methods as well. The additional procedure for lifting of actions is given in Figure 7-13.

$$
\begin{array}{rl}
LW_a[\![r := e]\!] = & (r := e') \textbf{ when } e_g \\
\text{where} & (e' \textbf{ when } e_g) = LW_e[\![e]\!] \\
\\
LW_a[\![a \textbf{ when } e]\!] = & a' \textbf{ when } (a_g \wedge e' \wedge e_g) \\
\text{where} & (a' \textbf{ when } a_g) = LW_a[\![a]\!] \\
& (e' \textbf{ when } e_g) = LW_e[\![e]\!] \\
\\
LW_a[\![\textbf{if } e \textbf{ then } a]\!] = & (\textbf{if } e' \textbf{ then } a') \textbf{ when} \\
& \quad (e_g \wedge (a_g \vee \neg e')) \\
\text{where} & (a' \textbf{ when } a_g) = LW_a[\![a]\!] \\
& (e' \textbf{ when } e_g) = LW_e[\![e]\!] \\
\\
LW_a[\![a_1 \mid a_2]\!] = & (a'_1 \mid a'_2) \textbf{ when } (a_{1g} \wedge a_{2g}) \\
\text{where} & (a'_1 \textbf{ when } a_{1g}) = LW_a[\![a_1]\!] \\
& (a'_2 \textbf{ when } a_{2g}) = LW_a[\![a_2]\!] \\
\\
LW_a[\![a_1 \textbf{ ; } a_2]\!] = & (a'_1 \textbf{ ; } LW_a[\![a_2]\!] \textbf{ when } a_{1g} \\
\text{where} & (a'_1 \textbf{ when } a_{1g}) = LW_a[\![a_1]\!] \\
\\
LW_a[\![t = e \textbf{ in } a]\!] = & ((t' = e') \textbf{ in } a') \textbf{ when} \\
& ((t' = e') \textbf{ in } (t_g = e_g) \textbf{ in } a_g) \\
\text{where} & (e' \textbf{ when } e_g) = LW_e[\![e) \\
& a_2 = a[(t' \textbf{ when } t_g)/t] \\
& (a' \textbf{ when } a_g) = LW_a[\![a_2]\!] \\
\\
LW_a[\![m.g(e)]\!] = & (m.g_b(e') \textbf{ when } e_g \wedge m.g_g(e') \\
\text{where} & (e' \textbf{ when } e_g) = LW_e[\![e]\!] \\
\\
LW_a[\![\textbf{ loop } e\, a]\!] = & \textbf{loop } LW_e[\![e]\!]\, LW_a[\![a]\!] \\
LW_a[\![\textbf{ localGuard } a]\!] = & \textbf{localGuard} LW_a[\![a]\!]
\end{array}
$$

Figure 7-13: Procedure to apply **when**-lifting to actions, referencing the procedure in Figure 7-1. Method Expression calls and bound variables are expected to already be split between body and guard.

7.4.4 Scheduling and Runtime Improvements

There is much space for providing further optimizations. Most obviously, by doing static analysis we can improve locality, and intra-rule parallelism by doing partial scheduling as discusses in Chapter 5. The simplest forms of these techniques would fuse logical rule pipelines into efficient sequential code. Another scheduling improvement that would be highly useful is to dynamically cache guard evaluations to improve selection of the next rule. Initial implementations of these techniques have been tried but a more in depth analysis is left as future work.

Chapter 8

Refinements and Correctness

An important aspect of the design process is the task of modifying a hardware-software design for better performance, area, or power. To be truly good for design exploration, BCL should allow us not only make refinements easily, but also to convince the designer that such refinements are correct formally. Generally, it is extremely difficult for a designer to give a full formal correctness specification for a system. Specifying correctness requires a level of knowledge of the overall system and familiarity with formal verification methods that few designers possess. As a consequence, common practice is to settle for partial verification via testing. This works, but as test suites tend to be built in conjunction with the design itself, designers rarely gain sufficient confidence in their refinements' correctness until near the end of the design cycle.

An alternative is to restrict the types of refinements to ones whose local correctness guarantees that the overall behavior will remain unaffected, and designs usually rely on the notion of equivalence supported by the design language semantics for proving or testing local equivalence.

The most common use of such techniques are in hardware design where the cost of mistakes are high enough to justify the cost. Most hardware description languages hardware description languages describe synthesizable systems at the level of gates and wires. This limits their language-level notion of equivalence to FSM (finite state machine) equivalence. Tools usually require the designer to specify the mapping of state elements (*e.g.,* flip-flops), and thus reduce the problem of FSM equivalence

to combinational equivalence, which can be performed efficiently. FSM-equivalence-preserving refinements have proven to be quite useful because tools are available to prove the local correctness automatically and there is no negative impact on the overall verification strategy. However, FSM refinement is too restrictive and disallows many desirable changes. For instance, adding a buffer to cut a critical path in a pipeline is prohibited. Thus these tools are limited to verification in the later stages of design when the timing has been decided.

In contrast, BCL's nondeterministic one-rule-at-a-time model offers a much different level of abstraction. For instance, the adding of a pipeline stage can be implemented in a natural way by splitting the rule corresponding to the appropriate stage into multiple rules, and introducing state to hold the intermediate results. As designs at this level are meant to be correct for all possible traces of execution, we can reason about whether refinements preserve all possible behaviors of the system.

In this chapter we discuss the sorts of refinements we wish to capture, how we can have the designer express the relationship between specification program and implementation program concisely.

8.1 Understanding Design Refinements

To understand the refinements we are considering it is helpful to consider the *final* implementation of the BCL program as its changes will be motivated by efficiency or performance concerns of the final result. For simplicity we will discuss our motivations of a single domain hardware design, *i.e.,* a synchronous FSM.

Consider the FSM shown in Figure 8-1. The FSM consists of two registers `r1` and `r2`, both initially zero, and some combinational logic implementing functions `f1` and `f2`. The critical path in this system goes from `r1` to `r2` via `f1` and `f2`. In order to improve performance, a designer may want to break this path by adding a buffer (say, a one element FIFO) on the critical path as shown in Figure 8-2. Though we have not shown the circuitry to do so, we will assume that `r2` does not change and the output `z` is not defined when the FIFO is empty.

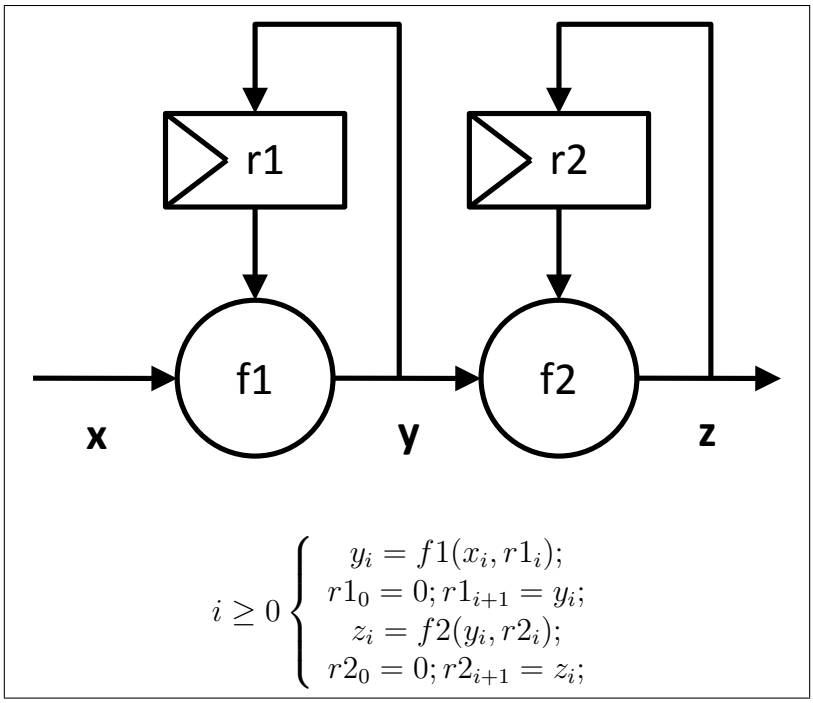


Figure 8-1: Initial FSM

In this refined FSM, the operation that was done in one cycle is now done in two; `f1` is evaluated in the first cycle, and `f2` in the second. The computation is fully pipelined so that each stage is always productive (except the first cycle of the second stage, when the FIFO buffer is empty) and we have the same cycle-level computation rate. However we have the benefit of increased system throughput, as the clock period in the refined system can be much shorter. The refined FSM no longer matches the input-output behavior of the initial FSM meaning that they are no longer FSM equivalent. However, a little analysis shows that the sequence of values assumed by `r2` and `z` are the same in both systems. In other words, the refined system produces the same answer as the original system but one cycle later. Therefore, in many situations such a refinement may be considered correct; our notion of equivalence should let us consider practical BCL programs that compile into these FSMs equivalent.

We can represent the original FSM design as a single-rule BCL program as shown in Figure 8-3. While it is reasonable to deal with streams of inputs in FSMs, it makes more sense in rule-based designs to think of input and output in terms of queues that we have added. For simplicity we can assume that `inQ` is never empty and `outQ`

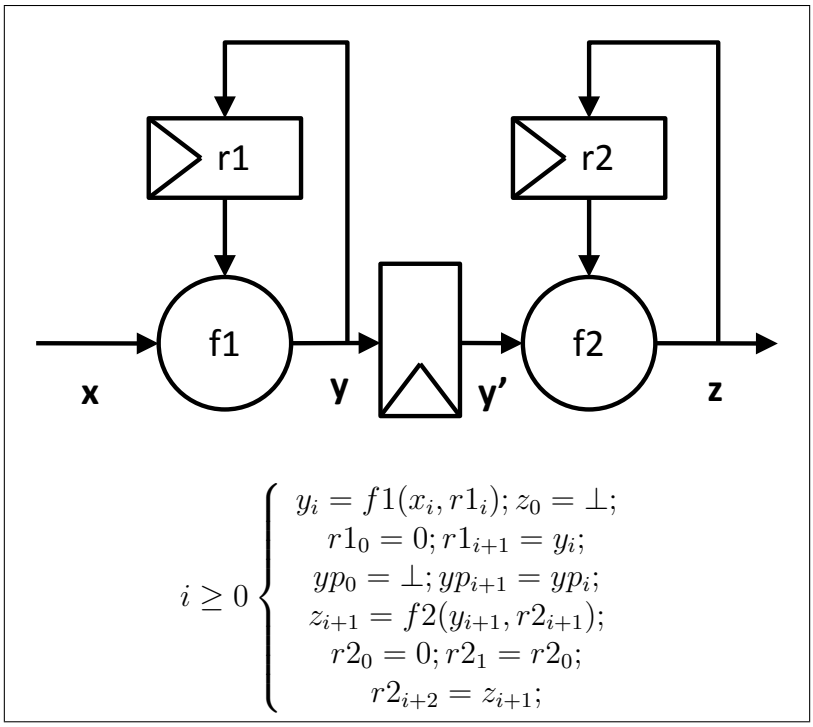
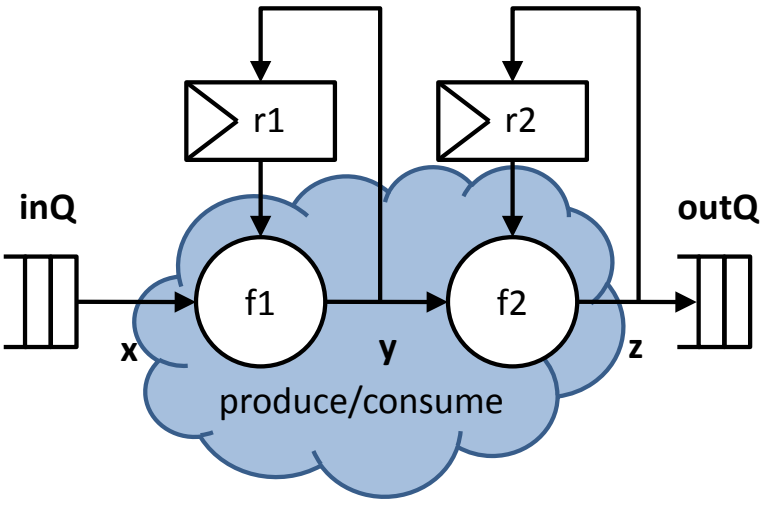


Figure 8-2: Refined FSM

is never full. If we assume that a rule executes in one clock cycle then the rule in Figure 8-3 specifies that every cycle `r1` and `r2` should be updated, one value should be dequeued from `inQ`, and one value should be enqueued in the `outQ`.

The refined FSM in Figure 8-2 may be described by splitting our single rule into two rules: `produce` and `consume`, which communicate via the FIFO `q` as shown in Figure 8-4. This refined system has choice and thus corresponds to many possible FSMs. The particular FSM we are considering can be obtained by constructing the derived rule that attempts to execute `consume` if possible, and then attempts to executes `produce`. In this sense we are checking a more general question than the FSM equivalence. Thus, we must account for each of the following possible initial sequences of the system:

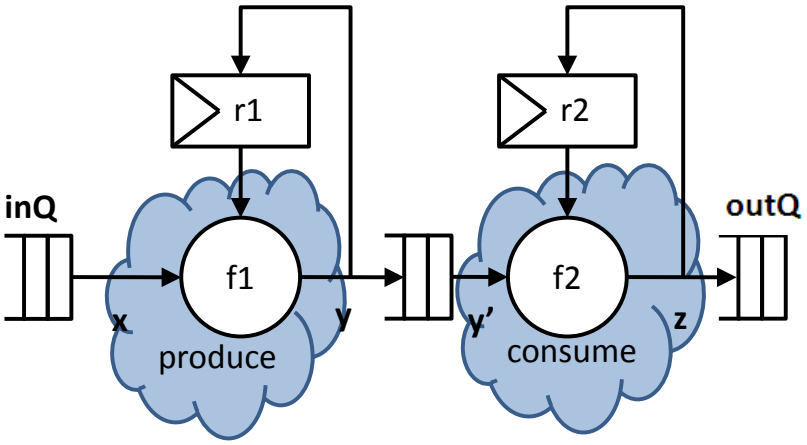


Program $P1$

```
Rule produce_consume:
   x = inQ.first() in
   y = f1(r1,x) in
   z = f2(y,r2) in
     (inQ.deq() |
      r1 := y |
      r2 := z |
      outQ.enq(z))

Register r1 (0)
Register r2 (0)
Module inQ ...
Module outQ ...
```

Figure 8-3: A Rule-based Specification of the Initial Design



Program *P1a*

```
Rule produce:
   x = inQ.first() in
   y = f1(r1,x) in
     (inQ.deq() |
      q.enq(y) |
      r1 := y)

Rule consume:
   y = q.first() in
   z = f2(y,r2) in
     (q.deq() |
      outQ.enq(z) |
      r2 := z)

Register r1 (0)
Register r2 (0)
Module q ...
Module inQ ...
Module outQ ...
```

Figure 8-4: A Refinement of the Design in Figure 8-3

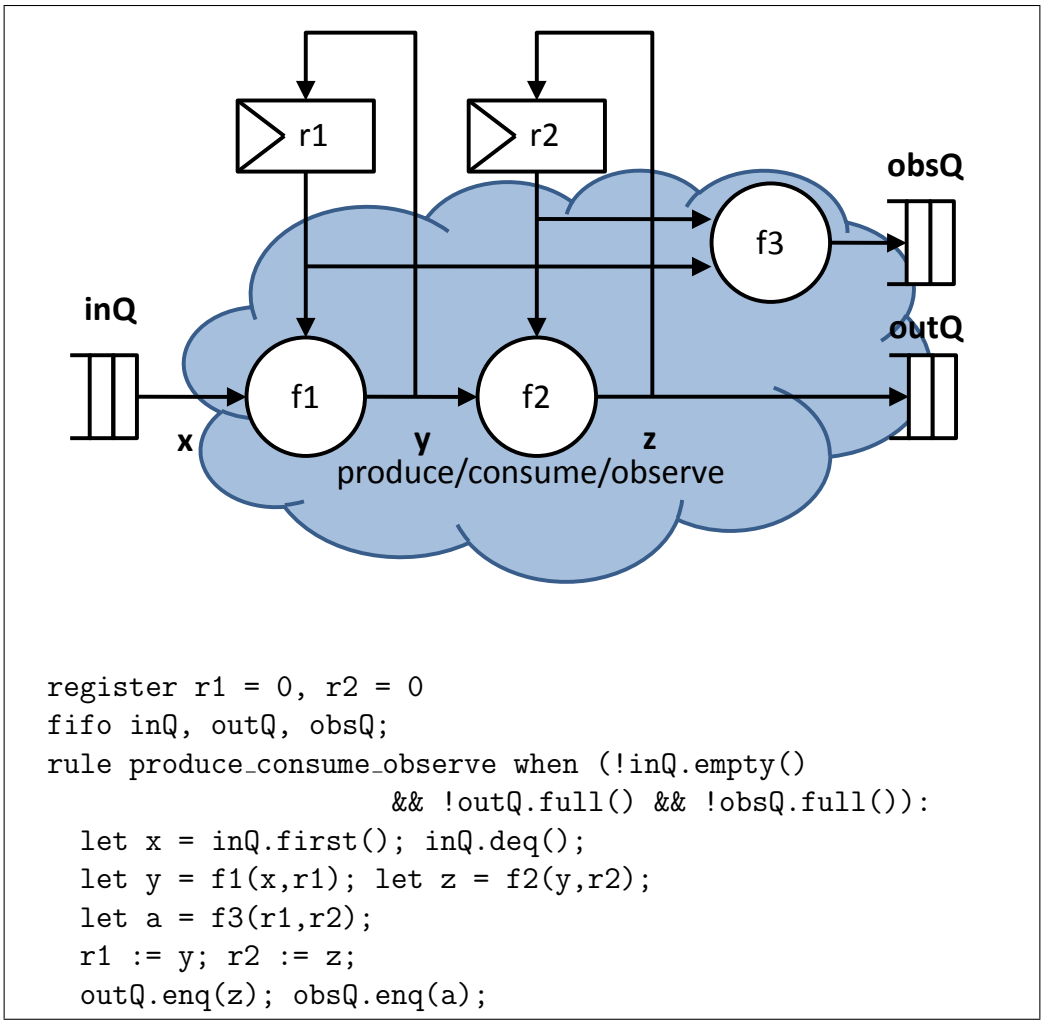
Behavior 1	Behavior 2	Behavior 3
`produce`	`produce`	`produce`
`consume`	`produce`	`produce`
`produce`	`consume`	`consume`
`consume`	`consume`	`produce`
...	...	`consume`
...	...	...

In the first execution, the program repeatedly enters a token into the FIFO and then immediately takes it out. This emulates the execution of the rule in the unrefined system (Figure 8-3) and leaves the FIFO `q` empty after each `consume` rule execution. The execution also does the same set of updates to registers `r1` and `r2` as the original system. The second execution repeatedly queues up two tokens before removing them. Note that, this schedule is valid only if `q` has space for two tokens. The third schedule, corresponds to the refined FSM (Figure 8-2); save for the initial state, there is always at least one token in `q`.

8.1.1 Observability

In what sense are the modules in Figure 8-3 and Figure 8-4 equivalent? Notice that given any sequence of inputs $x_0, x_1, x_2, x_3, ...$ both programs produce the same sequence of outputs $z_1, z_2, z_3, ...$ However, the relative order of the production and consumption of these values are different. Assuming all FIFOs are of size 1, both systems can observe the following sequences: $x_0, z_1, x_1, z_2, x_2, z_3...$ and $x_0, x_1, z_1, z_2, x_2, x_3, z_3, ...$ However, the sequence $x_0, x_1, x_2, z_1, z_2, z_3, ...$ can only be observed for the refined system, as the refined system has more buffering. In spite of this, we want a notion of equivalence that permits this refinement.

Our notion of equality applies only to full BCL programs. To express equality between systems that interact with the outside world, we need to construct a "generic" context that represents all possible interactions with the outside world. This can be done naturally by adding infinite source and sink queues to drive interactions and store results. It is easy to see why, this closed system models all possible interactions of `inQ` and `outQ` with the outside world. Practically we can always find a finite size




```
register r1 = 0, r2 = 0
fifo inQ, outQ, obsQ;
rule produce_consume_observe when (!inQ.empty()
                    && !outQ.full() && !obsQ.full()):
  let x = inQ.first(); inQ.deq();
  let y = f1(x,r1); let z = f2(y,r2);
  let a = f3(r1,r2);
  r1 := y; r2 := z;
  outQ.enq(z); obsQ.enq(a);
```

Figure 8-5: Program of Figure 8-3 with an Observer

approximation of these infinite state elements that will serve to guarantee the "never empty" and "never full" properties that we want for any particular result.

Under a weaker notion of equality, which relies on the transitive closure of rule applications instead of trace equivalence, the previously discussed refinement is correct. At the same time this weaker notion of equality can lead to errors if the module is used incorrectly (for instance if the input to the module changes depending on the number of values outstanding). We rely on the user to express desired distinctions programmatically. For instance if the user believes the relative order of inputs and outputs are necessary, he can add an additional FIFO to which we enqueue witnesses of both input and output events. We believe this is a good tradeoff between greater

flexibility of refinements and user-responsibility in expressing correctness [8].

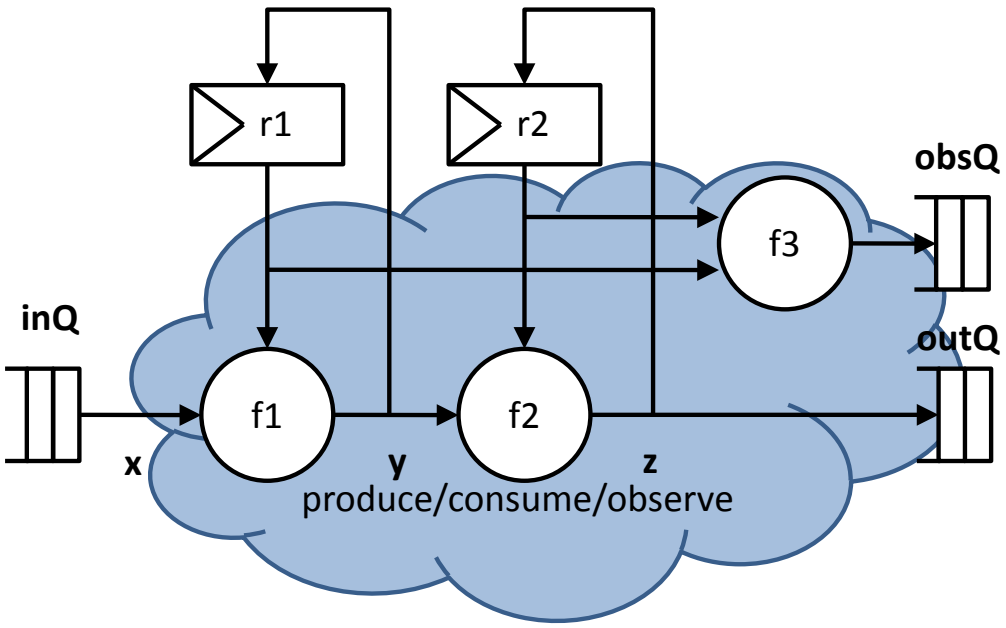
As another example of describing a context, consider refinements of a processor. To show the correctness of a refinement, it is sufficient to show that the refined processor generates the same sequence of instruction addresses of committed instructions as the original. As such we can add a single observation FIFO to the context to observe all relevant differences and consider all possible initial instruction and data memory configurations to verify correctness.

8.1.2 An Example of an Incorrect Refinement

While refinements are often easy to implement, it is not uncommon for a designer to make subtle mistakes that we would like to catch. Consider the original one-rule produce-consume example augmented with observation logic as shown in Figure 8-6. In addition to doing the original computation, this system computes a function of the state of `r1` and `r2`, and at each iteration inserts the result into a new FIFO queue(`obsQ`). A designer may want to do the same rule splitting refinement he had done with the first design, leading to the system in Figure 8-7.

While refinements are often easy to implement, it is not uncommon for a designer to make subtle mistakes. Consider the original one-rule produce-consume example augmented with observation logic as shown in Figure 8-6. In addition to doing the original computation, this system computes a function of the state of `r1` and `r2`, and at each iteration inserts the result into a new FIFO queue(`obsQ`). A designer may want to do the same rule splitting exercise he had done with the first design, leading to the system in Figure 8-7.

This refinement is clearly wrong; we can observe `r1` and `r2` out-of-sync via the new observer circuit. Thus, the sequence `produce observe consume` has no correspondence in the original system. For our tool to be useful to a designer, it must be able to correctly determine that this refinement is incorrect (or rather that it failed to find a matching behavior in the original system). A correct refinement is shown in Figure 8-8, where extra queues have been introduced to keep relevant values in sync. The correct solution would be obvious to an experienced hardware designer because

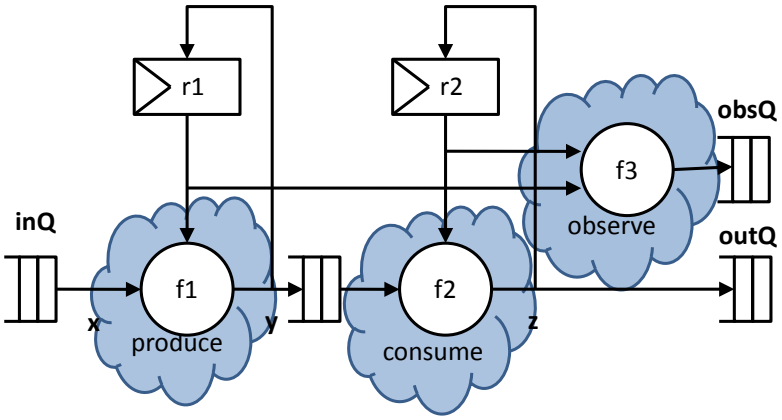


Program *P2*

```
Rule produce_consume_observe:
   x = inQ.first() in
   y = f1(r1,x) in
   z = f2(y,r2) in
   a = f3(r1,r2) in
     (inQ.deq() |
      r1 := y |
      r2 := z |
      outQ.enq(z) |
      obsQ.enq(a))

Register r1 (0)
Register r2 (0)
Module inQ ...
Module outQ ...
Module obsQ ...
```

Figure 8-6: System of Figure 8-3 with an Observer



Program *P2a*

```
Rule produce:
   x = inQ.first() in
   y = f1(r1,x) in
     (inQ.deq() |
      q.enq(y) |
      r1 := y)

Rule consume:
   y = q.first() in
   z = f2(y,r2) in
     (q.deq() |
      outQ.enq(z) |
      r2 := z)

Rule observe:
   a = f3(r1,r2) in
      obsQ.enq(a)

Register r1 (0)
Register r2 (0)
Module q ...
Module inQ ...
Module outQ ...
Module obsQ ...
```

Figure 8-7: An incorrect refinement of the system in Figure 8-6

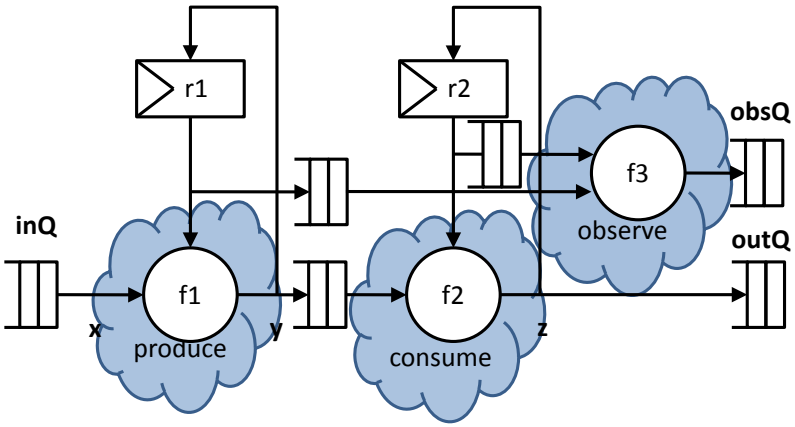
all paths in a pipeline have the same number of stages.

8.1.3 Refinements in the Context of Choice

The examples that we have considered so far have started with a single rule programs. Such systems be definition have no choice. Much of the value of rule-based systems comes from the ability to specify systems that can have multiple distinct executions. An example of a useful nondeterministic specification is that of a speculative processor whose correctness does not depend upon the number of instructions that are executed on the incorrect path. What does it mean to do a refinement in such a system?

Consider the example in Figure 8-9, which is a variation of our producer-consumer example with an observer (Figure 8-6). Unlike the lockstep version that records the state of the registers each iteration, in this system we are allowed to not only miss some updates of `r1` and `r2`, but are permitted to repeatedly make the same observations. An implementation, *i.e.,* a particular execution, of this rule-based specification would pick some deterministic sequence of observations from the allowed set. By giving such a specification, the designer is saying, in effect, that any execution of observations is acceptable. In that sense, the observations made in the system in Figure 8-6 are an acceptable implementation of this nondeterministic system. By the same reasoning we could argue that the refinement shown in Figure 8-8 is a correct refinement of Figure 8-9. But suppose we did not want to rule out any behaviors prematurely in our refinements, then a correct refinement will have to preserve *all* possible behaviors. This can be done by specifying a projection function by which state in two different systems can be related. The partial function relationship is both natural for designers to come up with and easy to specify. Having manually defined this function, the designer could conceivably pass it to a tool that would either tell them that the refinement was correct, or give them an execution from one system that can not be simulated by the other.

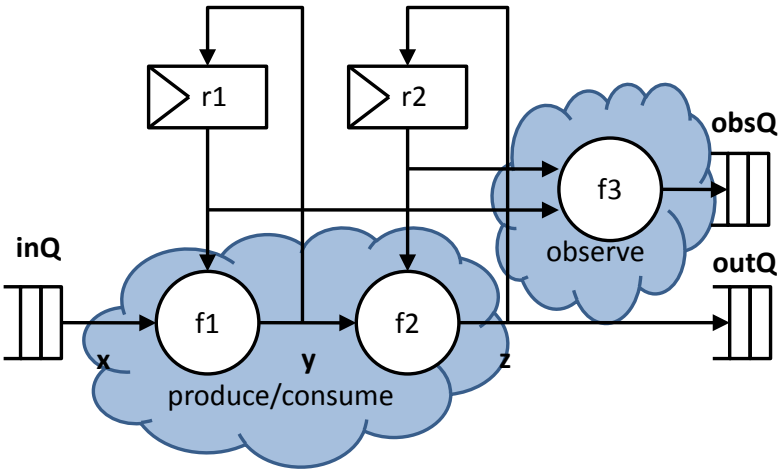
We show a correct refinement of the nondeterministic system in Figure 8-10, where we introduce an extra register, `r1p`, to keep a relevant copy of `r1` in sync with `r2` with which to make legal observations. It is nontrivial to show that all behaviors in new



Program *P2b*

```
Rule produce:
   x = inQ.first() in
   y = f1(r1,x) in
     (inQ.deq() |
      q.enq(y) |
      r1Q.enq(r1) |
      r1 := y)
Rule consume:
   y = q.first() in
   z = f2(y,r2) in
     (q.deq() |
      outQ.enq(z) |
      r2Q.enq(r2) |
      r2 := z)
Rule observe:
   a = f3(r1Q.first(),r2Q.first()) in
      (r1Q.deq() |
       r2Q.deq() |
       obsQ.enq(a))
Register r1 (0)
Register r2 (0)
Module q ...
Module inQ ...
Module outQ ...
Module obsQ ...
Module r1Q ...
Module r2Q ...
```

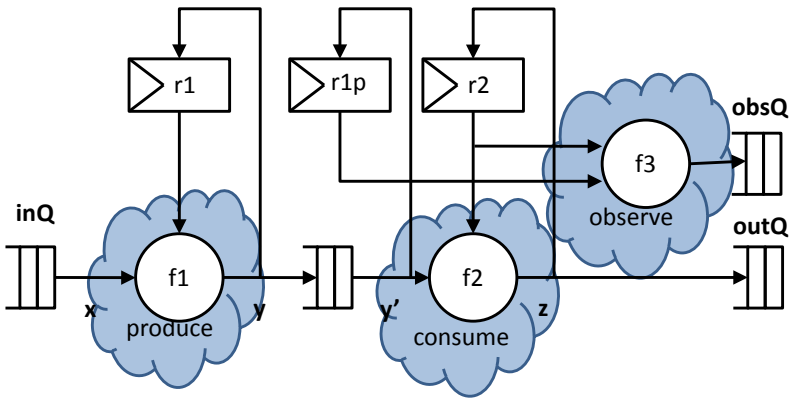
Figure 8-8: A correct refinement of the system in Figure 8-6



Program *P*3

```
Rule produce_consume:
   x = inQ.first() in
   y = f1(r1,x) in
   z = f2(y,r2) in
     (inQ.deq() |
      r1 := y |
      r2 := z |
      outQ.enq(z))
Rule observe:
   a = f3(r1,r2) in
      (obsQ.enq(a))
Register r1 (0)
Register r2 (0)
Module inQ ...
Module outQ ...
Module obsQ ...
```

Figure 8-9: A system with a nondeterministic observer



Program *P3a*

```
Rule produce:
   x = inQ.first() in
   y = f1(r1,x) in
     (inQ.deq() |
      q.enq(y) |
      r1 := y)
Rule consume:
   y = q.first() in
   z = f2(y,r2) in
     (q.deq() |
      outQ.enq(z) |
      r1p := y |
      r2 := z)
Rule observe:
   a = f3(r1p,r2) in
      (obsQ.enq(a))
Register r1  (0)
Register r2  (0)
Register r1p (0)
Module q ...
Module inQ ...
Module outQ ...
Module obsQ ...
```

Figure 8-10: Correct refinement of Figure 8-9

system can model can be modeled by the original nondeterministic specification and vice versa. As we show later, our tool can automatically perform such verification.

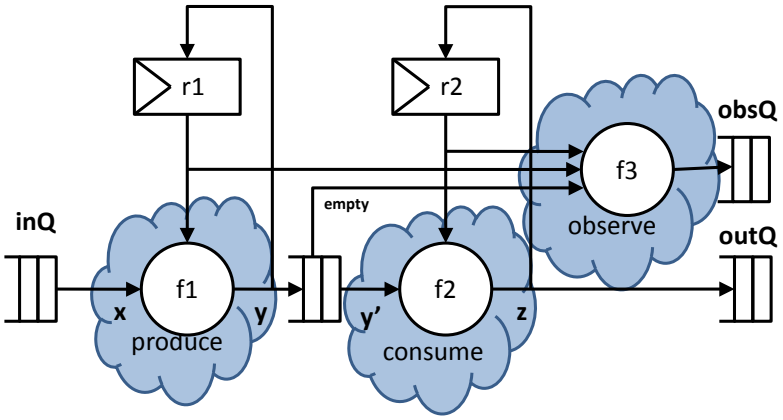
Figure 8-11 shows an alternative refinement. Instead of adding logic to let us emulate safe observations when we were in an unsafe state, we simply restrict when

the observer rule can operate to times when the FIFO is empty. Because we are always free to `consume` until we reach this state, we effectively can emulate any of the original traces by emptying the pipeline before observation.

Though these two refinements are the same and successfully accomplish the critical path splitting we wanted, they have very different properties. Since the second refinement requires us to empty the pipeline before observing, when we select a final scheduling strategy, we must balance between the desire to keep the pipeline fully fed (*i.e.,* always having a token in the FIFO) and being able to observe (*i.e.,* having no tokens in the FIFO). This leads to problems. For instance, the Bluespec compiler would choose to keep the design fully parallel leading tt observations *never* taking place (with the exception of the initial state). In contrast the first refinement can easily allow all rules to fire in parallel. While our notion of equivalence can help designers verify the correctness of their refinements, designers will still need to do this level of analysis is still necessary to do good refinements.

8.2 Establishing Correctness of Implementation

Correctness of a refinement is slightly more complicated than the notion of implementation as discussed in Chapter 5. While both must deal with different notions of state, our definition of implementation relies on a total function between the implementation and the specification. This makes it very easy to understand the execution of a rule in the implementation in terms of specification as it is always meaningful. However, a total function is often relatively difficult to construct. For instance, consider the refinement from the program in Figure 8-3 to the one Figure 8-4. To deal with the differences in relative updating of `r1` and `r2` the function relating the must "flush" partially completed functions, effectively folding in the operation of `consume`. This is too complicated to expect a average user to define. Instead we use an "obvious" *partial* relation and exploit the rule executions to fill in rules. In our concrete example we can relate states when `q` is empty. To generate a total function, we apply rules of the implementation program until there is a relation. This is the same as saying we



Program *P3b*

```
Rule produce:
   x = inQ.first() in
   y = f1(r1,x) in
     (inQ.deq() |
      q.enq(y) |
      r1 := y)
Rule consume:
   y = q.first() in
   z = f2(y,r2) in
     (q.deq() |
      outQ.enq(z) |
      r2 := z)
Rule observe:
   (a = f3(r1,r2) in
      (obsQ.enq(a))) when (q.empty())
Register r1  (0)
Register r2  (0)
Register r1p (0)
Module q ...
Module inQ ...
Module outQ ...
Module obsQ ...
```

Figure 8-11: Second correct refinement of Figure 8-9

consider finite executions between states the do have correspondences from our partial relation. Notice that this technique in general allows one state of the implementation to correspond to multiple states in the specification due to nondeterminism; our system needs to check *every* correspondence.

Recharacterizing our notion of implementation in terms of partial functions leads to our notion of partially correct refinement.

Definition 21 (**Partially Correct Refinement).** Let P be a program modeled by the transition system $\mathcal{S} = (S, S_0, \longrightarrow_{\mathcal{S}})$, P' be a program modeled by the transition system $\mathcal{T} = (T, T_0, \longrightarrow_{\mathcal{T}})$, and $p : T \rightharpoonup S$ a partial function relating states having the property that $T_0 \subseteq \mathrm{Dom}(p)$. *$P'$ is a partially correct refinement of P* exactly when the following conditions hold:

1. **Correspondence of Initial State:** $\{p(t) | t \in T_0\} = S_0$.
2. **Soundness:** For all $t_1, t_2 \in \mathrm{Dom}(p)$ such that $t_1 \twoheadrightarrow_{\mathcal{T}} t_2$, also $p(t_1) \twoheadrightarrow_{\mathcal{S}} p(t_2)$.
3. **Limited Divergence:** For all $t_0 \in T_0$ and $t_1 \in T$ such that $t_0 \twoheadrightarrow_{\mathcal{T}} t_1$, there exists $t_2 \in \mathrm{Dom}(p)$ such that $t_1 \twoheadrightarrow_{\mathcal{T}} t_2$. ■

The first clause states the initial states correspond to each other. The second clause states that every possible execution in the implementation whose starting and ending states have corresponding states in the specification must have a corresponding execution in the specification. The third clause states that from any reachable state in the implementation we can always get back to a state that corresponds to a state in the specification; thus we can never be in a state that has "no meaning" in relation to the specification.

Notice, that this definition guarantee that the implementation only guarantees that the implementation can always be understood in terms of the implementation. Total correctness additionally requires that all executions in the specification have a correspondence in the implementation.

Definition 22 (**Totally Correct Refinement).** A totally correct refinement is a partially correct refinement that, in addition, satisfies:

4. **Completeness:** For all $s_1, s_2 \in S$ and $t_1 \in \mathrm{Dom}(p)$ such that $s_1 \twoheadrightarrow_{\mathcal{S}} s_2$ and $p(t_1) = s_1$, there exists an $t_2 \in \mathrm{Dom}(p)$ such that $t_1 \twoheadrightarrow_{\mathcal{T}} t_2$ with $p(t_2) = s_2$. ■

Of the conditions for total correctness, correspondence of initial state the completeness are easy to verify in cases where we refine a program from a larger rules to more fine-grained rules. As such This leaves us only concerned with soundness and limited divergence.

8.3 Checking Simulation Using SMT Solvers

We can understand the execution of rule R as the application of a pure function f_R of type $S \longrightarrow S$ to the current state. When the guard of R fails, it causes no state change (*i.e.,* $f_R(s) = s$). We can compose these functions to generate a function f_σ corresponding to a sequence of rules σ. To prove the correctness of refinements, we pose queries about f_σ to an SMT solver.

SMT solvers are conceptually Boolean Satisfiability (SAT) solvers extended to allow predicates relating to non-boolean domains (characterized by the particular theories it implements). SMT solvers do not directly reason about computation, but rather permit assertions about the input and output relation of functions. They provide concrete counter-examples when the assertion is false. For example, suppose we wish to verify that some concrete function f behaves as the identity function. We can formulate a universal quantification representing the property: $\forall x, y.(x = f(y)) \wedge (x = y)$. An SMT solver can be used to solve this query, provided the domains of x and y are finite, and f is expressed in terms of boolean variables. If the SMT solver can find a counter-example, then the property is false. If not, then we are assured that f must be the identity. The speed of SMT solvers on large domains is due to their ability to exploit symmetries in the search space [29].

When we reason about rule execution it is often useful to discard all executions where a rule produces no state update (a *degenerate* execution); it is clearly equivalent to the same execution with that rule removed. As such, when posing questions to the solver it is useful to add clauses that state that sequential states of an execution are different. To represent this assertion for the rule R, we define the predicate function $\hat{f}_R(s_2, s_1)$ that asserts that the guard of rule R evaluates to true in s_1 and that s_2 is

the updated state:

$\hat{f}_R(s_2, s_1) = (s_2 = f_R(s_1)) \wedge (s_2 \neq s_1)$

As with the functions, we can construct a larger predicate $\hat{f}_\sigma(s_2, s_1)$ that is true when a non-degenerate execution of σ takes us from s_1 to s_2.

Now we explain how the propositions in Definition 21 can be checked via a small set of easily answerable SMT queries.

8.3.1 Checking Correctness

For this discussion let us assume we have a specification program P and a refinement P' and their respective transition systems $\mathcal{S} = (S, S_0, \longrightarrow_\mathcal{S}, \twoheadrightarrow_\mathcal{S})$ and $\mathcal{T} = (T, T_0, \longrightarrow_\mathcal{T}, \twoheadrightarrow_\mathcal{T})$ are related by the projection function $p : T \rightharpoonup S$.

Now let us consider the soundness proposition from Definition 21: $\forall t_1, t_2 \in \text{Dom}(p).(t_1 \twoheadrightarrow_\mathcal{T} t_2) \implies (p(t_1) \twoheadrightarrow_\mathcal{S} p(t_2))$.

A naïve approach to verifying this property entails explicitly enumerating all pairs (t_1, t_2) in the relation $\twoheadrightarrow_\mathcal{T}$ and checking the corresponding pair $(p(t_1), p(t_2))$ in the relation $\twoheadrightarrow_\mathcal{S}$. As the set of states in both systems are finite, both of these relations are similarly finite (bounded by $|T|^2$ and $|S|^2$, respectively) and thus we can mechanically check the implication.

We can substantially reduce this work by noticing two facts. First, because of transitivity, if we have already checked the correctness of $t_1 \overset{\sigma_1}{\twoheadrightarrow}_\mathcal{T} t_2$ and $t_2 \overset{\sigma_2}{\twoheadrightarrow}_\mathcal{T} t_3$, then there is no need to check the correctness of execution $\sigma = \sigma_1\sigma_2$. Second, if we have already found an execution σ such that $t \overset{\sigma}{\twoheadrightarrow}_\mathcal{T} t'$ then we can ignore all other executions $\sigma' \neq \sigma$ that have the same starting and ending states as they must also be correct. This essentially reduces the task from checking the entire transitive closure to checking only a *covering* of it. Unfortunately, the size of this covering is still very large.

The insight on which our algorithm is built is that proving this property for a small set of finite rule sequences is tantamount to proving the property for any execution. We explain this idea using the program in Figure 8-4.

- Let's begin by considering all rule sequences of length one: `produce` and `consume`.
- The sequence `consume` is never valid for execution starting in a relatable state so we need not consider it further.
- The sequence `produce` is valid to execute but does not take us to a relatable state, so we construct more sequences by extending it with each rule in the implementation. These new sequences are `produce produce` and `produce consume`.
- The sequence `produce consume` always takes a relatable state to another relatable state. We check that all concrete executions of `produce consume` have a corresponding execution in the specification. We do this check over a finite set of sequences in $\mathcal{S}$ (in this case: `produce_consume`), the selection of which we explain later. Since all executions of `produce consume` end in a relatable state, we need not extend it.
- `produce produce` never takes us from relatable state to relatable state, so again extend the sequence to get new sequences `produce produce produce` and `produce produce consume`.
- `produce produce produce` is degenerate if `q` is of length 2 (`q` has to have some known finite length).
- Suppose we could prove that the sequence `produce produce consume` always behaves like `produce consume produce`. Then any execution prefixed by `produce produce consume` is equal to an execution prefixed by `produce consume produce`. Notice that we need not consider any sequences prefixed by `produce consume produce` because itself has the prefix `produce consume`. Therefore we need not consider further sequences prefixed by `produce produce consume`.
- Because we have no new extension to consider, we have proved the correctness of this refinement.

Each of these steps involved an invocation of the SMT solver on queries that are much simpler than the general query presented previously, though the solver still must conceptually traverse the entire state space. The queries themselves are simple because they are always presented using rule sequences of concrete length, which are much smaller than the sequences in $\twoheadrightarrow_{\mathcal{T}}$. The only problem with this procedure is that in the worst case this algorithm will run for the maximum number of states in $\mathcal{S}$. If we give up before the correctly terminating condition, this only means we have failed to establish the correctness of the refinement. We think it is unlikely that the type of refinements we consider in this paper will enter this case. In fact most refinements can be shown to be correct with very small number of considered sequences.

8.3.2 The Algorithm

The algorithm constructs three sets, each of whose elements corresponds to a set of finite executions of $\mathcal{T}$. For each iteration, R_σ represents the set of finite sequences for which we have explicitly found a corresponding member, and U represents the set of finite executions we have yet to verify (each element of U conceptually represents all finite sequences starting with some concrete sequence of rule executions σ). NU is the new value of U being constructed for the next iteration of the execution.

The Verification Algorithm:

1. Initially: $R_\sigma := \emptyset$, $U := \{R_i | R_i \in R_{\mathcal{T}}\}$, $NU := \emptyset$
2. if $U = \emptyset$, we have verified all finite executions. Exit with Success.
3. Check if we have reached our iteration limit. If so, give up, citing the current U set as the cause of the uncertainty.
4. For each $\sigma \in U$:
   (a) Check if the execution of σ from a relatable state is ever non-degenerate:
   $\exists t_1 \in T, t_2 \in \mathrm{Dom}(p).(t_1 \overset{\sigma}{\twoheadrightarrow}_{\mathcal{T}} t_2)$
   If no execution exists we can stop considering σ immediately.

(b) Check if σ should be added to R_σ. That is, if some execution of σ should have a correspondence in $\mathcal{S}$:
$\exists t, t' \in \text{Dom}(p).(t \overset{\sigma}{\twoheadrightarrow}_{\mathcal{T}} t')$
If so $R_\sigma := R_\sigma \cup \{\sigma\}$.

(c) Check if all finite executions of σ that should have a correspondence in $\mathcal{S}$ have such a correspondence:
$\forall t, t' \in \text{Dom}(p).(t \overset{\sigma}{\twoheadrightarrow}_{\mathcal{T}} t') \implies \exists \sigma'.(p(t) \overset{\sigma'}{\twoheadrightarrow}_{\mathcal{S}} p(t'))$
If this fails due to some concrete execution of σ, exit with Failure providing the counter example as justification.

(d) For every execution where σ does not put us in a relatable state, we must show that extensions of the form $\sigma\sigma'$ have an execution taking us to the same state $\sigma_1\sigma_2\sigma'$, where σ_1 is a member of R_σ and $|\sigma_1\sigma_2| \leq |\sigma|$. Thus, the correctness of $\sigma\sigma'$ is reduced to the correctness of the shorter sequence $\sigma_2\sigma'$.
$\forall t \in \text{Dom}(p), t' \notin \text{Dom}(p).(t \overset{\sigma}{\twoheadrightarrow}_{\mathcal{T}} t') \implies$
$\exists \sigma_1 \in R_\sigma, \sigma_2.$
$(|\sigma_1\sigma_2| \leq |\sigma|) \wedge (\sigma_1(t) \in \text{Dom}(p)) \wedge (\sigma_2(\sigma_1(t)) = t').$
If this succeeds, we need not consider executions for which σ is a prefix. If not, extend the prefix σ by each of the rules in $R_{\mathcal{T}}$. $NU := NU \cup \{\sigma.R_i | R_i \in R_{\mathcal{T}}\}$.

5. $U := NU$, $NU := \emptyset$, Go to Step 2. ■

8.3.3 Formulating the SMT Queries

The four conditions in the inner-most loop of the algorithm can be formulated as the following SMT queries using the $\hat{f}_\sigma$ predicate and the computational version of projection function p, $\hat{p} : T \longrightarrow S$ and $rel : T \longrightarrow \{0, 1\}$ where p and $\hat{p}$ are the same if p is defined and $rel(t)$ returns true exactly when $p(t)$ is defined.

1. *Existence of valid execution of σ starting from a relatable state:*

$\exists t_1, t_2 \in T. \hat{f}_\sigma(t_2, t_1) \wedge rel(t_1)$

2. *Verifying that each execution of σ in the implementation starting and ending in a relatable state has a corresponding execution in the specification:*

$$\forall t_1, t_2 \in T.$$
$$(rel(t_1) \wedge rel(t_2) \wedge \hat{f}_\sigma(t_2, t_1)) \implies$$
$$\bigvee_{\sigma' \in EC(\sigma)} (\hat{f}_{\sigma'}(\hat{p}(t_2), \hat{p}(t_1)))$$

where EC is the "expected correspondences" function that takes a sequences of rules σ in $\mathcal{T}$ and returns a finite set of sequences in $\mathcal{S}$ to which σ is likely to correspond. This function can be easily generated by the tool or the user, since the refinements are rule splitting, it is easy to predict the candidates in the specification that could possibly mimic σ. For instance, consider the refinement of the program in Figure 8-3 to the one in Figure 8-4. Each occurrence of `produce` in the implementation should correspond to an occurrence of `produce_consume` in the specification. Thus, the sequence `produce produce consume produce`, if it has a correspondence at all, could only correspond to the sequence `produce_consume produce_consume produce_consume`.

3. *Checking that every valid execution of σ in the implementation has an equivalent sequence that is correct by concatenation of smaller sequences*:

$$\forall t_1, t_2, t_m \in T.$$
$$rel(t_1) \wedge \hat{f}_\sigma(t_2, t_1) \implies rel(t_m) \wedge$$
$$\bigvee_{\sigma_1 \in R_\sigma} (\bigvee_{\sigma_2 \in EA(\sigma, \sigma_1)} (\hat{f}_{\sigma_1}(t_m, t_1) \wedge \hat{f}_{\sigma_2}(t_2, t_m)))$$

Our algorithm requires us to find, given σ and σ_1 in $\mathcal{T}$, a σ_2 such that the execution of σ is the same as the execution of $\sigma_1\sigma_2$, and $|\sigma_1\sigma_2| \leq |\sigma|$. We assume the existence of a "expected alternatives" function EA that enumerates all possible σ_2 given σ and σ_1.

8.3.4 Step-By-Step Demonstration

For the sake of clarity, we provide an additional example of the algorithm's execution. Figure 8-12 gives the trace of reasoning through which our algorithm progresses in order to verify the refinement of the program in Figure 8-6 to the one in Figure 8-7. Each node represents an element in the algorithm's set U, and the path from the root to any node in the graph corresponds to the concrete value σ for that node. At each node, we verify the correctness of all corresponding finite executions of σ: nodes displayed as $\perp$ are vacuously true by Step 4a, while other leaf nodes are either true by Step 4d or incorrect by Step 4c. The program is ultimately rejected as the refinement being checked is *incorrect*:

- We begin by considering all rule sequences of length one executed in a relatable state: `produce`, `consume`, and `observe`. The rule `observe` always ends in a relatable state, and corresponds directly to the `observe` rule in the specification program. `consume` is never valid to execute, so the only sequence that we extend is `produce` since it never ends in a relatable state.

- We now extend `produce`, giving us three new sequences to consider: `produce produce`, `produce consume`, and `produce observe`. `produce consume` always ends in a relatable state and corresponds to the execution of `produce_consume` in the specification. Neither `produce produce`, nor `produce observe` ever end in a relatable state, and since we are unable to prove their equivalence to an execution we have already verified, we extend both.

- In the third iteration, we consider the sequence `produce observe consume`, which always ends in a relatable state. This exposes an error in the refinement since there is no possible sequence of rule in the specification that produces this final state (in this case, the implementation enqueues a value to `obsQ` that the specification is unable to replicate).

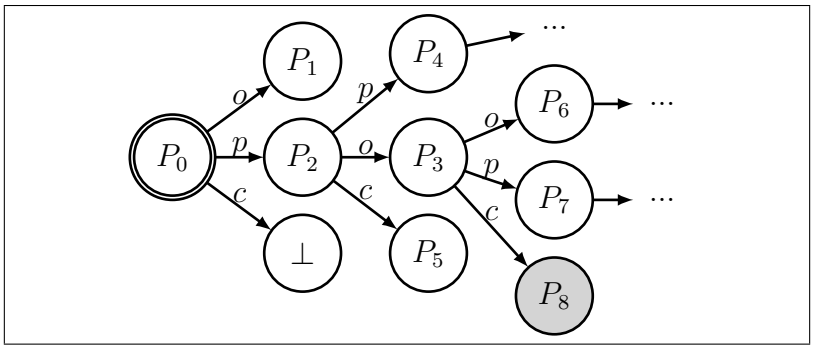


Figure 8-12: Tree visualization of the algorithmic steps to check the refinement of the program in Figure 8-6 to the one in Figure 8-7

8.4 The Debugging Tool and Evaluation

This algorithm was embodied in an initial tool based on the BCL frontend. The algorithm in Section 8.3 works more efficiently when rule sizes are small, therefore the first phase of the tool is to reduce the size of actions by action sequentialization, conditional merging, and "when lifting" [32]. Next, the tool generates the function f_R for each rule R. We use typed λ-calculus with let blocks to represent these functions and apply many small transformations to simplify them. As we have discussed, this algorithm makes many queries to an SMT solver; we use the STP SMT solver [47] for this purpose. By static analysis (*e.g.,* rule commutativity and sequence degeneracy) of the programs, we remove unneeded sequences from consideration in the sets EA and EC. This has substantial impact on the size of SMT queries.

To demonstrate our tool, we consider a refinement of a Simplified MIPS (SMIPS) processor, whose ISA contains a representative subset of 35 instructions from the MIPS ISA. While the ISA semantics are specified one instruction at a time, our program is pipelined with five stages in the style of the DLX processor [71], and resembles soft-cores used in many FPGA designs. The execution of the final implementation is split into the following five separate stages (see Figure 8-13(b)):

1. *Fetch* requests the next instruction from the instruction memory (`imem`) based on the `pc` register which it then updates speculatively to the next consecutive `pc`.

2. *Decode* takes the data from the instruction memory and the fetch stage, decodes the instruction, and passes it along to the execute stage. It also reads the appropriate locations in the register file `rf`, stalling to avoid data hazards (stall logic is not shown).

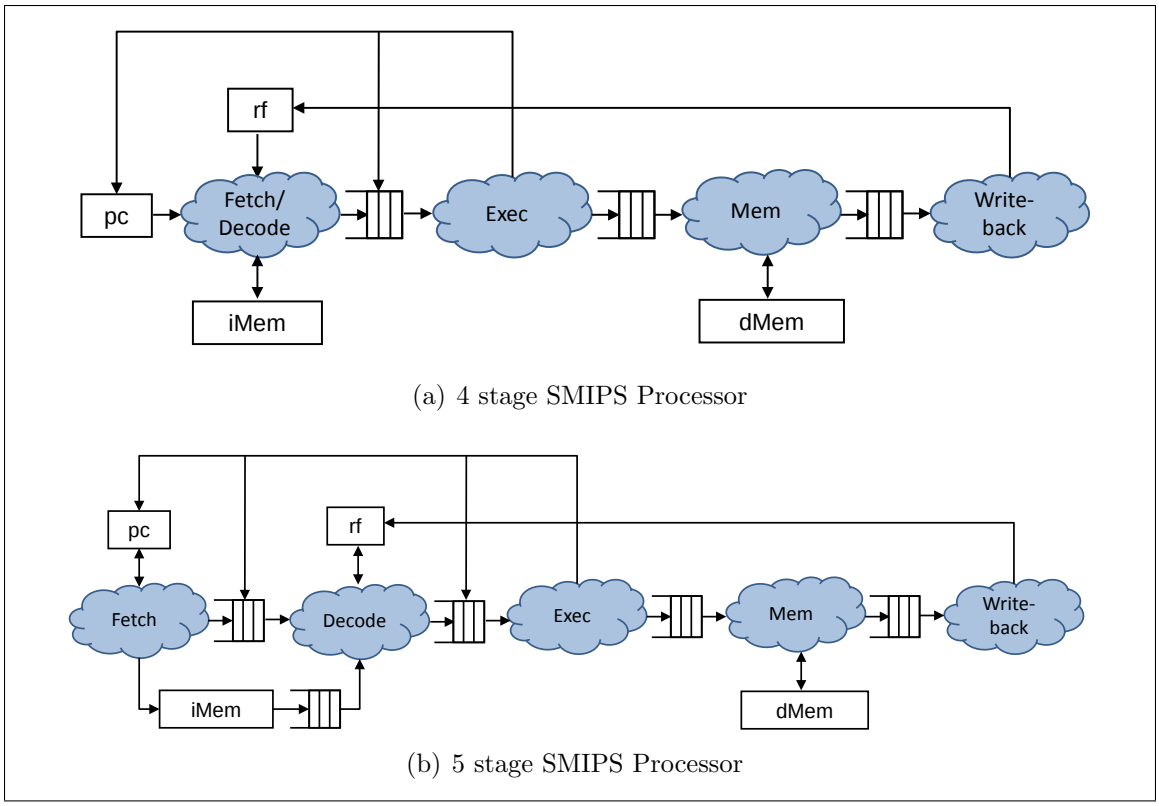
3. *Execute* gets decoded instructions from the execute queue, performs ALU operations and translates addresses for memory operations. To handle branch operations, it kills mispredicted instructions and sets the `pc`.

4. *Memory* performs reads and writes to the data memory, passing the data to the writeback state. (A further refinement might introduce a more realistic split-phase memory, which would move some of this functionality into the writeback stage).

5. *Writeback* gets instructions in the form of register destination and value pairs, performing the update on the register file.

The implementation program contains one rule per stage, and stages communicate via FIFO connections. If we were to executes the rules for each stage in reverse order (starting from writeback and finishing with fetch), the result is a fully pipelined system. If each FIFO is implemented as a single register with a *valid* bit, this is indistinguishable from the standard processor complete with pipeline stalls. If instead we execute the rules in pipeline order, we end up with a system where the instructions fly through the processor one-at-a-time. For code simplicity, our final implementation actually decomposes the execute stage into three mutually exclusive cases, implementing each with a separate rule(`exec, exec_branch, and exec_branch_mispredict`). Since the rule guards are mutually exclusive, this does not modify the pipeline structure, nor does it change the analysis.

Our implementation is relatively complicated and we would like to know if it matches the ISA. One way to achieve this is to start with a single-rule description of the behavior (transliterated directly from the documentation, which we consider to be correct), and incrementally refine the program towards the final five-stage implementation. After each refinement, our tool can be used to verify correctness with



(a) 4 stage SMIPS Processor

(b) 5 stage SMIPS Processor

Figure 8-13: SMIPS processor refinement

regards to the previous iteration. For the sake of brevity, we examine only the final refinement, which takes a four-stage processor (Figure 8-13(a)) and splits the fetch-decode stage. Though the transformation is straightforward, the tool must be able to correctly resolve the effect of speculative execution from branch prediction.

The tool is able to establish the correctness of this refinement step in under 7 minutes. To do so it needed to check 21 executions in the refined program of maximum length 3. In general for pipeline partitionings, the length is the maximum number of steps between states unrelated by our state relationship plus 1. In most pipelines this is the pipeline depth plus 1. In our example, we correspondences in the four-stage program for the 5 corresponding rules, `fetch_decode` for `fetch decode`, and `exec_branch_mispredict` for the mispeculating sequences `fetch exec_branch_mispredict` and `fetch fetch exec_branch_mispredict`.

The performance of this tool can be improved in three orthogonal dimensions. First, we currently only leverage the theory of bit vectors. By adding additional

theories of FIFOs, arrays, and uninterpreted functions [56] we can dramatically reduce the complexity of our SMT queries. Secondly, our interface with the SMT solver is inefficient, requiring file-level IO. More than half of the compute time comes from marshaling and unmarshaling the query representation. This clearly can be eliminated by directly integrating an SMT solver into the tool. Finally, our algorithm allows us to reason about each element of U in parallel. Exploiting all three forms have the possibility of making this tool fast enough to be viable for fast refinements.

Chapter 9

Conclusion

This thesis introduces BCL, a unified language for hardware-software codesign and addresses many important aspects of the design process needed for effective implementation.

To allow designers to explore what hardware-software partitioning is needed, BCL allows designers to maintain a more agnostic view on whether a computation should be placed in hardware or software and to annotate the decision directly via domains. The domain abstraction allows designers to isolate messy synchronization logic into easily-reused synchronizer modules, while still allowing important channel-multiplexing logic to be visible at the program level and thus available to be optimized and verified by the common user, not the synchronizer writer. It also factors the compilation task to a per-domain level which greatly simplifies compilation.

BCL is higher level than the standard low-level descriptions of hardware (RTL) and software (C code), factoring out many of the decisions in execution, and opting for a highly nondeterministic execution model. This allows us to handle many refinement problems as local rule refinements which both make the correctness understandable to the average user, and makes mechanical verification cheap enough to be move from post-facto step to a part of the design loop itself.

While nondeterminism makes BCL easier to understand and verify, it can be difficult to implement efficiently. As such it is restricted in implementations. To understand how this affects the correctness of the program we introduce a compositional

view of the scheduling task that encompasses both previous algorithms and permits multi-substrate algorithms.

Given a scheduled BCL program, we describe how to compile BCL rules into synchronous hardware efficiently. This view of implementation is conceptually simpler than the previous views given in the compilation of BSV, the predecessor hardware description language, without compromising on efficiency or expressivity. This makes BCL hardware designs as efficient as RTL equivalents. Similarly we describe how BCL rules can be compiled into high-quality C++.

The net result of this, is that BCL users can effectively tune their programs for hardware and software without losing precision in their meaning or having to resort to extra-lingual approaches or indirectly nudge their BCL design to get the correct result. In this sense, BCL is a perfect base on which to build further hardware-software designs frameworks; it expresses the computation and partitioning desired concisely and precisely making it easy to reason about correctness while keeping code/hardware generation efficient.

While this work deals with the fundamental aspects of the codesign problem, there are no evaluations in this thesis. This is not to say that no implementation work has been done. In fact, a major part of the effort has been devoted to implementing a full compiler. In fact complete BCL programs can now be compiled, partitioned, implemented in hardware and software and run on multiple FPGA platforms. However, substantial work is needed to make any even a slightly meaningful evaluation of the codesign problem. This is because to make a design truly efficient takes great deal of tuning and specialization. To properly do this in BCL we need efficient hardware-software synchronizers and other basic primitives, a good example application, quality schedules for various partitionings, and nontrivial analysis of the system as a whole. Much, but not all of this work has been done. When complete it will appear in Myron King's PhD thesis.

Beyond this work, there is significant room for improvement in the presentation of BCL to the user. This is necessary to make BCL a viable representation for industrial design.

The most obvious improvement to be made to BSV is the introduction of a type system, both for values and actions and for domains. Type systems are well understood and the type view of domains from Chapter 6 is so simple that little theoretical work would need to be done to properly give BCL a type system. A natural choice may be to simply copy BSV's type system which extends Haskell's type system with whole number "numeric types" to allow the expression of size, and important aspect for hardware design.

Another practical improvement for the user would be to effectively automate the scheduling task. Forcing the designer to select a schedule without assistance for each of the many design that they are exploring is impractical. In the context of single-cycle synchronous hardware, automatic algorithms have proven to be sufficient for automating the task. Though the task is still fundamentally the designer's responsibility, in almost all cases little to no user-given information need be conveyed to the algorithm to get it to give the desired implementation. Extensions to allow this for software as well as multi-cycle hardware would complete the picture and allow rapid implementation.

Perhaps the most important work needed to make BCL practical is a rich library of synchronizers and design components to enable the construction of designs for various platforms. Though this is already happening in the context of evaluation, even more work is necessary to fit into all the desired contexts and to be sufficiently specializable to achieve all desired behaviors. This task is not just a matter of wrapping current hardware-software communication into module. A proper synchronizer design will need to provide enough control to mimic virtual channels, fairness and prioritization, and bursting. Taking the related example of hardware clock synchronizer are an indication finding the best representation for this will take significant work.

A further extension that may be of interest is to extend the notion of partitions beyond synchronous hardware and software executed on a general-purpose processor. In principle a domain can be implemented in any computational substrate, *e.g.,* asynchronous hardware or GPUs. There may be significant value if BCL's model may be efficiently implemented in such a context.

These extensions – both the planned work for efficiency and the speculative work on the user-level representation – are needed to fully enable our proposed extreme design exploration approach to embedded implementations. If these extensions can be realized in a single language and compilation infrastructure, then the old unwieldy multi-language methodologies of hardware-software codesign may cease to be used.

Bibliography

[1] Carbon Design Systems Inc. `http://carbondesignsystems.com`.

[2] Matlab/Simulink. `http://www.mathworks.com`.

[3] Accelera Organization, Inc., Napa, CA. *SystemVerilog 3.1a Language Reference Manual*, May 2004.

[4] Shail Aditya, B. Ramakrishna Rau, and Vinod Kathail. Automatic architectural synthesis of vliw and epic processors.

[5] A. Agarwal, Man Cheuk Ng, and Arvind. A comparative evaluation of high-level hardware synthesis using reed-solomon decoder. *Embedded Systems Letters, IEEE*, 2(3):72 –76, 2010.

[6] Charles Andr and Marie agns Peraldi-frati. Behavioral specification of a circuit using syncharts: A case study. In *EUROMICRO*, 2000.

[7] Péter Arató, Zoltán Ádám Mann, and András Orbán. Algorithmic aspects of hardware/software partitioning. *ACM Trans. Des. Autom. Electron. Syst.*, 10:136–156, January 2005.

[8] Arvind, Nirav Dave, and Michael Katelman. Getting formal verification into design flow. In *Lecture Notes in Computer Science, FM 2008: Formal Methods Volume 5014/2008 pp.12-32*, 2008.

[9] Arvind, Rishiyur S. Nikhil, Daniel L. Rosenband, and Nirav Dave. High-level Synthesis: An Essential Ingredient for Designing Complex ASICs. In *Proceedings of ICCAD'04*, San Jose, CA, 2004.

[10] Arvind and R.S. Nikhil. Executing a program on the MIT Tagged-Token Dataflow Architecture . *Computers, IEEE Transactions on*, 39(3):300 –318, March 1990.

[11] Arvind and Xiaowei Shen. Using Term Rewriting Systems to Design and Verify Processors. *IEEE Micro*, 19(3):36–46, May 1999.

[12] Peter J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2001.

[13] AutoESL Desisgn Technologies, Inc. `http://www.autoesl.com`.

[14] Felice Balarin, Yosinori Watanabe, Harry Hsieh, Luciano Lavagno, Claudio Passerone, and Alberto Sangiovanni-Vincentelli. Metropolis: An integrated electronic system design environment. *Computer*, 36:45–52, 2003.

[15] Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, César Mu noz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, N. Shankar, Eli Singerman, and Ashish Tiwari. An overview of SAL. In C. Michael Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196, Hampton, VA, jun 2000. NASA Langley Research Center.

[16] Gérard Berry. Esterel on hardware. In *Mechanized Reasoning and Hardware Design*, pages 87–104. Prentice Hall, Hertfordshire, UK, 1992.

[17] Gérard Berry and Laurent Cosserat. The esterel synchronous programming language and its mathematical semantics. In *Seminar on Concurrency*, pages 389–448, 1984.

[18] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in haskell. In *ICFP*, pages 174–184, 1998.

[19] Bluespec Inc. `http://www.bluespec.com`.

[20] Bluespec, Inc., Waltham, MA. *Bluespec SystemVerilog Version 3.8 Reference Guide*, November 2004.

[21] Stephen D. Brown, Robert J. Francis, Jonathan Rose, and Zvonko G. Vranesic. *Field-programmable gate arrays*. Kluwer Academic Publishers, Norwell, MA, USA, 1992.

[22] Joseph T. Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogenous systems. *Int. Journal in Computer Simulation*, 4(2):0–, 1994.

[23] Andrew Butterfield. A denotational semantics for handel-c. *Formal Aspects of Computing*, 22, 2010.

[24] Cadence. NC-Verilog. `http://www.cadence.com`.

[25] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. Lustre: A declarative language for programming synchronous systems. In *POPL*, pages 178–188, 1987.

[26] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, Reading, Massachusetts, 1988.

[27] Karam S. Chatha and Ranga Vemuri. Magellan: multiway hardware-software partitioning and scheduling for latency minimization of hierarchical control-dataflow task graphs. In *CODES*, pages 42–47, 2001.

[28] H. K. Chaudhry, P. Eichenberger, and D. R. Chowdhury. Mixed 2-4 state simulation with vcs. In *Proceedings of the 1997 IEEE International Verilog HDL Conference (IVC '97)*, pages 77–, Washington, DC, USA, 1997. IEEE Computer Society.

[29] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, STOC '71, pages 151–158, New York, NY, 1971.

[30] Ed Czeck, Ravi Nanavati, and Joe Stoy. Reliable Design with Multiple Clock Domains. In *Proceedings of Formal Methods and Models for Codesign (MEMOCODE)*, 2006.

[31] Nirav Dave. Designing a Reorder Buffer in Bluespec. In *Proceedings of MEMOCODE'04*, San Diego, CA, 2004.

[32] Nirav Dave, Arvind, and Michael Pellauer. Scheduling as Rule Composition. In *Proceedings of Formal Methods and Models for Codesign (MEMOCODE)*, Nice, France, 2007.

[33] Nirav Dave, Kermin Fleming, Myron King, Michael Pellauer, and Muralidaran Vijayaraghavan. Hardware acceleration of matrix multiplication on a xilinx fpga. In *MEMOCODE*, pages 97–100. IEEE, 2007.

[34] Nirav Dave, Man Cheuk Ng, and Arvind. Automatic Synthesis of Cache-Coherence Protocol Processors Using Bluespec. In *Proceedings of Formal Methods and Models for Codesign (MEMOCODE)*, Verona, Italy, 2005.

[35] Jack B. Dennis, John B. Fosseen, and John P. Linderman. Data flow schemas. In *International Sympoisum on Theoretical Programming*, 1972.

[36] Edsger W. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM*, 18(8):453–457, 1975.

[37] David L. Dill. The murphi verification system. In *In Computer Aided Verification. 8th International Conference*, pages 390–393. Springer-Verlag, 1996.

[38] Bruno Dutertre and Leonardo de Moura. The yices smt solver. 2006.

[39] Stephen Edwards. High-Level Synthesis from the Synchronous Language Esterel. In *Proceedings of IWLS'02*, New Orleans, LA, 2002.

[40] Stephen A. Edwards and Olivier Tardieu. Shim: a deterministic model for heterogeneous embedded systems. *IEEE Trans. VLSI Syst.*, 14(8):854–867, 2006.

[41] K. Ekanadham, J. H. Tseng, P. Pattnaik, A. Khan, M. Vijayaraghavan, and Arvind. A PowerPC Design for Architectural Research Prototyping. *The 4th Workshop on Architectural Research Prototyping*, 2009.

[42] Rolf Ernst, Jorg Henkel, and Thomas Benner. Hardware-software cosynthesis for microcontrollers. *IEEE Des. Test*, 10(4):64–75, 1993.

[43] Thomas Esposito, Mieszko Lis, Ravi Nanavati, Joseph Stoy, and Jacob Schwartz. System and method for scheduling TRS rules. United States Patent US 133051-0001, February 2005.

[44] H. Fleisher and L. I. Maissel. An introduction to array logic. *IBM Journal of Research and Development*, 19:98–109, 1975.

[45] Kermin Fleming, Chun-Chieh Lin, Nirav Dave, Arvind, Gopal Raghavan, and Jamey Hicks. H.264 decoder: A case study in multiple design points. In *MEMOCODE*, pages 165–174. IEEE Computer Society, 2008.

[46] D. Gajski, N. Dutt, A. Wu, and S. Lin. High-level synthesis introduction to chip and system design. 1994.

[47] Vijay Ganesh and David L. Dill. A Decision Procedure for Bit-Vectors and Arrays. In *19th International Conference on Computer Aided Verification (CAV-07)*, volume 4590, pages 519–531, 2007.

[48] Sumit Gupta, Nikil Dutt, Rajesh Gupta, and Alex Nicolau. Spark: A high-level synthesis framework for applying parallelizing compiler transformations. In *In International Conference on VLSI Design*, pages 461–466, 2003.

[49] George Hadjiyiannis, Silvina Hanono, and Srinivas Devadas. Isdl: An instruction set description language for retargetability, 1997.

[50] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[51] James C. Hoe. *Operation-Centric Hardware Description and Synthesis*. PhD thesis, MIT, Cambridge, MA, 2000.

[52] James C. Hoe and Arvind. Operation-Centric Hardware Description and Synthesis. *IEEE TRANSACTIONS on Computer-Aided Design of Integrated Circuits and Systems*, 23(9), September 2004.

[53] Shan Shan Huang, Amir Hormati, David F. Bacon, and Rodric Rabbah. Liquid metal: Object-oriented programming across the hardware/software boundary. In *ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming*, pages 76–103, Berlin, Heidelberg, 2008. Springer-Verlag.

[54] Gilles Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.

[55] Michal Karczmarek. *Multi-cycle Synthesis from Bluespec Descriptions.* PhD thesis, MIT, Cambridge, MA, 2011.

[56] S. Lahiri, S. Seshia, and R. Bryant. Modeling and verification of out-of-order microprocessors in UCLID. In *FMCAD '02*, volume 2517 of *LNCS*, pages 142–159. Springer-Verlag, November 2002.

[57] Leslie Lamport. Specifying concurrent program modules. *ACM Trans. Program. Lang. Syst.*, 5:190–222, April 1983.

[58] Edward A. Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Computers*, 36(1):24–35, 1987.

[59] S. Y. Liao. Towards a new standard for system level design. In *Proceedings of the Eighth International Workshop on Hardware/Software Codesign*, pages 2–7, San Diego, CA, May 2000.

[60] David C. Luckham. Rapide: A language and toolset for causal event modeling of distributed system architectures. In *WWCA*, 1998.

[61] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989.

[62] Mentor Graphics. *Catapult-C Manual and C/C++ style guide*, 2004.

[63] Laurence W. Nagel. *SPICE2: A Computer Program to Simulate Semiconductor Circuits*. PhD thesis, EECS Department, University of California, Berkeley, 1975.

[64] P.G. Neumann, R.S. Boyer, R.J. Feiertag, K.N. Levitt, and L. Robinson. A Provably Secure Operating System: The system, its applications, and proofs. Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, May 1980. 2nd edition, Report CSL-116.

[65] P.G. Neumann and R.J. Feiertag. PSOS revisited. In *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC 2003), Classic Papers section*, pages 208–216, Las Vegas, Nevada, December 2003. IEEE Computer Society. http://www.acsac.org/ and http://www.csl.sri.com/neumann/psos03.pdf.

[66] P.G. Neumann and Robert N.M. Watson. Capabilities revisited: A holistic approach to bottom-to-top assurance of trustworthy systems. In *Fourth Layered Assurance Workshop*, Austin, Texas, December 2010. U.S. Air Force Cryptographic Modernization Office and AFRL. http://www.csl.sri.com/neumann/law10.pdf.

[67] Man Cheuk (Alfred) Ng, Kermin Elliott Fleming, Mythili Vutukuru, Samuel Gross, Arvind, and Hari Balakrishnan. Airblue: A System for Cross-Layer Wireless Protocol Development. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, La Jolla, CA, October 2010.

[68] Eriko Nurvitadhi, James C. Hoe, Shih-Lien L. Lu, and Timothy Kam. Automatic multithreaded pipeline synthesis from transactional datapath specifications. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 314–319, New York, NY, USA, 2010. ACM.

[69] Jingzhao Ou and Viktor K. Prasanna. Pygen: A matlab/simulink based tool for synthesizing parameterized and energy efficient designs using fpgas. In *Proc. Int.*

Symp. on Field-Programmable Custom Computing Machines, IEEE Computer, pages 47–56. Society Press, 2004.

[70] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: Combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.

[71] David A. Patterson and John L. Hennessy. *Computer Organization & Design: The Hardware/Software Interface, Second Edition*. Morgan Kaufmann, 1997.

[72] Michael Pellauer, Michael Adler, Michel Kinsy, Angshuman Parashar, and Joel S. Emer. Hasim: Fpga-based high-detail multicore simulation using time-division multiplexing. In *HPCA*, pages 406–417. IEEE Computer Society, 2011.

[73] J. Plosila and K. Sere. Action systems in pipelined processor design. In *In Proc. of the 3rd Int. Symp. on Advanced Research in Asynchronous Circuits and Systems*, pages 156–166. IEEE Computer Society Press, 1997.

[74] Dumitru Potop-Butucaru, Stephen A. Edwards, and Gérard Berry. *Compiling Esterel*. Springer, 2007.

[75] Anand Raghunathan and Niraj K. Jha. Behavioral synthesis for low power. In *International Conference on Computer Design*, pages 318–322, 1994.

[76] L. Robinson, K.N. Levitt, and B.A. Silverberg. *The HDM Handbook*. Computer Science Laboratory, SRI International, Menlo Park, California, June 1979. Three Volumes.

[77] Daniel L. Rosenband. The Ephemeral History Register: Flexible Scheduling for Rule-Based Designs. In *Proceedings of MEMOCODE'04*, San Diego, CA, 2004.

[78] Daniel L. Rosenband. *A Performance Driven Approach for Hardware Synthesis of Guarded Atomic Actions*. PhD thesis, MIT, Cambridge, MA, 2005.

[79] Daniel L. Rosenband and Arvind. Modular Scheduling of Guarded Atomic Actions. In *Proceedings of DAC'04*, San Diego, CA, 2004.

[80] Daniel L. Rosenband and Arvind. Hardware Synthesis from Guarded Atomic Actions with Performance Specifications. In *Proceedings of ICCAD'05*, San Jose, CA, 2005.

[81] J.M. Rushby, F. von Henke, and S. Owre. An introduction to formal specification and verification using EHDM. Technical Report SRI-CSL-91-02, Menlo Park, California, February 1991.

[82] Oliver Schliebusch, Andreas Hoffmann, Achim Nohl, Gunnar Braun, and Heinrich Meyr. Architecture implementation using the machine description language lisa. In *Asia and South Pacific Design Automation Conference*, pages 239–244, 2002.

[83] Xiaowei Shen, Arvind, Xiaowei Shen, Larry Rudolph, and Larry Rudolph. Cachet: An adaptive cache coherence protocol for distributed shared-memory systems. In *In International Conference on Supercomputing*, pages 135–144, 1999.

[84] Satnam Singh. Designing reconfigurable systems in lava. In *Proceedings of the 17th International Conference on VLSI Design*, VLSID '04, pages 299–, Washington, DC, USA, 2004. IEEE Computer Society.

[85] Jrgen Staunstrup and Mark R. Greenstreet. From high-level descriptions to vlsi circuits. *BIT*, pages 620–638, 1988.

[86] Synfora. PICO Platform. `http://www.synfora.com/`.

[87] Jean-Pierre Talpin, Christian Brunette, Thierry Gautier, and Abdoulaye Gamatié. Polychronous mode automata. In *EMSOFT*, pages 83–92, 2006.

[88] Marek Telgarsky, James C. Hoe, and José M. F. Moura. Spiral: Joint runtime and energy optimization of linear transforms. In *International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, volume 3, 2006.

[89] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In R. Nigel Horspool, editor, *CC*, volume 2304 of *Lecture Notes in Computer Science*. Springer, 2002.

[90] Donald E. Thomas and Philip Moorby. *The Verilog hardware description language (3. ed.)*. Kluwer, 1996.

[91] Cheng tsung Hwang, Jiahn humg Lee, and Yu chin Hsu. A formal approach to the scheduling problem in high level synthesis. *IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems*, 10:464–475, 1991.

[92] W. Snyder and P. Wasson, and D. Galbi. Verilator. `http://www.veripool.com/verilator.html`, 2007.

[93] Albert Wang, Earl Killian, Dror Maydan, and Chris Rowen. Hardware/software instruction set configurability for system-on-chip processors. In *Design Automation Conference*, pages 184–188, 2001.

[94] Stephen Williams and Michael Baxter. Icarus verilog: open-source verilog more than a year later. *Linux J.*, 2002:3–, July 2002.