A Formal Framework for Modular Synchronous System Design ⋆

Maria-Cristina V. Marinescu and Martin C. Rinard

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
{cristina, rinard}@lcs.mit.edu

Abstract. We present the formal framework for a novel approach for specifying and automatically implementing systems such as digital circuits and network protocols. The goal is to reduce the design time and effort required to build correct, efficient, complex systems and to eliminate the need for the designer to deal directly with global synchronization and concurrency issues. Our compiler automatically transforms modular and asynchronous specifications of circuits written in our specification language, into tightly coupled, fully synchronous implementations in synthesizable Verilog. We formally state the correctness theorems and give an outline of the correctness proofs for two of the three main techniques that our compiler implements.
Keywords: formal, modular, asynchronous, system design

1 Introduction

We present the formal framework for our novel approach for specifying and automatically implementing efficient systems such as digital circuits and network protocols. We formally state our correctness theorems and give an outline of the correctness proofs for two of the three primary implementation techniques.

Our goal is to reduce the design time, effort, and expertise required to build correct and efficient systems. The key challenge is to reconcile the three goals of (1) shielding the developer from having to deal with difficult global issues such as coordinating the timing of events in widely separated parts of the system, (2) supporting a broad class of systems, and (3) enabling the automated synthesis of systems that are as efficient as the corresponding manually-developed versions. To meet this challenge, we have designed a specification language that is concise, expressive, and simple to use and implemented a compiler able to deliver efficient synchronous implementations of these specifications. Our language supports the following features:

- **Modular Specification via FIFO Queues:** The designer specifies a system as a set of modules connected by conceptually unbounded FIFO queues. These queues *temporally* separate the modules at the design level and enable meaningful local reasoning about the behavior of each module.

⋆ This research was supported in part by NSF Grant CCR-9702297 and the Singapore-MIT Alliance.

- **Atomic Updates:** The designer uses a set of atomic update rules to specify the behavior of each module. Atomic execution allows the developer to focus on one rule at a time when reasoning about the behavior of the system, without the need to consider the complex non-local interactions that occur with explicitly parallel models. This approach also facilitates the automated analysis and transformation of the specification.

The key implementation challenge is to construct the synchronization and scheduling details otherwise given explicitly by the designer. Three techniques together meet this challenge: (1) *relaxation*, which automatically extracts the concurrency from the specification, (2) *global scheduling*, which transforms the specification to implement each unbounded queue as a finite buffer, and (3) *pipelining*, which automatically transforms the base specification to obtain a more efficient pipelined implementation. Except for the last step of our synthesis algorithm, which generates Verilog from symbolic expressions, our compiler technology is not specially targeted to circuit design.

The primary contribution of this paper is the formalization of our design approach. Specifically, we provide a formal definition of our target class of systems and the algorithms that our compiler uses to implement these systems. We also sketch correctness proofs for two of our three primary compiler algorithms. This formal foundation gives the designer the guarantee that, if she starts from a correct initial specification, the resulting implementation is also correct (assuming the compiler is implemented correctly).

The remainder of the paper is organized as follows. Section 2 presents a simple example that illustrates our approach. Section 3 reviews our specification language and the basic idea behind the synthesis and pipelining algorithms. Section 4 presents the formal framework and the correctness proofs. Section 5 presents experimental results. Section 6 discusses related work; we conclude in Section 7.

2 Example

In general, a system consists of state and computation. Our language enables the designer to specify the computation as a set of *modules*. Each module performs local computation and interacts with other modules by reading and writing parts of the state. For circuits, the state holds values across clock cycles, is distributed throughout the circuit, and is implemented as hardware registers and memory. The computations are implemented as combinational logic that transforms data during each clock cycle.

We illustrate this approach by presenting a simple circuit example: a linear pipelined datapath with associated control, which implements a very reduced instruction set: an 'increment' instruction `INC`, and a 'jump if register value zero' instruction `JRZ`. We next present the specification for a three-stage pipelined implementation of this instruction set.

2.1 State

The state consists of all the state variables used to specify the system. Figure 1 presents the state declarations for our example. Line `1` declares three type names:

```
1 type reg = int(3), val = int(8), loc = int(8);
2 type ins = <INC reg> | <JRZ reg loc>;
3 type irf = <INC reg val> | <JRZ val loc>;
4 var pc : loc, im : ins[N], rf : val[8];
5 var iq = queue(ins), rq = queue(irf);
```

Fig. 1. State Variables and Type Declarations in Example

a 3 bit integer value reg which is used to represent architecture register names, an 8 bit integer val which is used for the values in the register file, and an 8 bit integer loc, which represents the locations of the instructions in the instruction memory. To represent a data type with several different formats, introduce a tagged union type similar to those found in ML [23] and Haskell [16]. Line 2 declares a tagged union type ins which represents instructions. An ins type instruction can have of one of two data formats: the format <INC reg> has an INC tag and a register name field of type reg, while JRZ reg loc has the tag JRZ, a register name field of type reg, and a branch target field of type loc. Line 3 declares the type irf for instructions whose register operands the processor has already fetched from the register file. This type declaration is also of tagged union type and reads similarly to the type declaration on line 2.

Line 4 declares the following state variables: a program counter pc of type loc declared on line 1, an instruction memory im of type array of N instructions of type ins, and a register file rf of type array of 8 values of type val. The declarations on line 5 use a predefined data type: a queue is a conceptually unbounded first-in first-out (FIFO) queue that carries values between modules. Our language supports the following primitive operations on queues:

- head(q): Retrieves the first element in the queue q.
- tail(q): Returns the rest of q after the first element.
- insert(q,e): Returns the queue after inserting element e at the end of q.
- replace(e1,e2,q): Returns q after replacing all entries e1 by e2. This operation can involve a partial match if e1 contains don't care fields.
- notin(q,e): Returns *true* if the element e is not in q; otherwise returns *false*. This operation can involve a partial match if e contains don't care fields.
- q = nil: Resets the queue to be empty.

We implement queues in the hardware as a number of registers equal to the length of the queue. Line 5 declares a queue iq of instructions of type ins, for fetched instructions, and a queue rq of instructions of type irf, for instructions whose register operands have been fetched.

2.2 Modules

Our circuit executes a sequence of instructions. To execute each instruction, the circuit performs the following steps:

- It reads the instruction from the instruction memory.
- If the instruction is an INC instruction, it reads a value from the register file, increments it and stores it back into the register file.
- If the instruction is a JRZ instruction with the value in its register argument zero, it jumps to the location argument and continues execution from there.

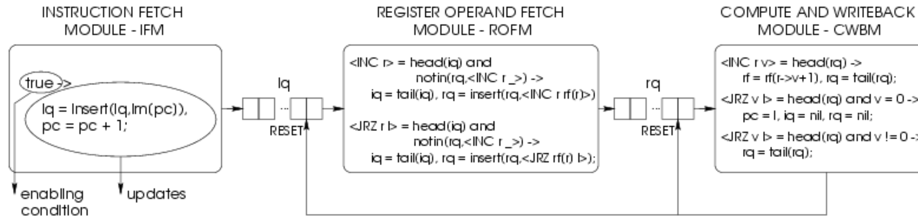

Fig. 2. Specification Example

- If the instruction is a JRZ instruction with the value in its register argument different from zero, it does nothing.

As illustrated in Figure 2, there is one module per pipeline stage: an instruction fetch module IFM, a register operand fetch module ROFM and a compute and write back module CWBM. To keep the pipeline full, the circuit uses a speculative approach that starts the execution of the next instruction before it knows whether the instruction should execute or not. If the circuit starts executing an instruction but later determines that the instruction should not execute, it restores the state of the circuit to reflect the values before the instruction started executing, then restarts the execution. If the speculation was correct, the circuit performs useful computation for each clock cycle that otherwise would have been spent stalling. Pipelining combined with speculation can increase the parallelism in the system and therefore boost the throughput of the circuit.

Modules interact with other modules by reading and writing shared state. The rules in each module read values from input queues and other state variables, perform local computations, then write results to output queues and other state variables. Each rule has an enabling condition and a set of updates to the state, separated by an arrow. When the rules enabling condition evaluates to true, it is enabled and can atomically apply its updates to the current state to obtain the next state. We illustrate this conceptual model of execution by discussing the operation of the rules in ROFM.

The two rules in the module ROFM remove instructions from iq, fetch the register operands, and insert them into rq. The first rule processes INC instructions, and the second one processes JRZ instructions. Both rules use a form of pattern matching similar to that found in ML and Haskell. The enabling condition of the first rule is <INC r> = head(iq) and notin(rq, <INC r _>).

- Its first clause, <INC r> = head(iq), requires that the instruction at the head of iq be an increment instruction with register name r. If this is true, the clause matches and *binds* the variable r to the register name argument of the INC instruction, to be used later in the rule when referring to this operand.
- The second clause, notin(rq, <INC r _>), uses the binding to test that the queue rq does not contain an increment instruction whose first argument is r. The second argument is irrelevant; we denote this using the _ sign. This clause implements a test for a read after write hazard (RAW); if there is a pending instruction waiting to execute that will write the register r, the machine delays

the operand fetch so that it fetches the value after the write (this translates into *stalling*). The clause notin(rq, <INC r _>) implements this check.

3 Background

Before presenting the formal framework and correctness proofs, we informally present the key elements of our approach and outline the algorithms in our compiler.

3.1 Specification Language

Timing issues such as synchronization, concurrency, and execution order are a primary source of complexity for circuit designers. For performance reasons, most languages for designing synchronous circuits are based on a synchronous model of execution. The advantage is that the developer can tightly control the specification to deliver an efficient circuit. The disadvantage is that the tight temporal coupling of this model makes local reasoning difficult, undercutting the advantages of modular design.

Our approach addresses this problem by presenting an asynchronous abstract model of execution. The designer specifies a circuit as a set of modules connected by conceptually unbounded FIFO queues; the update rules that model the computations of the modules execute atomically and asynchronously with regard to each other. This is a standard model of asynchronous execution found in systems such as Unity [8] and term rewriting systems [2]. Our synthesis algorithm eliminates the potential inefficiency associated with a direct asynchronous implementation by automatically generating a coordinated global schedule for all operations in the system. This schedule is used to generate an efficient synchronous implementation in synthesizable Verilog.

Advantages: Using conceptually unbounded queues to connect the modules has two benefits: (1) it enables the designer to reason about and develop each module in isolation, then compose the modules together into a complete system without the need to deal with complex global issues such as the coordinated assignment of operations to clock cycles, and (2) it enables the designer to reason about the correctness of the specifications without reasoning about the concurrent execution of the composed modules. In this sense, queues localize the temporal aspects of the design.

3.2 Implementation

A novelty of our approach is that it takes a modular specification with asynchronous execution semantics and converts it into a synchronous, parallel implementation. It is the compiler's job to efficiently bridge the gap between these models of execution. The key idea of the synthesis algorithm is to automatically compose the module specifications to derive, at the granularity of individual

clock cycles, a global schedule for the operations of the entire system, including the removal and insertion of queue elements. The resulting implementation executes efficiently in a completely synchronous, pipelined manner [20].

To decrease the clock cycle of the generated circuit, our compiler implements a technique we call relaxation, which makes it possible to evaluate many or all of the enabling conditions immediately and in parallel rather than sequentially. The compiler also implements a set of techniques geared towards optimizing the generated combinational logic[1].

These algorithms are all based on the underlying assumption that each individual operation takes less time than the clock cycle time to complete. For the case in which this assumption does not hold, we have developed an automated technique that implements functional pipelining for sequential circuits [19]. The existing synthesis algorithm would need to be extended if certain individual basic operations are too expensive to implement in combinational logic.

Relaxation: The operation preceding relaxation in our synthesis algorithm orders all the rules of the specification for symbolic execution. The specification produced by this operation may suffer from an excessively long clock cycle, as the execution of the rules modifying shared state is completely sequentialized. Our implementation executes multiple rules at each clock cycle as follows. For each set that consists of rules with no data dependencies within the set, the compiler can execute all the rules in the set in parallel. If a set of rules has data dependencies, our compiler transforms the rules, when possible, so that their enabling conditions test the state of the system at the beginning of the clock cycle rather than the state created by the previously executing rule. For a rule R that updates state variables tested by a subsequent rule R', if we can prove that the execution of R will not disable the enabling condition of R', we can modify the precondition of R' to test the state before R executes. To preserve correctness, the updates still execute sequentially if they operate on the same state variable. This transformation ensures that each element of data traverses at most one module per clock cycle, producing an acceptable critical path for the circuit. By eliminating unnecessary serialization, we expose the additional parallelism in the specification and shorten the clock cycle of the circuit, and, indirectly, increase its throughput. Relaxation does not insert or remove delays in or from the circuit.

Global Scheduling: In the initial specification, queues have unbounded length. But the implementation must have a finite, specific number of entries allocated for each hardware buffer implementing a queue. The designer decides on the amount of memory elements he or she is willing to spend on each of these buffers, and the compiler generates an implementation based on this budget that, for any execution instance of the system, does not exceed that length.

[1] These optimizations include common subexpression elimination and mutual exclusion testing. The former avoids unnecessary replication of hardware, while the latter eliminates false paths in the implementation.

The global scheduling algorithm can handle specifications whose rules can have both acyclic and cyclic queue insertion and removal dependencies. For cyclic specifications, the compiler looks at groups of rules that must execute together to maximize the concurrency and avoid both deadlock and overflow of the queues in the system; for acyclic specifications, it only needs to consider each rule in isolation. The scheduler augments each rule that inserts an element into a queue to ensure that it never causes any of the corresponding finite hardware buffers to overflow. The basic approach is to assume all queues are within length at the beginning of the clock cycle and schedule only those rules for firing that are 1) enabled and 2) whose combined execution leaves the queues within their length at the *end* of the clock cycle. As a result, the circuit can perform single or multiple reads or writes from and into each queue in the same clock cycle, even if the queues are initially full. The condition is that enough rules will execute that remove elements from queues, therefore making space for new elements to be inserted. Queues can get arbitrarily[2] large during the clock cycle as long as they are within the maximum specified length at the end of the cycle.

The generated global schedule enables the synchronous and concurrent execution of multiple rules per clock cycle. In hardware, global scheduling corresponds to generating the control signals for the combinational logic, and a given length of 1 for each queue translates into the synthesis of a standard pipeline. The global scheduling algorithm is the key to efficient pipelining; it also may reduce the area of the resulting circuit.

Pipelining: This transformation automatically generates a pipelined specification from a non-pipelined or insufficiently pipelined specification. The pipelining algorithm repeatedly shortens the clock cycle of the circuit by extracting a computation from the critical path and moving it into a new pipeline stage. The new stage precomputes the result of the selected expression, using a new queue to pass the result to the module from which the expression was removed. To keep the pipeline full, the new stage must produce the next value of the expression before the final values of the variables it accesses become available. The algorithm achieves this goal by speculating on these values, using state retention and recovery to respond to incorrect speculations.

Our algorithm uses several techniques to improve the quality of the pipelined circuit. If the amount of state necessary to recover from an incorrect speculation is excessive, our algorithm can generate stall logic that causes the pipeline stage to stall until the new values are available. This technique eliminates the need for retaining recovery state, as the execution of the pipeline stage will never need to roll back. Our algorithm can also generate circuits that forward the correct value to preceding pipeline stages. This technique increases the throughput of the circuit by reducing the amount of time that the circuit spends recovering

[2] The maximum number of elements in a queue at any time is the maximum number of non-mutually exclusive rules that append elements into the queue plus the length of the buffer implementing the queue. We communicate these values within a clock cycle via wires.

from incorrect speculations or waiting for correct values to become available. The pipelining transformation preserves the property that every register and memory variable in the circuit specification observes the same sequence of values after as before pipelining.

Our pipelining algorithm reduces the designer time and effort by automating complex techniques. The less obvious advantage, though equally important, is that it increases the confidence in the correctness of the resulting implementation. Pipelining starts from an easy specification with little or no concurrency, which is easy to verify for correctness; the resulting pipelined version needs to be highly concurrent, and therefore manually developed versions are a lot harder to verify.

4 Formalism

The first part of this section describes the general formal framework necessary for proving the correctness of the relaxation and global scheduling algorithms. We then define the specification of the system before and after applying each algorithm, as direct or extended instances of the general framework. The second part of the section states and gives a short outline of the proofs for theorems regarding the correctness properties of our algorithms.

4.1 Formal Definitions

We define a **system** to be a tuple $System\ G = \langle T, ex, f, g \rangle$.
A **transition system** is a set of **transitions** $T = \{t_i \equiv l_i : c_i \Rightarrow u_i\}$.
A **transition** $t \equiv l : c \Rightarrow u \in T$ has a label $l \in Label = \{1, ..., |T|\}$, a condition $c \in C$ and an update $u \in U$.
We have an **external** function $ex : t \rightarrow Bool$ s.t. $ex(t)$ is true if the transition t is observable from the exterior and false otherwise. We say that a transition $t \equiv l : c \Rightarrow u$ is **external** iff $ex(t)$ is true and **internal** otherwise.
The **set** S **of states** $s \in S$ is the set of functions $S = Vars \rightarrow Vals$, where $Vars$ is the set of all register, memory state variables and queues ($Queues$) in the circuit specification, and $Vals$ is the set of all values that the state variables in $Vars$ can take.
We define two functions f and g as follows:

- We assume a set of conditions C, a set of expressions E and a set of updates U. The evaluation of some $c \in C$ in some state returns a $Bool$ value. The evaluation of some $e \in E$ in some state returns a value in $Vals$. To make the notation more concise, we extend $Vals$ to include $Bool$. The evaluation function $f : E \times S \rightarrow Vals$ returns the value of expression $e \in E$ in $s \in S$.
- An update function $g : U \times S \rightarrow S$. $g(u, s) = s^{'}$ applies the updates in u to the state s and returns the modified state $s^{'}$.

Each transition system T defines a **transition relation** $R \subseteq S \times T \times S$.

$$R = \{\langle s, t, s^{'} \rangle.\ t \equiv l : c \rightarrow u \in T \wedge f(c, s) \wedge g(u, s) = s^{'}\}$$

Assume there exists an **initial state** s_0 of T. In s_0 the following are true:

- $\forall v_k \in Vars - Queues,\ s_0(v_k) = initialVal_k$, where $initialVal_k \in Vals$ are the initial values of the registers and memories in the circuit.
- All the queues are empty: $\forall k$ s.t. $1 \leq k \leq maxQueue,\ s_0(queue_k) = \{\}$

An **execution fragment** is a finite alternating sequence of states and transitions $frag = \{s_1 t_1 s_2 t_2 s_3 ... s_n \mid s_i \in S \ .\ t_i \in T \ .\ \langle s_i, t_i, s_{i+1}\rangle \in R\}$.
An **execution** is an execution fragment starting in the initial state s_0.
A state s is **reachable** if s is the final state of some finite execution.
An **execution sequence** is the sequence of states in an execution obtained after dropping the intermediate transitions:

$$\tau = \{s_1, s_2, ..., s_n \mid s_1 t_1 s_2 t_2 ... s_n \text{ is an execution}\}$$

SPEC: is an instance of $System$ of the form $\langle T^S, ex^S, f^S, g^S\rangle$. T^S is a transition system that represents the nondeterministic specification of the circuit. We skip over the actual definitions for the set of expressions E^S, the set of updates U^S, and the functions f^S and g^S. For more details see [18].
RELAX: is an instance of $System$ of the form $\langle T^R, ex^R, f^R, g^R\rangle$. T^R is a transition system that represents the circuit implementation obtained as the result of relaxation.
We define a new **external** function ex^R that has the same values as $ex^S(t_i)$ for each transition $t_i \in T^R$ with the same label as the transition in T^S. We defer the definition of the external function for the newly introduced transitions to the construction of the new transition system.
The **set** S^R **of states** $s^R \in S^R$ is the set of functions $S^R = (Vars \times Versions) \cup \{\mathsf{pc}\} \rightarrow Vals$, where $Versions$ is a set of integers and pc is a variable of type integer that represents the ordinal number of the transition in the relaxed implementation that is currently under evaluation. pc provides a way to express the deterministic execution of the transitions in the relaxed circuit, following the order used by the relaxation transformation.
The set of expressions E^R in RELAX is $e^R ::= c \mid (v, n, p) \mid \mathsf{pc} \mid \rho({e^R}_1,...,{e^R}_n)$
The triple (v, n, p) stands for a variable $v \in Vars$, its version $n \in Versions$ and position $p \in Position$. $Position$ is a set of integers. Our transition system contains positions, while the state does not. A position and variable name pair uniquely denotes a state variable instance within the condition of a transition.
We also define the set of updates U^R in RELAX

$$ur ::= \mathsf{pc}{+}{+} \mid \mathsf{pc} = 1 \mid (v_1, nr_1, p_1) = {e^R}_1, ..., (v_n, nr_n, p_n) = {e^R}_n, \mathsf{pc} = \mathsf{pc}{+}{+}$$
$$\mid (v_1, nr_1, p_1) = {e^R}_1, ..., (v_n, nr_n, p_n) = {e^R}_n, \mathsf{pc} = 1$$

We introduce a function **update**: $U^R \times S^R \rightarrow \mathcal{P}(Vars)$ which takes an update $u^R \in U^R$ and a state $s^R \in S^R$ and returns the set of variables in $Vars$ that get updated.
We define a numbering function as follows:
RN: $(E^S \cup U^S) \times$ Version $\rightarrow E^R \cup U^R$
$RN(c)(n) = c$
$RN(v)(n) = (v, n, 0)$
$RN(\rho({e^S}_1, ..., {e^S}_n))(n) = \rho(RN({e^S}_1)(n), ..., RN({e^S}_n)(n))$
$RN(\mathsf{pc}++)(n) = \mathsf{pc}++$

$RN(\mathsf{pc}=1)(n) = \mathsf{pc}=1$
$RN((v_1, nr_1, p_1) = e^R{}_1, ..., (v_n, nr_n, p_n) = e^R{}_n, \mathsf{pc}++)(k) =$
$\quad (v_1, k+1, 0) = RN(e^S{}_1)(k); ...; (v_n, k+1, 0) = RN(e^S{}_n)(k), \mathsf{pc}++$
$RN((v_1, nr_1, p_1) = e^R{}_1, ..., (v_n, nr_n, p_n) = e^R{}_n, \mathsf{pc}=1)(k) =$
$\quad ((v_1, k+1, 0) = RN(e^S{}_1)(k); ...; (v_n, k+1, 0) = RN(e^S{}_n)(k), \mathsf{pc}=1$
We define a **variable positioning** function as follows:
$VP:\ E^R \times Position \rightarrow E^R \times Position$
$VP(c, i) = (c, i)$
$VP((v, n, 0), i) = ((v, n, i), i+1)$
$VP(\mathsf{pc}, i) = (\mathsf{pc}, i)$
$VP(\rho(e^R{}_1, ..., e^R{}_n), i) = \text{let } (a_1, b_1) = VP(e^R{}_1, i), (a_2, b_2) = VP(e^R{}_2, b_1), \ldots,$
$\quad (a_n, b_n) = VP(e^R{}_n, b_{n-1}) \text{ in } (\rho(a_1, a_2, ..., a_n), b_n)$
We can now define a relaxation function RE that replaces the current version of each variable $v \in Vars$ in position $p \in Position$ with its relaxed version. We obtain the relaxed version of a variable $v \in Vars$ with original version $n \in Versions$ in position $p \in Position$ by invoking a function $\sigma \in \Sigma : Vars \times Versions \times Position \rightarrow Versions$. The original version n is equal to $l \bmod |T^S|$, where l is the label of the transition that invokes $\sigma \in \Sigma$. From the construction of the new transition system we will see that the tuple (v, n, p) is unique within the set of all the external transitions in T^R.
$RE : E^R \times \Sigma \rightarrow E^R$
$RE(c, \sigma) = c$
$RE((v, n, p), \sigma) = (v, \sigma(v, n, p), p)$
$RE(\mathsf{pc}, \sigma) = \mathsf{pc}$
$RE(\rho(e^R{}_1, ..., e^R{}_n), \sigma) = \rho(RE(e^R{}_1, \sigma), ..., RE(e^R{}_n, \sigma))$
We define a new **transition system** $T^R = \{t_i \equiv l_i : c_i \Rightarrow u_i | i \in \{1, ..., n\}\}$ by modifying the previous transition system T^S as follows:

- For each transition $t \equiv l : c \Rightarrow u \in T^S$, construct two transitions in T^R, one external, one not external, as follows:
  $t_l \equiv l : (\mathsf{pc} == l)\ \wedge\ RE(\pi_1(VP(RN(c^S)(l), 1)), \sigma)\ \Rightarrow RN(u^S)(l), \mathsf{pc}++;$
  $t_{|T|+l} \equiv |T|+l : (\mathsf{pc} == l)\ \wedge \overline{RE(\pi_1(VP(RN(c^S)(l), 1)), \sigma)}\ \Rightarrow \mathsf{pc}++;$
  $|T|+l$ is a fresh label and $ex^R(t_{|T|+l}) = \mathsf{false}$. We use π_1 for the projection of the first element of a tuple. For a pair (a, b) we have $\pi_1(a, b) = a$.
- Create a new transition $t_{2*|T|+1}$ to express the wrap-around after the relaxation algorithm tried all other transitions and either executed them or not. The new transition is not external.

  $$\forall v \in Vars\, t_{2*|T|+1} \equiv 2*|T|+1 : (\mathsf{pc} == |T|+1) \Rightarrow s(v, 1) = s(v, s(\mathsf{pc})),\ \mathsf{pc} = 1;$$

  $2*|T|+1$ is a fresh label and $ex^R(t_{2*|T|+1}) = \mathsf{false}$.

Assume there exists an **initial state** $s^R{}_0$ of T^R. In $s^R{}_0$ the following are true:

- $\forall v_k \in Vars - Queues, \forall i \in \{1, ..., |T|+1\},\ s^R{}_0(v_k, i) = initialVal_k$, where $initialVal_k \in Vals$ are the initial values of the registers and memories in the circuit.
- $s^R{}_0(\mathsf{pc}) = 1$
- All the queues are empty: $\forall k$ s.t. $1 \leq k \leq maxQueue,\ s^R{}_0(queue_k) = \{\}$

We skip over the actual definitions for functions f^R and g^R. For details see [18].
RELAXQ: is a set of two inputs, a system $\langle T^R, ex^R, f^R, g^R\rangle$ of the same type with RELAX, and a user-defined length $maxLength(queue_k)$ for each queue $queue_k, k \in \{1, ..., maxQueue\}$. The semantics of a transition in this system is the same as the semantics of a transition in RELAX with one difference. If, for the given queue lengths, executing the update of the transition would overflow at least one of the queues, that transition is not enabled for execution, i.e:

$$[\![\langle c \Rightarrow u,\ T^R\rangle]\!] = \{\langle s,\ c \Rightarrow u,\ g^R(u,s)\rangle\ .\ f^R(c,s) \text{ and } \ u \Rightarrow \text{overflow}(s_i) \text{ is } \mathsf{false}\}$$

where

$$\forall i \in \{s(\mathsf{pc})+1, ..., |T|\}, s_i = \begin{cases} g^R(u_{i-1}, s_{i-1}) \text{ if } f^R(c_{i-1}, s_{i-1}) \\ s_{i-1} \qquad\qquad\quad \text{otherwise} \end{cases} \tag{1}$$

and overflow(s) is only defined for the states in which $s(pc) = |T|+1$ as:

$$\text{overflow}(s) = \begin{cases} \exists queue_k \in Queues\ .\ length(queue_k, s) > maxLength(queue_k) \\ \qquad\qquad\qquad\qquad\quad \text{if } s(\mathsf{pc}) = |T|+1 \\ undefined \qquad\qquad\quad \text{otherwise} \end{cases}$$

$length : Queues \times S^R \to Int$ is a function that returns the current length of the queue in the given state. We also define a function $room^R : Queues \times S^R \to Int$ that returns the number of empty slots of the given queue in the given state.
A transition system with this semantics defines a transition relation $R^R_q \subseteq S^R \times T^R \times S^R\ .\ R^R_q = \{\langle s^R, t^R, s^{R'}\rangle\ .\ t^R \equiv l_r : c^R \to u^R \in T^R\ \wedge\ f^R(c^R, s^R)\ \wedge\ g^R(u^R, s^R) = s^{R'}\ \wedge\ u^R \Rightarrow \text{overflow}({s^R}_i) \text{ is } \mathsf{false}\}$, where ${s^R}_i$ is defined as in (1).
FINIT: is an instance of $System$ of the form $\langle T^F, ex^F, f^F, g^F\rangle$. T^F is a transition system that represents the circuit implementation after global scheduling.
We define two functions, $tail : t \times Queues \to Bool$ and $append : t \times Queues \to Bool$ that take a transition and a queue and return true iff the transition contains a $tail$ or, correspondingly, an $append$ operation on the queue given as parameter.
We also define a function $room^F : Queues \times S^F \to Int$ that returns the number of empty slots of the given queue in the given state.
We call a transition $t \equiv l : c \Rightarrow u \in T^F$ **appending** if $\exists queue_k\ \in\ Queues\ .\ append(t, queue_k) = \mathsf{true}$.
We define a new **transition system** $T^F = \{t_i \equiv l_i : c_i \Rightarrow u_i | i \in \{1, ..., n\}\}$ by modifying the previous transition system T^R such that for every appending transition $t\ \equiv l:\ c\ \Rightarrow\ u\ \in\ T^R$, we construct a transition in T^F of the form

$$t\ \equiv l:\ c\ \wedge\ finalLengthOK(s^F, T^F, \text{currentPath})\ \Rightarrow\ u$$

where currentPath starts as [](nil).
Let a function $eval : Bool \to \{0,1\}$ return 1 for true and 0 for false. For some appending transition $t^F \equiv l^F : c^F \Rightarrow u^F\ \in\ R^F$ from state s^F, we define:

$$finalLengthOKqueue(s^F, T^F, queue_k, \text{currentPath}) =$$
$$\sum_i eval(tail(t_i, queue_k) \bigwedge$$
$$(f^F(c^F{}_i \wedge finalLengthOK(s^F{}_i, T^F, newPath), s^F{}_i) \vee (t_i \in \text{currentPath})))$$
$$+ room(queue_k, s^F) > 0, \ \forall i \in \{l^F + 1, ..., |T|\} \ . \ tail(t_i, queue_k) = \mathsf{true},$$

$$\text{where } s^F{}_{l^F} = s^F \text{ and } s^F{}_i = \begin{cases} g^F(u^F{}_{i-1}, s^F{}_{i-1}) & \text{if } f^F(c^{F'}_{i-1}, s^F{}_{i-1}) \\ s^F{}_{i-1} & \text{otherwise} \end{cases}$$

$$newPath = \begin{cases} \text{currentPath} & \text{if } t_i \in \text{currentPath} \\ \text{currentPath} \vee t_i & \text{otherwise} \end{cases}$$

$$finalLengthOK(s^F, T^F, \text{currentPath}) =$$
$$\bigwedge finalLengthOKqueue(s^F, T^F, queue_k, \text{currentPath}),$$
$$\forall queue_k \in Queues \ . \ append(t^F, queue_k) = \mathsf{true}$$

Here currentPath holds the set of currently explored transitions for each starting transition in the system. This set is necessary for cyclic specifications.
The semantics of a transition in T^F is the following. If, for the given queue lengths, executing the update of the transition does not ensure that all the queues are within their maximum lengths when $s^F(\mathsf{pc}) = |T| + 1$, then T^Fgoes into an ERROR state s_{ERROR}.
We will show that the condition $finalLengthOK(s^F, T^F, \text{currentPath})$ makes sure that FINIT never goes into an ERROR state provided that the designer specifies appropriate lengths for all the queues.
Assume there exists an **initial state** s^F_0 of T^F. In s^F_0 the following are true:

- $\forall v_k \in Vars - Queues, \forall i \in \{1, ..., |T| + 1\}$, $s^F_0(v_k, i) = initialVal_k$, where $initialVal_k \in Vals$ are the initial values of the registers and memories in the circuit.
- $s^F_0(\mathsf{pc}) = 1$
- All the queues are empty: $\forall k$ s.t. $1 \leq k \leq maxQueue$, $s^F_0(queue_k) = \{\}$

4.2 Correctness

We prove two properties for each algorithm: (1) simulation: the transformed and the original circuit are in a simulation relation, and (2) non-termination: the transformed circuit preserves the non-termination property of the original circuit. Relaxation takes as input a system of type SPEC, and outputs a system of type RELAX. Global scheduling takes a set of two inputs of the form described in RELAXQ, and outputs a system of type FINIT.

1. Relaxation - Simulation

We want to prove that the behavior of the resulting specification after relaxation never does anything that the specification before relaxation could not do. This

means that we want to prove that for any execution in RELAX, we can find an execution in SPEC with the same execution sequence.
We first define the abstraction function AF that maps each state of RELAX to a state of SPEC: $AF(s^R) \rightarrow \{\forall v \in Vars \mid s^S(v) = s^R(v, s^R(\mathsf{pc}))\}$.

Theorem 1: (Simulation) $\forall s^R \in S^R, \forall s^S \in S^S, \forall t^R, \forall s^{R'} \in S^R \;.\; s^S = AF(s^R)$
if $\langle s^R, t^R, s^{R'}\rangle \in R^R$ then
$(\exists t^S, \exists s^{S'} \in S^S . \langle s^S, t^S, s^{S'}\rangle \in R^S$ and $s^{S'} = AF(s^{R'}))$ or $s^S = AF(s^{R'})$

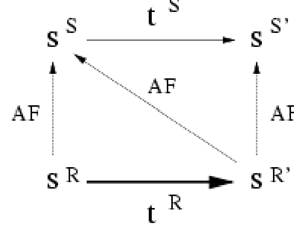

Fig. 3. Commutative Diagram for Simulation

The proof goes by induction on the length of the execution sequence. The induction step is a case analysis on the form of the transition t^R. It uses the definitions of the functions f^R, g^R, AF and the following property:

$$\forall s^R \in S^R,\ \forall i \;.\; \text{ConsistentExecutionTrace}(s^R, T^R) \text{ and}$$
$$f^R(RE(\pi_1(VP(RN(c^S{}_i)(l_i), 1)), \sigma), s^R) \;\wedge\; (\mathsf{pc} \geq l_i)$$
$$\rightarrow f^R(\pi_1(VP(RN(c^S)(l_i), 1)), s^R)$$

where ConsistentExecutionTrace(s^R,T^R) is an invariant that ensures that the execution trace $T^R$is valid.
The intuition is that we can follow the constructive steps of the relaxation algorithm to prove that, if the enabling condition of some relaxed transition evaluates to true, than the enabling condition of the original transition also evaluates to true. Therefore, the transformed specification never takes any step that that the original specification cannot take.

2. Relaxation - Non-termination

We want to prove that relaxation cannot stop the progress in the execution of the system. Non-termination says that if from any state s of SPEC there exists an execution step that can be taken, there exists an execution step in RELAX that can be taken from some state in RELAX reachable only by internal transitions from any state that maps to s using AF.

$$\text{Let } R_0 = \{\langle s_1,\ s_2\rangle | \langle s_1,\ t,\ s_2\rangle \in R^R \text{ and } ex^R(t) = \mathsf{false}\}$$

Theorem 2: (Non-termination) $(\forall s^S \in S^S, \forall s^R \in S^R)$ such that (ConsistentExecutionTrace(s^R, T^R) and $AF(s^R) = s^S$,
if $\exists t^S \;.\; \langle s^S,\ t^S,\ s^{S'}\rangle \in R^S$ then $\exists t^R\ \exists s^{R*}$.
$\langle s^R,\ s^{R*}\rangle \in R_0^{3*|T|+1}$ and $\langle s^{R*},\ t^R,\ s^{R'}\rangle \in R^R$ and $ex^R(t^R) = \mathsf{true})$

Lemma 1:

For any $k \geq 0$ and some state $s^R \in S^R$ s.t. $AF(s^R) = s^S$ and $l = s^R(\mathsf{pc})$, there exists a sequence of states s^{R0},...,s^{Rk} in RELAX starting from $s^{R0} = s^R$ s.t. the following hold: $\langle s^{R^{k-1}}, s^{R^k} \rangle \in R_0^1$, $\langle s^R, s^{R^k} \rangle \in R_0^k$, $s^S = AF(s^{R^k})$, and $\forall v \; s^{R^k}(v, l + d(k,l)) = s^R(v,l)$, where k is the number of transition executions in RELAX and

$$d(k,l) = \begin{cases} k & \text{if } k \leq |T| + 1 - l \\ k - (|T|+1) & \text{otherwise} \end{cases}$$

The proof for *Lemma 1* goes by induction on the number of transition executions in RELAX. For k in *Lemma 1* s.t. $k = |T| + minI - l + 1$, where $minI$ is the smallest integer label of all the transitions in SPEC that can execute from s^S, we can prove the non-termination theorem by contradiction. The intuition behind the non-termination proof is that since a rule R in the transformed numbering tests a state previous to the current state, if R is enabled in the original numbering but not in the transformed one, *some* rule *does* execute in the transformed numbering and modifies the state originally tested by R.

3. Global Scheduling - Simulation

To prove that for any execution in FINIT, we can find an execution in RELAXQ with the same execution sequence, we first define an abstraction function AF^F that maps each state of FINIT to a state of RELAXQ.

$$FUNC\ AF^F(s^F) \rightarrow \{\forall v \in Vars | s^F(v,n) = s^R(v,n)\}$$

Theorem 3: (Simulation) $\forall s^F \in S^F,\ \forall s^R \in S^R,\ \forall t^F,\ \forall s^{F'} \in S^F$,

if $s^R = AF(s^F)$ and $\langle s^F, t^F, s^{F'} \rangle \in R^F$, then

$\exists t^R,\ \exists s^{R'} \in S^R \;.\; s^{R'} = AF(s^{F'})$ and $\langle s^R, t^R, s^{R'} \rangle \in {R^R}_q$.

The proof proceeds by induction on the length of the execution sequence. The induction step is a case analysis on the form of the transition t^F and it is similar (and less difficult) then the simulation proof for the relaxation algorithm. The idea is that if a transition in the transformed system executes, then it would have also executed in the original system. This is true because the enabling condition is strictly stronger after than before applying the scheduling transformation. Global scheduling will disable those rules whose execution would result in overflowing at least one queue in the system, but will never enable a rule that was not enabled when the queues were unbounded.

4. Global Scheduling - Non-termination

We want to prove that if from any state s of RELAXQ there exists an external execution step that can be taken, then there exists an external execution step in FINIT that can be taken from any state that maps to s using AF s.t. the system does not go into an ERROR state. We also want to prove that FINIT goes into an ERROR state iff the only transition from the state s would overflow at least one of the queues. In other words, given enough buffer space such that executing the original specification on this budget does not exceed the given queue lengths, global scheduling does not introduce deadlock.

Theorem 4: (Non-termination) $\forall s^R \in S^R,\ \forall s^F \in\ S^F$,

if $\exists t^R\ .\ AF(s^F) = s^R$ and $\langle s^R,\ t^R,\ s^{R'} \rangle\ \in\ {R^R}_q$ and $ex^R(t^R) = \mathsf{true}$ then $\exists t^F,\ \exists s^{F'} \in S^F . \langle s^F, t^F, s^{F'} \rangle\ \in\ R^F$ and $ex^F(t^F)$ and $s^{F'} \neq s_{ERROR}$.

Also, $\forall s^R \in S^R, \forall s^F \in\ S^F, \forall t^R$,

if $AF(s^F) = s^R$ and $\langle s^R,\ t^R,\ s^{R'} \rangle\ \in\ R^R$ and $ex^R(t^R)$) then $\langle s^R, t^R, s^{R'} \rangle\ \notin\ {R^R}_q$ iff $finalLengthOK(s^F, T^F, \text{currentPath}) = \mathsf{false}$.

Our global scheduling algorithm generates a correct transformed specification that correctly deadlocks if the designer specifies lengths for the queues that are not large enough for the particular application. It is the designer's responsability to know what queue length values are enough for the given circuit not to deadlock, we prove that, given such lengths, our scheduling algorithm does not introduce deadlock in the system.

The proof for (4.2) proceeds by contradiction to show that if an external transition t takes place in RELAXQ, we can infer that $room^F(q, s) > 0$, and therefore there is space in q for a transition t' in FINIT to execute. To prove (4.2), we start from the current states in RELAXQ and FINIT, in which we know that all corresponding queues have the same number of elements. We then only have to prove that corresponding rules in these two systems, following transitions t in RELAXQ and t' in FINIT, either both execute or neither does. This is sufficient because it proves strict equality between the lengths of corresponding queues in RELAXQ and FINIT, at the end of the cycle. Let r and r' be the corresponding rules immediately following t in RELAXQ and t' in FINIT. If r and r' are not appending rules, they both execute if their enabling conditions — which are identical — evaluate to true. Otherwise, none of them executes. If r and r' are appending rules, we reduced proving (4.2) for t and t' to proving (4.2) for r and r'. Because the number of rules following t and t', correspondingly, is finite, we will eventually reach the last rules in the two systems, where (4.2) holds, since there are no more following rules.

5. Global Scheduling - Correctness for Groups of Transitions

For cyclic specifications, the algorithm considers the coordinated execution of groups of transitions, rather than transitions in isolation. We call a **phase** a sequence of external transition executions, such that each transition executes at most once. The simulation theorem states that for any phase in FINIT, we can find a phase in RELAXQ with the same execution sequence. The proof is virtually identical to the simulation proof for one transition, and is based on the fact that the enabling condition of each of the transitions is strictly stronger in FINIT than in RELAXQ.

We can formulate a new non-termination theorem for groups of transitions which states that, if from any state s of RELAXQ there exists a phase that takes the system into a new state in which none of the queues overflows its designer-specified length, then from any state in FINIT, s', that maps to s using AF, there exists a phase in FINIT which does not take the new system into an ERROR state. The proof goes by contradiction and works on a phase instead of a single transition at a time. The idea is to infer that the transition from s'

in the FINIT phase corresponding to the executing RELAXQ phase would have its enabling condition satisfied, and therefore execute.

5 Experimental Results

We have implemented a synthesis and pipelining system based on the algorithms presented in Sections 3.2 and 4. Our experiments are designed to investigate two aspects of using our system: (1) how natural and concise it is for the designer to write circuit specifications in our language, and (2) how well the resulting implementations perform. To evaluate our system, we developed a set of benchmarks in our specification language and used our system to produce synthesizable Verilog implementations at the RTL level. We then synthesized the resulting implementations using the Synopsys Design Compiler to an industry standard .25 micron standard cell process. We obtained manually written Verilog descriptions of the same or functionally equivalent circuits as the ones in our benchmark set, and we synthesized them in the same environment as the automatically generated versions. This is our reference point for performance evaluation.

Our benchmark set contains a processor and a few standard DSP applications: a bubblesort network, a butterfly network like the ones used in bitonic sort and FFTs, and a cascaded FIR filter. The processor is a 32-bit datapath, RISC-style, linearly pipelined processor with a complete instruction set. We obtained manually written versions of bubblesort and butterfly sort networks from the RAW benchmark suite at MIT. We were unable to obtain a free manually developed FIR application to match against our automatically generated FIR circuit. We obtained the processor benchmark off the web, from Santa Clara University; this is a standard 32-bit fixed point DSP that implements the same basic functionality as our processor. Figure 4 and Figure 5 show cycle time (MHz), total circuit area and and register area numbers for our four benchmarks and the corresponding manually written Verilog versions.

Benchmark	Cycle	Area	Register Area
Bubble Sort	324.67	1803.75	1371
Butterfly	204.08	1881.125	969
FIR filter	103.41	7384	3529
Pipelined Processor	88.89	28845	7533

Fig. 4. Clock Cycle and Area Estimates for Automatically Generated Versions

Benchmark	Cycle	Area	Register Area
Bubble Sort	308.64	1475.75	1192
Butterfly	120.34	2041.125	1348
FIR filter	—	—	—
SCU RTL 98 DSP	90.91	28359.75	7147

Fig. 5. Clock Cycle and Area Estimates for Manually Written Versions

5.1 Design Effort Evaluation

It took us less than 5 hours to develop the specification for the processor, and about 10 minutes for each of the other benchmarks. We believe this is significantly faster than developing the corresponding models by hand. Our processor specification contains 13 type and state declarations and 29 rule definitions for module specifications. The SCU RTL 98 DSP application, on the other hand, consists of approximately 885 lines of Verilog code. Our automatically generated implementation consists of about 1200 lines of synthesizable Verilog. The bubblesort benchmark has 2 multiple state declarations and 12 very simple rule definitions. The butterfly network has 3 multiple state declarations and 13 simple rule definitions. The FIR filter benchmark has 5 multiple state declarations and 4 rule definitions. The manually written specifications have 200 lines of Verilog code for bubblesort and 378 for butterfly.

The specification-to-Verilog synthesis time is roughly proportional to the complexity of the generated control. For all applications except the pipelined processor, our system required less than one minute to generate the Verilog output. For the processor, it took roughly half an hour. The synthesis times for the corresponding automatically generated Verilog versions, and manually written versions is comparable, and last roughly from 1 to 4 minutes for bubblesort and butterfly, while the automated version for the FIR filter takes about 15.00 minutes to synthesize. Our automatically generated RISC processor benchmark takes about 3:17 hours to synthesize; the functionally equivalent, manually developed SCU RTL 98 DSP application takes about 27.00 minutes to synthesize.

5.2 Performance Evaluation

For the bubblesort network, our compiler generates a circuit that is about 5 percent faster, and about 22 percent larger than the equivalent manually written version. The number of registers generated in the automatically synthesized version is about 15 percent larger than the equivalent number of registers in the manually written application. The extra register area comes from the counters and valid bits associated with each of the pipeline queues. Since the length of each such queue is given by the designer, the number of extra registers for the automatically generated application does not vary with the number of elements sorted by the bubblesort network. This means that the larger the number of elements sorted, the closer the gap in the total register area between the automatically generated and equivalent manually-written versions.

For our second benchmark, we took a manually written version of a bitonic sort network, and we introduced pipeline registers in the same places as in our high-level specification used as source for the automatically generated bitonic sort circuit. After synthesis, the manually written bitonic sort network application yields a circuit that is about 8.5 percent larger, and about 69.59 percent slower, than our automatically generated implementation. The circuit obtained after introducing the pipeline registers into the manually written application is only about 8.2 percent faster than the original manually written application. We stress here that we did not specify the same logic for this application in our language, as the one that is coded in the manually written version; rather, we

designed and specified the bitonic sort network our own way, keeping the same number of numbers to be sorted, and the same width for the data paths.

In the case of our last, and biggest application, the RISC-style, linear pipelined processor, notice that the synthesized area is roughly the same, while the clock cycle of our processor is within 3 percent of the manually coded version.

6 Related Work

High-level synthesis approaches are based on a variety of languages such as concurrent languages, hardware description languages, software languages, data flow languages, and others. We can further distinguish different approaches within these categories. Concurrent languages consist of synchronous languages, protocol specification languages, and others like CSP, Occam, ADA, CCS, Unity, CRP, POLIS. Synchronous languages include Esterel [5], Lustre, Argos, Signal, RSML and Statecharts. Protocol specification languages include SDL, Lotos [26] and Estelle.

Software approaches are generally of one of three types: the library extension, the language extension, or the new language approach. The library extension approach includes systems like Scenic, work by Young et al. [27], SystemC, Lava and Hawk. The language extension approach includes Transmogrifier-C, Programmable Active Memory (PAM), Reactive-C, SpecCharts, ECL, SpecC, Data Parallel C. The new language approach includes the Olympus/Hercules system based on HardwareC, Superlog, V++, OpenJ, Rapide. There are also other systems like Compilogic, SpC, ADAS, RAW, and Fiper and Piper which use a specification language which is a subset of Standard Prolog. In industry, the hardware design languages that are heavily used are VHDL and Verilog.

Systems like Ptolemy, GRAPE, Warp at CMU, SPW from Cadence or COSSAP from Synopsys start from block diagram languages based on a dataflow semantics and are targeted to DSP design. Several specification and verification systems have taken an approach similar to ours, based on describing the behavior of a system by a state transition system [8, 14]. Closely related to our research, Hoe and Arvind develop a method for hardware description and synthesis based on an operation-centric approach. The hierarchical Production Based Specifications (PBS) model has similarities with our approach in that it enables temporal modularity when designing a circuit.

Traditionally, the correctness of a design was tested by simulation. Bryant's [7] introduction of reduced, ordered BDDs for circuit verification renewed interest in symbolic execution. Success in verification can be attributed to the development of formal methods like theorem provers and model checkers. Model checkers include EMC [9], Caesar [25], SMV [22], RuleBase [4], Spin [15], Murphi [10]. Theorem provers usually work using either the Boyer-Moore [6] system or the HOL [13] system. Other well-known theorem provers include LP [12], Nuprl [11], PVS [24], VERIFY [3], Esterel [5]. FoCs [1] (Formal Checkers) takes properties in CTL logic and automatically generates VHDL checkers from them, then integrates them into the simulation environment.

What is different about our approach is that we start from an initial specification in a high-level language, and generate a circuit implementation by applying

algorithms that are formally correct. This ensures that, given a correct specification, the resulting implementation is also correct; no need for verification. In this way, our approach is closer to the formal synthesis work by Manohar [17] and Martin [21].

7 Conclusions

This paper presents the formal framework for our novel approach for specifying and automatically implementing efficient systems such as digital circuits and network protocols. Our goal is to reduce the design time, effort, and expertise required to build correct and efficient systems and to eliminate the need for the designer to deal directly with complex issues like global synchronization and explicit concurrency. Our approach uses a compiler to automatically transform modular, asynchronous specifications into efficient, tightly-coupled, synchronous implementations. Our results show that our specifications are roughly an order of magnitude shorter than corresponding synchronous specifications that deal directly with global timing issues, and that our compiler is capable of producing implementations that are of comparable efficiency.

We provide a formal definition of our target class of systems and the algorithms that our compiler uses to implement these systems. We also sketch correctness proofs for two of our three primary compiler algorithms. This formal foundation gives the designer the guarantee that, if a correct compiler starts from a correct initial specification, the resulting implementation is also correct.

References

1. Y. Abarbanel, I. Beer, L. Glushovsky, S. Keidar, and Y. Wolfsthal. Focs: Automatic generation of simulation checkers from formal specifications. In *CAV*, pages 538–542, 2000.
2. F. Baader and T. Nipkow. *Term rewriting and all that.* Cambridge University Press, 1998.
3. H. G. Barrow. Verify: A program for proving correctness of digital hardware designs. *Artificial Intelligence, 24:437-91*, 1984.
4. I. Beer, S. Ben-David, C. Eisner, and A. Landver. Rulebase: An industry-oriented formal verification tool. In *Proceedings of the 9th Design Automation Conference (DAC)*, pages 655–660. Association for Computing Machinery, Inc., June 1996.
5. F. Boussinot and R. de Simone. The ESTEREL language. In *Proceedings of the IEEE*, pages 79(9):1293–1304, Sept. 1991.
6. R. S. Boyer and J. S. Moore. *Computational Logic.* Academic Press, New York, 1979.
7. R. E. Bryant. A methodology for hardware verification based on logic simulation. *Journal of the ACM (JACM)*, 38(2):299–328, 1991.
8. K. M. Chandy and J. Misra. *Parallel program design: a foundation.* Addison-Wesley, Reading, Mass., 1988.
9. E. M. Clarke and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs: Workshop, Yorktown Heights, NY*, May 1981.

10. D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, 1992.
11. R. C. et al. *Implementing Mathematics with the NuPRL Proof Development Environment*. Prentice-Hall, 1986.
12. S. J. Garland and J. V. Guttag. Inductive methods for reasoning about abstract data types. In *Proceedings of the 15th Symposium on Principles of Programming Languages*, pages 219–228, 1988.
13. M. Gordon. Hol: A proof generating system for higher-order logic. In *VLSI Specification, Verification and Synthesis*. Kluwer, 1987.
14. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, N.J., 1985.
15. G. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Englewood Cliffs, N.J., 1991.
16. P. Hudak et al. Report on the programming language Haskell: a non-strict, purely functional language (version 1.2). *SIGPLAN Notices*, 27(5), May 1992.
17. R. Manohar. A case for asynchronous computer architecture. In *Proceedings of the ISCA Workshop on Complexity-Effective Design*, June 2000.
18. M.-C. Marinescu. *Synthesis of Synchronous Pipelined Circuits from High-Level Modular Specifications*. PhD thesis, University of California, Santa Barbara, Dec. 2002.
19. M.-C. Marinescu and M. C. Rinard. High-level automatic pipelining for sequential circuits. In *Proceedings of the 14th International Symposium on System Synthesis*, Montreal, Canada, Oct. 2001.
20. M.-C. Marinescu and M. C. Rinard. High-level specification and efficient implementation of pipelined circuits. In *Proceedings of the ASP-DAC*, Yokohama, Japan, Jan. 2001.
21. A. J. Martin. Synthesis of asynchronous vlsi circuits. In *Formal Methods for CLSI Design*. North-Holland, 1990.
22. K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993.
23. R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.
24. S. Owre, J. Rushby, and N. Shankar. Pvs: A prototype verification system. In *11th International Conference on Automated Deduction (CADE)*, number 607 in Lecture Notes in Artificial Intelligence, pages 748–752, 1992.
25. J. Quielle and J. Sifakis. Specification and verification of concurrent systems in caesar. In *Proceedings of 5th ISP*, 1982.
26. K. Turner and M. van Sinderen. *Lotos specification style for OSI, The LOTOSPHRE Project*. KLUWER, London, UK, 1995.
27. J. Young, J. MacDonald, M. Shilman, A. Tabbara, P. Hilfinger, and A. Newton. Design and specification of embedded systems in Java using succesive, formal refinement. In *Proceedings of the 35th ACM/IEEE Design Automation Conference*, June 1998.