Protocol Selection, Implementation, and Analysis for Asynchronous Circuits

Thesis · August 2002

Source: CiteSeer

CITATIONS
3

READS
55

1 author:

Eric R. Peskin
New York University
14 PUBLICATIONS **213** CITATIONS

SEE PROFILE

PROTOCOL SELECTION, IMPLEMENTATION, AND ANALYSIS FOR ASYNCHRONOUS CIRCUITS

by

Eric Robert Peskin

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

August 2002

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a dissertation submitted by

Eric Robert Peskin

This dissertation has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

	Chair: Chris J. Myers
	Erik Brunvand
	Al Davis
	Ganesh Gopalakrishnan
	Christian Schlegel

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the dissertation of Eric Robert Peskin in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

Chris J. Myers
Chair, Supervisory Committee

Approved for the Major Department

Thomas C. Henderson
Chair/Dean

Approved for the Graduate Council

David S. Chapman
Dean of The Graduate School

ABSTRACT

This dissertation presents new methods for *handshaking expansion* of asynchronous circuits. Handshaking expansion includes *protocol selection*, *reshuffling*, and *state variable insertion*. The starting point is a channel-level specification of a design. The goal is a signal-level description of the given design that is correct, synthesizable, and efficient. This dissertation studies the impact of protocol selection and implementation on *deadlock* avoidance, *complete state coding* (CSC), CPU time required to compile a given example, and the quality of the circuit.

This dissertation treats reshuffling and state-variable insertion as special cases of concurrency reduction. Prior work in the field has also taken this approach. However, this dissertation extends this approach and applies it to specifications that contain quantitative timing assumptions.

The concurrency reduction algorithms have been implemented within a *computer aided design* (CAD) tool. Starting from a signal-level specification that contains the constraints of the desired protocol, these algorithms search the concurrency reduction design space, guided by an estimate of the performance of the final circuit. The CAD tool that this dissertation presents also contains a front end that, given a channel-level specification, produces the starting point for concurrency-reduction. This front end currently handles only pure synchronization channels, using one protocol.

Finding all possible ways to reduce concurrency of a specification is a fundamentally exponential problem. However, this dissertation presents techniques to dramatically prune the search space. This dissertation demonstrates that these techniques are capable of reducing the search space by several orders of magnitude compared to the theoretical upper bound – and by one order of magnitude beyond existing techniques – without significantly impacting the quality of the solutions.

To Yu-Ying and Jay

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENTS

I wish to thank my committee members Chris Myers, Erik Brunvand, Al Davis, Ganesh Gopalakrishnan, and Christian Schlegel for taking the time to read my dissertation so thoroughly, for providing so many comments, and for giving me the opportunity to address them. I especially want to thank my advisor, Chris Myers, for taking me on late in my graduate career, and providing the guidance and support necessary for me to navigate my change in dissertation topics. Finally, I wish to thank my family for all their love and support throughout this process.

CHAPTER 1

INTRODUCTION

Asynchronous circuits are circuits that have no global clock. Instead of using a global clock to synchronize events, asynchronous circuits use local *handshaking* between modules to synchronize when and where necessary. There has been a resurgence of interest in the design of asynchronous circuits due to their potential advantages in performance, robustness, modularity, and reduced power consumption.

One can specify or model asynchronous circuits at many levels. Typically, higher-level specifications involve *processes* that communicate over *channels*. Each channel connects multiple processes (usually two). This dissertation uses a *zero-slack communication* model. In this model, the number of communication actions that each process has completed on a given channel must equal the number of communication actions that the other process has completed on that channel. Thus, zero-slack communication provides a mechanism for synchronization between otherwise concurrent processes. Once a process *initiates* a communication action on a given channel, it must wait until all other processes that share that channel have also *initiated* the corresponding communication on that channel. Only then is the communication action *completed*, and all processes that share the channel are free to proceed [35, 36, 48, 49, 50].

Channel communications can also exchange data between two processes. Consider two processes, P and Q, connected by a channel c. Suppose that P has a local variable x, and Q has a local variable y. Further suppose that P executes a command such as `send`(c, x) while Q executes a matching `receive`(c, y). After the communication *completes*, the variable y in Q has the value that variable x had in

P before the communication.

It is convenient for designers to specify designs at the channel level. However, this level does not correspond directly to wires in a circuit. Thus, before circuit synthesis, it is necessary to translate the specification to the *signal level*. A signal-level specification does not contain high-level abstractions such as channels. Instead, a signal-level description specifies the actual *signals* (or wires) that exist in the circuit that implements the specification. Furthermore, the signal-level description specifies the causal relationships between transitions on these signals. Circuit synthesis can proceed from the signal-level description.

Given a channel-level specification of an asynchronous circuit, *handshaking expansion* must produce a semantically equivalent signal-level description of the circuit. This means that the signal-level description must preserve the synchronization and data-exchange patterns between processes of the channel-level specification. Where the channel-level specification contains serial chain of actions, these actions — including any implied synchronization and data-transfer actions — must occur in the order specified by that sequence. The definition of channel communication given in the above paragraphs as well as in [35, 36, 48, 49, 50] must be maintained. Section 1.1 describes a problem called *deadlock* in which a system is incapable of making further progress. Freedom from deadlock must also be preserved. If there is no deadlock inherent in the channel-level specification, then the signal-level specification must also not enter deadlock.

In the channel-level specification, processes communicate over channels using operations such as `send` and `receive`. In the signal-level description, signals on individual wires undergo transitions. The goal of this transformation is to produce a signal-level description of the given design that is correct, synthesizable, and efficient.

The first step of handshaking expansion is to pick a communication protocol for each channel. This decision allocates signals to the channel and determines the interpretation of these signals. There are many possible protocols for any given communication action [51, 63, 58]. Data validity can be handled through

bundled data or *data encoding* (for example, with a *dual-rail* code). Any given `send` or `receive` operation can be either *active* or *passive* [50]. Handshaking can be *two-phase* or *four-phase*. Even within four-phase handshaking, there are many possible conventions that define when data must be valid [63, 20].

There are also more exotic choices. *Single-track* [12] protocols use one wire for both *requests* and *acknowledgements*, letting each process drive the wire in turn. *Pulse-mode* [64, 65] protocols allow a single process to assume an *acknowledgement* after a certain delay. Furthermore, complex protocols can use a single handshake to exchange both query data and response data (for example, a memory address and the value at that location) between two processes. Other protocols can broadcast data from one process to multiple processes as in Akella and Gopalakrishnan's hopCP [1, 32, 33, 2]. Prosser et al. [63] evaluate and study tradeoffs involved in sequencing communication actions.

Any protocol can yield a correct and synthesizable description (assuming correct completion of the remaining steps of handshaking expansion). However, the choice of protocol can affect the resources necessary to find a synthesizable description. Furthermore, the choice of protocol can have a significant impact on the efficiency of the resulting circuit. These effects are highly dependent on the given example and even on the context of the particular channel communication.

Given a choice of protocol, one must decide on the order in which these signals undergo transitions. *Reshuffling* [51, 58, 20] is the process of changing this order. Just one protocol can have many possible *reshufflings*. For example, Martin et al. [53] define a particular *delay-insensitive*, dual-rail protocol. The authors then use three different reshufflings of the same protocol. They call these *half-buffer*, *precharged half-buffer*, and *precharged full-buffer*. Reshuffling can affect whether the signal-level description is correct and synthesizable. It also affects the efficiency of the final circuit. Again, these effects are highly dependent on the context. This dissertation presents algorithms to automatically search the solution space of possible reshufflings, guided by an estimate of the performance of the final circuit.

In the style of circuit design used here, the first attempt at circuit synthesis

uses signals allocated by protocol selection (as well as any other outputs dictated by the specification) as the only state bits of the implementation. However, it may be that these bits are not sufficient to distinguish the different states of circuit. Even if they are technically sufficient, relying on these state bits alone may be suboptimal, because it may result in logic that too complex. Hence, it in some cases, it may be necessary or desirable to insert additional state variables [51, 58].

This dissertation treats both reshuffling and state-variable insertion as special cases of concurrency reduction. It presents algorithms for searching the concurrency reduction design space to find efficient circuits.

These algorithms handle specifications that contain explicit, quantitative timing assumptions. This allows optimizations at several levels of the design process. Section 6.1.1 shows that timing information can reduce the size of the search space for concurrency reduction itself. This is because events that would otherwise be concurrent may be ordered under the given timing assumptions. Furthermore, some handshaking expansions that would not be synthesizable without timing assumptions are synthesizable under appropriate timing assumptions. Finally, prior work by Myers et al. [60, 59, 7, 58] has shown that quantitative timing information can speed up the synthesis process and result in better circuits.

1.1 Related Work

Hoare introduced a language called *Communicating Sequential Processes* (CSP) [35, 36] that inspired the model of communication used in this dissertation. Today, there exist several languages and dialects derived from CSP that are used for software programming and hardware design. Martin's *Communicating Hardware Processes* (CHP) [48, 49, 50, 52] is the first to be specialized for the design of asynchronous circuits and hardware systems. Akella and Gopalakrishnan's hopCP language [1, 32, 33, 2], are based in part on CSP. Brunvand and Sproull [18, 16, 17] developed algorithms to translate programs written in *Occam*, a parallel programming language based on CSP, into asynchronous circuits of the *micropipeline* [66] style. Van Berkel et al. [13, 11] developed the VLSI programming language

Tangram, which is also based on CSP and Occam.

As Hoare [35] points out, a group of processes using this communication model may enter a situation in which they are all trying to communicate, but none of the pending communication actions correspond. At this point, no further progress can be made. This situation is called *deadlock*. Several researchers studied essentially the same deadlock issue even earlier. Friedman and Menon studied *blocking conditions* in systems of asynchronously operating modules [27]. Bruno and Altman studied systems composed of five basic control modules and showed the conditions under which they would "hang up" [15]. Jump and Thiagarajan studied deadlock in *MG-control systems* (MGCS) [39]. Deadlock was also studied by Genrich and Lautenbach [29, 30, 31] who proved the presence or absence of deadlock for specific annotated marked graphs.

Martin describes handshaking expansion in [51, 52]. Burns [19] discusses how to automate Martin's techniques and implemented a subset of these techniques in a working automatic compiler. However, Burns' method does not include any explicit timing information. In contrast, the method in this dissertation supports explicit timing information.

Lines [45] discusses various implementations of buffers with and without processing logic. He enumerates (by hand) the possible reshufflings of a buffer without logic under certain constraints and evaluates their various merits. Then he shows how to add logic to these buffers. He demonstrates a systematic way to introduce data into a given reshuffling of a data-less buffer. His techniques provided inspiration for some of the heuristics that Chapter 6 presents. However, Lines applied these techniques only by hand. This dissertation presents an automated approach to searching the reshuffling solution space.

Manohar [46] describes a strategy for transforming reshuffled handshaking expansions back into higher-level CHP notation. He uses this notation to analyze correctness and performance of the reshuffled handshaking expansions. In particular, he uses *Two-Phase CHP*, which is essentially the shorthand that Chapter 5 uses. The high-level analysis enabled by the CHP leads to conclusions that also

inspire the heuristics that Chapter 6 presents. Again, the main difference is that this dissertation presents automatic techniques.

Several researchers have used *syntax-directed translation* to translate language-based specifications into asynchronous circuits. Examples include Brunvand's work on translating Occam into *macromodular micropipelines* [18, 16, 17], and also the Tangram work at Philips [13]. The CAD tool `Balsa` from Bardsley and Edwards [6] takes a very similar approach to that of `Tangram`. These techniques map program structures directly into a library of circuit structures. Similarly, Kim et al. [41] map program structures into a library of small *signal transition graphs* (STGs). These syntax-directed techniques allow the user to start from a channel-level specification as does the CAD tool that this dissertation presents. Furthermore, syntax-directed translation achieves a high degree of modular design, which is a goal of the work of this dissertation. These techniques do not include an explicit handshaking-expansion step during compilation. Instead, the creator of the library makes many decisions — including those of handshaking expansion — in advance for each library element separately. In contrast, this dissertation presents algorithms that search the reshuffling and state-variable-insertion design space to tailor the solution to the specification.

It is also worth mentioning that [41] discusses an interleaving strategy similar to the one that Chapter 5 advocates. The main similarity is that they segregate the *working phase* from the *idling phase* of the communication in question. Again, the main difference is that they apply this strategy to each library element separately before compilation.

Akella and Gopalakrishnan designed a system called `SHILPA` [1, 32, 33, 2], which accepts a high-level algorithmic description written in hopCP and produces a circuit. Like the specification language that Section 2.1 presents, hopCP allows both channels and levels in the same language. Gopalakrishnan and Akella [32] discuss how to transform a hopCP specification to obtain software pipelining. One could also transform the specifications of Section 2.1 in this way. However, neither `SHILPA` nor the tool that this dissertation presents automate such transformations.

Like the tool that this dissertation presents, SHILPA translates its language based specification into a series of graph-based intermediate forms and analyzes these graphs before finally deriving a circuit from these graphs. This is in contrast to certain syntax-directed approaches (for example, Brunvand's) that derive a circuit directly from the language-based specification. Also, the modular, syntax-directed translation that SHILPA employs operates on a finer grain than that in Brunvand's work. The initial graph that SHILPA derives has channel communications as primitive operations in the graph. This is similar to the graph-based form presented in Chapter 3. Like the tool that this dissertation presents, SHILPA then performs handshaking expansion to derive a signal-level graph.

In several respects, SHILPA is a more complete system than the tool that this dissertation presents. SHILPA implements both control and data path, while the tool that this dissertation presents currently implements only control. Both hopCP and the language of Section 2.1 allow variables to be shared between processes. In hopCP such variables are called *asynchronous ports*. However, SHILPA has additional features to check whether the use of such shared variables is safe, beyond what the tool that this dissertation presents provides. At its core, this check is based on checking whether two events are guaranteed to be serial. Section 6.1.1 does present a way to check whether two events are guaranteed to be serial, however it applies this to a different purpose, namely avoiding redundant concurrency reduction attempts. hopCP supports broadcast and multicast communication, whereas currently the tool that this dissertation presents supports only point-to-point channels. SHILPA can synthesize circuits that require arbiters for correct operation. The synthesis engine to which the tool that this dissertation presents interfaces does not support arbiters. Instead it simply identifies and rejects specifications that would require them.

SHILPA targets two-phase communication, whereas the front end to the tool that this dissertation presents currently targets solely four-phase communication. This is significant, because four-phase communication brings with it a much larger search space for the reshuffling problem. Much of the contribution of this dissertation con-

cerns dealing with the four-phase reshuffling problem. hopCP has no function that allows a process to check whether a communication is pending on a channel without committing to complete that communication before performing other actions. In contrast, the language of Section 2.1 adopts Martin's *probe* function [48] for this purpose, and the tool that this dissertation presents supports it throughout. Finally, hopCP and `SHILPA` do not support specifications that make explicit, quantitative timing assumptions.

Bachman designed a *Computer Aided Design* (CAD) tool called `Mercury` [4, 5], which does scheduling and resource allocation for asynchronous circuits. It generates a structural view of the *data path* and a behavioral view of *control*. Currently the tool that this dissertation presents does not implement the data path. One approach to a complete system would be to use the techniques that this dissertation presents for control and `Mercury` for data path.

Jacobson et al. [38] present a tool for asynchronous system-level design called `ACK`, which takes design from high level to layout. It produces hybrid circuits that contain combinations of *macromodules*, AFSMs, and *microengines*. It interactively explores various design tradeoffs with tight feedback between the user and the tools. `ACK` supports both two-phase and four-phase protocols, allowing the user to choose between them. `ACK` implements both control and data path. However, `ACK` does not support specifications that contain explicit, quantitative timing assumptions.

Some synthesis CAD tools can take a signal-level specification with complete state coding violations, and attempt to solve these violations as in [68, 70, 69, 43, 34, 22, 42]. However, such approaches are often prohibitively expensive [14], because the space of solutions they must search is quite large. Without the channel-level specification, there is little information available to guide this search.

Vanbekbergen et al. [67] and Ykman-Couvreur et al. [71] developed a method to reduce concurrency on *signal transition graphs* (STGs). Thus, this method operates at essentially the same level of abstraction as the concurrency reduction method that Chapter 5 presents. However, the method of [67, 71] is restricted to *marked graphs*. The method of Chapter 5 supports a much broader class of

specifications. Ykman-Couvreur et al. [72] and Lin et al. [44] also developed a method to reduce concurrency in *state graphs*. The authors combined this method with their own state-variable insertion method. The authors showed that concurrency reduction can lead to designs that are smaller and — surprisingly — faster than those in which all concurrency is retained. However, their method does not handle specifications that make explicit, quantitative timing assumptions. In contrast, this dissertation addresses concurrency reduction in the context of *timed circuit design* [59, 60, 58], in which the specification can make explicit, quantitative timing assumptions. Supporting such timing assumptions requires performing at least part of the concurrency reduction at higher levels of abstraction than that of the state graph. This leads to a different set of problems and solutions.

Cortadella et al. [23] present a technique that treats reshuffling as a special case of concurrency reduction. The CAD tool `Petrify` implements this technique. This technique starts from a specification that includes channel communications in a graphical representation. At the signal-level, `Petrify` supports an extension to Petri nets that allows either standard transitions or toggle transitions that toggle the value of a signal. The `Petrify` tool supports both four-phase and two-phase expansion. In contrast, the tool that this dissertation presents currently has a front end that targets only four-phase expansion, although the concurrency reduction engine that this dissertation presents is applicable in either case.

For four-phase communication, Cortadella et al. [23] assume that return-to-zero actions are not significant for data-integrity, and hence may be reshuffled without restriction. Therefore, the initial expansion in [23] inserts the return-to-zero events such that they are maximally concurrent with events from other channel communications. The technique determines the reachable state space for this maximally concurrent specification. Then the technique searches for ways to reduce the concurrency of the state space to produce a synthesizable solution. This dissertation takes a very similar approach to the reshuffling problem. However, there are important differences. The graphical representation that this dissertation uses [8, 7, 9] can refer to the current levels of the signals in the systems and

not just to transitions on those signals. This is important, because it makes it possible to partition the representation into a separate component for each process in the specification. This is significant in the domain of channel communications, because one of the main reasons to use a channel abstraction is to achieve modular design. At the signal level, `Petrify` does support similar specifications with levels. However, it is unclear how to use this feature at the channel level using `Petrify`. Furthermore, this dissertation introduces handshaking expansion techniques that support *timed circuit design* [59, 60, 58], in which the specification can make explicit, quantitative timing assumptions. Supporting such timing assumptions requires performing at least part of the concurrency reduction at higher levels of abstraction than that of the state space. This leads to a different set of problems and solutions. Like [23], the front end to the tool that this dissertation presents currently makes the assumption that return-to-zero events are never important for data integrity. However, this dissertation also presents a method for specifying which relationships between events are important for data integrity that is more general than this assumption.

1.2 Contributions

This dissertation presents new algorithms to search the handshaking expansion design space. These algorithms achieve *complete state coding* (CSC) and avoid *deadlock*. Both of these goals are necessary to synthesize a circuit. The usual approaches to this problem include reshuffling and state-variable insertion. This dissertation treats reshuffling and state-variable insertion as special cases of concurrency reduction. This approach is similar to that taken by the CAD tool `Petrify` [23], but this dissertation applies it to specifications that contain quantitative timing assumptions.

Supporting quantitative timing assumptions allows optimizations at several levels of the design process. Section 6.1.1 and Section 7.2.3 show that timing information can reduce the size of the search space for concurrency reduction itself. This is because events that would otherwise be concurrent may be ordered under the

given timing assumptions. Exactly determining all timed-concurrency information requires timed-state-space exploration. This is an expensive operation. Therefore the heuristic of Section 6.1.1 computes this information just once on the initial most-concurrent starting point that captures the constraints of the protocol, and uses this information as an approximation to the timed-concurrency information for the reduced structures. This approximation allows the cost of the timed-state-space exploration to be amortized over many pruning decisions. Section 7.2.3 shows that this does reduce total run time, and for the examples considered, the approximation does not impair the ability of the search to find the optimal solution. Furthermore, some handshaking expansions that would not be synthesizable without timing assumptions are synthesizable under appropriate timing assumptions. Finally, prior work by Myers et al. [60, 59, 7, 58] has shown that quantitative timing information can speed up the synthesis process and result in better circuits. By bringing together the study of concurrency reduction and the study of timed circuits, this dissertation allows each field to benefit from the other.

Supporting such timing assumptions requires performing at least part of the concurrency reduction at higher levels of abstraction than that of the state graph. Working strictly at the state-graph level alone, it is unclear whether modifications to the state graph still meet the timing specifications stated in the higher-level specifications. The need to perform concurrency reduction at higher levels of abstraction leads to different problems from those faced by concurrency reduction methods that operate at the state-graph level alone. This dissertation contributes solutions to these different problems. It is true that Vanbekbergen et al. [67] and Ykman-Couvreur et al. [71] developed a method to reduce concurrency on STGs. STGs have essentially the same level of abstraction as the structures used in this dissertation. However, the method of [67, 71] is restricted to marked graphs. This dissertation contributes methods that support a much broader class of specifications.

The tool that this dissertation presents operates within a framework that supports hierarchy at multiple levels of the design process. The specification that

Section 2.1 introduces is an extension of the *Very High-Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL)*, which contains ample support for hierarchical design. Furthermore, the extensions that Section 2.1 presents support channel communications and hence the Communicating Sequential Processes [35, 36] paradigm, which encourages modular design. This specification level, as well as the graphical, intermediate forms that this dissertation uses (Belluomini's *timed event/level (TEL) structures* [8, 7, 9]) support specifications that depend on signal levels. This allows the graphical intermediate forms to retain the modularity present in the specification. Each process results in a distinct connected component in the graphical representation. The abstraction techniques of Zheng et al. [75] and the modular synthesis techniques of Mercer et al. [57, 56] allow the synthesis engine to exploit this hierarchy information.

To test the utility of the concurrency-reduction algorithms that this dissertation presents, we have implemented them as extensions to the CAD tool *Automatic Timed Asynchronous Circuit Synthesis* (`ATACS`) [59]. Given a signal-level specification that contains the constraints of the given protocol, these algorithms search the concurrency-reduction design space to find solutions that are correct, synthesizable, and efficient. These algorithms model concurrency reduction as adding rules to TEL structures. Thus, the concurrency reduction method that this dissertation presents is applicable to any combination of example and protocol that can be expressed as a signal-level TEL structure.

Finding all possible ways to reduce concurrency in a specification is a fundamentally exponential problem. However, this dissertation presents techniques to dramatically prune the search space. The techniques of Chapter 5 can reduce the number of possibilities to be considered by many orders of magnitude compared to the theoretical upper bound. Chapter 6 presents heuristics that may not find all solutions, but reduce the size of the search space even further. For the examples considered, Chapter 7 finds that these heuristics are capable of reducing the search space by an additional two orders of magnitude beyond the techniques of Chapter 5 — and by one order of magnitude beyond existing techniques — without

significantly impacting the quality of the solutions found.

The tool that this dissertation presents also includes a front end that, given a channel-level specification, produces the starting point for concurrency-reduction. This front end currently handles only pure synchronization channels, using one, fixed, *four-phase* protocol with *narrow sequencing*. However, the front end is separate from the concurrency-reduction engine, which is more general. The user who wants to try other protocols can bypass the front end entirely, by writing the signal-level input to the concurrency-reduction engine. This input is signal-level, and it must contain the constraints of the desired protocol or protocols. However, even in this case, the concurrency-reduction engine frees the user from the concerns of reshuffling and state-variable insertion. `ATACS` supports a wide variety of signal-level specification formats, any of which can be used to write the signal-level input to the concurrency reduction engine. These include the handshake-level VHDL that Section 2.2 presents, which can be simulated using commercial VHDL simulators. It also includes a CSP-like language that supports timed handshaking expansions. One can also provide the signal-level specification directly as a TEL structure. Chapter 4 presents extensive examples of this. Provided that one can obtain the initial, most-concurrent, signal-level specification that contains all the constraints of the channel-level specification and also the desired protocol, then the concurrency reduction method and the pruning techniques that this dissertation presents are applicable.

Figure 1.1 outlines the overall design flow for timed circuit design in `ATACS`. The rectangles in this figure represent processing steps. The rounded rectangles represent data. The arrows represent the flow of information. Except where noted, Chapter 2 describes each step and data structure.

The user provides a specification in a hardware description language. `ATACS` currently accepts several languages for specification. The examples in this dissertation use VHDL.

The compilation step [74] converts the language-based specification into an internal, graphical representation called a *timed event/level (TEL) structure* [8, 7, 9]

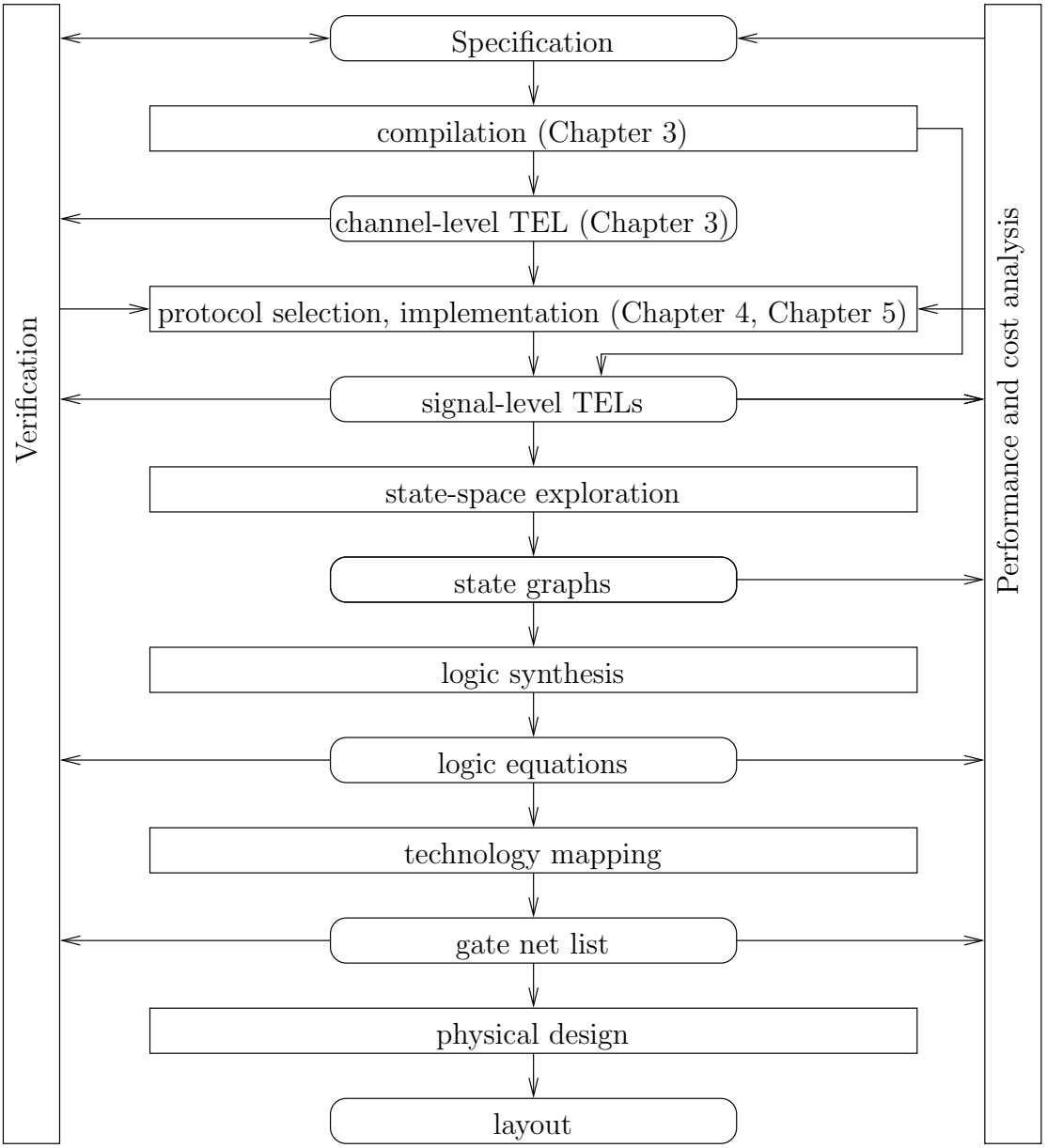

Figure 1.1. Timed circuit design in `ATACS`.

(see Chapter 2). This dissertation presents extensions to the compilation step to support channel communications. Chapter 3 introduces the channel-level TEL structure, which models the semantics of channel communications.

Given a channel-level TEL structure, the protocol selection, and implementation step produces corresponding signal-level TEL structures. This step is the focus of this dissertation. It includes the steps of selecting a protocol, initial expansion based on the protocol, reshuffling, and state-variable insertion. If the specification contains no channel communications, this step is bypassed. In this case, the compiler produces a signal-level TEL structure directly.

Given a signal-level TEL structure, state-space exploration applies timing analysis methods to find the set of reachable timed states of the system. The result is a *state graph* in which each node represents a state, and each edge represents a possible transition between states.

The logic synthesis step derives logic equations that implement the state graph. This implementation is largely technology independent. There is no guarantee that the logic gates implied by the equations are feasible.

The technology mapping step maps a set of logic equations into a given gate library. The result is a gate net list that implements the logic equations using the given technology. Finally, the physical design net list transforms the topology of the gate net list into the geometry of layout for an integrated circuit.

Verification compares various intermediate forms (as well as the final implementation) to the original specification. Verification also checks that each of these intermediate forms has certain desirable properties, such as freedom from deadlock. For example, if the channel-level TEL structure suffers from deadlock, this indicates a fundamental problem in the channel-level specification. The user must use this information to modify the specification. Signal-level TEL structures that fail verification must be rejected. The protocol implementation step responds by finding other signal-level TEL structures. Verification makes sure that the logic equations indeed implement the behavior of the specification. Furthermore, the gate net list must conform to the timing constraints imposed by the specification.

Performance analysis guides the search of protocol implementation. This is currently based on Mercer's stochastic cycle period analysis [55]. This method provides an estimate of the average cycle period of the design, based on the current TEL structure. Note that the tool that this dissertation presents is modular in this respect as well. Any other cost function could be used instead of the average cycle period.

1.3 Overview

Chapter 2 gives an overview of the background material necessary to understand this dissertation. In particular, this chapter describes the design flow of the CAD tool that this dissertation presents. The chapter defines the input to the tool, the intermediate forms that the tool uses, the output from the tool, and methods for analyzing the output.

Chapter 3 discusses the compilation of the specification that uses channels into a graphical representation that models the channel communication in the specification. This chapter first addresses the semantic issues of channel-based specifications that the compiler must check and record. Then the chapter defines a graphical model of channel communications and describes how to derive this model from the specification.

Chapter 4 presents the initial, most-concurrent signal-level specification that forms the starting point for the concurrency reduction techniques of Chapter 5 and Chapter 6. This starting point is the most concurrent signal level specification that still captures all of the constraints of the channel-level specification and of the desired protocol. This chapter presents an overview of the different types of constraints that a protocol imposes on the signal-level implementation. The chapter presents a model for expressing such constraints. The tool that this dissertation presents does include a simple automatic expander that, given the graphical channel-level specification that is the result of Chapter 3, produces the starting point for concurrency reduction in format described in Chapter 4. This front end currently targets pure synchronization channels, using one four-phase protocol. For

other protocols, the user must provide the starting point for concurrency reduction as a signal-level specification. Section 4.3 presents an extensive overview of how do this. It provides a survey of many protocols that have been used in the literature, showing how the constraints of each can be expressed by the model. In particular, it shows how a target data path imposes constraints on control. The reader who is concerned only with what is currently automated in the tool can skip Section 4.3. However, Section 4.3 is still useful to the user of the current tool who wants to bypass the front end to the tool and specify the starting point for concurrency reduction directly. It is also useful to future developers who want to extend the front end to target more of these protocols. The chapter concludes by showing how the could form the basis for an extendable library of protocols for CAD tools.

Chapter 5 presents techniques for reducing the concurrency in a design. It treats the process of deciding the order in which circuit signals undergo transitions as one special case of concurrency reduction. This is also the view taken by [23]. In particular, this chapter discusses how to search the space of all such orders that satisfy the constraints of Chapter 4. The chapter also treats state-variable insertion as a special case of concurrency reduction. The chapter discusses the size of the theoretical search space. It then presents techniques to reduce the size of this search space.

The techniques of Chapter 5 may not be sufficient to reduce the search space to a manageable size. Therefore, Chapter 6 presents heuristics that further reduce the search space. These heuristics are known to be inexact in the sense that sometimes prune away solutions. In this case, *exact* means finding the exactly optimal solution or finding the entire solution space. If a pruning technique is *inexact* it just means that it might prune away a solution that would have been synthesizable. This issue is distinct from *correctness*. A solution is correct if it still meets the constraints of the given channel-level specification and of the target protocol. The heuristics of Chapter 6 are useful, because in practice they tend to leave near-optimal solutions in the search space, and the remove many possibilities that are not solutions from the search space. This chapter presents both *static heuristics* that reduce the number

of levels in the search tree before the search even begins, and *dynamic heuristics* that make pruning decisions at each interior node of the search tree.

Chapter 7 presents and analyzes the results of several case studies. This chapter tests the effectiveness of the pruning techniques in Chapter 5 and Chapter 6 by comparing the results for various combinations of these pruning techniques. This chapter also compares the tool that this dissertation presents to `Petrify`. This section attempts to quantify the similarities and differences by comparing the two tools on a set of examples. This chapter emphasizes the CPU time required to process a given example and the average cycle period of the solutions found.

Chapter 8 presents conclusions and ideas for future research.

CHAPTER 2

BACKGROUND AND DESIGN FLOW

This chapter gives an overview of the background material necessary to understand this dissertation. In particular, this chapter describes the design flow of the CAD tool that this dissertation presents. Section 2.1 defines the channel-level specification that is input to the tool. Section 2.2 defines a signal-level specification. The user can use the this signal-level specification as input to bypass the channel-level portion of the tool. Furthermore, in certain cases, the tool can translate a signal-level representation that it has derived back into the format of Section 2.2. Section 2.3 defines TEL structures [8, 7, 9], the graphical representation used within the tool. Section 2.4 defines the state graphs that `ATACS` derives from the TEL structures. Section 2.5 defines the production rules that `ATACS` derives from the state graphs. Finally, Section 2.6 presents the methods used to analyze the results of this dissertation. Figure 2.1 outlines the portion of the `ATACS` design flow that is most relevant to this dissertation.

2.1 Channel-Level Specification

This section describes the particular form of channel-level specification that serves as the input to the tool that this dissertation presents. We have developed a *channel package* [58] for VHDL. For purposes of simulation, this operates as a standard VHDL package, using only standard VHDL. Thus, one can simulate designs at the channel level using commercially-available VHDL simulators. For purposes of synthesis, `ATACS` recognizes the operations provided by the channel package as if they were VHDL primitives. In this sense, `ATACS` recognizes an extended subset of VHDL.

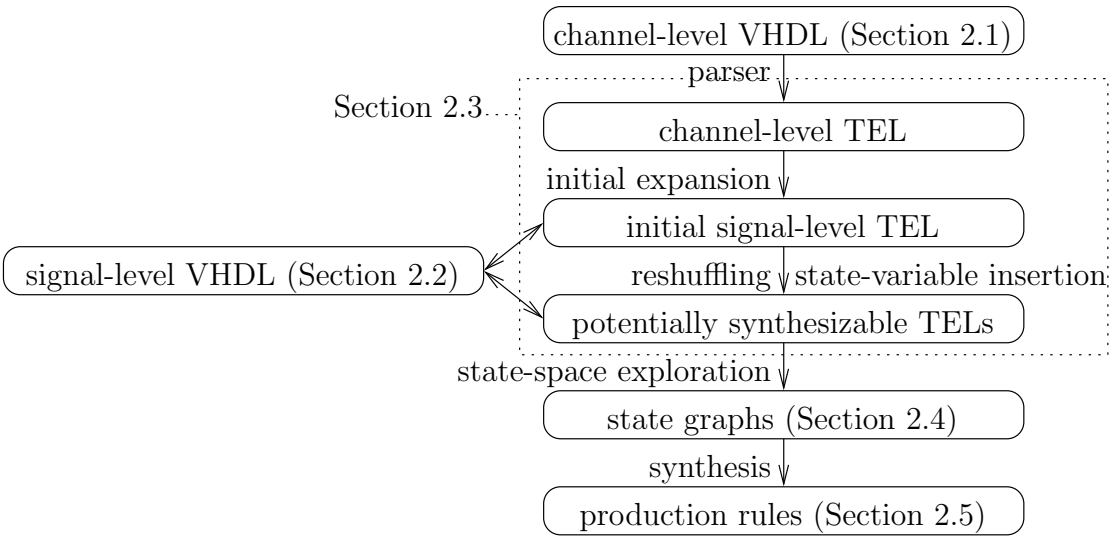

Figure 2.1. Design flow.

The channel package defines a new data type called `channel`. Signals and ports of this data type can be initialized with the function `init_channel`, which takes several optional parameters that determine properties of the channel. The channel package also defines `send`, `receive`, and `probe` operations on the `channel` data type. Consider the VHDL below.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use work.nondeterminism.all;
use work.channel.all;
entity simple is
end simple;
architecture behavior of simple is
  signal x, y :  std_logic_vector( 2 downto 0 ) := "000";
  signal C : channel := init_channel(
    sender => timing(rise_min => 2, rise_max => 4,
                     fall_min => 1, fall_max => 3),
    receiver => timing(rise_min => 6, rise_max => 10,
                       fall_min => 5, fall_max => 7));
begin
  P : process
  begin
    send(C, x);
    --@synthesis_off
    x <= x + 1;
    --@synthesis_on
    wait for delay(1, 2);
```

```
    end process P;
    Q : process
    begin
      receive(C, y);
    end process Q;
  end behavior;
```

Processes P and Q operate concurrently. However, they synchronize when they communicate over channel C. If P reaches its `send` first, it waits for Q to reach its `receive`. If Q reaches its `receive` first, it waits for P to reach its `send`. After the communication *completes*, the variable y in Q has the value that variable x had in P before the communication. The value of variable x of P is unchanged by the communication.

This data transfer is currently implemented only in simulation. The synthesis tool that this dissertation presents implements only the control (synchronization) portion of channel communications and not the data path. However, these operations allow the user to write a VHDL description that mixes the two, and simulate the whole description, but then let the tool that this dissertation presents extract the part it can handle and synthesize that part. In theory, another tool could extract the data path, implement that, and then the two pieces could work together. Alternatively, the data path could be implemented by hand. Section 4.3 shows how the tool could extended to handle data.

The above `init_channel` command specifies the timing of `send` and `receive` operations on the channel C. In particular, it specifies that the sender's handshake signals should have a rise time of at least two time units and at most four time units and a fall time of at least one time unit and at most three time units. Similarly, the receiver's handshake signals should have a rise time of at least six time units and at most ten time units and a fall time of at least five time units and at most seven time units. Chapter 3 specifies other optional parameters that the `init_channel` function can take.

The `delay` function is also used to make quantitative timing assumptions. It is defined in the nondeterminism package. `delay`(l, u) returns a randomly selected time between l time units and u time units. Note that for purposes of simulation,

we pick a default time unit, which is currently set to one nanosecond. For synthesis purposes, what matters is the ratios between the various timing assumptions in the specification.

This example uses two ATACS compiler directives. --@synthesis_off tells the compiler to ignore any code until the next occurrence of --@synthesis_on. The directives simply appear as comments to commercial simulators. Thus, during simulation, the directives themselves (but not the code between them) are ignored. Thus, the user encloses code that is not currently synthesizable by ATACS between these directives. Typically, this is code that specifies the data path. Currently, the tool that this dissertation presents can implement only control and not data path. However, these directives allow the user to write a VHDL description that mixes the two, and simulate the whole description, but then let ATACS extract the part it can handle and synthesize that part. In theory, another tool could extract the data path, implement that, and then the two pieces could work together. Alternatively, the data path could be implemented by hand. It is also worth noting that ATACS is capable of handling certain, very small pieces of mixed control and data-path, as described in Section 4.3.2. However, channel-level VHDL front end does not currently target such structures. If future versions do, then --@synthesis_off and --@synthesis_on may be needed in fewer places.

The channel package also provides a probe function [48]. The probe function returns a boolean value that indicates whether a communication is pending on a channel. Any boolean expression can use the probe function. Another useful procedure found in other examples is the await procedure, which waits until the corresponding probe is *true*, that is, until there is a communication pending on the given channel. The await_any procedure waits until there is a pending communication on any of the given channels. For example, await_any(X, Y) waits until the condition probe$(X) \vee$ probe(Y) is satisfied. The await_all procedure waits until there is a communication pending on each of the given channels. For example, await_all(X, Y) waits until the condition probe$(X) \wedge$ probe(Y) is satisfied.

An example using the await_any function is shown in the VHDL code below.

This example represents a bit slice (the *slice* process) of a very simple shift register, and the environment (the *produce* and *consume* processes) of that bit slice. The processes communicate over three channels. The L is used to load a new value into the bit slice, overwriting the current value. The S_{in} channel is used to shift values serially into the shift register. The S_{out} channel is used to shift values serially out of the shift register. The VHDL code below divides places each process in its own component. Each component uses the `channel` data type for its ports. Although data flow in only one direction on each channel, control information must flow in both directions on each channel. Hence, each channel port must be declared in the `inout` mode.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use work.nondeterminism.all;
use work.channel.all;
entity produce is
  port(L, S_in :  inout channel := init_channel(sender => timing(4, 7)));
end produce;
architecture behavior of produce is
  signal x :  std_logic := '0';
begin
  produce :  process
    variable z :  integer;
  begin
    z := selection(2);
    if (z = 1) then
      send(L, x);  -- Must complete data transfer before continuing.
    else
      send(S_in, x);  -- Must complete data transfer before continuing.
    end if;
    x <= not(x);
    wait for delay(3, 5);
  end process produce;
end behavior;
--------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use work.channel.all;
entity slice is
  port(L : inout channel := init_channel(receiver => timing(1, 2));
       S_in :  inout channel := init_channel(receiver => timing(1, 2));
       S_Out :  inout channel := init_channel(sender => timing(2, 3)));
end slice;
architecture behavior of slice is
  signal y :  std_logic;
```

```
begin
  slice : process
  begin
    await_any(L, S_in);  --Wait for pending communication on L or S_in.
    -- produce ensures that L and S_in are never both pending at once.
    -- At this point, exactly one of L,S_in has a pending communication.
    if (probe(S_in)) then  -- It is the S_in channel that is pending.
      send(S_out, y);
      receive(S_in, y);
    else  -- It is the L channel that is pending.
      receive(L, y);
    end if;
  end process slice;
end behavior;
--------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use work.nondeterminism.all;
use work.channel.all;
entity consume is
  port(S_out : inout channel := init_channel(receiver => timing(3, 7)));
end consume;
architecture behavior of consume is
  signal z : std_logic;
begin
  consume : process
  begin
    receive(S_out, z);
  end process consume;
end behavior;
--------------------------------------------------------------------------
use work.channel.all;
entity shift is
end shift;
architecture new_structure of shift is
  component produce
    port(L, S_in : inout channel);
  end component;
  component slice
    port(L, S_in, S_out : inout channel);
  end component;
  component consume
    port(S_out : inout channel);
  end component;
  signal L, S_in, S_out : channel := init_channel;
begin
  source : produce port map (L => L, S_in => S_in);
  UUT : slice port map (L => L, S_in => S_in, S_out => S_out);
  sink : consume port map (S_out => S_out);
end new_structure;
```

The *produce* process continually sends data to either the L or S_{in} channel. It chooses which channel to use for each datum at random, using the `selection` function from the *nondeterminism package* [58]. `selection`(n) returns a random integer such that $1 \leq$ `selection`$(n) \leq n$. Note that once the *produce* process has chosen which channel to use, it must complete the `send` operation, thus synchronizing and exchanging data with the *slice* process, before it can make any further progress. Thus, it is never the case that both the L and S_{in} channels have communications pending at the same time. The *slice* process probes the L and S_{in} channels to see which channel, if any, has a datum. If the L channel has a datum, the *slice* process simply receives it, overwriting its local variable y. If the S_{in} channel has a datum, the *slice* process first sends the current value of y out on the S_{out} channel, and then receives the new value of y from the S_{in} channel, thus completing the communication on S_{in} The *consume* process simply receives data from the S_{out} channel. Finally, the structural VHDL code connects the three processes.

The *slice* process uses the `await_any` function to wait until there is an incoming datum on either channel L or channel S_{in}. Because of the behavior of the *produce* process, there cannot be pending communications on both the L and S_{in} channel. Therefore, in this case, once the `await_any` function completes, there is a pending communication on exactly one of the channels. The *slice* process then uses the `probe` function to determine which channel has the datum, before it enters the code to handle that channel. If it tried to communicate with the other channel first, deadlock would result. Thus, the `probe` command is necessary for this application. Also, when the S_{in} channel has a pending communication, the *slice* process first sends the old value of y out on the S_{out} channel before it executes the `receive` operation on the S_{in} channel. If it did not do this, the current value of the y variable would be lost. This ability to find out that a communication is pending on some channel, but then do other work before completing the communication on that channel is an important feature of the `probe` function [48].

2.2 Signal-Level Specification

We have also developed a *handshake package* [58] for VHDL. This allows the user to produce a signal-level specification directly in VHDL. Furthermore, if the user instead provides a channel-level specification, `ATACS` can apply the methods that this dissertation presents to derive signal-level solutions to this channel-level specification. In some cases, `ATACS` can then express these solutions in VHDL using the handshake package.

For purposes of simulation, the handshake package operates as a standard VHDL package, using only standard VHDL. Thus, just as the channel package of Section 2.1 enables channel-level simulation using commercially-available VHDL simulators, the handshake package of this section supports signal-level simulation of handshaking protocols using commercially-available VHDL simulators. For purposes of synthesis, `ATACS` recognizes the operations provided by the handshake package as if they were VHDL primitives. In this sense, `ATACS` recognizes an extended subset of VHDL.

VHDL already supports signals, so no new data types are necessary. However, the handshake package defines new operations on signals: `guard` and `assign`. The `guard`(s, v) operation waits for signal s to reach the value v. If it is already the case that $s = v$, then `guard`(s, v) exits immediately. In this special case, its behavior is different from that of a simple VHDL `wait` statement. `guard_and`(s_1, v_1, s_2, v_2) waits until s_1 has value v_1 and s_2 has value v_2. Similarly, `guard_or`(s_1, v_1, s_2, v_2) waits until s_1 has value v_1 or s_2 has value v_2.

The `assign`(s, v, l, u) operation has a precondition of $s \neq v$. It changes s to have value v after a delay that is at least l and at most u time units long. Furthermore, the `assign` operation does not return control to its caller until s has attained the value v. Thus, `assign`(s, v, l, u) has a postcondition of $s = v$. The `vassign` operation handles possibly *vacuous* assignment. A vacuous assignment is one that has no effect, because the variable already has the desired value. Thus, `vassign` does not share the precondition of the `assign` statement. `vassign`(s, v, l, u) assigns s the value v after l to u time units no matter what

the previous value of s was. Like the `assign` statement, `vassign`(s, v, l, u) does not return control to the caller unless and until signal s has the value v. Thus, `vassign`(s, v, l, u) has the postcondition that $s = v$. The handshake package also supports parallel assignment. `assign`$(s_1, v_1, l_1, u_1, s_2, v_2, l_2, u_2)$ is equivalent to executing `assign`(s_1, v_1, l_1, u_1) and `assign`(s_2, v_2, l_2, u_2) in parallel. Similarly, `vassign`$(s_1, v_1, l_1, u_1, s_2, v_2, l_2, u_2)$ is equivalent to executing `vassign`(s_1, v_1, l_1, u_1) and `vassign`(s_2, v_2, l_2, u_2) in parallel. Currently, the handshake package supports parallel assignments of up to three different signals.

For example, the following VHDL code using the handshake package represents one possible handshaking expansion of the channel-level specification of *simple* from Section 2.1.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use work.handshake.all;
use work.nondeterminism.all;
entity simpleHand is
end simpleHand;
architecture behavior of simpleHand is
  signal cReq, cAck :  std_logic;
  signal cData, x, y :  std_logic_vector( 2 downto 0 ) := "000";
begin
  P : process
  begin
    --@synthesis_off
    cData <= x;
    x <= x + 1;
    --@synthesis_on
    wait for delay(1, 2);
    assign(cReq, '1', 2, 4);
    guard(cAck, '1' );
    assign(cReq, '0', 1, 3);
    guard(cAck, '0' );
  end process P;
  Q : process
  begin
    guard(cReq, '1' );
    --@synthesis_off
    y <= cData;
    --@synthesis_on
    assign(cAck, '1', 6, 10);
    guard(cReq, '0' );
    assign(cAck, '0', 5, 7);
  end process Q;
end behavior;
```

Process P places a new valid datum on $cData$. Then, it raises the corresponding request line, $cReq$ to indicate that $cData$ holds a new, valid datum. When process Q detects that $cReq$ is high, it copies the datum from $cData$ into the variable y. Then, it raises $cAck$ to indicate that it has received the datum. Then process P lowers $cReq$, and then process Q lowers $cAck$. This completes the communication, and control returns to the beginning of each process. Figure 2.2 illustrates the first two iterations of this example.

2.3 Timed Event/Level Structures

The `ATACS` compiler accepts the textual specifications of Section 2.1 and produces a graphical representation called a *timed event/level (TEL) structure* [8, 7, 9]. TEL structures most closely model signal-level specifications. However, this dissertation shows that they can also model channel-level specifications and even protocol specifications. The following definition is adapted from [7].

Definition 2.1 A TEL structure is a 7-tuple $T = (N, S_0, A, E, R, R_0, \#)$ where:

1. N is the set of signals;
2. $S_0 \subseteq N$ is the initial state;
3. $A \subseteq (N \times \{+, -\}) \cup \{\$\}$ is the set of actions;
4. $E \subseteq A \times (\mathcal{N} = \{0, 1, 2, \ldots\})$ is the set of events;
5. $R \subseteq E \times E \times \mathcal{N} \times (\mathcal{N} \cup \{\infty\}) \times 2^{2^N}$ is the set of rules;
6. $R_0 \subseteq R$ is the set of initially marked rules;

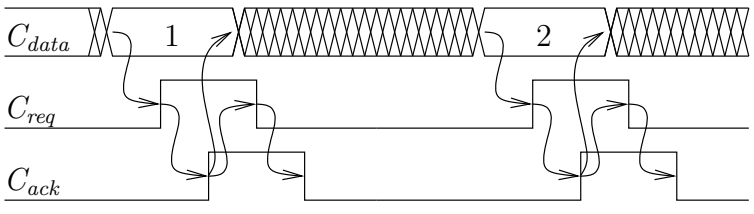

Figure 2.2. The first two iterations of the *simpleHand* example.

7. $\# \subseteq E \times E$ is the irreflexive and symmetric conflict relation.

Each signal, $x \in N$, represents a wire in the specification. The boolean state of the system is modeled as the set of signals, $S \subseteq N$, that are currently high. In particular, the initial state, $S_0 \subseteq N$, is the set of signals that are initially high. For each signal $x \in N$, the action set A can contain a rising transition $(x, +)$, denoted $x+$, and a falling transition $(x, -)$, denoted $x-$. The action $x+$ adds x to the current boolean state: $S = S \cup \{x\}$. The action $x-$ removes x from the current boolean state: $S = S - \{x\}$. The action set A can also contain the *sequencing action*, \$, which does not cause a transition on any signal, and hence leaves S unchanged. Each event, $(a, i) \in E$, is an action paired with a nonnegative integer and refers to a particular instance of that action. This dissertation treats TEL structures as finite, cyclic structures. The instance number i is necessary to distinguish multiple instances of a given action within one iteration of the TEL structure. A *sequencing event* is any event of the form $(\$, i)$. For clarity, this dissertation often assigns a symbolic name to a sequencing event of the form $\$_s$, where s is a string. Each rule, $r \in R$, is of the form (e, f, l, u, B) where:

1. $e \in E$ is the enabling event of r;
2. $f \in E$ is the enabled event of r;
3. $l \in \mathcal{N}$ is the lower timing bound of r;
4. $u \in (\mathcal{N} \cup \{\infty\})$ is the upper timing bound of r;
5. $B \subseteq 2^N$ is the set of boolean states in which the rule is *level-satisfied*.

Definition 2.2 A rule (e, f, l, u, B) is *level-satisfied* in boolean state S if and only if $S \in B$.

This dissertation often expresses the set of boolean states in which a rule is level-satisfied as a boolean expression, using the following operators.

Definition 2.3 Given a TEL structure, $T = (N, S_0, A, E, R, R_0, \#)$, a signal $x \in N$, and boolean expressions e, e_1, and e_2, let $[e]$ be defined recursively as follows:

$$[true] = 2^N \tag{2.1}$$

$$[x] = \{S \subseteq N \mid x \in S\} \tag{2.2}$$

$$[\sim e] = [true] - [e] \tag{2.3}$$

$$[e_1 \ \& \ e_2] = [e_1] \cap [e_2] \tag{2.4}$$

$$[e_1 \mid e_2] = [e_1] \cup [e_2] \tag{2.5}$$

A rule is *enabled* if its enabling event has occurred and it is level-satisfied. There are two possible semantics to handle the situation in which a rule ceases to be level-satisfied before its enabled event occurs. In *nondisabling semantics*, once a rule becomes enabled, it cannot lose its enabling just because it is no longer level-satisfied. However, in *disabling semantics*, once the rule is no longer level-satisfied, it is no longer enabled. A single specification can include rules with both types of semantics. This dissertation indicates a rule that uses disabling semantics by appending a "d" to its expression. Nondisabling semantics are typically used to specify environment behavior. Disabling semantics are typically used to specify logic gates. During verification, the disabling of a boolean expression on a rule using disabling semantics can be treated as one type of failure, since it corresponds to a glitch on the input to a gate. A rule is *satisfied* if it has been enabled for at least l time units and *expired* if it has been enabled for at least u time units. Excluding conflicts, an event cannot occur until every rule enabling it is satisfied, and it must occur before every rule enabling it has expired.

The conflict relation, #, is used to model disjunctive behavior and choice. If two events *conflict* with each other, this means that both events cannot occur in the same iteration of the TEL structure. Taking conflicts into account, if two rules have the same enabled event, but conflicting enabling events, then only one of the two enabling events needs to occur to cause the enabled event. In general, an event can occur once and only once every rule in a maximal set of rules that enable the event and whose enabling events do not conflict with each other is satisfied. The ability

of an event to occur when only a subset of its enabling rules have become satisfied models a form of disjunctive causality. An event that is enabled by multiple rules whose enabling events conflict with each other is similar to a *merge place* in a Petri net. Choice is modeled when multiple rules share the same enabling event but have conflicting enabled events. In this case, only one of the enabled events can occur in any given iteration of the TEL structure. Once the common enabling event occurs, all the rules from this event become enabled. However, as soon as any one of the events that these rules enable occurs, all the rules that lead to conflicting enabled events lose their enabling. An event that is the enabling event of multiple rules that have conflicting enabled events is similar to a *choice place* in a Petri net. This dissertation assumes that the conflict relation is irreflexive and symmetric.

Definition 2.4 Define the conflict relation for sets, $\#_{set}$ as follows, such that two sets of events are in conflict if and only if each member event of each set conflicts with each member event of the other set:

$$X\#_{set}Y \iff \# \supset X \times Y$$

Note that because the conflict relation for events, #, is irreflexive and symmetric, the conflict relation for sets of events, $\#_{set}$, is also irreflexive and symmetric.

Definition 2.5 Let $x\pm$ be shorthand for the set $\{x+, x-\}$.

Suppose that $r = (e, f, l, u, B) \in R$, and $r' = (e', f, l', u', B') \in R$. If e does not conflict with e', then f cannot occur until both r and r' are satisfied. This situation models a *join*. However, if e conflicts with e', which is written $e\#e'$, then f can occur as soon as either r or r' is satisfied. This situation models a *merge*.

A rule ceases to be enabled once its enabled event or any event that conflicts with its enabled event occurs. For example, suppose that $r = (e, f, l, u, B) \in R$, and $r' = (e, f', l', u', B') \in R$. Once f occurs, r is no longer enabled. If f does not conflict with f', and r' is enabled, then r' remains enabled until f' occurs. Similarly, once f' occurs, r' is no longer enabled, but if f does not conflict with f', and r is enabled, then r remains enabled until f occurs. This situation models a

fork. However, if f conflicts with f', which is written $f\#f'$, then as soon as either f or f' occurs, neither r nor r' is enabled. This situation models a *choice*.

Definition 2.6 In a *one-safe* TEL structure, once the enabling event of a rule occurs, it cannot occur again until either the enabled event of the rule occurs, or an event that conflicts with its enabled event occurs [7]. Initially marked rules behave as if their enabling events have already occurred. Therefore, in a one-safe TEL structure, the enabling event of an initially-marked rule cannot occur until the first occurrence of the enabled event of that rule or of an event that conflicts with that enabling event. This property is similar to the one-safe property of Petri nets, which prevents places from containing multiple tokens. This dissertation calls any violation of the one-safe property for TEL structures a *safety violation*.

A TEL structure can be represented graphically as an annotated, directed graph, in which nodes represent events and edges represent rules. For each event $(a, i) \in E$, there is a node labeled "a/i". Recall that i is an instance number that is used to distinguish this event from others with the same action in the same iteration of the TEL structure. If $i = 1$, the "/1" may be omitted. For example, the TEL structure of Figure 2.3 contains two instances of $L_{req}+$ and two instances of $L_{req}-$. In each case, the second instance is distinguished with a "/2". For each rule $(e, f, l, u, B) \in R$, there is a directed arc from the node representing event e to the node representing f. This arc is labeled with "$[l, u]$" and the expression for B using the rules of Definition 2.3 on page 29. If $B = [true]$, then the "$[true]$" may be omitted. For each initially marked rule, $r \in R_0$, the arc representing r is dotted. Conflicts, if any, are explicitly listed.

Assuming no conflicts, the TEL structure of Figure 2.4 shows a fork followed by a join. First the event $a+$ occurs. Then the sequence $b+; b-$ and the sequence $c+; c-$ occur in parallel. Once $b-$ and $c-$ have both completed, then $a-$ can happen. Once $a-$ has happened $a+$ can happen again. This whole sequence repeats forever.

Assume that $a+$ first occurs at time 5. This enables the rules $(a+, b+, 1, 3, [true])$ and $(a+, c+, 2, 4, [true])$. At time 6, the rule $(a+, b+, 1, 3, [true])$ becomes satis-

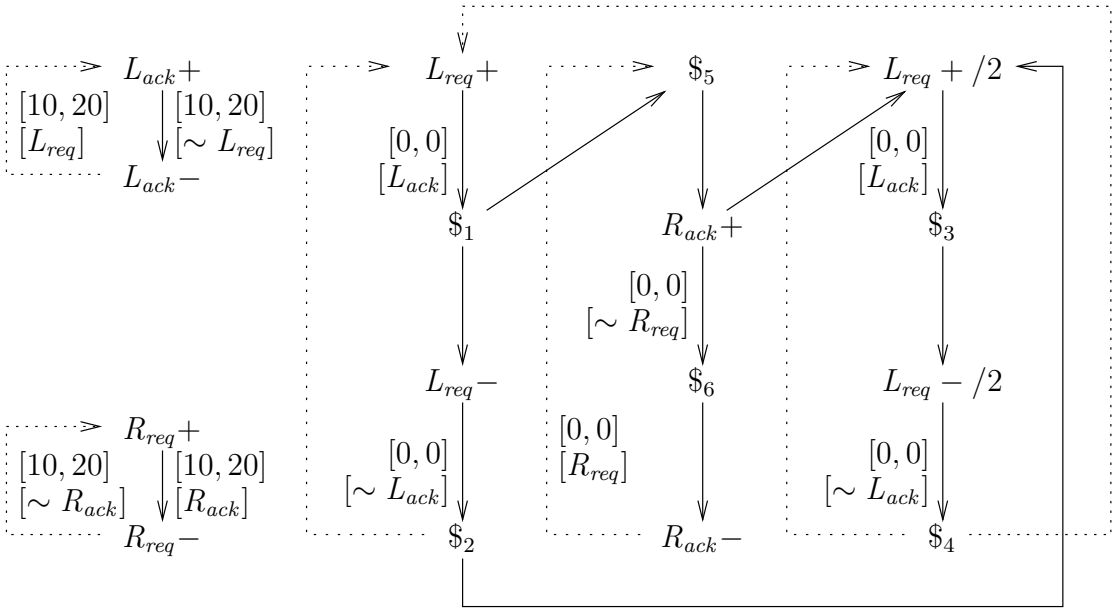

Figure 2.3. Example TEL structure. Except where indicated, each rule has timing bounds $[1, 2]$.

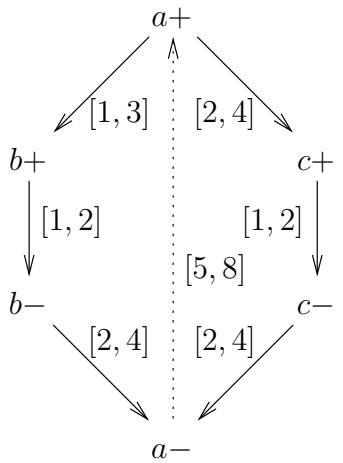

Figure 2.4. TEL structure for a fork followed by a join ($\# = \emptyset$) or a choice followed by a merge ($b \pm \#_{set} c\pm$).

fied. Thus, this is the first time at which $b+$ can occur. At time 7, the rule $(a+, c+, 2, 4, [true])$ becomes satisfied. Thus, time 7 is the first time that $c+$ can occur. At time 8, the rule $(a+, b+, 1, 3, [true])$ expires. This means that $b+$ must occur by time 8. Taken together, the above information shows that $b+$ occurs some time in the range $[6, 8]$. Whenever $b+$ occurs, it enables the rule $(b+, b-, 1, 2, [true])$. One time unit later, $(b+, b-, 1, 2, [true])$ becomes satisfied. One time unit after that, $(b+, b-, 1, 2, [true])$ expires. Thus, $b-$ must occur some time in the range $[7, 10]$. Similarly, $c+$ must occur in the during $[7, 9]$ and $c-$ must occur during $[8, 11]$.

As one possible case, suppose that $b-$ occurs at time 8.2. This enables the rule $(b-, a-, 2, 4, [true])$. Further suppose that $c-$ occurs at time 9.7. This enables the rule $(c-, a-, 2, 4, [true])$. Under these assumptions, $(b-, a-, 2, 4, [true])$ becomes satisfied at time 10.2, and $(c-, a-, 2, 4, [true])$ becomes satisfied at time 11.7. This would mean that the earliest time at which $a-$ can occur is time 11.7. Continuing with these assumptions, $(b-, a-, 2, 4, [true])$ expires at time 12.2, and $(c-, a-, 2, 4, [true])$ expires at time 13.7. Thus, under these assumptions, $a-$ must occur during the range $[11.7, 13.7]$. Note that each end of this range is controlled by the later arriving signal, in this case $c-$. Whenever $a-$ occurs, it enables $(a-, a+, 5, 8, [true])$. Five time units later, $(a-, a+, 5, 8, [true])$ becomes satisfied. Three time units after that, $(a-, a+, 5, 8, [true])$ expires. So under these assumptions $a+$ occurs for the second time, starting a new iteration, sometime in the range $[16.7, 21.7]$.

On the other hand, if the events on signal b conflict with the events on signal c, the TEL structure of Figure 2.4 shows a choice followed by a merge. In this case, in any given iteration of Figure 2.4, once $a+$ has happened, either $b+$ can happen or $c+$ can happen, but not both. Furthermore, whichever one of $b-$ or $c-$ occurs is sufficient to enable $a-$. The process repeats forever, but each iteration makes an independent choice between the sequence $b+; b-$ and the sequence $c+; c-$.

Figure 2.5 shows the TEL structure for a timed version of a rendezvous element from [58], demonstrating the use of level expressions and timing on rules. This is a rendezvous element in the sense of a Muller C-element used in an environment

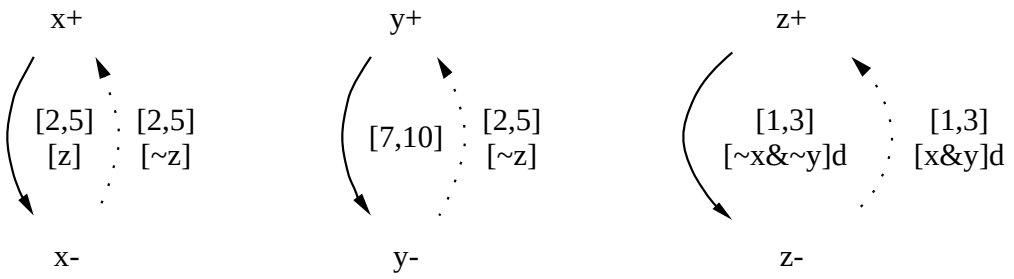

Figure 2.5. TEL structure for a timed *rendezvous element* and its environment [58].

that will never change an input twice without an intervening change on the output. This dissertation calls the assumption that the environment behaves in this way the *rendezvous assumption*.

Assume that in the initial state, all signals are low. In this state, the rules $(x-, x+, 2, 5, [\sim z])$ and $(y-, y+, 2, 5, [\sim z])$ are enabled. At time 2, they become satisfied, and at time 5 they expire. Thus, $x+$ and $y+$ must each occur in the range $[2, 5]$. Once $x+$ occurs, it enables the rule $(x+, x-, 2, 5, [z])$. The level expression $[z]$ ensures that this rule cannot violate the rendezvous assumption. In particular, this rule is not satisfied until two time units after z becomes high.

Once $y+$ occurs, it enables the rule $(y+, y-, 7, 10, [true])$. This rule has no level expression. Therefore, ignoring timing, $(y+, y-, 7, 10, [true])$ would be satisfied at this time. This would allow $y-$ to occur immediately, even if $z+$ had not yet occurred. This would cause the rule $(z-, z+, 1, 3, [x \ \& \ y]\, d)$ to become disabled. This would violate the rendezvous assumption and require a true C-element. (Recall that the "d" after the expression on this rule means that this rule uses disabling semantics.) However, taking time into account, the rule $(y+, y-, 7, 10, [true])$ does not become satisfied until seven time units after it becomes enabled. Suppose that $y+$ occurs at time 2, and that $x+$ occurs at time 5. (This is the worst case.) In this case, the rule $(z-, z+, 1, 3, [x \ \& \ y]\, d)$ is enabled at time 5, satisfied at time 6, and expired at time 8. Thus, $z+$ must occur by time 8. However, the rule

$(y+, y-, 7, 10, [true])$ is enabled at time 2, and is not satisfied until time 9, because of the lower timing bound on the rule $(y+, y-, 7, 10, [true])$. Thus, $y-$ cannot occur until time 9, by which time $z+$ is guaranteed to have occurred. Thus, the given timing constraints enforce the rendezvous assumption for y.

The user can enter TEL structures directly, or the `ATACS` compiler can derive them from higher-level specifications. For example, consider the signal-level VHDL code of Section 2.2. From this, the compiler derives the TEL structure of Figure 2.6.

2.4 State Graphs

In order to design a circuit from a TEL, it is necessary to find its *reduced state graph* (RSG). An reduced state graph is essentially a graph in which each vertex represents a state of the system. Each state is labeled with the state of each signal wire. The edges in the graph represent possible transitions between states.

Definition 2.7 A reduced state graph is a 6-tuple $RSG = (I, O, T, S, \delta, \lambda_S)$ where:

1. I is the set of input signals;
2. O is the set of output signals;
3. $T \subseteq (I \cup O) \times \{+, -\} \times \mathcal{N}$ is the set of transition events;
4. S is the set of states;

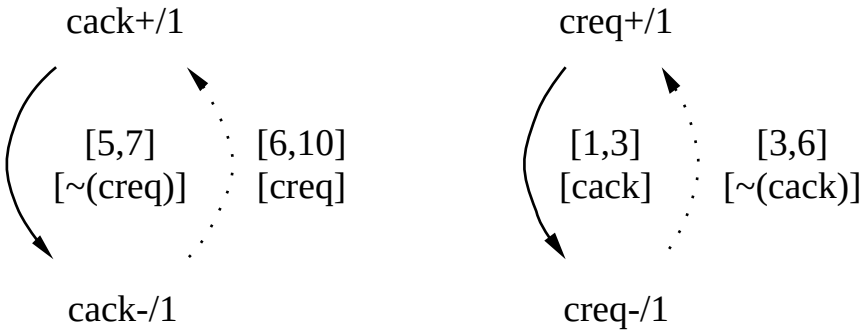

Figure 2.6. TEL structure derived from the *simpleHand* example of Section 2.2.

5. $\delta \subseteq S \times T \times S$ is the set of state transitions.

6. $\lambda_S : S \rightarrow (I \cup O \rightarrow \{0, R, 1, F\})$ is the *state labeling function*

The state labeling function λ_S labels each state $s \in S$ with a function that maps a signal into the value of that signal. In particular, for each $u \in I \cup O$,

$$\lambda_S(s)(u) = \begin{cases} 0 & \text{if } u \text{ is stable low} \\ R & \text{if } u \text{ is untimed-enabled to rise} \\ 1 & \text{if } u \text{ is stable high} \\ F & \text{if } u \text{ is untimed-enabled to fall} \end{cases}$$

In speed-independent design, the enablings could be inferred from the edges in the state graph. However, in timed design, it is possible that some states (and the associated edges) are eliminated by timing constraints. Therefore, information about the untimed enablings cannot be inferred from the edges alone, and it must be represented in the states themselves. Nevertheless, it is useful to define the following function that strips the enabling information from a state.

Definition 2.8 For each $u \in I \cup O$ and $s \in S$,

$$\text{val}(\lambda_S(s)(u)) = \begin{cases} 0 & \text{if } \lambda_S(s)(u) \in \{0, R\} \\ 1 & \text{if } \lambda_S(s)(u) \in \{1, F\} \end{cases}$$

Figure 2.7 shows the reduced state graph for the TEL structure of Figure 2.5. Each node s is labeled with $\lambda_S(s)(x)\lambda_S(s)(y)\lambda_S(s)(z)$. In the states labeled $RF0$ and $1FR$, $\lambda_S(s)(y) = F$. This indicates that the signal y is untimed enabled to fall. However, the only arc leaving $RF0$ is the arc for $x+$, and the only arc leaving $1FR$ is the arc for $z+$. That is because the timing bounds on the TEL structure of Figure 2.5 ensure that $z+$ must occur before $y-$. Thus, timing enforces the rendezvous assumption for y. In contrast, in states $1R0$ and state $1FR$, $\lambda_S(s)(x) = 1$. This indicates that even in the untimed sense, x is stable high in these states. This is because the level expressions of Figure 2.5 ensure that $z+$ must occur before $y-$. Thus, the level expressions enforce the rendezvous assumption for x.

Figure 2.8 shows the reduced state graph for the TEL structure of Figure 2.6. Each node s is labeled with $\lambda_S(s)(c_{ack})\lambda_S(s)(c_{req})$.

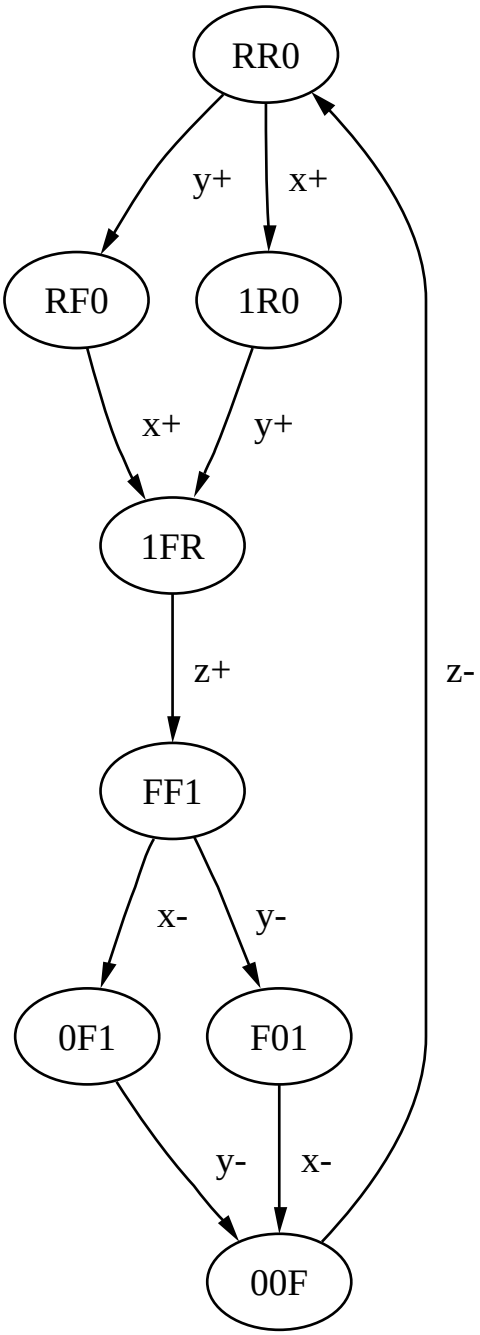

Figure 2.7. Reduced state graph for the timed rendezvous element. Each state s is labeled with $\lambda_S(s)(x)\lambda_S(s)(y)\lambda_S(s)(z)$.

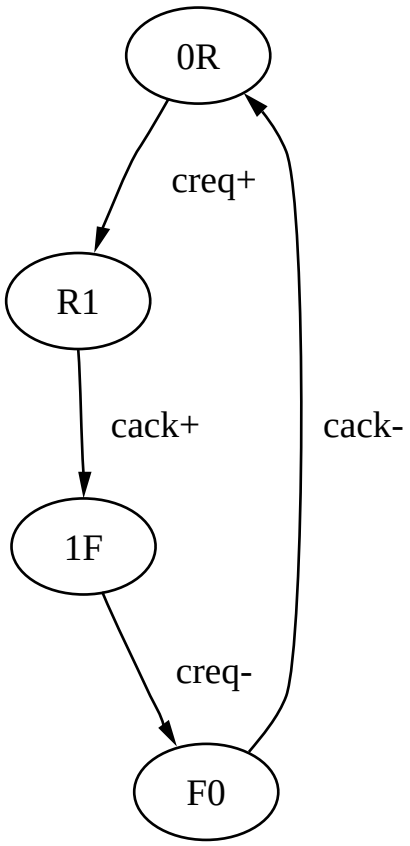

Figure 2.8. Reduced state graph for the *simpleHand* example. Each state s is labeled with $\lambda_S(s)(c_{ack})\lambda_S(s)(c_{req})$.

In order to be synthesizable, an reduced state graph must have the *Complete State Coding* (CSC) property. This means that if any two states in the reduced state graph have the same underlying binary value, they must also have identical output enablings.

Definition 2.9 A reduced state graph $RSG = (I, O, T, S, \delta, \lambda_S)$ has the *complete state coding* property if and only if for any two states $s, t \in S$, either $\exists u \in I \cup O$. $\text{val}(\lambda_S(s)(u)) \neq \text{val}(\lambda_S(t)(u))$, or $\forall o \in O$. $\lambda_S(s)(o) = \lambda_S(t)(o)$. Any pair of states that violates this property is called a *complete state coding (CSC) violation.*

For example, the reduced state graph in Figure 2.8 is complete state coded. However, consider the reduced state graph in Figure 2.9. The inputs are w and z. The outputs are x and y. Each state s is labeled with $\lambda_S(s)(w)\lambda_S(s)(z)\lambda_S(s)(x)\lambda_S(s)(y)$. The two highlighted states, those labeled with $10RR$ and $1R00$, have the same underlying value for each signal: 1000. However, in state $10RR$ the outputs are enabled to rise, but in state $1R00$ they are not.

2.5 Production Rules

From the Reduced State Graph, `ATACS` attempts synthesis. If synthesis is successful, the result is a set of *production rules.* Each production rule takes the form of a guarded command. The guard is a boolean expression. The command is an action (the raising or lowering of a signal) that should be performed whenever the guard evaluates to *true*. For example, from the reduced state graph of Figure 2.7, the method obtains the following production rules:

$$x \ \& \ y \rightarrow z+ \tag{2.6}$$

$$\sim x \ \& \ \sim y \rightarrow z- \tag{2.7}$$

These production rules specify the circuit of Figure 2.10.

From the reduced state graph of Figure 2.8, synthesis derives the following production rules:

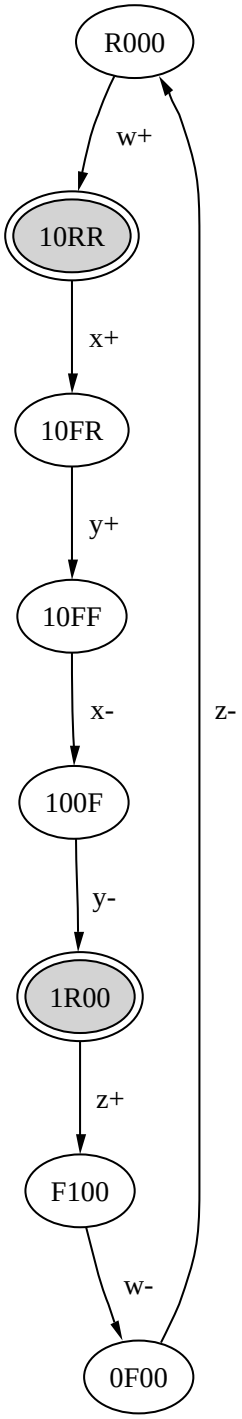

Figure 2.9. Reduced state graph with a complete state coding violation. The inputs are w and z. The outputs are x and y. Each state s is labeled with $\lambda_S(s)(w)\lambda_S(s)(z)\lambda_S(s)(x)\lambda_S(s)(y)$.

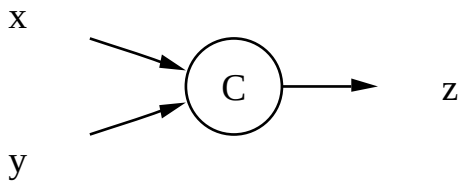

Figure 2.10. Circuit derived for the timed rendezvous element.

$$c_{req} \rightarrow c_{ack}+ \tag{2.8}$$

$$\sim c_{req} \rightarrow c_{ack}- \tag{2.9}$$

$$\sim c_{ack} \rightarrow c_{req}+ \tag{2.10}$$

$$c_{ack} \rightarrow c_{req}- \tag{2.11}$$

In this case, c_{ack} always follows c_{req}. Hence a simple wire can implement c_{ack}. Furthermore, c_{req} responds to any change in the value of c_{ack} by executing the opposite transition. Thus, an inverter can implement c_{req}. Figure 2.11 illustrates the circuit. In general, whenever the guard to raise a signal is the complement of the guard to lower that signal, a combinational gate is sufficient to implement that signal.

2.6 Performance Analysis

To help the user find an efficient solution, this dissertation presents techniques that automatically consider many different implementations of a given specification For this search to be effective, the evaluation of any given point in the design space must be quick [38]. In other words, quick estimates of the metrics under optimization are necessary. If comparing two alternatives requires full synthesis

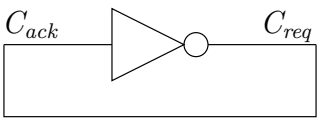

Figure 2.11. Circuit for the *simpleHand* example.

of each alternative, it is not practical to compare many alternatives. This section discusses a strategy for rapid preliminary analysis of alternatives.

Section 5.3 presents a branch-and-bound algorithm [25] to guide the choices made during concurrency reduction. This requires the ability to assess quickly the quality of a given TEL structure. To estimate the performance of a given alternative, this technique uses Mercer's stochastic cycle period analysis [55]. This can estimate the performance of a design, based solely on its TEL structure. It can do so in significantly less time than synthesis would take.

CHAPTER 3

COMPILATION OF CHANNEL-LEVEL SPECIFICATION

This chapter describes how our tool compiles the channel-level specification of Section 2.1 into a TEL structure that models the behavior of that specification. Section 3.1 describes how the specification can constrain the implementation of each channel. It also defines the requirements for such constraints to be consistent. Section 3.2 describes how to model the behavior of the channel-level specification as a TEL structure.

3.1 Semantic Issues

This section describes the various choices that must be made before a channel communication can be implemented. It also describes how the channel-level specification can place constraints on these decisions and the requirements for such specifications to be consistent. Section 3.1.1 discusses the decision about which process initiates any given channel communication. Section 3.1.2 discusses the direction of data transmission. Section 3.1.3 discusses the number and significance of each phase of a channel communication. Section 3.1.4 discusses how the receiver can determine whether the incoming data are valid.

3.1.1 Active vs. Passive

If two processes communicate over a channel, one process must initiate the communication. This process is called the *active* process with respect to that channel. The other process simply waits for a communication to begin and then responds. This second process is called the *passive* process with respect to that channel [50].

The specification can declare any given port to be active or passive. For example, the following VHDL entity declares the x port to be active and the y port to be passive, but it makes no commitment about the z port.

```
entity declare is
  port(x :  inout channel := active;
       y :  inout channel := passive;
       z :  inout channel := init_channel);
end declare;
```

The `active` and `passive` declarations are allowed only in entity declarations, since only one port on a given channel may be active. However, the `init_channel` declaration is also allowed in signal declarations.

It is also possible for the usage of the channel to implicitly determine which side of a channel is active and which is passive. By using a `probe` command, the specification implicitly declares the probed channel to be passive on the side that uses the probe. Hence, when the compiler encounters a probe command, it responds as if the corresponding port had explicitly been declared passive. If in fact, that port was declared active, the `probe` command is an error.

This dissertation assumes that exactly one side of each channel must be active, and the other side must be passive. In other styles, it is possible to relax this constraint. For example, `Tangram` supports channels on which both the sender and receiver are considered to be active [40]. However, this requires a handshake circuit called a *passivator* to be inserted on the channel between the sender and the receiver. This dissertation treats any such circuitry as part of either the sender or receiver. Assigning the passivator to one of the communicating processes (sender or receiver) makes that process passive. Within this context, the assumption that exactly one side of a channel is active, and the other side is passive, is valid. Therefore, if the specification constrains (either explicitly or implicitly) both sides of any given channel to be passive, or if it constrains both sides of any given channel to be active, the compiler rejects the specification.

If, on the other hand, the specification constrains exactly one side of a given channel, the specification is acceptable. In this case, the compiler responds by constraining the other side of the channel accordingly. For example, if the specification

constrains one side to be to be passive, the compiler constrains the other side to be active. Similarly, if the specification constrains one side to be active the compiler constrains the other side to be passive.

Finally, there may be cases in which the specification does not constrain either side of any given channel. In this case, the compiler simply notes this fact. This means that the methods presented in the later chapters are free to make either choice as to which side of the channel in question is active and which side is passive. Currently, these methods make channels be *push-type* channels by default. Hence, if the specification does not constrain which side of a given channel is active, the tool that this dissertation presents currently implements `send` operations on that channel as active and `receive` operations as passive.

3.1.2 Data Direction

Using a `send` command implicitly declares the process that uses it to be the producer of data for the channel in question. Using a `receive` command declares the process to be the consumer of data for the channel in question.

This dissertation assumes that one side of any given channel should contain `send` commands and no `receive` commands and that the other side should contain `receive` commands and no `send` commands. Therefore, if the compiler detects `send` commands on both sides of any given channel, or `receive` commands on both sides of any given channel, it reports an error. Furthermore, if it detects both `send` and `receive` commands on the same channel in any given process it reports an error.

The choice of this section is independent of the choice between active and passive of Section 3.1.1. In other words, any given channel may have active `send` operations and passive `receive` operations, in which case it is a *push channel*, or it may have passive `send` operations and active `receive` operations, in which case it is a *pull channel*. The tool that this dissertation presents supports both types. The specification can determine the type of a given channel through the declarations of Section 3.1.1 combined with the usage of `send`, `receive`, and `probe` commands.

The tool that this dissertation presents does not currently support multicast,

broadcast, or bidirectional channels. Section 1.1 discusses other tools that do, and Section 8.1 discusses the possibility of extending the tool that this dissertation presents to support such channels.

3.1.3 Two-phase vs. Four-phase

The first communication on any given channel typically begins as follows. The active process raises a *request* line. The passive process responds by raising an *acknowledge* line. There are two possibilities for what happens next.

In *two-phase* communication or *transition signaling* the communication is considered complete at this point. The next time that the processes communicate over the same channel, the request and acknowledge lines are already high. Therefore, this second communication must proceed as follows. The active process lowers its request line. The passive process responds by lowering its acknowledge line. At this point the second communication is considered complete. Both the request and the acknowledge line are now low, so the third communication can proceed exactly as the first. Thus, in transition signaling, any transition on a given control wire is treated in the same way. A rising transition and a falling transition have exactly the same significance. Figure 3.1, adapted from [28], illustrates two-phase communication. Note that each communication requires just two transitions on control wires. It is not the values on the control wires that is significant but rather the relationship between these values. If the control wires have the same value, no communication is pending on the channel. If they have they have opposite values a communication is pending on the channel. These two possibilities lead to the name *two phase*.

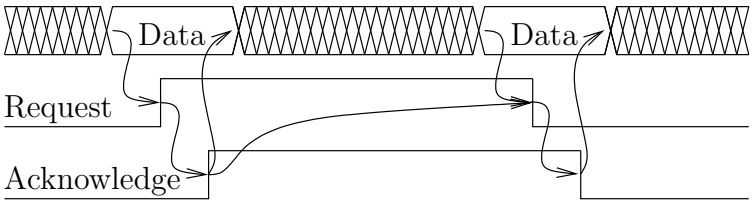

Figure 3.1. Two-phase communication.

In *four-phase* communication, each communication must begin with rising transitions. Thus, once request and acknowledge are both high, the processes must reset the control wires to the low state before the next communication on the same channel can begin. The active process lowers its request wire and the passive process responds by lowering its acknowledge wire. In this case, the lowering of these wires is part of the first communication. The second communication on this channel can then proceed in exactly the same way as the first communication. Thus, in four-phase communication, any communication on a given channel is just like any other communication on that channel, but a rising transition on a control signal is different from a falling transition on that control signal. Figure 3.2, adapted from [28], illustrates one type of four-phase communication. In this case, each communication requires four transitions on control wires. Furthermore, each communication contains all four possible combinations of levels for the control wires, hence the name *four phase*.

As mentioned in Chapter 1 and [63, 20], even within four-phase protocols, there are several possible conventions about when data are valid. These conventions must result in correct operation of the target data path. These conventions also place constraints on how two different communications may be correctly sequenced.

Consider a simple, four-phase communication protocol. For purposes of this illustration, consider only the control portion of the communication. Thus, a single communication action on channel A expands the following sequence of events:

$$A_{req}+; A_{ack}+; A_{req}-; A_{ack}-$$

Now consider the sequential composition of two independent communication

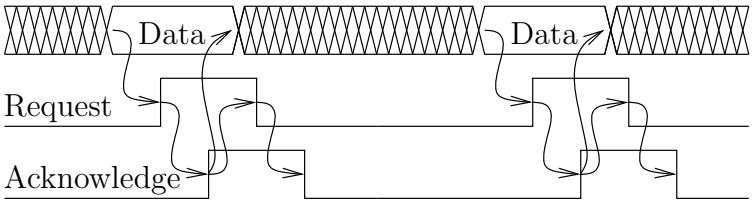

Figure 3.2. Four-phase communication.

actions on two separate channels A and B. Let us start with the most aggressive (most concurrent) possibility for the control, which is *narrow sequencing* [63]. This assumes that once the *acknowledge* signal has been raised ($A_{ack}+$), the communication on B can start. Thus, the *return-to-zero* events ($A_{req}-; A_{ack}-$) may be overlapped with the events of the communication on channel B. Figure 3.3(a) illustrates narrow sequencing. However, in *weak-broad sequencing*, the communication on channel B must not start until the *request* line for A has returned to zero ($A_{req}-$). So only the return to zero of the acknowledge line ($A_{ack}-$) may be overlapped with the communication on channel B. Figure 3.3(b) illustrates weak-broad sequencing. Finally, in *broad sequencing*, the communication on B must wait for $A_{ack}-$. Thus, no overlap between A and B is possible. Figure 3.3(c) illustrates broad sequencing.

In all these protocols, the *req* signal is controlled by the active process, and the *ack* signal is the response from the passive process. However, data can flow in either direction. On a push channel, data flow in the direction of the *req* signal. On a pull channel, data flow in the direction of the *ack* signal. Section 3.1.1 and Section 3.1.2 show how the specification can constrain the type of each channel.

Currently, the front end for the tool that this dissertation presents always targets pure synchronization channels using a four-phase protocol with narrow sequencing. However, other protocols can be handled by bypassing the front end and providing signal-level input directly to the concurrency reduction engine. Section 4.3 demonstrates signal-level specifications for many different types of sequencing. In case future developers automate the generation of the most-concurrent starting point for the other protocols described in this section, the channel package has reserved several directives for future expansion. The remaining paragraphs of this section describe these directives. They are currently accepted but ignored.

The initialization functions, `active`, `passive`, and `init_channel` can accept one of the following optional arguments. `2phase` declares that the channel should use two-phase communication. `4phase` declares four-phase operation, but does not commit to a particular type of sequencing. `narrow`, `weak-broad`, and `broad` each imply four-phase operation and further restrict the type of sequencing used.

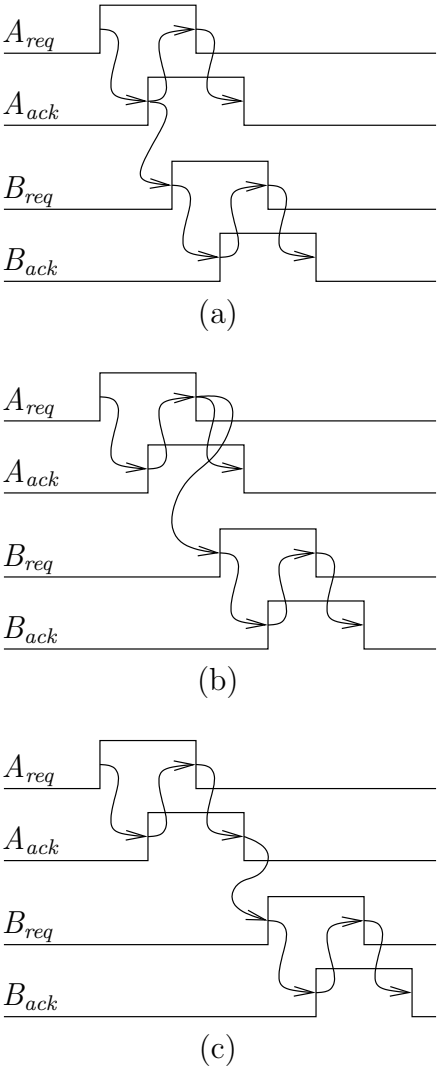

Figure 3.3. Four-phase sequencing constraints. Part (a) shows narrow sequencing. Part (b) shows weak-broad sequencing. Part (c) shows broad sequencing.

For example, the following signal declaration specifies that channel C must use four-phase communication, with weak-broad data sequencing.

```
signal C : channel := init_channel(sequencing => weak_broad);
```

It is also possible for an entity declaration to place such constraints on its ports. If both sides of a given channel are so constrained, the compiler must check the declarations for consistency. For example, if the specification declares one side of a channel to be two-phase, but the other to be four-phase, the compiler rejects the specification. If, on the other hand, the specification constrains only one side of a channel, the compiler simply propagates the protocol constraint to the other side of the channel. Finally, if the specification constrains neither side of a given channel, the compiler notes that the decision of whether to use two-phase or four-phase (and if four-phase, what type of data constraints to use) is still an open decision.

3.1.4 Bundled Data vs. Data Encoding

When data transfer occurs during a channel communication, the sender's assertion that the data are valid must not reach the receiver before the new, valid data themselves reach the receiver. There are many approaches to guaranteeing this. They generally fall into two categories: *bundled data* and *data encoding*.

Figure 3.4 illustrates the bundled-data approach. In this approach, the sender's assertion that the data are valid is carried on a control wire separate from the data wires. In the case of a push channel, this is the *request* signal. In the case of a pull channel, this is the *acknowledge* signal. The data path consists of standard,

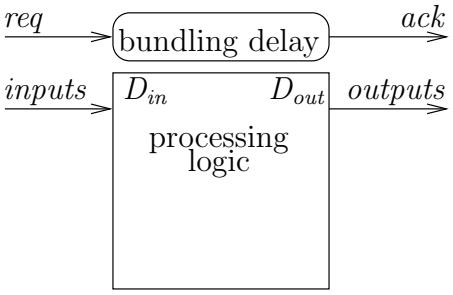

Figure 3.4. Bundled data.

combinational logic blocks. The designer or the CAD tools must estimate the worst-case delay though each logic block. The control path simply uses a delay element to model this worst-case delay. This delay element is inserted on the wire that carries the sender's assertion that the data are valid. Even if a given logic block is multiple bits wide, one delay element is associated with the entire block. Thus, all the data bits of the block (or bus) are "bundled" together with one delay, hence the name *bundled data*. The common delay element is called the *bundling delay*. Each channel that carries data using the bundled-data approach relies on the assumption that all valid data arrive at the receiving end of the channel before the relevant control signal (that announces the arrival of the data) gets through the bundling delay and reaches the receiver. This one-sided timing assumption is called the *bundling constraint*.

Figure 3.5 illustrates the data-encoding approach. In this approach, data are encoded in such a way that the receiver can determine whether they are valid by directly examining the data. Hence the sender's assertion that the data are valid is encoded in the data themselves. Thus, no timing assumption is necessary. This approach requires using more data wires than there are information bits in each datum to be sent. For example, consider the problem of encoding one bit of information. A common form of data encoding, known as *dual-rail*, does this in the following way. Two physical data wires, d_0 and d_1 are allocated to the channel. When $d_0 = d_1 = 0$, this means that there is no valid datum on the channel. When $d_0 = 1$, but $d_1 = 0$, this means that a valid datum of "0" is present on the channel. If $d_0 = 0$, but $d_1 = 1$, this means that a valid datum of "1" is present on the channel.

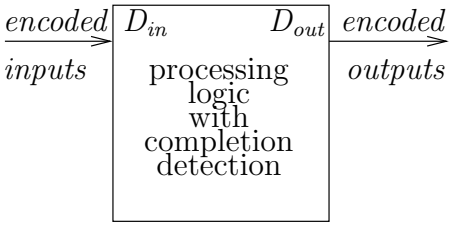

Figure 3.5. Data encoding.

This can be generalized to multi-bit channels in several ways. The approach can simply be repeated for each bit, using a total of $2i$ wires to transmit i bits of information. The entire data word is valid if and only if each pair of wires has one of the wires high. For narrow data words, a *unary* [21] or *one-hot* [26] encoding scheme is feasible. In this scheme, one wire is allocated for each possible datum value that the channel can transmit. The data are valid when exactly one of these wires is high. Transmitting an i-bit wide data word using this scheme requires 2^i data wires. In general, data can be encoded using an m/n code, in which the data are considered valid when m out of n wires are high [3]. In this terminology, each bit of a dual-rail code uses a $1/2$ code. A unary or one-hot scheme uses a $1/2^i$ code to send i bits of information.

Currently, the tool that this dissertation presents implements only the control portion of channel communications and not the data path. However, Section 4.3 shows how data could be implemented in the future, and how the target data path should affect the control. These effects can be achieved by bypassing the front end and providing the signal-level starting point directly to the concurrency reduction engine. In case this is automated in the future, the channel package reserves several directives to indicate how data should be handled for future expansion. The remaining paragraphs of this section describe these directives. They are currently accepted but ignored.

The specification can constrain these choices as well. The initialization functions, `active`, `passive`, and `init_channel` can accept one of the following optional arguments. `bundled` declares that the channel should use bundled data communication. `encoded`(m, n) declares data encoding, using an m/n code for each field. Thus, each field is implemented with n wires, m of which are 1 for any valid code word.

These declarations may be combined with those of Section 3.1.3. For example, the following signal declaration specifies that channel C must use four-phase communication, with narrow data sequencing, and dual-rail encoding.

```
signal C : channel := init_channel(sequencing => narrow,
                                   data => encoded(1, 2));
```

3.2 Channel-Level TEL

The first step toward the graphical manipulations of Chapter 4 and Chapter 5 is to produce a TEL structure that models the channel-level specification. This TEL structure is not meant to be a signal-level specification. It is meant to be only a channel-level model of the behavior of the specification. At such a behavioral level, all that is important about `send` and `receive` operations is that matching `send` and `receive` operations should synchronize. In other words, whichever operation is executed first among a matching pair of `send` and `receive` operations must wait until its matching partner is executed. This property is used to perform a channel-level verification of the specification. If verification finds a deadlock in the channel-level TEL that the technique of this section produces, this means that there is a deadlock inherent in the channel-level specification. In this case, the user must change the channel-level specification before any signal-level implementation can be found.

Simple syntax-directed translation produces the desired TEL structure. For each channel C in the channel-level specification, the `ATACS` parser adds two signals, $c!$ and $c?$ to the TEL structure. A high value of signal $c!$ indicates that there is a pending `send` operation on channel C. A high value on signal $c?$ indicates that there is a pending `receive` operation on channel C.

When a given process executes a `send` operation on channel C, it must indicate that the `send` is pending by raising the signal $c!$. Then it must wait until the corresponding `receive` operation is pending before proceeding. In other words, it must wait until the signal $c?$ is high. Once $c?$ is high, the process must indicate that the `send` operation is no longer pending by lowering the signal $c!$. It then must wait until the `receive` operation is no longer pending. In other words, it must wait until $c?$ is low. Thus, for each `send` operation on channel C in the specification, the `ATACS` parser introduces the following rules into the channel-level TEL structure: $(c!+, c!-, 0, \infty, [c?])$ and $(c!-, \$_{sink}, 0, \infty, [\sim c?])$. What rules should enable $c!+$ depends on what precedes the `send` operation in the channel-level specification. Similarly, what rules $\$_{sink}$ should enable depends on what follows the `send` operation

in the channel-level specification. This scheme does not commit the lower levels of the tool to any particular protocol or implementation. It is used only for channel-level verification. For purposes of the rest of the tool that this dissertation presents, the events $c!+$ and $c!-$ and the above rules are simply place holders for the `send` command itself. The handshaking expansion phase could replace these with any signal-level structure that still meets the semantics of the `send` operation. It is the initial handshaking expansion step that occurs after the compilation described here but before concurrency reduction that is currently limited to four-phase communication. The channel-level TEL structure itself imposes no such limitation.

In the above discussion, $\$_{sink}$ actually represents a distinct sequencing event for each channel communication. In simple cases, this sequencing event can be removed in a *postprocessing* step [76, 75].

When a given process executes a `receive` operation on channel C, it must indicate that the `receive` is pending by raising the signal $c?$. Then it must wait until the corresponding `send` operation is pending before proceeding. In other words, it must wait until the signal $c!$ is high. Once $c!$ is high, the process must indicate that the `receive` operation is no longer pending by lowering the signal $c?$. It then must wait until the `send` operation is no longer pending. In other words, it must wait until $c!$ is low. Thus, for each `receive` operation on channel C in the specification, the `ATACS` parser introduces the following rules into the channel-level TEL structure: $(c?+, c?-, 0, \infty, [c!])$ and $(c?-, \$_{sink}, 0, \infty, [\sim c!])$. What rules should enable $c?+$ depends on what precedes the `receive` operation in the channel-level specification. Similarly, what rules $\$_{sink}$ should enable depends on what follows the `receive` operation in the channel-level specification. This scheme does not commit the lower levels of the tool to any particular protocol or implementation. It is used only for channel-level verification. For purposes of the rest of the tool that this dissertation presents, the events $c?+$ and $c?-$ and the above rules are simply place holders for the `receive` command itself. The handshaking expansion phase could replace these with any signal-level structure that still meets the semantics of the `receive` operation. It is the initial handshaking expansion step

that occurs after the compilation described here but before concurrency reduction that is currently limited to four-phase communication. The channel-level TEL structure itself imposes no such limitation.

For example, consider the following channel-level specification.

```
architecture behavior of PAex is
  signal x, y, z :  std_logic_vector( 2 downto 0 ) := "000";
  signal L, R : channel := init_channel;
begin
  producer :  process
  begin
    send(L, x);
    --@synthesis_off
    x <= x + 1;
    wait for delay(4, 5);
    --@synthesis_on
  end process producer;
  FIFO : process
  begin
    receive(L, y);
    wait for delay(1, 2);
    send(R, y);
  end process FIFO;
  consumer :  process
  begin
    receive(R, z);
    wait for delay(4, 6);
  end process consumer;
end behavior;
```

From this, the parser derives the channel-level TEL structure of Figure 3.6. Note that post processing has removed the $\$_{sink}$ sequencing events.

If the compiler determines the specification has constrained the protocol for a channel communication, it annotates the channel level TEL with information about the constraints. Furthermore, the compiler also annotates each event with information about which datum is being sent or received. This annotation is currently implemented by appending a number containing a bit vector with flags for each of the options to the name of each event involved in a channel communication. This information is not shown in Figure 3.6. Furthermore, this information is ignored during verification of the channel-level TEL. However, the expansion techniques of Chapter 4 use this information to constrain their decisions about which process is the active participant and which process is the passive participant in each channel communication.

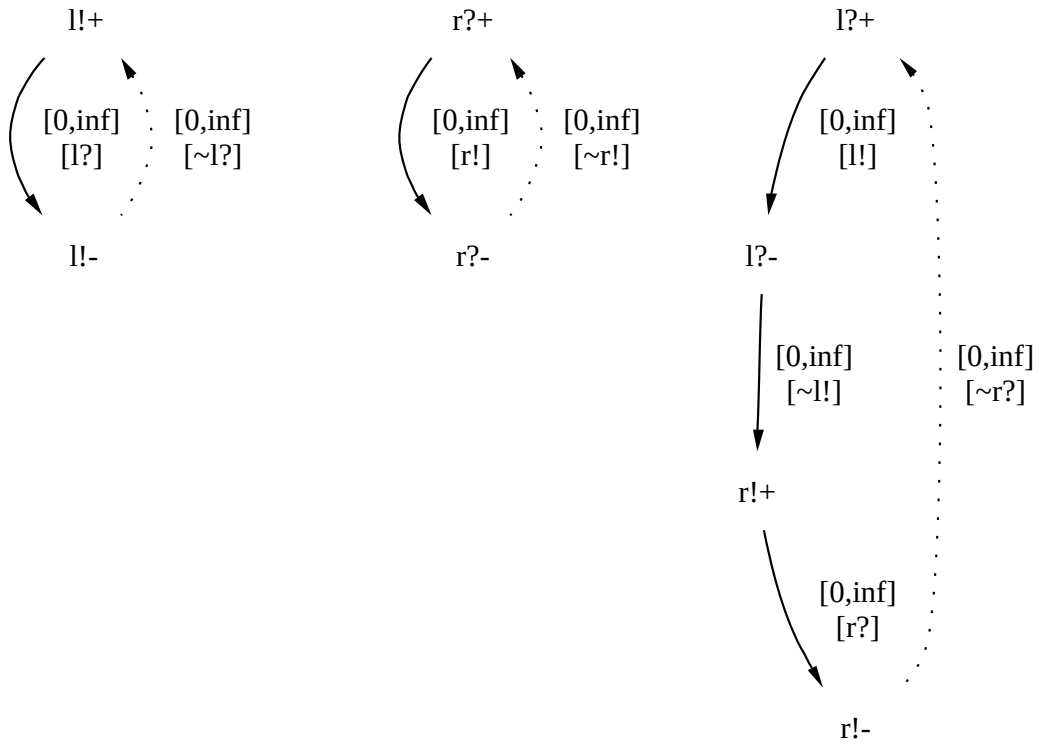

Figure 3.6. Channel-level TEL model of the *producer*, *consumer*, and *FIFO*.

The channel-level TEL structure represents an abstract model of the channel-level specification. If there is deadlock inherent in the original channel-level specification, that deadlock is detectable from this TEL structure. `ATACS` can detect such a deadlock before the graph transformations and searches of Chapter 4 and Chapter 5 even begin.

CHAPTER 4

PROTOCOL SPECIFICATION

Given the channel-level model of Chapter 3, the techniques of this chapter derive an initial signal-level model. This requires selecting a communication protocol for each channel. Given the channel-level TEL structure, and the selection of a protocol, the techniques of this chapter must find the most concurrent TEL structure that still meets the constraints of both the original specification and the given protocol.

This chapter presents an overview of the different types of constraints that a protocol imposes on the signal-level implementation. It also presents a model for expressing such constraints using TEL structures. In particular, Section 4.1 shows the difference in the constraints on signal-level implementations of an active and a passive protocol.

The techniques of this chapter have been automated for only one protocol. The tool that this dissertation presents does include a simple automatic expander that, given the graphical channel-level specification that is the result of Chapter 3, produces the starting point for concurrency reduction in format described in this chapter. This front end currently targets pure synchronization channels, using one four-phase protocol with narrow sequencing. For other protocols, the user must provide the starting point for concurrency reduction as a signal-level specification. Section 4.2 and Section 4.3 present an extensive overview of how do this. They provide a survey of many protocols that have been used in the literature, showing how the constraints of each can be expressed by the model. In particular, Section 4.3 shows how a target data path imposes constraints on control. It covers the constraints necessary to ensure integrity of any data transferred during channel

communications. This section provides a survey of many protocols that have been used in the literature, showing how the constraints of each can be expressed by the model. The survey is not meant to describe every possible protocol. Rather it is meant to cover a broad enough range of examples from the literature to demonstrate how the choice of target data path impacts the constraints on control. This section uses a simple one-place buffer as a running example. Section 4.4 shows how to apply the same model to more complex examples. Section 4.5 shows how the model could form the basis for an extendible library of protocols for CAD tools.

The reader who is concerned only with what is currently automated in the tool can skip Section 4.2 (except for the first example), Section 4.3, and Section 4.5 However, these sections (and indeed this entire chapter) are still useful to anyone who extends the front end in the future to implement these techniques to support more of these protocols. Furthermore, until then, this section is also useful to the user who bypasses the front end, directly inputting the most-concurrent, signal-level specification that still meets the constraints of the user's chosen protocol. For such a user, these sections outline a systematic approach to setting up the starting point for the concurrency reduction techniques of Chapter 5 and Chapter 6.

4.1 Active vs. Passive

As Section 3.1.1 states, when two processes (for example, a circuit and its environment) communicate over a channel, one process is active and initiates the communication. The other process is passive and simply responds to the actions of the active process. For example, consider the following channel-level VHDL. There is no data transfer in this example. The communication actions represent pure synchronization in this example.

```
architecture behavior of AP is
  signal C : channel := init_channel(
    sender => timing(rise_min => 3, rise_max => 4,
                     fall_min => 1, fall_max => 2),
    receiver => timing(rise_min => 4, rise_max => 9,
                    fall_min => 1, fall_max => 2));
begin
  A : process
  begin
    send(C);
```

```
  end process A;
  P : process
  begin
    receive(C);
  end process P;
end behavior;
```

For example, for a push channel, A is the active process, and P is the passive process. In this case, following signal-level VHDL illustrates the sequence of assignments and guards that occur using four-phase handshaking.

```
architecture behavior of handAP is
  signal Creq, Cack :  std_logic;
begin
  A : process
  begin
    assign(Creq, '1', 3, 4);
    guard(Cack, '1');
    assign(Creq, '0', 1, 2);
    guard(Cack, '0');
  end process A;
  P : process
  begin
    guard(Creq, '1');
    assign(Cack, '1', 4, 9);
    guard(Creq, '0');
    assign(Cack, '0', 1, 2);
  end process P;
end behavior;
```

Figure 4.1 shows the corresponding TEL structure.

4.2 Sequencers

Given the channel-level TEL structure of Chapter 3, the initial expander that this dissertation presents targets a particular form of pure synchronization channel, using a four-phase, narrow protocol. For example, consider four channel communications in series. The tool that this dissertation presents currently sequences these

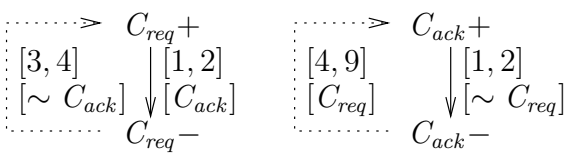

Figure 4.1. TEL structure for a four-phase expansion of a channel communication.

as shown in Figure 4.2. Note that for each four-phase communication, the first two phases complete before the next communication can begin. The return-to-zero transitions are free to be interleaved with the communications that follow them. This is the same sequencing that Cortadella et al. [23] use.

Many other sequencers are possible. The front-end that automates the initial expansion process currently targets only the above scheme. However, one can handle other protocols, by bypassing this front end and providing the constraints on reshuffling directly to the concurrency reduction engine. The remainder of this section presents some examples of how to do this.

Prosser et al. [63] present a study of many possible sequencers. They illustrate each possibility using a sequencer with the following interface behavior. A communication on a passive port L initiates the step of an algorithm for which this sequencer is responsible. The sequencer initiates a communication on port X to do the work of this step of the algorithm. Finally, the sequencer initiates a communication on port R to pass control to the next step of the algorithm.

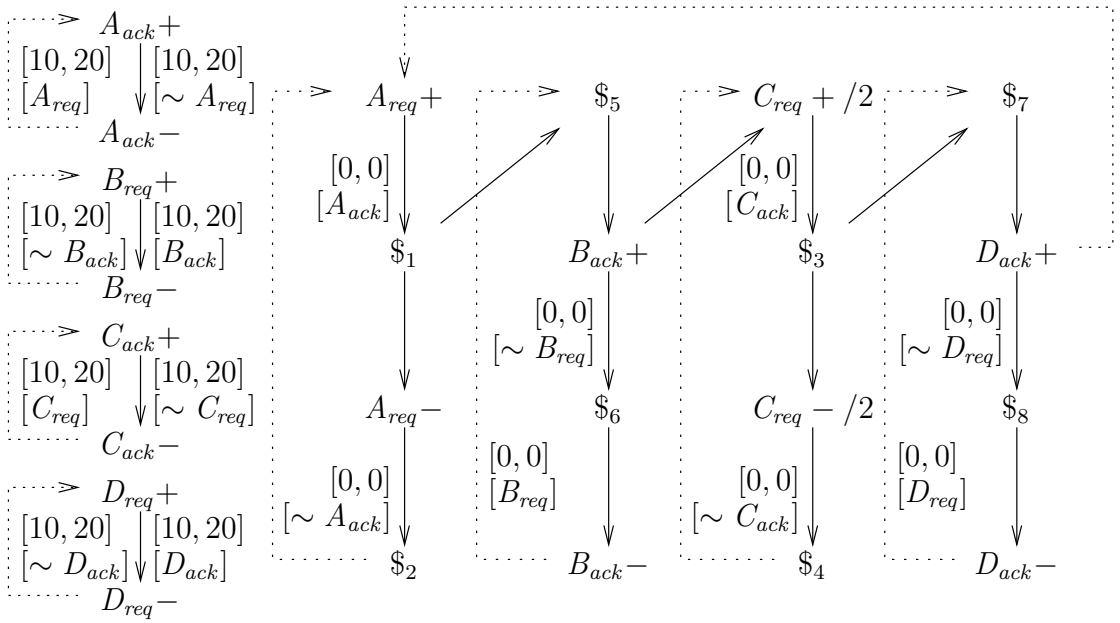

Figure 4.2. The sequencing currently targeted by the automatic tool that this dissertation presents.

Let $a \prec b$ mean that a must occur before b. Let a^{-1} stand for the occurrence of a in the previous iteration. In the following, $[e]$ is shorthand for the step in the protocol at which the process waits for the guard (Definition 2.3 on page 29) to become satisfied. For the sequencer, L is passive and the other ports are active. This gives the following constraints:

$$L_{ack}^{-1}- \prec [L_{req}] \prec L_{ack}+ \prec [\sim L_{req}] \prec L_{ack}- \tag{4.1}$$

$$\left[\sim R_{ack}^{-1}\right] \prec R_{req}+ \prec [R_{ack}] \prec R_{req}- \prec [\sim R_{ack}] \tag{4.2}$$

$$\left[\sim X_{ack}^{-1}\right] \prec X_{req}+ \prec [X_{ack}] \prec X_{req}- \prec [\sim X_{ack}] \tag{4.3}$$

The above constraints apply two both two-phase communication and also to four-phase communication. In the case of two-phase communication, each equation represents two communications. In the case of four-phase communication, each equation represents just one communication.

The two-phase sequencer of Prosser et al. [63] enforces the following additional constraints:

$$[L_{req}] \prec X_{req}+ \tag{4.4a}$$

$$[\sim L_{req}] \prec X_{req}- \tag{4.4b}$$

$$[X_{ack}] \prec R_{req}+ \tag{4.4c}$$

$$[\sim X_{ack}] \prec R_{req}- \tag{4.4d}$$

One can manually construct a TEL structure that represents these constraints. Each precedence ($\prec$) relationship leads to a rule in the TEL structure. The TEL structure of Figure 4.3 represents the constraints of Equations (4.1), (4.2), (4.3), and (4.4) simultaneously. In Figure 4.3, the protocol is unrolled such that each iteration of the TEL structure covers two iterations of the protocol.

Within four-phase sequences, Prosser et al. [63] classify the possibilities as broad, weak-broad, and narrow. For example, the sequencer of van Berkel [11] is a broad sequencer. It uses the following constraints:

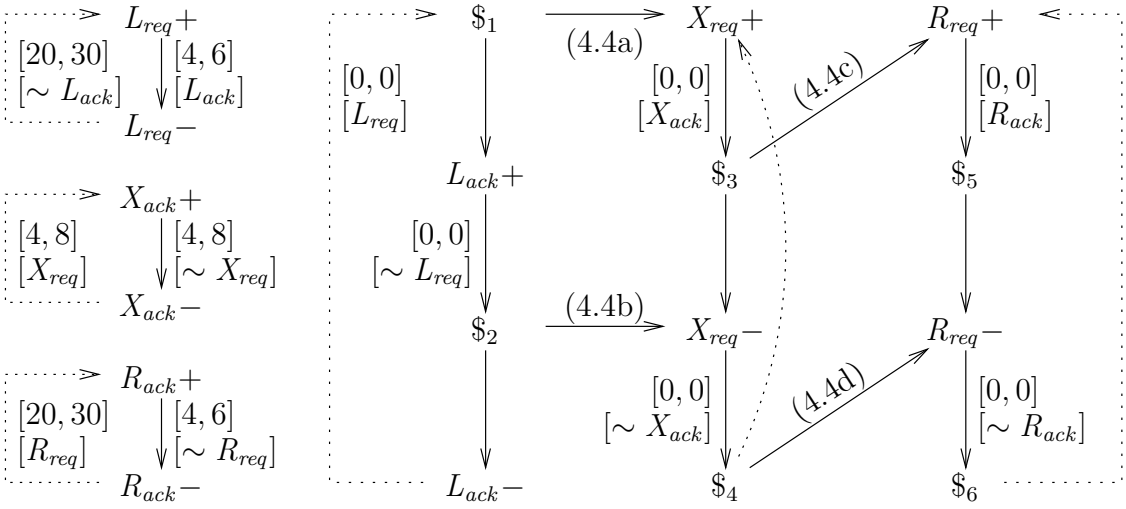

Figure 4.3. Constraints on two-phase sequencer.

$$[L_{req}] \prec X_{req}+ \tag{4.5a}$$

$$[\sim X_{ack}] \prec R_{req}+ \tag{4.5b}$$

$$[R_{ack}] \prec L_{ack}+ \tag{4.5c}$$

$$[\sim L_{req}] \prec R_{req}- \tag{4.5d}$$

$$[\sim R_{ack}] \prec L_{ack}- \tag{4.5e}$$

The TEL structure of Figure 4.4, adapted from Prosser et al. [63], represents the constraints of Equations (4.1), (4.2), (4.3), and (4.5) simultaneously.

The Winkel weak-broad sequencer uses the following constraints:

$$[L_{req}] \prec X_{req}+ \tag{4.6a}$$

$$[X_{ack}] \prec L_{ack}+ \tag{4.6b}$$

$$[\sim L_{ack}] \prec R_{req}+ \tag{4.6c}$$

$$[\sim L_{req}] \prec X_{req}- \tag{4.6d}$$

$$[\sim X_{ack}] \prec L_{ack}- \tag{4.6e}$$

$$\left[\sim R_{ack}^{-1}\right] \prec X_{req}+ \tag{4.6f}$$

The TEL structure of Figure 4.5, adapted from Prosser et al. [63], represents the constraints of Equations (4.1), (4.2), (4.3), and (4.6) simultaneously.

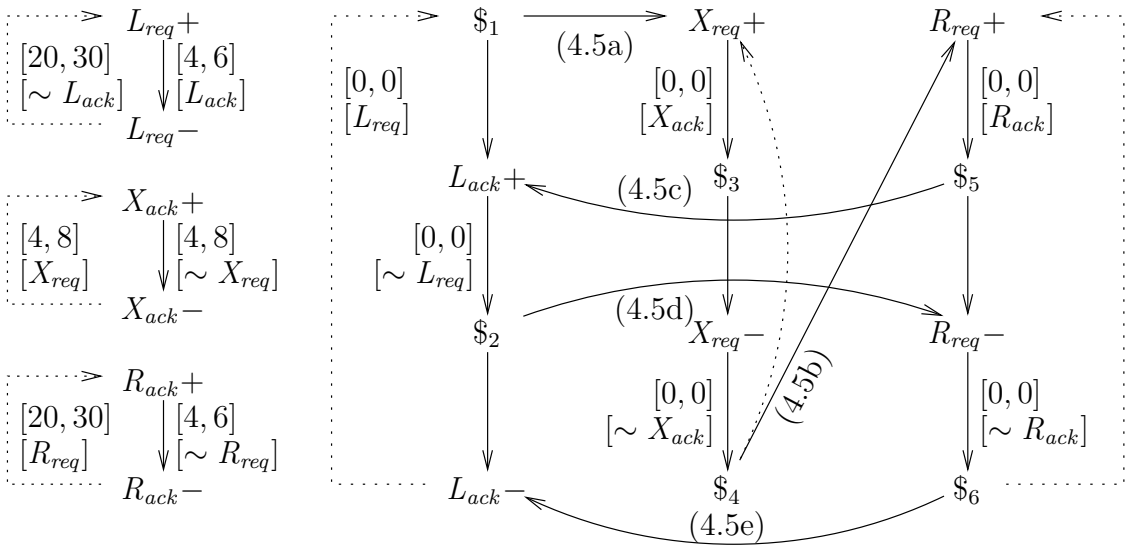

Figure 4.4. Constraints on the van Berkel sequencer.

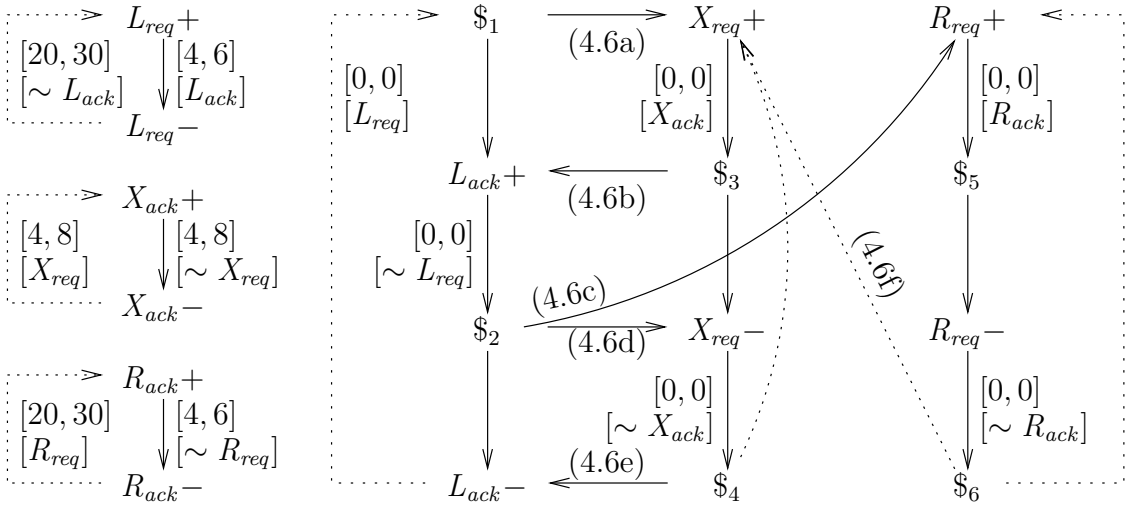

Figure 4.5. Constraints on the Winkel weak-broad sequencer.

The Brunvand narrow sequencer [16] uses the following constraints:

$$[L_{req}] \prec X_{req}+ \tag{4.7a}$$

$$[X_{ack}] \prec L_{ack}+ \tag{4.7b}$$

$$[X_{ack}] \prec R_{req}+ \tag{4.7c}$$

$$[\sim L_{req}] \prec X_{req}- \tag{4.7d}$$

$$[\sim X_{ack}] \prec L_{ack}- \tag{4.7e}$$

$$[\sim X_{ack}] \prec R_{req}- \tag{4.7f}$$

$$\left[\sim R_{ack}^{-1}\right] \prec X_{req}+ \tag{4.7g}$$

The TEL structure of Figure 4.6, adapted from Prosser et al. [63], represents the constraints of Equations (4.1), (4.2), (4.3), and (4.7) simultaneously.

4.3 Data Constraints

The front-end that automates the initial expansion process currently targets pure synchronization channels, using one, fixed, four-phase protocol in which the return-to-zero actions are assumed to be free to be interleaved with other communications. However, one can handle other protocols, by bypassing this front end and providing the constraints on reshuffling directly to the concurrency reduction

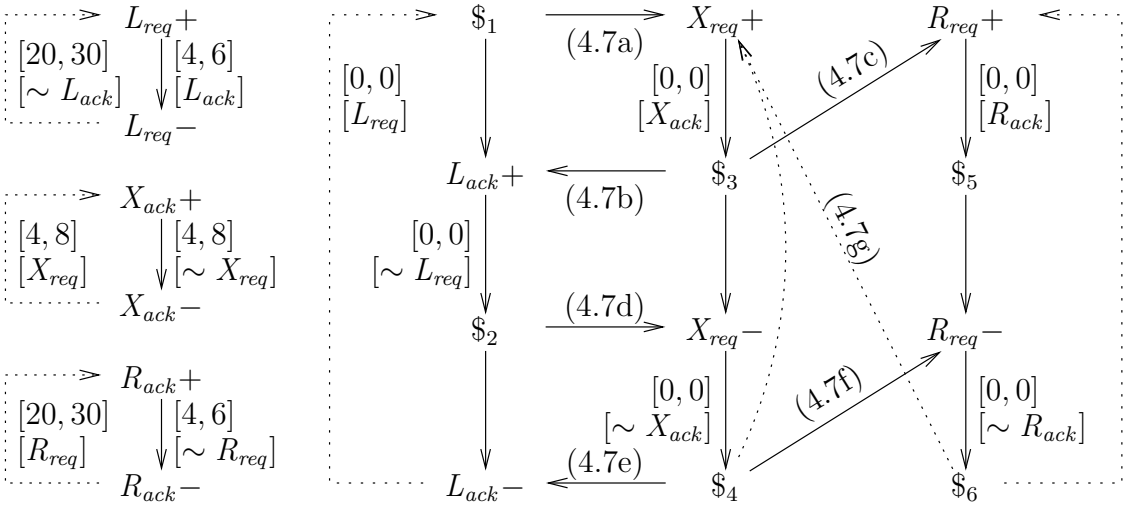

Figure 4.6. Constraints on the Brunvand narrow sequencer.

engine. This section presents an extensive survey of how to do this, focusing on how the target data path imposes constraints on control.

The TEL structures in this section were designed by hand to meet the constraints imposed by each target data path. The reader who is interested only in what is currently automated in the tool can skip this section. However, this section (and indeed this entire chapter) is still useful to anyone who extends the front end in the future to implement these techniques. Furthermore, until then, this section is also useful to the user who bypasses the front end, directly inputting the most-concurrent, signal-level specification that still meets the constraints of the user's chosen protocol. For such a user, this section shows how the choice of target data path impacts the constraints on control for a one-place buffer. Furthermore, this section outlines a systematic approach to incorporating these constraints into a signal-level TEL structure that captures the constraints of the chosen protocol for that buffer.

If channel communication is used not merely for synchronization, but also for data transfer, the choice of data-path style imposes constraints on the controller and *vice versa*. This section considers such constraints in the context of a buffer. Section 4.4 extends the model to include more complicated constructs. A simplified buffer process is shown below.

```
buf :  process
begin
  receive(L, x);
  send(R, x);
end process buf;
```

By default, the tool that this dissertation presents assumes that an otherwise unconstrained channel is a push type channel. So, if the specification does not specify otherwise, L above is passive and R above is active. However, the specification could use the declarations of Section 3.1.1 to declare that L should be active and R should be passive. In this case, because of the `receive` on L and the `send` on R, both channels would be pull type channels.

In any case, the examples in this chapter will consist of FIFO queues that are constructed by tiling multiple, identical instances of a buffer process together. This

requires the buffer process to have one active port and one passive port. One could instead alternate between two types of buffers. In one type, both ports would be active, and in the other type, both ports would be passive. However, this chapter does not include examples of such buffers.

The following subsections consider two main approaches for handling data. Section 4.3.1 considers the case in which control and data path are two separate processes. Section 4.3.2 considers the case in which control and data are unified. Both sections use four-phase handshaking for purposes of illustration.

4.3.1 Separate Control and Data Path

The interface to the control portion of the above buffer is shown in Figure 4.7(a) for L passive and R active and in Figure 4.7(b) for L active and R passive. This is similar to the six-terminal sequencer model of Prosser et al. [63]. The variable x has been replaced with an additional communication on the channel X. The X channel connects the control and data path processes. Intuitively, X_{req} tells the data path to store the incoming value from channel L into variable x, and X_{ack} indicates that this has been accomplished. However, the details of what each event on the X channel means depend on the implementation of the data path. Regardless of the choice of data path, each of the three channels must obey its protocol.

The constraints of Equation (4.3) of Section 4.2 applies to the buffer. If L is passive and R is active, the Equations (4.1) and (4.2) from Section 4.2 also apply

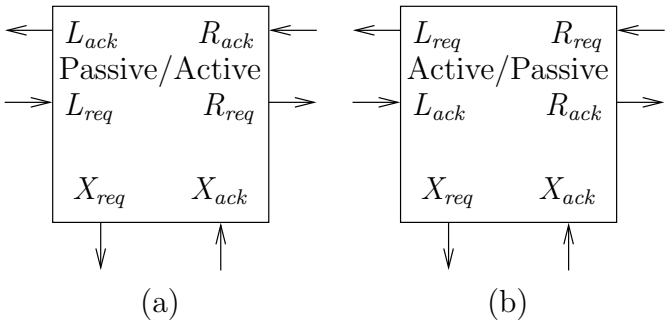

Figure 4.7. Generic interfaces to the control portion of a buffer. In part (a), L is passive and R is active. In part (b), L is active and R is passive.

to the buffer. If instead, L is active and R is passive, the constraints on L and R become

$$\left[\sim L_{ack}^{-1}\right] \prec L_{req}+ \prec [L_{ack}] \prec L_{req}- \prec [\sim L_{ack}] \tag{4.8}$$

$$R_{ack}^{-1}- \prec [R_{req}] \prec R_{ack}+ \prec [\sim R_{req}] \prec R_{ack}- \tag{4.9}$$

The above constraints apply two both two-phase communication and also to four-phase communication. In the case of two-phase communication, each equation represents two communications. In the case of four-phase communication, each equation represents just one communication.

The remainder of this section considers several possible implementations for the data path and discusses the additional constraints that each implementation imposes on the control.

Consider a two-phase, bundled-data, FIFO, using dual-edge-triggered flip-flops as shown in Figure 4.8(a). This figure shows both control and data path for two stages a FIFO buffer. The dual-edge-triggered flip-flops are used as two-phase, transition-sensitive storage elements. Any transition on the *clk* input causes the current value of D to overwrite the value of Q.

The delay element between X_{req} and X_{ack} must conservatively match the *clk* to Q delay of the flip-flop. In this model, it must also be at least as long as the required hold time of the flip flop. The *clk* to Q delay and the hold time are technically two separate parameters of the flip-flop. An optimization would be to use separate bundling delays to model these two parameters. However, in this simplified model, the single bundling delay between X_{req} and X_{ack} is set to conservatively match the maximum of the two parameters. Thus X_{ack} indicates both that a new datum is available at the Q output of the register and also that it is safe to change the D input to the register.

The delay element between R_{req} and L_{req} must conservatively match the processing delay of the combinational logic between stages. To consider the impact on control assume a controller in which L is passive and R is active. To ensure that

the incoming data from the L channel are stable before X_{req} clocks the register, the protocol must enforce the following:

$$[L_{req}] \prec X_{req}+ \tag{4.10a}$$

$$[\sim L_{req}] \prec X_{req}- \tag{4.10b}$$

Furthermore, this datum must be available at the Q output of the register before the buffer notifies the R channel that it is available. In this model, X_{ack} is the indication that a *clk* to Q delay has passed and the datum is available. Therefore, the protocol must enforce the following:

$$[X_{ack}] \prec R_{req}+ \tag{4.10c}$$

$$[\sim X_{ack}] \prec R_{req}- \tag{4.10d}$$

The datum must also be safely stored before the buffer notifies the L channel that it may overwrite the datum. In this model, X_{ack} also indicates that a hold time has passed, and it is safe to change the D input to the register. Thus, the protocol must enforce the following:

$$[X_{ack}] \prec L_{ack}+ \tag{4.10e}$$

$$[\sim X_{ack}] \prec L_{ack}- \tag{4.10f}$$

This is necessary, because in this push protocol, L_{ack} notifies the preceding stage that it is free to change L_{data}. Finally, the receiver on the other end of the R channel must receive this datum before the buffer overwrites the contents of the data-path register. This requires the following:

$$[R_{ack}] \prec X_{req}- \tag{4.10g}$$

$$\left[\sim R_{ack}^{-1}\right] \prec X_{req}+ \tag{4.10h}$$

One can manually construct a TEL structure to represent these constraints. Each precedence ($\prec$) relationship becomes a rule in the TEL structure. For example, the TEL structure of Figure 4.8(b) represents the constraints of Equations 4.1, 4.2, and 4.3 plus the data and safety constraints listed for this protocol simultaneously.

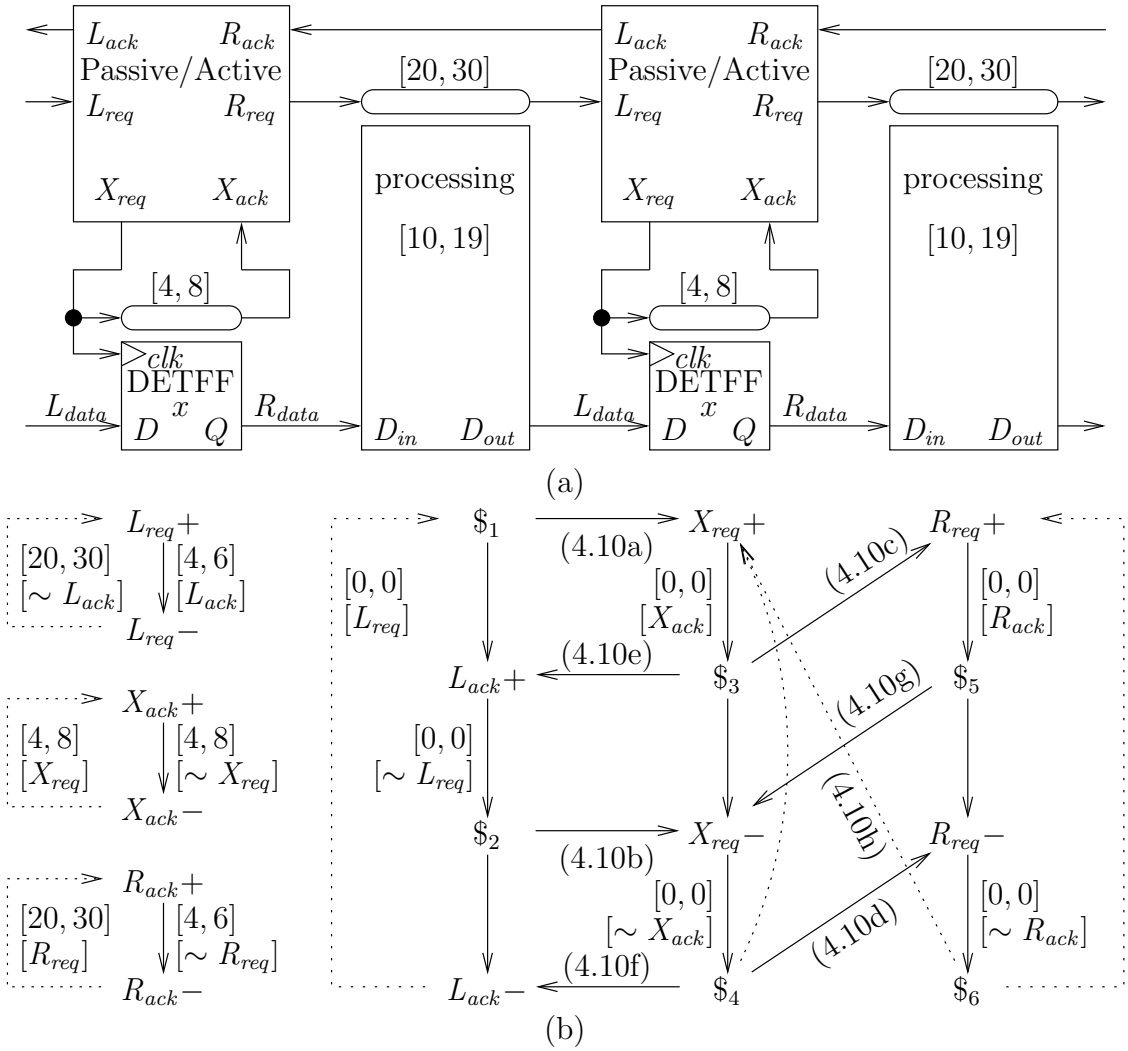

Figure 4.8. Two-phase, bundled-data, push FIFO using dual-edge-triggered flip-flops. Part (a) shows a schematic of the control and data path. Part (b) shows a TEL structure for the constraints on control.

In this TEL structure, the two-phase protocol has been unrolled such that each iteration of the TEL structure of Figure 4.8 covers two iterations of the protocol. Except where indicated, each rule has timing bounds $[1, 2]$. Alternatively, one could also put the constraints of each four phase protocol (each of 4.1, 4.2, and 4.3) into its own separate connected component of the TEL structure and then use level expressions to express the data constraints. However, the format of Figure 4.8 is more uniform in that each constraint is represented as a rule in the TEL structure.

Now consider a four-phase, bundled-data implementation, using a simple normally transparent latch, as shown in Figure 4.9(a). This figure shows both control and data path for two stages a FIFO buffer. The latches are level-sensitive. The value of the C input determines whether there is a combinational path from the D input to the Q output. (Some gate libraries would call the control input G instead of C.) When C is active, Q simply follows D. When C is inactive, Q holds its current value. In Figure 4.9(a), there is an inverting bubble in front of the C input to the latch. Therefore $C = \neg X_{req}$. When X_{req} is low, the latch is transparent. When X_{req} is high, the latch is opaque.

The delay element between X_{req} and X_{ack} must conservatively match the delay from $X_{req}+$ until the latch is completely opaque. The delay element between R_{req} and L_{req} has separate delay ranges for rising and falling delay. The rising delay must conservatively match the processing delay of the combinational logic between stages. The falling delay must conservatively match the recovery (return-to-zero) delay of the combinational logic. This is essentially the data path used in [28]. To consider the impact on control, first assume a controller in which L is passive and R is active. To ensure that there is a new, valid datum to send before the `send` operation on the R channel commences, the protocol must enforce the following:

$$[L_{req}] \prec R_{req}+ \tag{4.11a}$$

This datum must also enter the data-path latch before the latch becomes opaque. Thus, the protocol must enforce the following:

$$[L_{req}] \prec X_{req}+ \tag{4.11b}$$

To ensure that the latch has stored the datum before the L channel overwrites it, the protocol must enforce the following:

$$[X_{ack}] \prec L_{ack}+ \tag{4.11c}$$

The latch must remain opaque until the recipient at the other end of the R channel has received the data. Thus, the protocol must enforce the following:

$$[R_{ack}] \prec X_{req}- \tag{4.11d}$$

Finally, the latch must return to its transparent state to let new data through before the next `send` operation on channel R commences. This requires the following:

$$\left[\sim X_{ack}^{-1}\right] \prec R_{req}+ \tag{4.11e}$$

One can manually construct a TEL structure to represent these constraints. Each precedence ($\prec$) relationship becomes a rule in the TEL structure. The TEL structure of Figure 4.9(b) represents the constraints of Equations 4.1, 4.2, and 4.3 plus the data and safety constraints listed for this protocol simultaneously. The correspondence is straightforward, except in the case of Equation (4.11e) Conceptually, this would indicate an initially-marked rule from $\$_4$ to $R_{req}+$. However, for the TEL structure that represents these constraints to be one-safe (Definition 2.6 on page 32), this rule must be redirected to $\$_1$.

Now consider the case in which L is active and R is passive, as shown in Figure 4.10(a). To ensure that there is a new, valid datum to send before the `send` operation on the R channel commences, the protocol must enforce the following:

$$[L_{ack}] \prec R_{ack}+ \tag{4.12a}$$

This datum must also enter the data-path latch before the latch becomes opaque. Thus, the protocol must enforce the following:

$$[L_{ack}] \prec X_{req}+ \tag{4.12b}$$

To ensure that the latch has stored the datum before the L channel overwrites it, the protocol must enforce the following:

$$[X_{ack}] \prec L_{req}- \tag{4.12c}$$

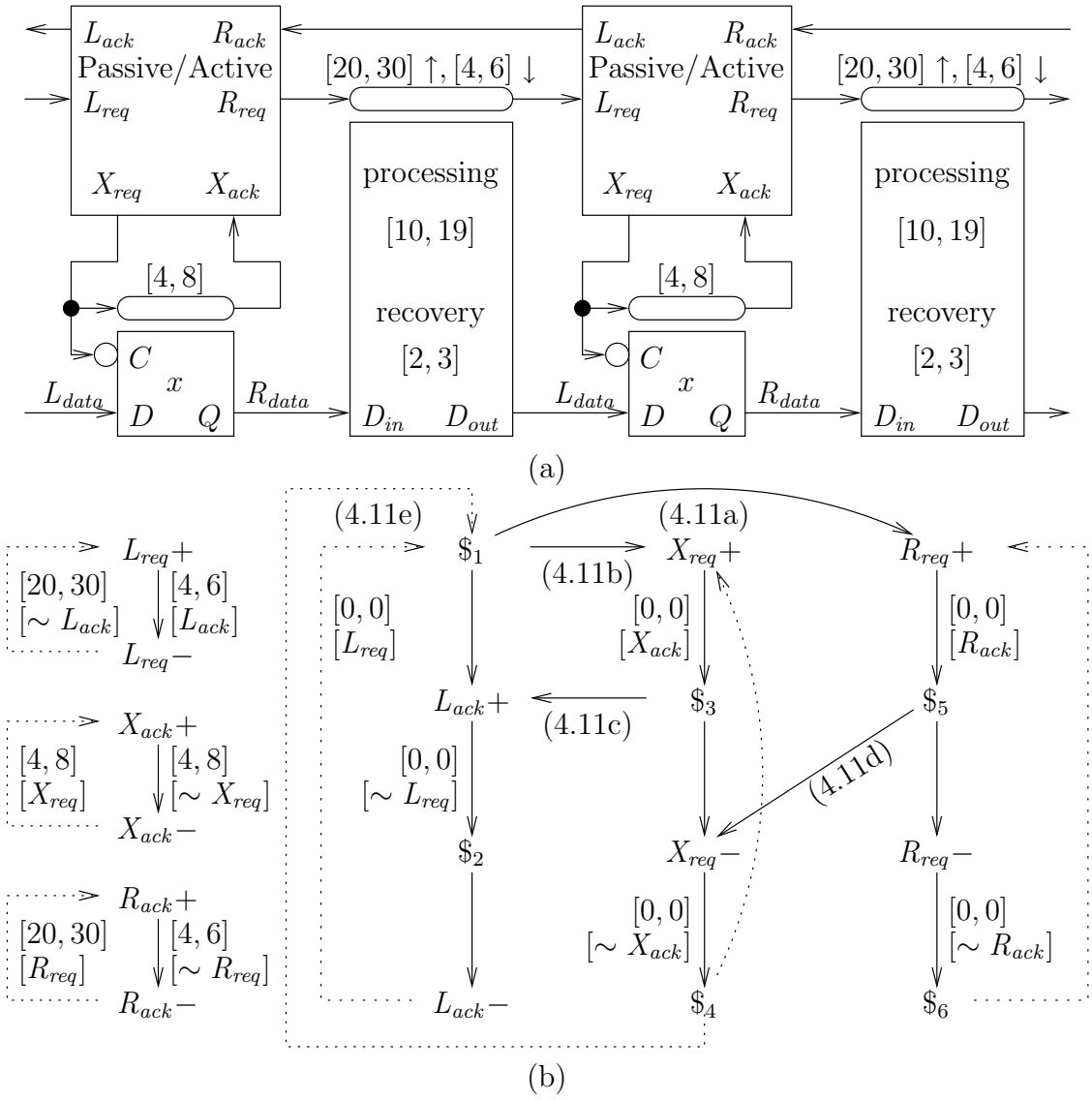

Figure 4.9. Four-phase, bundled-data, push FIFO using normally transparent latches. Part (a) shows a schematic of the control and data path. Part (b) shows a TEL structure for the constraints on control.

The latch must remain opaque until the recipient at the other end of the R channel has received the data. Thus, the protocol must enforce the following:

$$[\sim R_{req}] \prec X_{req}- \tag{4.12d}$$

Finally, the latch must become transparent again to let new data through before the next `send` operation on channel R commences. Hence, the protocol must enforce the following:

$$\left[\sim X_{ack}^{-1}\right] \prec R_{ack}+ \tag{4.12e}$$

The TEL structure of Figure 4.10(b) represents all these constraints simultaneously.

Now consider the normally opaque latches of Figure 4.11(a). In Figure 4.11(a), there is no inverting bubble in front of the C input to the latch. Therefore, $C = X_{req}$. When X_{req} is low, the latch is opaque. When X_{req} is high, the latch is transparent. The delay element between X_{req} and X_{ack} must conservatively match the delay from $X_{req}+$ until the latch is completely transparent. To consider the impact on control, first assume a controller in which L is passive and R is active. To ensure that there is a new, valid datum to store in the x register, the protocol must enforce the following:

$$[L_{req}] \prec X_{req}+ \tag{4.13a}$$

This datum must also enter the data-path latch before the buffer notifies the R channel that it is available. Thus, the protocol must enforce the following:

$$[X_{ack}] \prec R_{req}+ \tag{4.13b}$$

To ensure that the latch has stored the datum before the L channel overwrites it, the protocol must enforce the following:

$$[\sim X_{ack}] \prec L_{ack}+ \tag{4.13c}$$

Finally, the latch must remain opaque until the recipient at the other end of the R channel has received the data. Thus, the protocol must enforce the following:

$$\left[R_{ack}^{-1}\right] \prec X_{req}+ \tag{4.13d}$$

The TEL structure of Figure 4.11(b) represents all these constraints simultane-

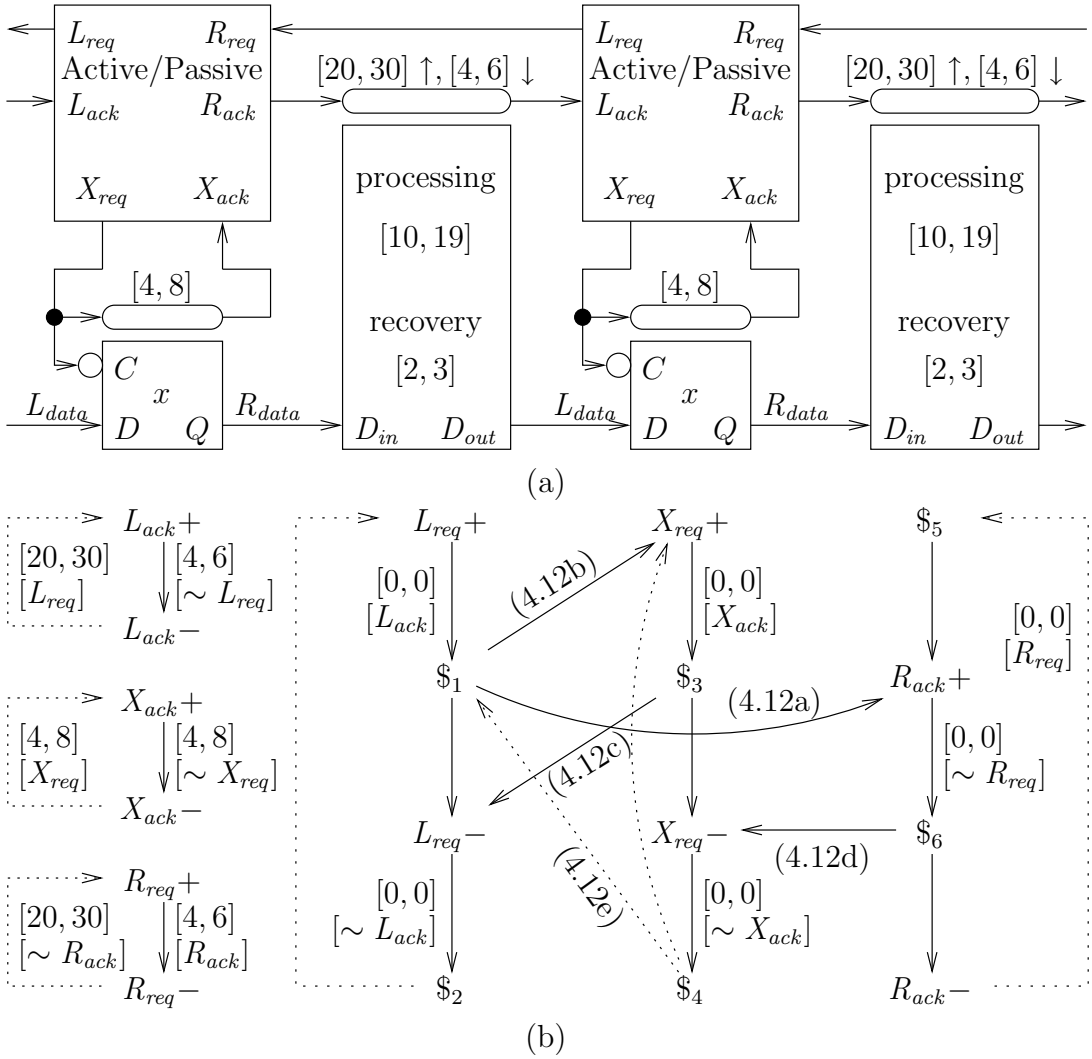

Figure 4.10. Four-phase, bundled-data pull FIFO using normally transparent latches. Part (a) shows a schematic of the control and data path. Part (b) shows a TEL structure for the constraints on control.

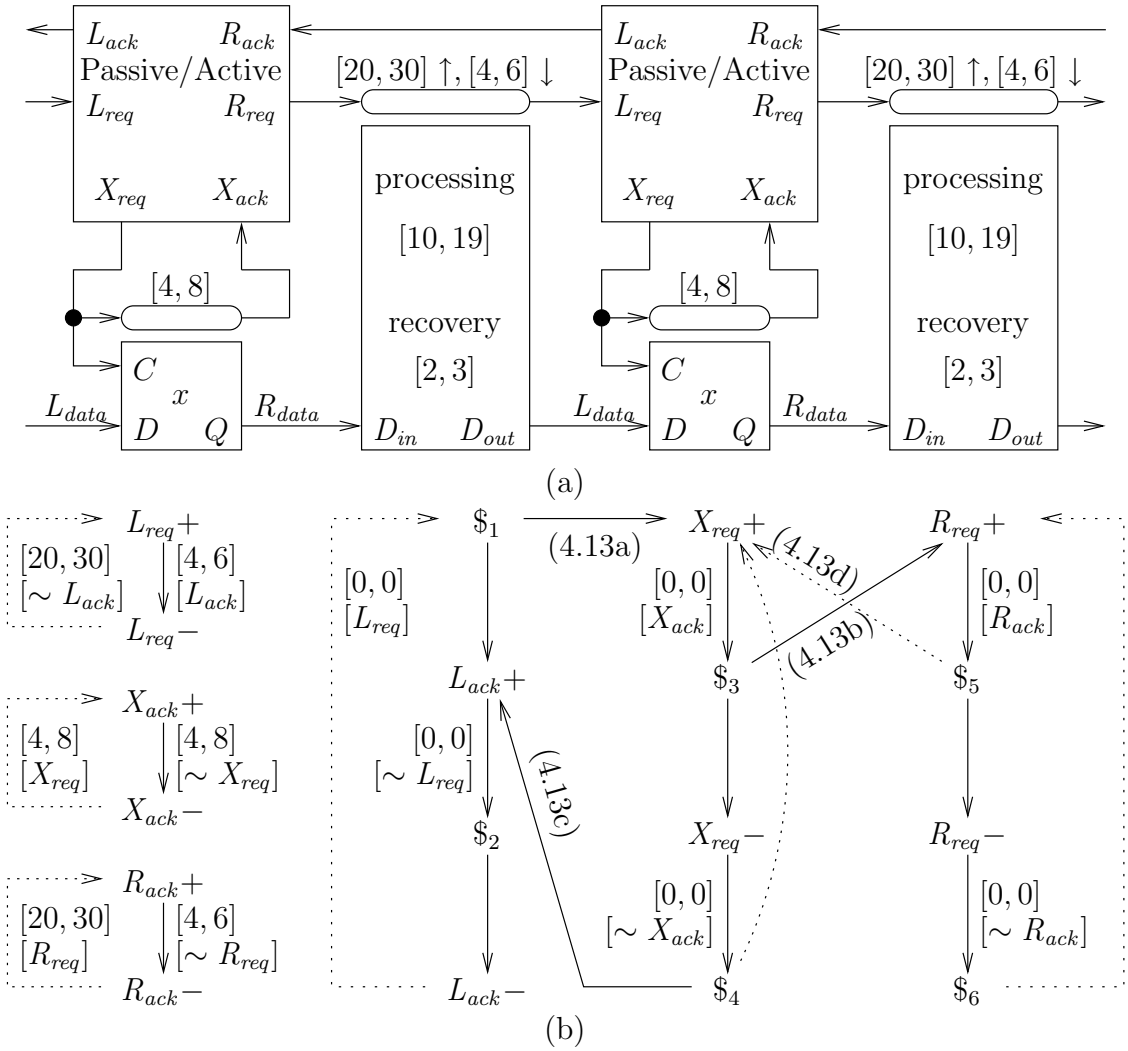

Figure 4.11. Four-phase, bundled-data push FIFO using normally opaque latches. Part (a) shows a schematic of the control and data path. Part (b) shows a TEL structure for the constraints on control.

ously.

Now consider the case in which L is active and R is passive, as shown in Figure 4.12(a). To ensure that there is a new, valid datum to store in the x latch, the protocol must enforce the following:

$$[L_{ack}] \prec X_{req}+ \tag{4.14a}$$

This datum must also enter the data-path latch before the buffer notifies the R channel that it is available. Thus, the protocol must enforce the following:

$$[X_{ack}] \prec R_{ack}+ \tag{4.14b}$$

To ensure that the latch has stored the datum before the L channel overwrites it, the protocol must enforce the following:

$$[\sim X_{ack}] \prec L_{req}- \tag{4.14c}$$

Finally, the latch must remain opaque until the recipient at the other end of the R channel has received the data. Thus, the protocol must enforce the following:

$$\left[\sim R_{req}^{-1}\right] \prec X_{req}+ \tag{4.14d}$$

The TEL structure of Figure 4.12(b) represents all these constraints simultaneously.

Now consider a data path using rising-edge-triggered flip-flops, as shown in Figure 4.13(a). This is essentially the data path used in [54, 37], except that Figure 4.13(a) uses a bundling delay to generate X_{ack}, instead of completion detection. This bundling delay between X_{req} and X_{ack} must conservatively match the *clk* to Q delay of the flip-flop. In this model, it must also be at least as long as the required hold time of the flip flop. The *clk* to Q delay and the hold time are technically two separate parameters of the flip-flop. An optimization would be to use separate bundling delays to model these two parameters. However, in this simplified model, the single bundling delay between X_{req} and X_{ack} is set to conservatively match the maximum of the two parameters. Thus X_{ack} indicates both that a new datum is

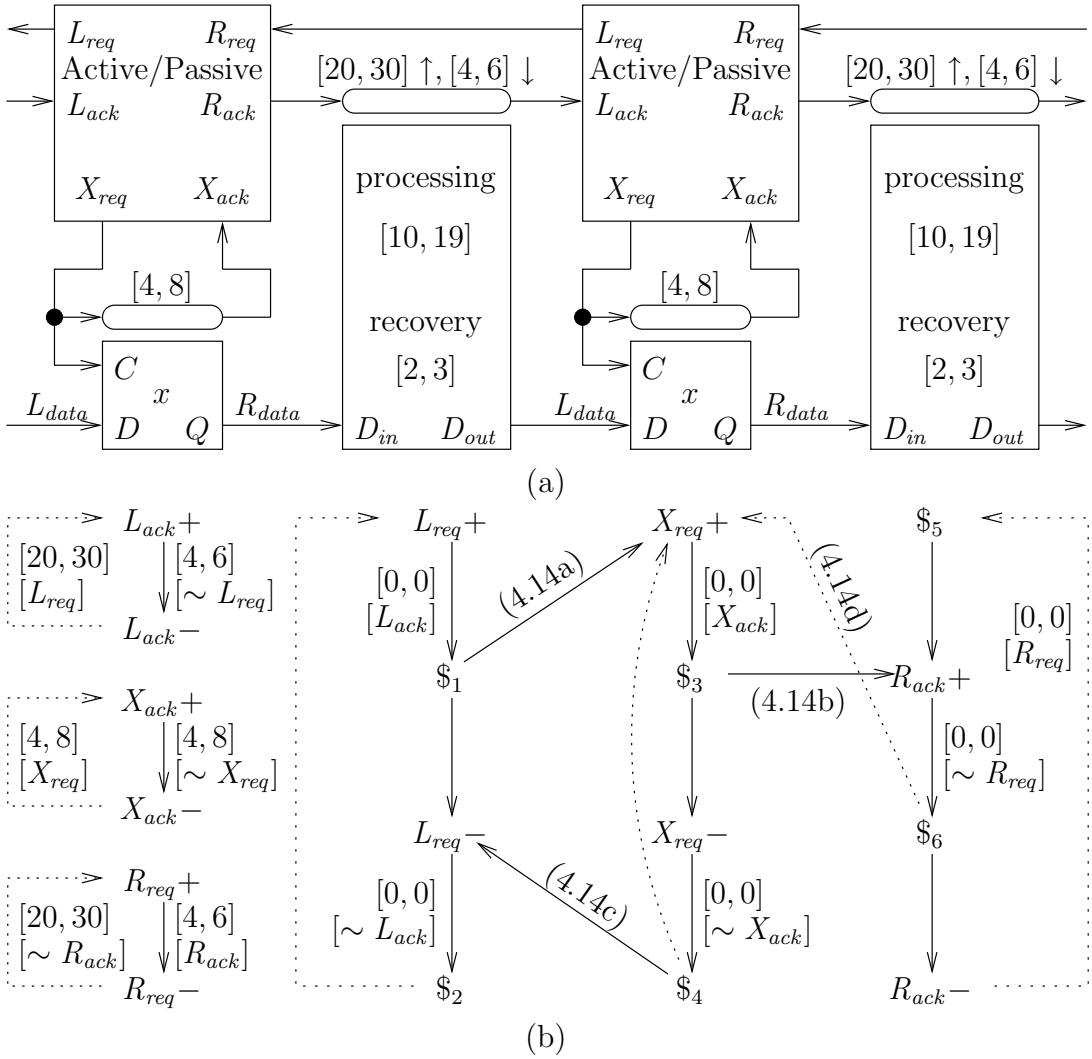

Figure 4.12. Four-phase, bundled-data pull FIFO using normally opaque latches. Part (a) shows a schematic of the control and data path. Part (b) shows a TEL structure for the constraints on control.

available at the Q output of the register and also that it is safe to change the D input to the register.

To consider the impact on control, first assume a controller in which L is passive and R is active. To ensure that the incoming data from the L channel are stable before X_{req} clocks the register, the protocol must enforce the following:

$$[L_{req}] \prec X_{req}+ \tag{4.15a}$$

Furthermore, this datum must be available at the Q output of the register before the buffer notifies the R channel that it is available. In this model, X_{ack} is the indication that a *clk* to Q delay has passed and the datum is available. Therefore, the protocol must enforce the following:

$$[X_{ack}] \prec R_{req}+ \tag{4.15b}$$

The datum must also be safely stored before the buffer notifies the L channel that it may overwrite the datum. In this model, X_{ack} also indicates that a hold time has passed, and it is safe to change the D input to the register. Thus, the protocol must enforce the following:

$$[X_{ack}] \prec L_{ack}+ \tag{4.15c}$$

This is necessary, because in this push protocol, $L_{ack}+$ notifies the preceding stage that it is free to change L_{data}. Finally, the receiver on the other end of the R channel must receive this datum, before the buffer overwrites the contents of the data-path register. This requires the following:

$$\left[R_{ack}^{-1}\right] \prec X_{req}+ \tag{4.15d}$$

The TEL structure of Figure 4.13(b) represents all these constraints simultaneously.

Now consider the case in which L is active and R is passive, as shown in Figure 4.14(a). To ensure that the incoming data from the L channel are stable before X_{req} clocks the register, the protocol must enforce the following:

$$[L_{ack}] \prec X_{req}+ \tag{4.16a}$$

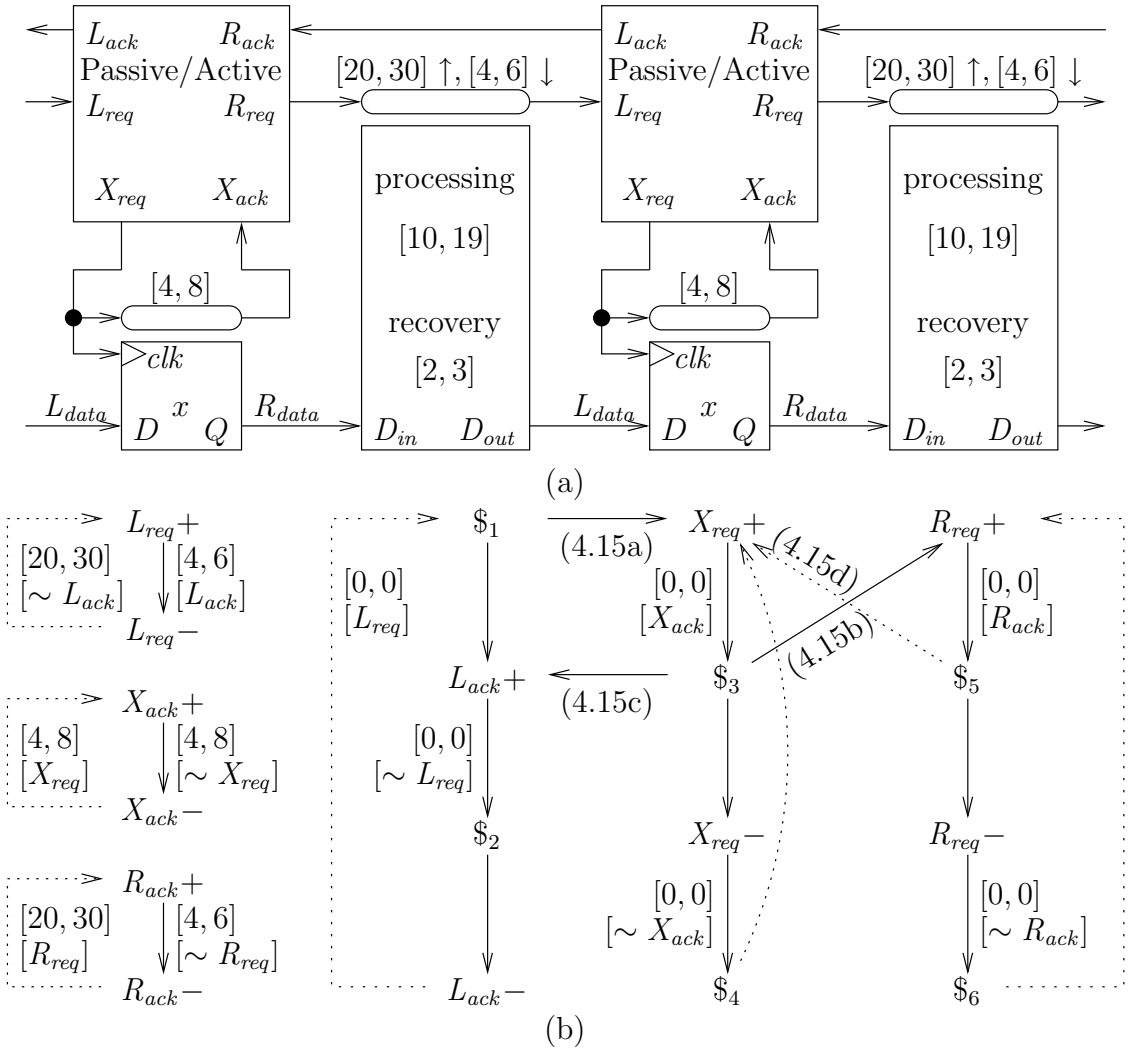

Figure 4.13. Four-phase, bundled-data, push FIFO using edge-triggered flip-flops. Part (a) shows a schematic of the control and data path. Part (b) shows a TEL structure for the constraints on control.

Furthermore, this datum must be available at the Q output of the register before the buffer notifies the R channel that it is available. In this model, X_{ack} is the indication that a *clk* to Q delay has passed and the datum is available. Therefore, the protocol must enforce the following:

$$[X_{ack}] \prec R_{ack}+ \tag{4.16b}$$

The datum must also be safely stored before the buffer notifies the L channel that it may overwrite the datum. In this model, X_{ack} also indicates that a hold time has passed, and it is safe to change the D input to the register. Thus, the protocol must enforce the following:

$$[X_{ack}] \prec L_{req}- \tag{4.16c}$$

This is necessary, because in this pull protocol, $L_{req}-$ notifies the preceding stage that it is free to change L_{data}. Finally, the receiver on the other end of the R channel must receive this datum before the buffer overwrites the contents of the data-path register. This requires the following:

$$\left[\sim R_{req}^{-1}\right] \prec X_{req}+ \tag{4.16d}$$

The TEL structure of Figure 4.14(b) represents all these constraints simultaneously.

Even a design that uses data encoding can partition control and data path. For example, Burns [20] defines the data-path components shown in Figure 4.15 for encoding and decoding data. Component *Output* has the following interface behavior. The environment raises req. *Output* responds by raising x_n, where n is the current value on the single rail input x. The environment then lowers req, and *Output* responds by lowering x_n. For correct operation of the *Output* unit, x must not change while req is high. Component *Latch* has the same behavior, except that *Latch* allows x to change as soon as it has raised x_n. The *Receive* component has the following interface behavior. When the environment raises x_n, *Receive* latches the value n into the variable x. It then raises *ack*. The environment then lowers x_n, and *Receive* responds by lowering *ack*. The *Passive* component has the following

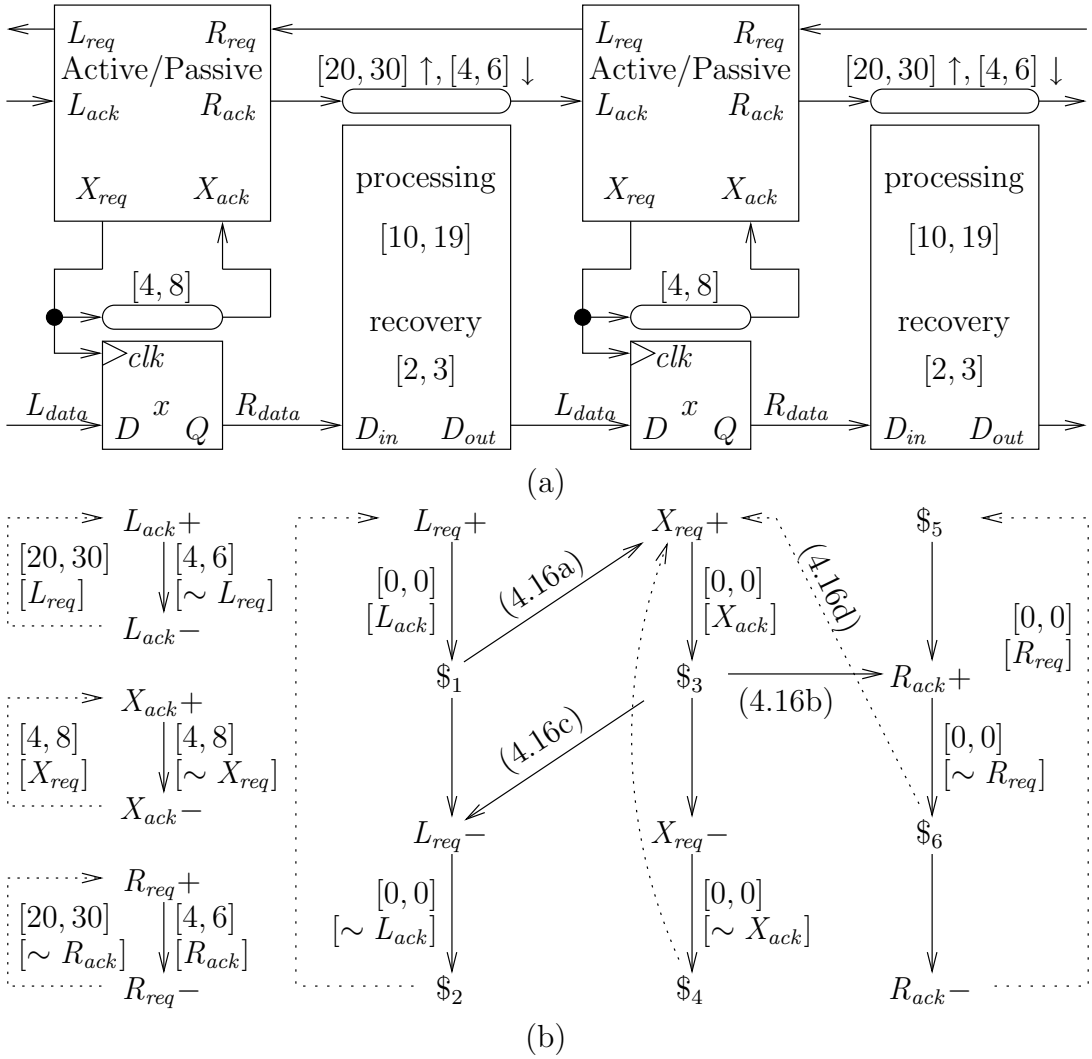

Figure 4.14. Four-phase, bundled-data, pull FIFO using edge-triggered flip-flops. Part (a) shows a schematic of the control and data path. Part (b) shows a TEL structure for the constraints on control.

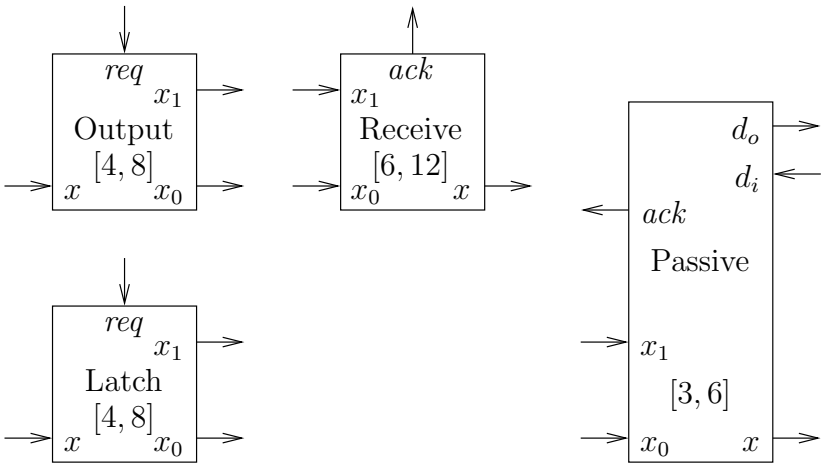

Figure 4.15. Data-path components from [20].

interface behavior. The environment raises x_n, and *Passive* responds by raising d_o. Then, the environment raises d_i. *Passive* then latches the value n into the variable x. Then, it raises *ack*. The environment then lowers x_n, and *Passive* responds by lowering d_o. The environment then lowers d_i, and *Passive* responds by lowering *ack*. Burns used these components to build several buffers, such as the push buffer in Figure 4.16(a). The control portion, the *pa-no-iso* process, uses a push protocol. Figure 4.16(a) shows two stages of the FIFO.

Data transfer in this FIFO operates as follows. When the stage on the left has a datum to send, it asserts its R_{req} signal, which is connected to the L_{req} input of the stage on the right. When the stage on the right is ready to accept the datum, it asserts its L_{ack} signal, which instructs the *Output* component on the left to convert the datum to dual-rail format. This dual-rail datum is received by the *Receive* component on the right, which converts it back to a signal-rail value x. Once it has done so, it asserts its *ack* signal which is connected to the R_{ack} input of the stage on the left. The stage on the left responds by lowering its R_{req} output, which lowers the L_{req} input to the stage on the right. This is the only way that the stage on the right can determine that the datum is available at the x output of its *Receive* component. Hence, to ensure that the stage on the right has a new datum to send before it notifies the next stage that a new datum is available, the protocol must

enforce the following:

$$[\sim L_{req}] \prec R_{req}+ \tag{4.17a}$$

The stage on the right responds to $L_{req}-$ by lowering its L_{ack} output. This tells the *Output* unit on the left to clear its x_n outputs. The *Receive* unit responds by lowering its *ack* output, which lowers the R_{ack} input to the stage on the left. This is the only way that the stage on the left can determine that it is now safe to change the x input to the *Output* unit on the left. Thus, to ensure that the next stage has successfully received the current datum before the next datum is received, the protocol must enforce the following:

$$\left[\sim R_{ack}^{-1}\right] \prec L_{ack}+ \tag{4.17b}$$

[20]. The TEL structure of Figure 4.16(b) represents all these constraints simultaneously.

Burns [20] also presents the above buffer modified such that L is active and R is passive, as shown in Figure 4.17(a). In this design, the buffer requests data from the L channel, and it can determine that it has received a datum when it receives an acknowledge. This means the protocol must enforce the following:

$$[L_{ack}] \prec R_{ack}+ \tag{4.18a}$$

To ensure that the next stage has successfully received the current datum before the next datum is received, the protocol must enforce the following [20]:

$$[R_{req}] \prec L_{req}+ \tag{4.18b}$$

The TEL structure of Figure 4.17(b) represents all these constraints simultaneously.

Burns [20] also presents a push buffer that requires an isochronic fork between control and the data path, as shown in Figure 4.18(a). In this design, the control portion of the buffer cannot determine that the data path has latched the datum

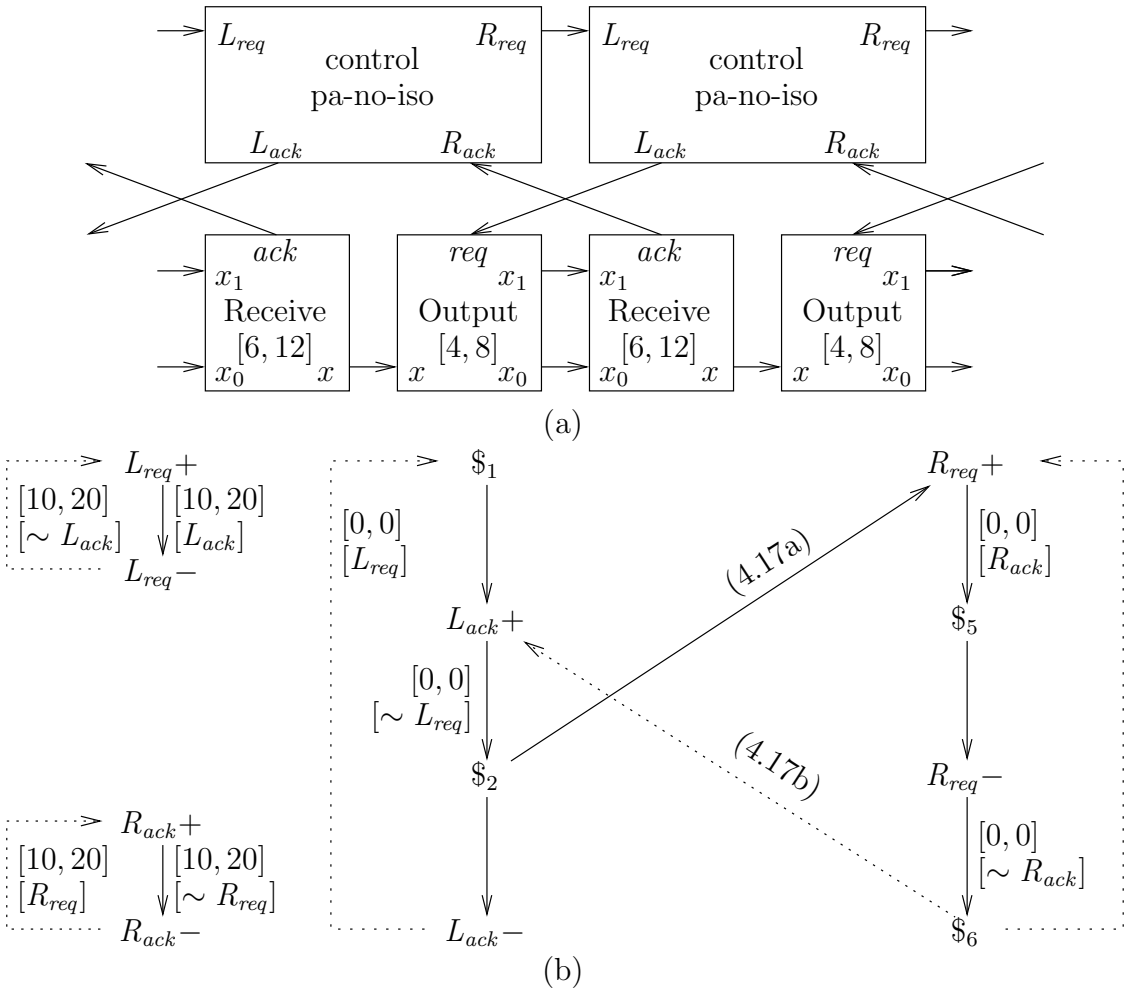

Figure 4.16. Push FIFO requiring no isochronic fork between control and data path [20]. Part (a) shows a schematic of the control and data path. Part (b) shows a TEL structure for the constraints on control.

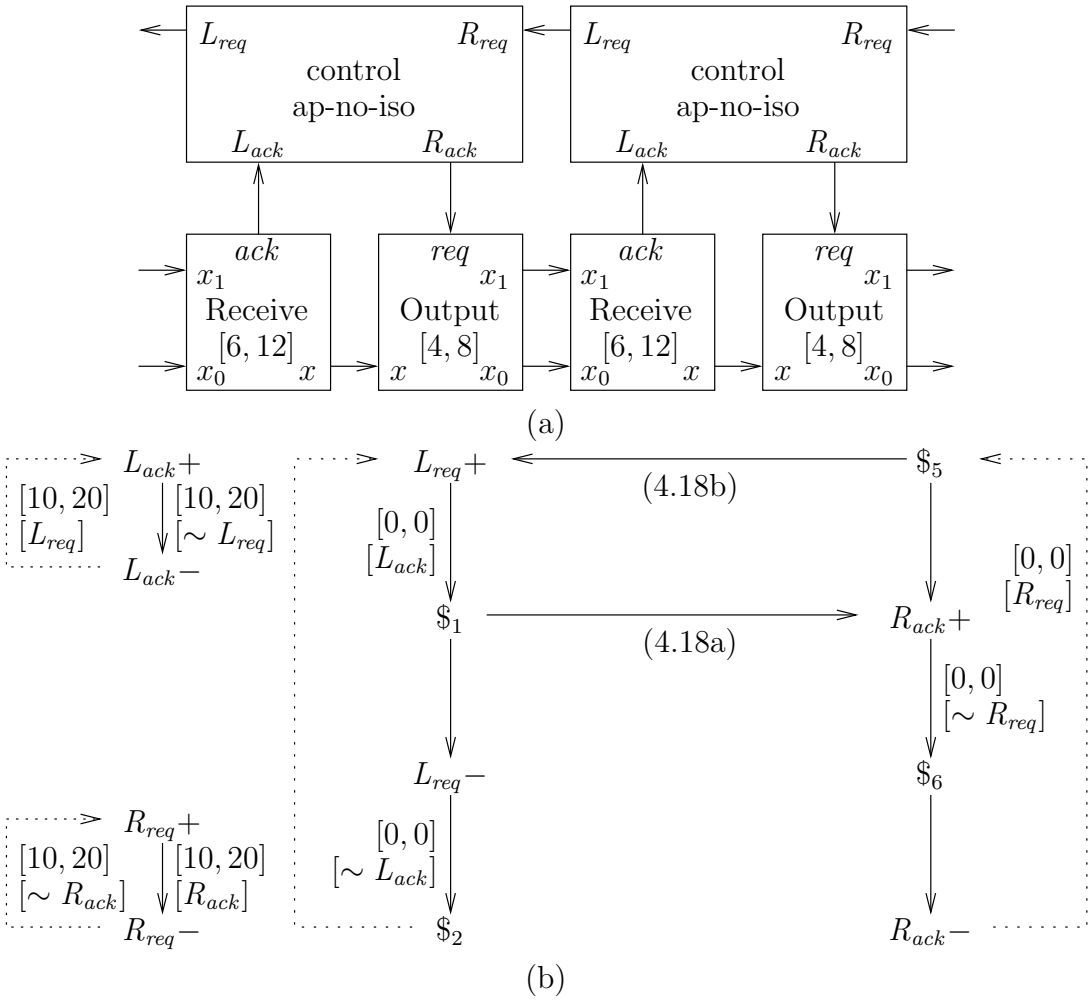

Figure 4.17. Pull FIFO requiring no isochronic fork between control and data path [20]. Part (a) shows a schematic of the control and data path. Part (b) shows a TEL structure for the constraints on control.

until the preceding buffer lowers its request and this propagates through the *Output* and *Passive* units. This means the protocol must enforce the following:

$$[\sim L_{req}] \prec R_{req}+ \tag{4.19a}$$

Recall that the x input to the *Output* component must not change while the req input to the *Output* component is high. The R_{req} output from the control portion of the buffer drives the req input to the *Output* component. The L_{ack} output from the control portion of the buffer drives the d_i input to the *Passive* component. Hence, when control lowers L_{ack}, this allows the *Passive* component to lower x, which in turn lowers the x input to the *Output* component. Meanwhile, when control lowers R_{req}, that lowers the req input to the *Output* component. Therefore, to ensure that the next stage has successfully received the current datum and that it is safe to change the x input to the *Output* unit before the next datum is received, the protocol must enforce the following [20]:

$$R_{req}^{-1}- \prec L_{ack}+ \tag{4.19b}$$

This is sufficient, assuming an isochronic fork between the path from R_{req} to the *Output* unit and the internal path within control from R_{req} to the gate for L_{ack}. Thus, the assumption is that the constraint $R_{req}^{-1}- \prec L_{ack}+$ implies that the *Output* component perceives $req- \prec x-$. The TEL structure of Figure 4.18(b) represents all the constraints on control simultaneously.

Burns also presents buffers that use the *Latch* component in the data path, such as the push buffer in Figure 4.19(a). In this design, the control portion of the buffer cannot determine that the data path has latched the datum until the preceding buffer lowers its request. This means the protocol must enforce the following:

$$[\sim L_{req}] \prec R_{req}+ \tag{4.20a}$$

To ensure that the next stage has successfully received the current datum before the next datum is received, the protocol must enforce the following [20]:

$$\left[R_{ack}^{-1}\right] \prec L_{ack}+ \tag{4.20b}$$

The TEL structure of Figure 4.19(b) represents all these constraints simultane-

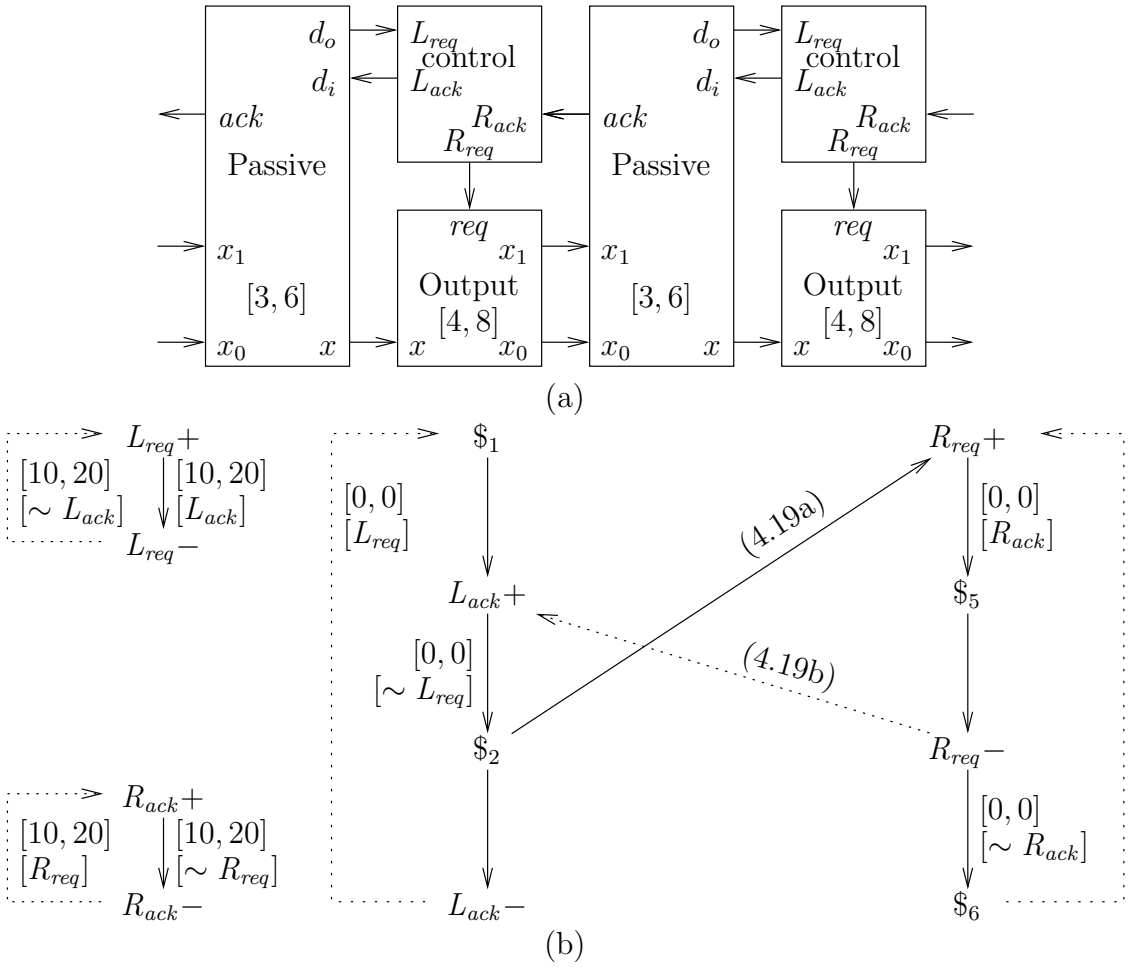

Figure 4.18. Push FIFO requiring an isochronic fork between control and data path[20]. Part (a) shows a schematic of the control and data path. Part (b) shows a TEL structure for the constraints on control.

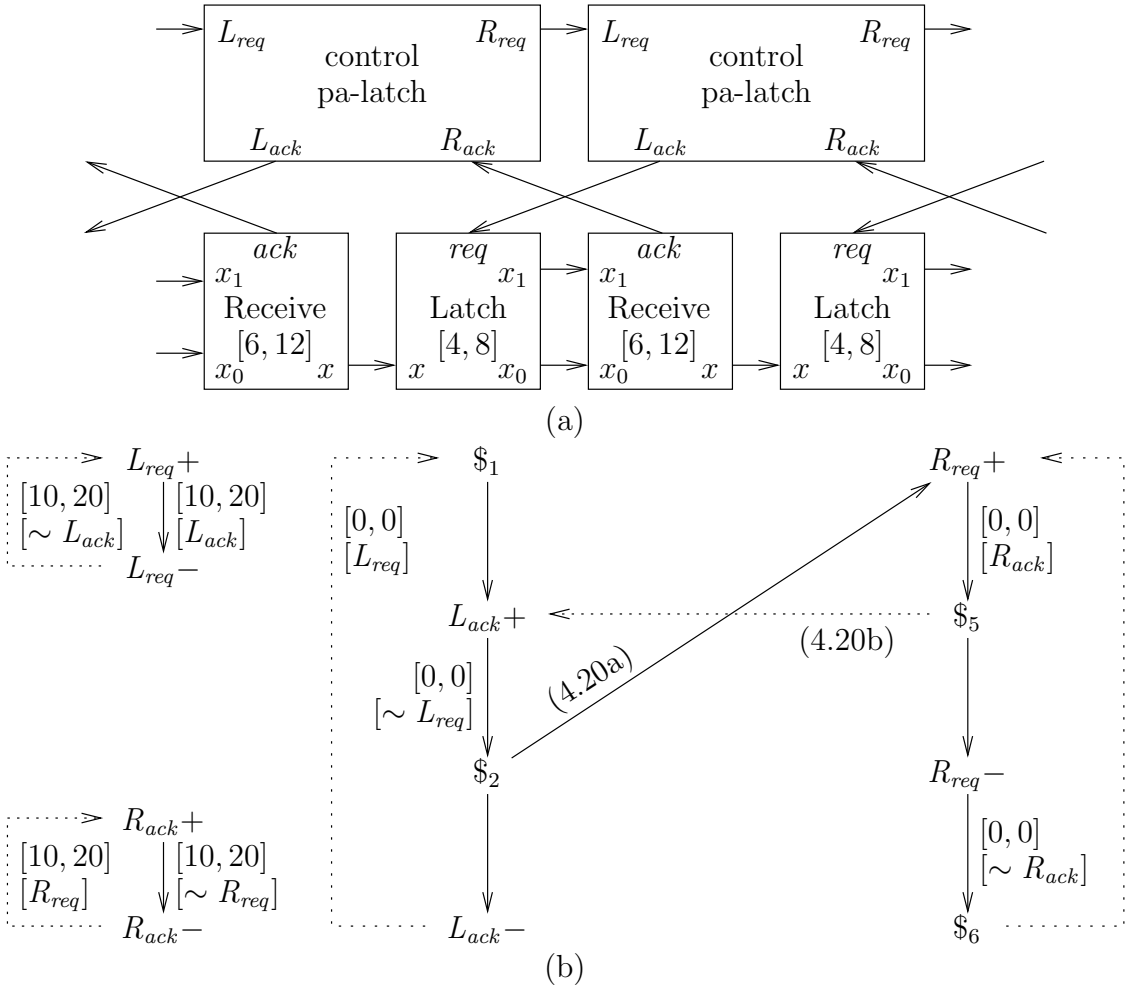

Figure 4.19. Push FIFO requiring the *Latch* component [20]. Part (a) shows a schematic of the control and data path. Part (b) shows a TEL structure for the constraints on control.

ously.

Burns [20] also presents the above buffer modified such that L is active and R is passive, as shown in Figure 4.20(a). In this design, the buffer requests data from the L channel, and it can determine that it has received a datum when it receives an acknowledge. This means the protocol must enforce the following:

$$[L_{ack}] \prec R_{ack}+ \tag{4.21a}$$

To ensure that the next stage has successfully received the current datum before the next datum is received, the protocol must enforce the following [20]:

$$\left[\sim R_{req}^{-1}\right] \prec L_{req}+ \tag{4.21b}$$

The TEL structure of Figure 4.20(b) represents all these constraints simultaneously.

4.3.2 Unified Control and Data Path

This section considers a design style that uses data encoding and does not separate control and data path at all. Instead, this approach simply allocates signals for each rail of the encoded data, and treats these signals as part of the controller. Cummings et al. [24], Martin et al. [53], Lines [45] and Manohar and Tierno [47] present examples of this design style. For example, consider the buffer consisting of a passive `receive` operation followed by an active `send` operation. For dual-rail data encoding, Figure 4.21 illustrates possible interfaces to one stage of a buffer (both data and control). Figure 4.21(a) shows the interface with L passive and R active. Figure 4.21(b) shows the case in which L is active and R is passive.

First consider the push case of Figure 4.21(a). On the channel L, a four-phase protocol is as follows, assuming all signals are initially low. The environment raises either L_0 or L_1. The buffer responds by raising L_{ack}. The environment lowers whichever of L_0 or L_1 it had raised, and the buffer responds by lowering L_{ack}. This requires the constraints $[L_n] \prec L_{ack}+ \prec [\sim L_n] \prec L_{ack}-$. Similarly, on the right side, the buffer raises either R_0 or R_1, and the environment responds by raising R_{ack}. Then the buffer lowers whichever of R_0 or R_1 it has raised, and the environment

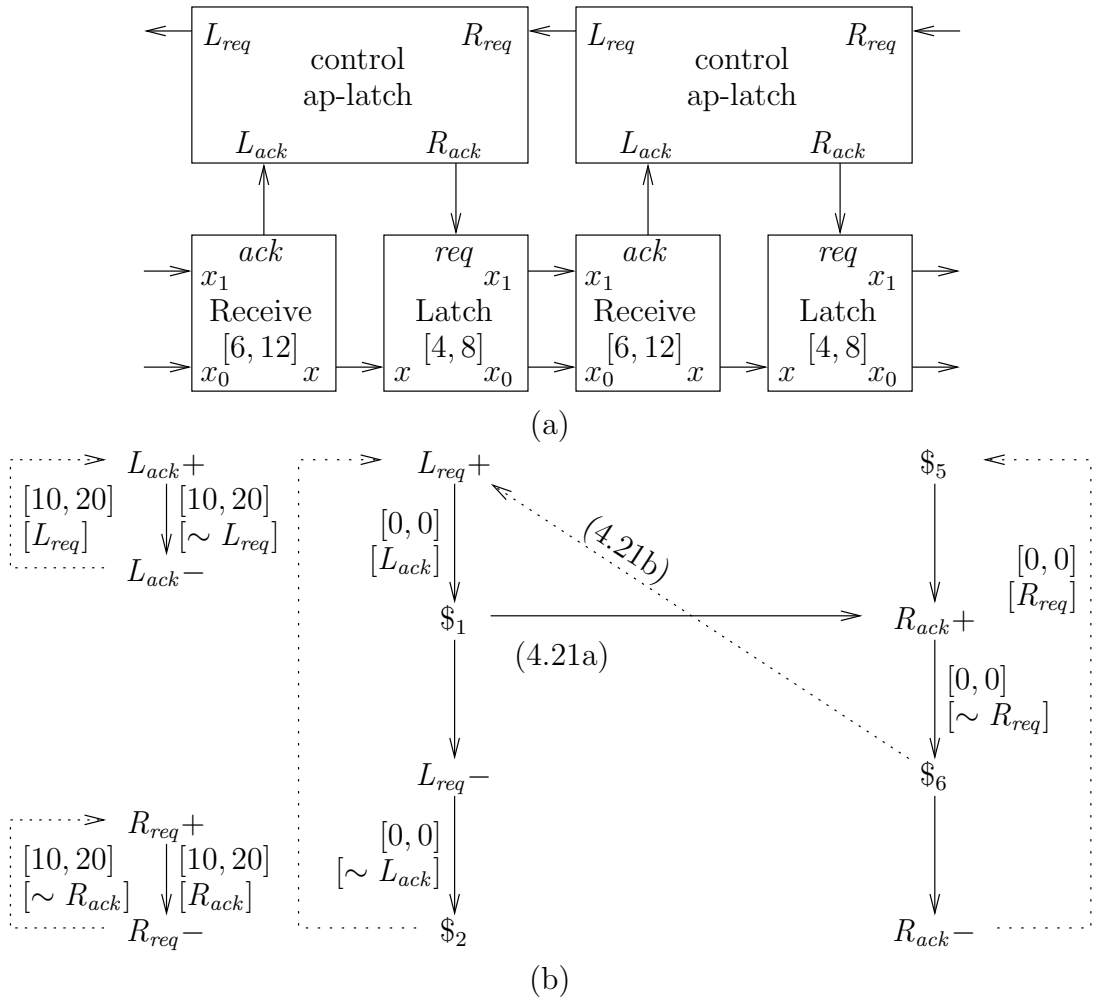

Figure 4.20. Pull FIFO requiring the *Latch* component [20]. Part (a) shows a schematic of the control and data path. Part (b) shows a TEL structure for the constraints on control.

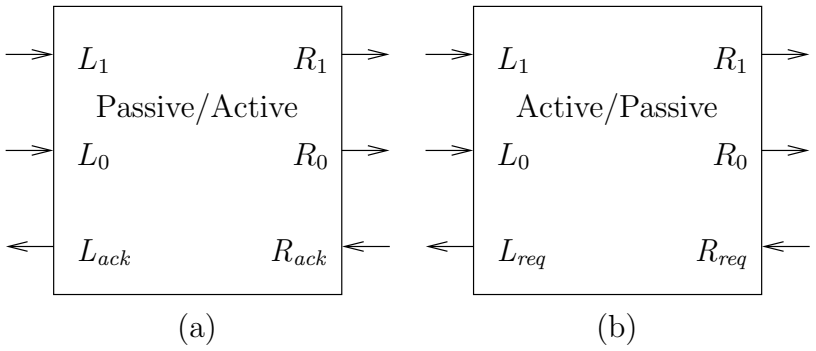

Figure 4.21. Four-phase, dual-rail buffer interfaces. Part (a) uses a push protocol. Part (b) uses a pull protocol.

responds by lowering R_{ack}. This requires the constraints $R_n+ \prec [R_{ack}] \prec R_n- \prec [\sim R_{ack}]$. Now consider the relationship between the two communication actions and how control and data impact each other. To ensure that the buffer has received a datum from the left before it tries to send it to the right, the protocol must enforce the following:

$$[L_n] \prec R_n+ \tag{4.22a}$$

Furthermore, in the absence of an extra state variable to store the value of x, the buffer must copy x from L to R before it acknowledges L. Otherwise L could erase the datum before it had been copied anywhere. Thus, the protocol must enforce the following:

$$R_n+ \prec L_{ack}+ \tag{4.22b}$$

The TEL structure of Figure 4.22 represents all these constraints simultaneously. This is a general scheme that applies to any $1/n$ code (any m/n code for which $m = 1$). For example, expanding to dual rail (the 1/2 code), L_n splits into L_0 and L_1, and R_n splits into R_0 and R_1. This results in the TEL structure of Figure 4.23.

Now suppose that L is active and R is passive. This corresponds to the interface of Figure 4.21(b). On channel L, the protocol is as follows, assuming all signals are

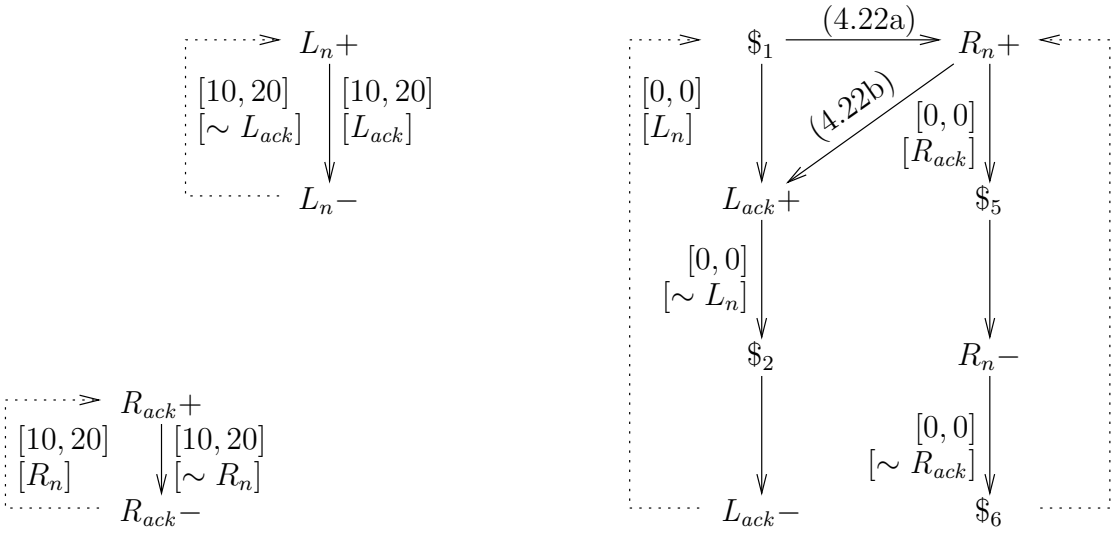

Figure 4.22. TEL structure for constraints on four-phase, push buffer.

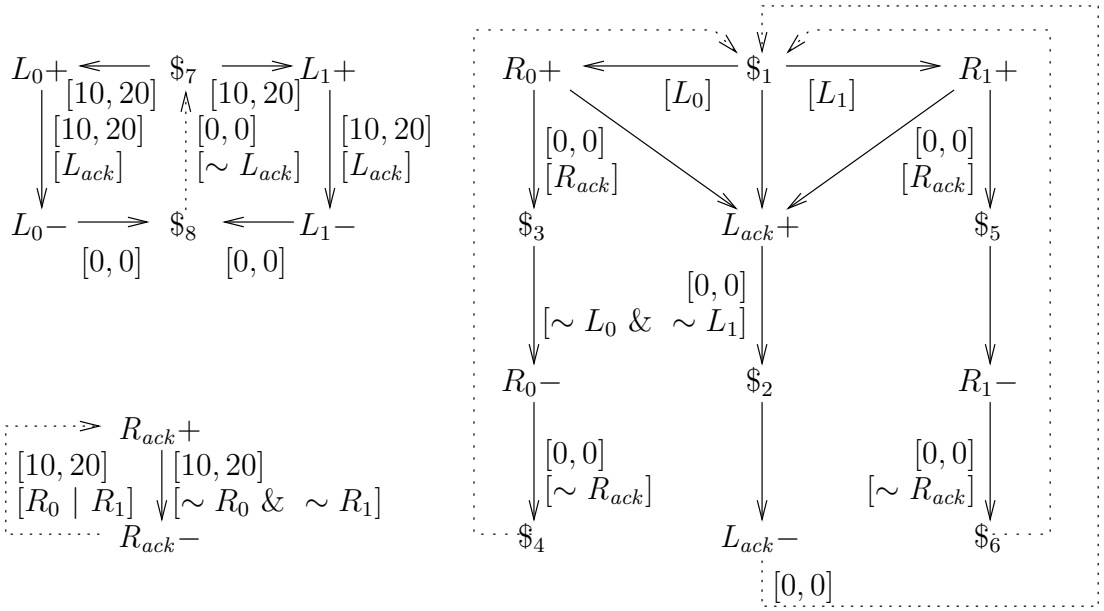

Figure 4.23. TEL for the dual-rail expansion of a four-phase, push buffer. $(L_0 \pm \#_{set} L_1 \pm \;\wedge\; \{R_0+, \$_3, R_0-, \$_4\} \#_{set} \{R_1+, \$_5, R_1-, \$_6\})$.

initially low. The buffer raises L_{req}. The environment responds by raising either L_0 or L_1. Then, the buffer lowers L_{req}, and the environment responds by lowering whichever of L_0 or L_1 it had raised. This requires the constraints $L_{req}+ \prec [L_n] \prec L_{req}- \prec [\sim L_n]$. Similarly, on the right side, the environment raises R_{req}, and the buffer responds by raising either R_0 or R_1. Then the environment lowers R_{req}, and the buffer responds by lowering whichever of R_0 or R_1 it has raised. This requires the constraints $[R_{req}] \prec R_n+ \prec [\sim R_{req}] \prec R_n-$. Now consider the relationship between the two communication actions and how control and data impact each other. To ensure that the buffer has received a datum from the left before it tries to send it to the right, the protocol must enforce the following:

$$[L_n] \prec R_n+ \tag{4.23a}$$

Furthermore, in the absence of an extra state variable to store the value of x, the buffer must copy x from L to R before it ceases to request x from L. Otherwise L could erase the datum before it had been copied anywhere. Thus, the protocol must enforce the following:

$$R_n+ \prec L_{req}- \tag{4.23b}$$

The TEL structure of Figure 4.24 represents all these constraints simultaneously. This is a general scheme that applies to any $1/n$ code (any m/n code for which $m = 1$). For example, expanded to dual rail (the 1/2 code), L_n splits into L_0 and L_1, and R_n splits into R_0 and R_1. This results in the TEL structure of Figure 4.25.

This unified scheme also works for pipeline stages that do some computation on the data instead of just buffering the data. Lines [45] shows that a buffer such as that of Figure 4.25 can be extended to do computation by replacing the expressions involving L_0 and L_1 with more general functions. Suppose that a process is to compute a function of three input bits, which arrive on three input channels A, B, and C. Let $\mathrm{f}_0(A, B, C)$ be a boolean function that is *true* when and only when each of the channels A, B, and C carries a valid input datum, and the corresponding output datum should be 0. Similarly, let $\mathrm{f}_1(A, B, C)$ be a boolean function that is *true* when and only when each input channel contains a valid datum and the corresponding output datum should be 1. Let $\mathrm{rtz}(A, B, C)$ be a boolean function that is *true* when and only when each of the channels A, B, and C is in the idle state. Given these definitions, and assuming pull type channels, Figure 4.26

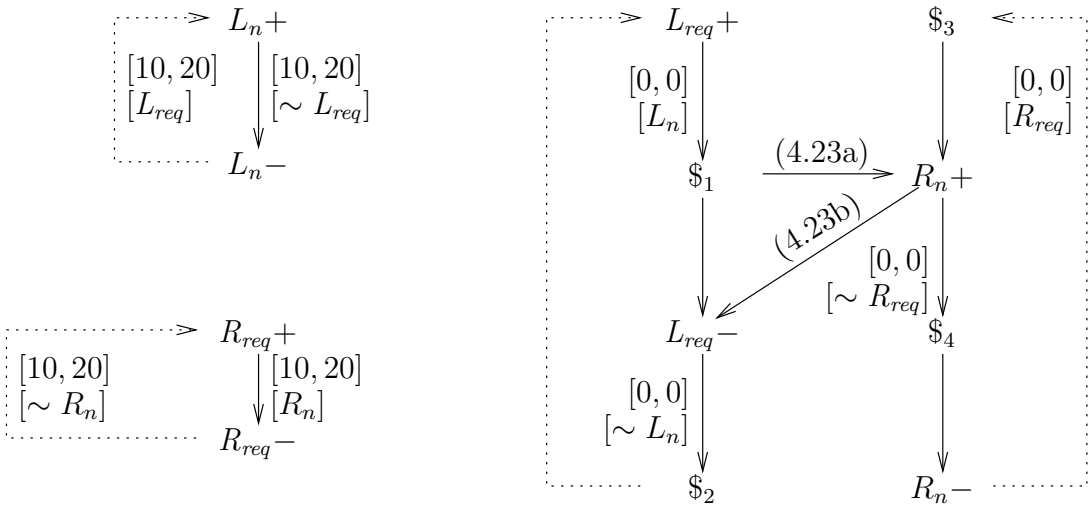

Figure 4.24. TEL structure for constraints on four-phase, pull buffer.

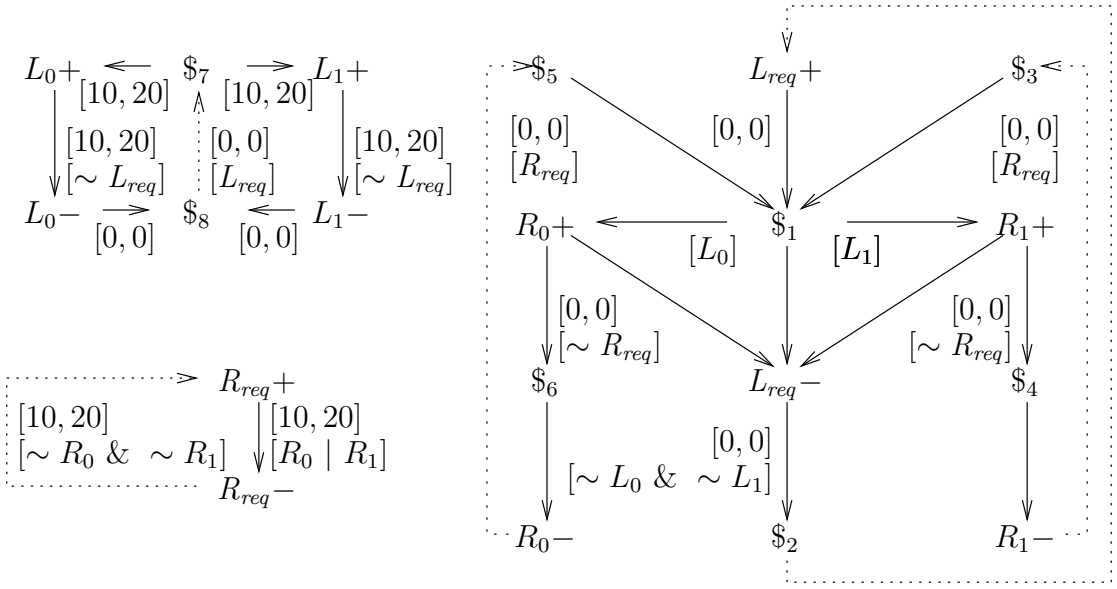

Figure 4.25. TEL for the dual-rail expansion of a four-phase pull buffer. $(L_0 \pm \#_{set} L_1 \pm \ \wedge \ \{\$_3, R_0+, \$_4, R_0-\} \#_{set} \{R_1+, \$_5, R_1-, \$_6\})$.

represents the constraints and behavior of this system. In this TEL structure, the requests for channels A, B, and C have been merged into one signal L_{req}.

For example, consider the following extension of the buffer example to compute the sum of three bits.

```
sum :  process
begin
  receive(A, aa, B, bb, C, cc);
  send(S, aa xor bb xor cc);
end process sum;
```

In this case, the functions are defined as follows.

$$\mathrm{f}_1(A,B,C) = A_0B_0C_1 \vee A_0B_1C_0 \vee A_1B_0C_0 \vee A_1B_1C_1 \tag{4.24}$$

$$\mathrm{f}_0(A,B,C) = A_1B_1C_0 \vee A_1B_0C_1 \vee A_0B_1C_1 \vee A_0B_0C_0 \tag{4.25}$$

$$\mathrm{rtz}(A,B,C) = \neg A_0 \wedge \neg A_1 \wedge \neg B_0 \wedge \neg B_1 \wedge \neg C_0 \wedge \neg C_1 \tag{4.26}$$

4.4 Extending Constraints Across Actions

While this chapter looks at buffers in detail, not all specifications are as simple as a buffer. This section considers how the constraints that this chapter presents could be generalized to more complicated examples. Section 4.4.1 discusses a possible

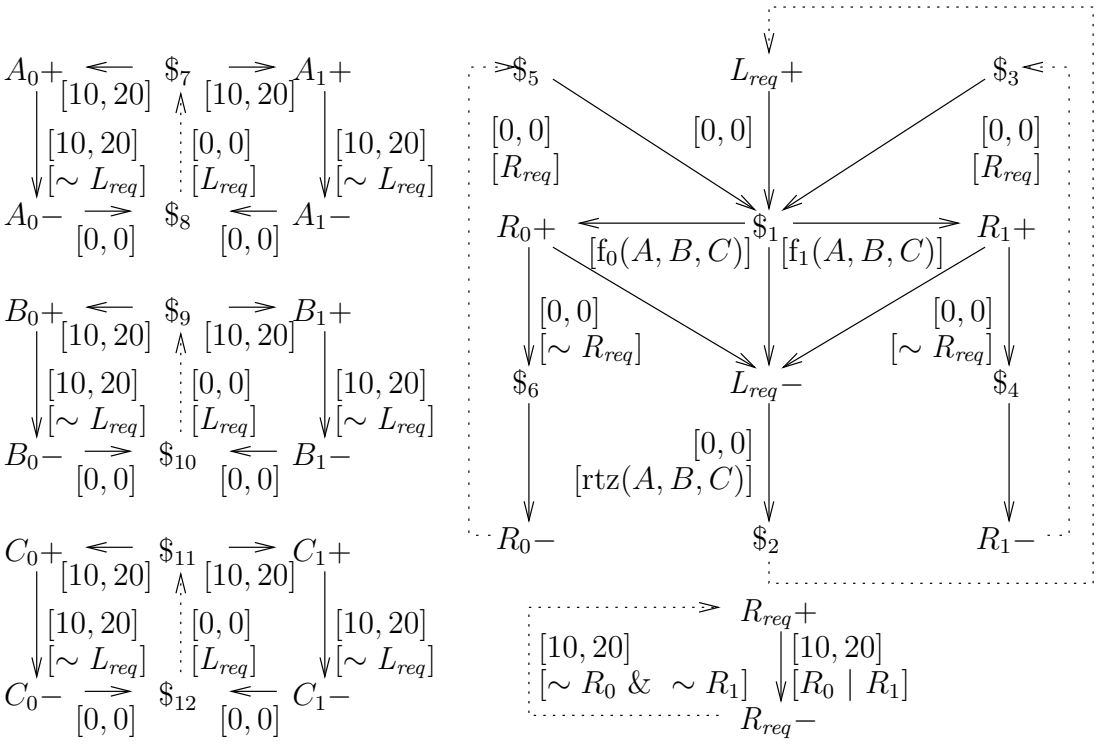

Figure 4.26. Constraints on the dual-rail circuit for the pull *sum* process. $(A_0 \pm \#_{set} A_1 \pm \ \wedge\ B_0 \pm \#_{set} B_1 \pm \ \wedge\ C_0 \pm \#_{set} C_1 \pm \ \wedge$
$\{\$_5, R_0+, \$_6, R_0-\} \#_{set} \{\$_3, R_1+, \$_4, R_1-\})$.

improvement to the tool that this dissertation presents. In contrast, the technique of Section 4.4.2 is fully implemented.

4.4.1 Same Variable, Different Channels

The tool that this dissertation presents applies data constants between every consecutive pair of communication actions. This is actually more conservative than necessary. These constraints could be relaxed in the following way.

The data constraints presented in this chapter should be applied whenever a `receive` operation is followed by a `send` operation that uses the same data variable. This still applies even if there are other actions between the `receive` operation and the `send` operation, provided that such operations do not use the channels or the data variable in question. This requires a recursive search of the channel-level TEL structure to find relevant pairs of `receive` and `send` actions. Each such pair essentially forms a buffer.

If there are intervening operations that use the data variable (but not the channels in question), the datum must be received before such operations attempt to read the variable. If there are intervening operations that write the data variable, such operation must complete, before the `send` operation announces the validity of the data.

4.4.2 Same Channel, Different Variables

A channel must be completely reset before any subsequent use of the same channel can occur. Figure 2.3 on page 33 shows a communication on channel L, followed by a communication on channel R, followed by another communication on channel L. The edges from $\$_2$ to $L_{req}+/2$, from $\$_4$ to $L_{req}+$, and from $R_{ack}-$ to $\$_5$ are necessary to ensure that each channel has reset before it is used again.

In general, adding all arcs necessary to make sure that each use of a given channel has completed before any other use of that channel requires a recursive search of the channel-level TEL structure. From each channel communication, the search must consider each outgoing path. On each of these paths, it finds the next use of the given channel, and adds an arc to that use. In some cases, the next use

of the channel may be in the current communication.

4.5 A Library of Protocols

This section discusses how the observations of this chapter could form the basis for a general extensible library of protocols. This has not been implemented in the tool that this dissertation presents. The current compiler targets only a single, fixed, four-phase protocol with narrow sequencing. Furthermore, data path is not implemented. However, this section (and indeed this entire chapter) is still useful to anyone who extends the compiler in the future to implement these techniques. Furthermore, until then, this section is also useful to the user who bypasses the channel-level compiler, directly inputting the most-concurrent, signal-level specification that still meets the constraints of the user's chosen protocol. For such a user, this section outlines a systematic approach to obtain such a signal-level starting point for an entire channel-level specification, given a signal level specification for a buffer using the desired protocol.

Section 4.3 shows that TEL structures can model the data constraints that a given data-path approach imposes on the control protocol. That section demonstrates this over a wide variety of buffer implementations from the literature. Furthermore, Section 4.4 shows how to extend this technique beyond simple buffers. Therefore, it is possible to model a library of protocols using a library of TEL structures. In such a library, each element would be the TEL structure for constraints on a simple buffer consisting of a `receive` operation followed by a `send` operation, much like the examples from Section 4.3.

Each library element would need to be defined by writing a signal-level description of the constraints for a buffer using that protocol. Even if the techniques of this section were automated, the signal-level description for a buffer in each protocol would still need to be written by hand. It could be written in any of a number of signal-level specification languages, including the VHDL handshake package of Section 2.2. Alternatively, the TEL structures of Section 4.3 could be used directly.

However, once such a library element is in place, the techniques that this section

proposes could automate the process of using an element from the library, together with a channel-level specification of the example to be synthesized (written using the channel-level VHDL package of Section 2.1), to produce the signal-level starting point for concurrency reduction. The channel-level VHDL specification itself would be protocol independent (except, optionally, for directives like those in Section 3.1). For example, in Section 4.3, the channel-level specification at the beginning of the section does not mention any channel named X. However, many — but not all — of the defined protocols introduce a communication with the data path on channel X as part of the process of ensuring data integrity. This X channel would appear in the library elements. The only way it would appear in the channel-level specification would be as the variable x and not as a channel X.

The situation would be roughly analogous to the system of Brunvand and Sproull [18, 16, 17], in which each library element is designed by hand, but then an automated procedure maps a specification onto the library. It might be even closer to the system of Kim et al. [41], in which each library element is an STG rather than actual circuit. One difference between these methods and the method that this section proposes is that the method that this section proposes would be used just to set up the starting point for the concurrency reduction engine. The actual circuit — or circuits — found would depend on the result of concurrency reduction, which is a global process not restricted to individual library elements. However, another important difference is that the methods of [18, 16, 17, 41] are already automated. The library method of this section is currently just a framework and a proposal for future automation. However, the concurrency reduction engine of the remaining chapters of this dissertation is fully automated and independent of the method used to set up its starting point.

Whether by hand or through future automation, the protocol constraints can be applied to any pair of communications consisting of a `receive` into a variable, followed by a `send` of the same datum as shown in Section 4.4.

To use a protocol from the library proposed here, a CAD tool would proceed as follows. Whenever it encountered a pair of communication actions that meet

the criteria of Section 4.4, it would instantiate the library TEL structure for the chosen protocol into the signal-level TEL structure, at the position corresponding to that pair of communication actions. For each new communication action, this procedure would add new events into the signal-level TEL structure. These events would be renamed copies of the events in the TEL structure corresponding to the chosen protocol. For each relevant pair of communication actions, this procedure would add rules to the signal-level TEL structure. These rules would correspond to the rules in the TEL structure corresponding to the chosen protocol. This approach is similar to that for component instantiation in [74].

The procedure would be repeated until it had expanded all communication actions in the channel-level TEL structure. The result would be an initial signal-level TEL structure that was the most concurrent possible that still contained all the constraints of the channel-level specification and the chosen protocol.

The user could select one protocol from the library, or some small set of protocols. If more than one protocol is selected, the tool could include the selection of one of these protocols as part of the search for solutions to the specification.

The creators of the tool could provide a small, initial library of protocols, and the user could extend this library at any time by specifying the TEL structure that corresponds to a new protocol.

CHAPTER 5

CONCURRENCY REDUCTION

The TEL structures of Chapter 4 are semantically correct expansions of a one-place buffer that communicates on channel L and then on channel R, for their assignments of active and passive channels. (As part of the expansion, many of the TEL structure of Chapter 4 also introduce a data-path handshake on channel X, to enforce data constraints of the protocol. However, this is not part of the channel-level specification.) Unfortunately, complete state coding violations prevent these TEL structures from being directly synthesizable. For example, consider the TEL structure of Figure 4.20 on page 92. The problem is that the entire four-phase handshake on L can complete before the handshake on R starts. In other words, the sequence of events in each iteration of the complete circuit could be as follows:

$$L_{req}+; L_{ack}+; L_{req}-; L_{ack}-; R_{req}+; R_{ack}+; R_{req}-; R_{ack}-$$

Manohar [46] defines a notation called *two-phase CHP* which can express such a sequence in a more compact form. This notation groups the upward going transitions and the downward going transitions of each four-phase communication. The two-phase CHP for the above sequence is as follows:

$$L\uparrow; L\downarrow; R\uparrow; R\downarrow$$

$L\uparrow$ represents the *working phase* [41] of the L communication, and $L\downarrow$ represents the *idling phase* [41] of the L communication. The above expansion is semantically correct. Unfortunately, it has complete state coding violations. Therefore, synthesis cannot proceed from this expansion. The problem is that after $L\downarrow$ and before $R\uparrow$, all of the control wires associated with channels L and R are all zero. This is the same state that the control wires are in at the start of the process (before $L\uparrow$).

Thus, when all signals are low, the circuit does not know whether to expect $L\uparrow$ or $R\uparrow$ next.

One solution is to interleave the two communications like this:

$$L\uparrow; R\uparrow; L\downarrow; R\downarrow$$

This is a form of narrow sequencing [63]. This expansion still meets the data constraints from Figure 4.20 on page 92. Furthermore, the above interleaving can be enforced by adding rules to the TEL structure of Figure 4.20 on page 92. Figure 5.1 shows the result. In effect, each communication provides state variables for the other communication, so no new state variables are necessary.

This is one reshuffling that solves this case. However, there may be other reshufflings of the same example that are also synthesizable. Furthermore, this strategy is directly applicable only to series chains of communications on independent channels using narrow sequencing. Our hypothesis is that this reshuffling is representative of a general type of reshuffling that is effective for a broad class of specifications. The idea is to force the working phase of the next communication in series to complete before the idling phase of the current communication. However, instead of simply assuming that this strategy will always be effective, a major

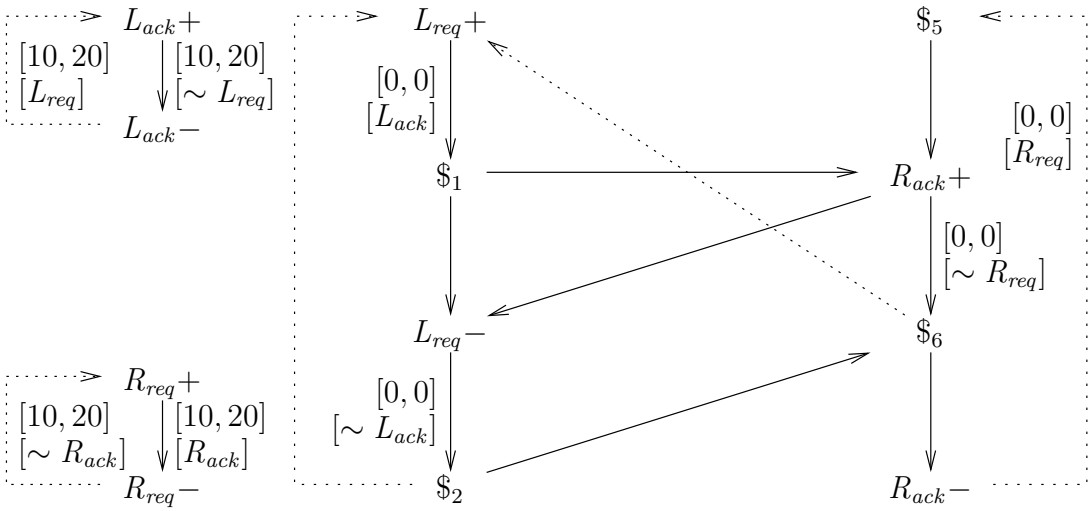

Figure 5.1. TEL structure with concurrency reduced for complete state coding.

goal of this dissertation is to conduct experiments to compare different types of reshuffling. Each reshuffling corresponds to a permutation of the events. To find literally all reshufflings one could let each pair of events be concurrent, and then find all possible ways to reduce the concurrency in the specification. However, any legal reshuffling must still meet the constraints of the channel level specification and the chosen protocol. Therefore, the starting point for concurrency reduction should be a signal-level specification that contains exactly these constraints. From this starting point, finding all possible ways to reduce concurrency finds all legal reshufflings.

Another approach to achieve complete state coding is to insert state variables. Section 5.4 shows that this can be accomplished by inserting an initially unconstrained state variable into the TEL structure, and then proceeding with concurrency reduction.

Note that concurrency reduction involves a trade-off. If concurrency reduction serializes operations on the critical path, it can degrade performance. Even in this case, performance degradation is not inevitable. Ykman-Couvreur et al. [72] and Lin et al. [44] show that for some circuits, the concurrency reduction can streamline the implementation, speeding it up enough to more than compensate for the serialization. Thus, some reduced circuits are smaller and faster than the corresponding circuits before reduction. However, if the serialized operations are slow, for example operations that must wait for slow functional units in the data path, the impact can be severe. Therefore, the goal is to reduce concurrency enough to make it feasible to implement the circuit, but to avoid reducing concurrency in ways that would significantly degrade performance. Section 5.3 shows how the concurrency reduction process can be guided by an estimate of the system performance, rejecting solutions that would perform poorly.

5.1 The Search Space

The most general approach considers all possible reshufflings that meet the given data and reuse constraints. These constraints are given by the initial TEL

structure (for example, any of the TEL structures from Chapter 4). Furthermore, the initial TEL structure represents the most concurrent possibility that meets all the constraints. Thus, any reshuffling that meets the constraints can be obtained by adding some set of rules to the initial TEL structure.

Thus, one way to find all possible reshufflings is to consider every possible combination of rules that can be added to the initial TEL structure. Unfortunately, the set of such combinations is quite large. Consider an initial TEL structure $T = (N, S_0, A, E, R, R_0, \#)$. Suppose that T is the TEL structure for just one output process. Concurrency reduction adds some set of rules to this structure that simply reduce the concurrency in this structure. In other words, each added rule should simply force its enabling event to precede its enabled event. Such a rule has the expression $[true]$. Choosing timing bounds for such a rule is more problematic. Section 6.2.2 shows that under timing, there is no guarantee that adding a rule strictly reduces the concurrency of a TEL structure. In some cases, adding a rule between two events introduces new states no matter what timing bounds are chosen for the rule. Furthermore, unrealistic timing bounds make it impossible to implement the circuit.

Definition 5.1 Let *min* be the minimum delay of any gate in the selected technology library. Let *max* be the maximum delay of any gate in the selected technology library. Let $e \rightarrow f$ stand for $(e, f, min, max, [true])$.

The techniques of this chapter attempt to reduce concurrency by adding rules from the set of *candidate rules*, $R_c = \{e \rightarrow f \mid e, f \in E\}$. Note that $|R_c| = |E \times E|$. Given the TEL structure T, concurrency reduction finds the set of modified TEL structures $S = \{(N, S_0, A, E, R \cup R', R_0 \cup R'_0, \#) \mid R'_0 \subseteq R' \subseteq R_c\}$. For any given $T' \in S$ and $r \in R_c$, there are three possibilities:

1. $r \notin R'$ means that r is omitted from T';
2. $r \in R' \wedge r \notin R'_0$ means that r is included in T', but not initially marked;
3. $r \in R'_0$ means that r is included in T' and initially marked.

Figure 5.2 represents the search space as a tree. In this tree, each node represents a TEL structure. The root node represents the initial TEL structure, that is, the most concurrent TEL structure that meets the constraints of the specification, as well as the data constraints from Chapter 4. The tree has a level of edges for each candidate rule in R_c. Consider the level of edges corresponding to rule $r \in R_c$. An edge in this level labeled "omit" corresponds to the case $r \notin R'$. An edge labeled with 0 corresponds to the case $r \notin R'_0 \wedge r \in R'$. Finally, an edge labeled with 1 corresponds to the case $r \in R'_0$. Each leaf node uniquely determines the values of R' and R'_0. Thus, each leaf represents a potentially synthesizable TEL structure. The row of leaf nodes represents the set S.

The tree of the search space has a branching factor of three and a level of edges for each member of R_c. Therefore, $|S| = 3^{|R_c|} = 3^{|E \times E|}$. Even for the small example of Figure 4.20 on page 92, the TEL structure for the output process has eight events. This means there are 64 candidate rules, which leads to $3^{64} \approx 3 \times 10^{30}$ combinations to try. Clearly, it is necessary to prune the search space.

5.2 Pruning Redundant Possibilities

This section presents techniques to prune the search space. Certain candidate rules can be removed from consideration before the algorithm even constructs a search tree such as Figure 5.2 and begins to search. This reduces the number of levels in the search tree. The resulting reduction in the size of the search

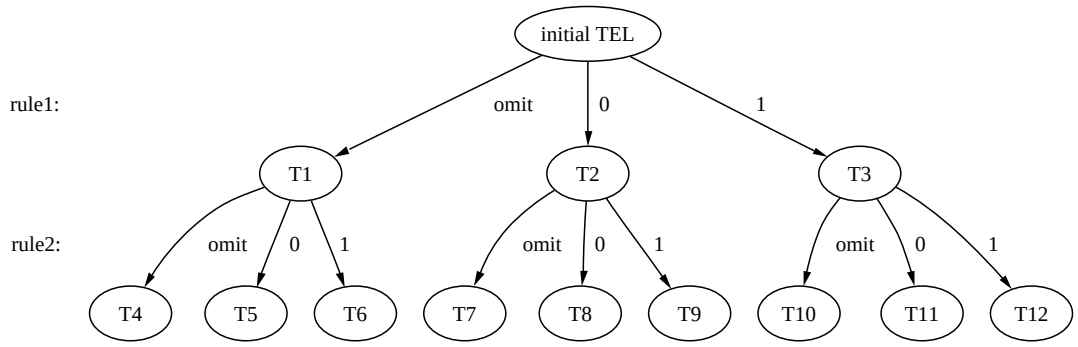

Figure 5.2. Search space for reshuffling (two candidate rules shown).

tree is exponential in the number of candidate rules removed from consideration. Section 5.2.1 and Section 5.2.2 present criteria for removing candidate rules before the search begins. In contrast, Section 5.2.3 shows how to prune sections of the search tree dynamically during the search.

5.2.1 Reflexive Loops

There is no need to add a rule from an event back to itself. Figure 5.3 illustrates a TEL structure with such a rule. If such a rule is not initially marked, it requires a given occurrence of an event to happen before itself, which is clearly impossible. If such a rule is initially marked, it requires an occurrence of an event to happen before the next occurrence of the same event. This is guaranteed even without the rule. Using timing bounds, an initially marked rule from an event back to itself could set a minimum separation between successive occurrences of that event. However, if the existing rules in the TEL structure already guarantee a minimum cycle time greater than the lower timing bound on the self-loop rule, the self-loop rule will not influence the cycle time. Suppose that the gate library determines the timing bounds for the self loop rule $e \rightarrow e$, as suggested in Section 5.1. If, even without $e \rightarrow e$, there is a loop of rules that includes event e and contains exactly one initially marked rule and at least one rule with lower timing bound *min*, then $e \rightarrow e$ will have no effect. Eliminating self-loop rules from consideration reduces the number of possibilities to $3^{|E \times E| - |E|}$.

5.2.2 Sequencing Events

It is not necessary to consider adding any rule from a sequencing event for which all of the enabling rules have the expression [*true*]. Such a rule could always

Figure 5.3. TEL structure with a self-loop rule.

be replaced by other, equivalent rules. Consider a sequencing event \$, such that $\forall\,(e, \$, l, u, B) \in R\,.\,B = [true]\ \wedge\ l = u = 0$. Note that it is reasonable for the rules that enable a sequencing event to have their timing bounds set to zero, because a sequencing event does not change the value of any signal. Consider a candidate rule $r_{new} = \$ \rightarrow f$. Adding this rule is equivalent to adding the set of rules $R_{new} = \{e \rightarrow f \mid (e, \$, l, u, B) \in R\}$. Since the two possibilities are equivalent, it is not necessary to explore both possibilities. One way to avoid such redundant exploration is to avoid adding any rule from a sequencing event for which all of the enabling rules have the expression $[true]$. The tool can identify and remove each such candidate rule before creating the search tree. Checking instead for a set of rules such as R_{new} would require a check at each node of the search tree.

For example, given the TEL structure shown in Figure 5.4, adding the rule $\$ \rightarrow z-$ produces the TEL structure of Figure 5.5(a). Alternatively, adding the rules $x+ \rightarrow z-$ and $y+ \rightarrow z-$ to the TEL structure of Figure 5.4 produces the TEL structure of Figure 5.5(b), which is equivalent to that in Figure 5.5(a). The TEL structure of Figure 5.5(a) is simpler but easier to avoid than that in Figure 5.5(b).

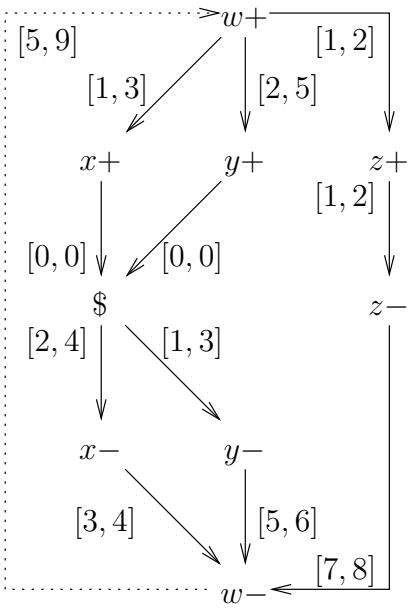

Figure 5.4. TEL structure with a sequencing event.

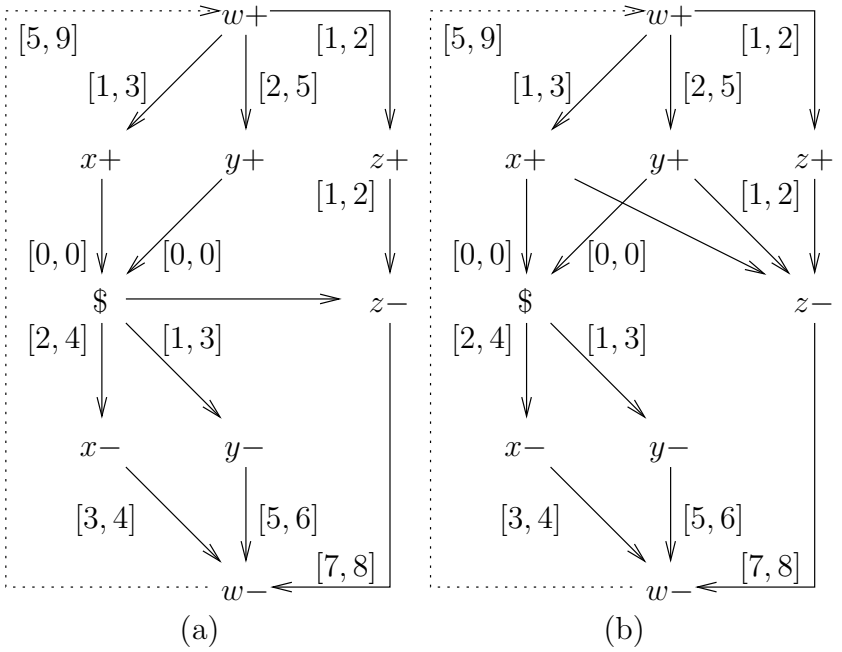

Figure 5.5. TEL structures derived from that of Figure 5.4 by adding the set of rules (a) $\{\$ \rightarrow z-\}$, and (b) $\{x+ \rightarrow z-, y+ \rightarrow z-\}$.

Similarly, it is not necessary to consider adding any rule to a sequencing event for which all of the enabled rules have the expression $[true]$. Such a rule could always be replaced by other, equivalent rules. Consider a sequencing event \$, such that $\forall\,(\$, f, l, u, B) \in R\;.\; B = [true]$. Adding the candidate rule $r_{new} = (e, \$, 0, 0, [true])$ is equivalent to adding the set of rules $R_{new} = \{(e, f, l, u, B) \mid (\$, f, l, u, B) \in R\}$. Since the two possibilities are equivalent, it is not necessary to explore both possibilities. One way to avoid such redundant exploration is to avoid adding any rule to a sequencing event for which all of the enabled rules have the expression $[true]$. The tool can identify and remove each such candidate rule before creating the search tree. Checking instead for a set of rules such as R_{new} would require a check at each node of the search tree.

For example, given the TEL structure shown in Figure 5.4, adding the rule $z- \rightarrow \$$ produces the TEL structure of Figure 5.6(a). Alternatively, adding the rules $z- \rightarrow x-$ and $z- \rightarrow y-$ to the TEL structure of Figure 5.4 produces the TEL structure of Figure 5.6(b), which is equivalent to that in Figure 5.6(a). The TEL

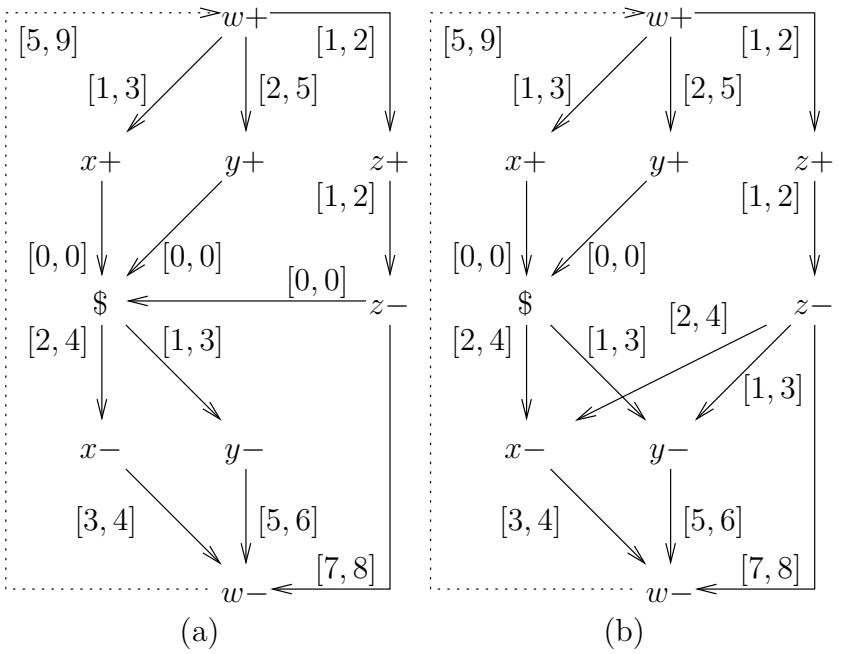

Figure 5.6. TEL structures derived from that of Figure 5.4 by adding the set of rules (a) $\{z- \rightarrow \$\}$, (b) $\{z- \rightarrow x-, z- \rightarrow y-\}$.

structure of Figure 5.6(a) is simpler but easier to avoid than that in Figure 5.6(b).

5.2.3 Reachability

While the above pruning techniques remove rules from consideration before the search even begins, it is also possible to prune certain branches from the search tree dynamically during the search.

It is not necessary to add a rule from event e to event f if existing rules already ensure that e must precede f. It may be that in any given iteration of the TEL structure, e must precede f. This is the case if there is already some path of rules from e to f, in which none of the rules on this path are initially marked. This guarantees that a given occurrence of e always occurs before the corresponding occurrence of f. With e and f already ordered, adding a rule from e to f would be redundant. For example, the TEL structures of Figure 5.7 are equivalent.

If there is a path in the other direction, from f to e, then any rule from e to f must be initially marked. Otherwise, adding a rule from e to f would complete

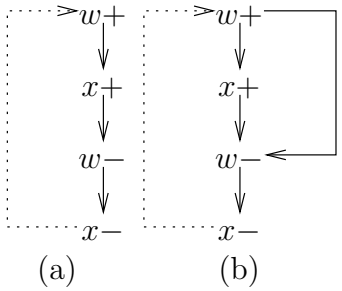

Figure 5.7. Adding a redundant rule. Part (a) shows a schematic of the control and data path. Part (b) shows a TEL structure for the constraints on control.

a cycle of unmarked rules. This causes a deadlock. Intuitively, this is because for any event e that is on a cycle of unmarked rules, the rules on the cycle mean that e must occur before itself, which is clearly impossible. This sufficient condition for deadlock in a TEL structure is analogous to that given in [39] for marked graphs. For example, Figure 5.8 shows two TEL structures. Starting with the TEL structure of Figure 5.8(a), adding the rule $w- \rightarrow w+$ to R but not to R_0 produces TEL structure of Figure 5.8(b). The TEL structure of Figure 5.8(a) is deadlock-free. However, that of Figure 5.8(b) contains a cycle of unmarked rules, and hence suffers from deadlock.

Furthermore, if there is a path from e to f that contains one initially marked rule, adding an initially marked rule from e to f would be redundant. For example, the TEL structures of Figure 5.9 are equivalent. In this case, the loop of rules from

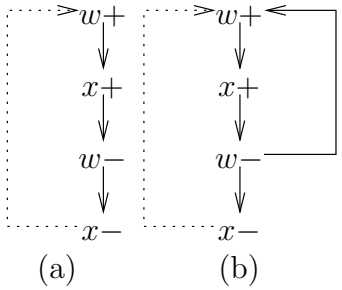

Figure 5.8. Adding a rule that creates a cycle of unmarked rules, causing deadlock.. Part (a) shows the TEL structure before adding $w- \rightarrow w+$. Part (b) shows the TEL structure after adding $w- \rightarrow w+$.

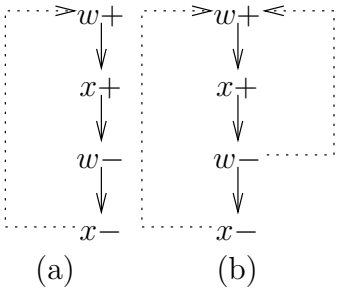

Figure 5.9. Adding an initially-marked, redundant rule. Part (a) shows a schematic of the control and data path. Part (b) shows a TEL structure for the constraints on control.

$w+$ back to itself contains an initially marked rule. Therefore, deadlock is not the problem is this case.

If there is a path from e to f that contains no initially marked rules, adding an initially marked rule from e to f introduces a safety violation (Definition 2.6 on page 32). For example, starting from the one-safe TEL structure of Figure 5.10(a), adding $w+ \rightarrow w-$ produces the TEL structure of Figure 5.10(b), which has a safety violation. In the initial state $w+$ can occur, because its only enabling rule, $(x-, w+, 0, \infty, [true])$ is initially marked. However, the rule $(w+, w-, 0, \infty, [true])$ is also initially marked. Therefore, the occurrence of $w+$ is a safety violation.

The following definitions formalize these notions.

Definition 5.2 The $\rightarrow_R$ relation is defined as follows:

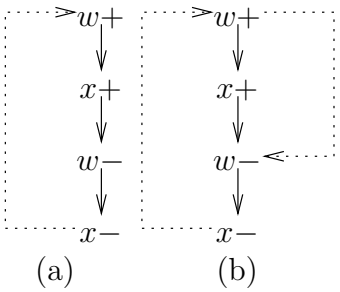

Figure 5.10. Adding an initially-marked rule that introduces a safety violation. Part (a) shows the TEL structure before adding $w+ \rightarrow w-$. Part (b) shows the TEL structure after adding $w+ \rightarrow w-$.

$$e \rightarrow_R f \iff \exists\,(e, f, l, u, B) \in R$$

Definition 5.3 Let $\rightarrow_R^*$ be the reflexive and transitive closure of $\rightarrow_R$

Definition 5.4 The oneToken function is defined as follows

$$\begin{array}{c} \text{oneToken}(e, f, R, R_0) \\ \Updownarrow \\ \neg\left(e \rightarrow_{R-R_0}^* f\right) \wedge \left(\exists\,(e', f', l, u, B) \in R_0 \;.\; e \rightarrow_{R-R_0}^* e' \wedge f' \rightarrow_{R-R_0}^* f\right) \end{array}$$

Consider adding a candidate rule $e \rightarrow f$ to R. If $e \rightarrow_{R-R_0}^* f$, then adding $e \rightarrow f$ to R but not to R_0 would be *redundant*. This corresponds to the situation in Figure 5.7. If oneToken(e, f, R, R_0) then adding $e \rightarrow f$ to R_0 would be redundant. This corresponds to the situation in Figure 5.9. Therefore, instead of adding this rule, the algorithm records the fact that it would be redundant. If $e \rightarrow_{R-R_0}^* f$, then adding $e \rightarrow f$ to R_0 would introduce a safety violation. This corresponds to the situation in Figure 5.10. If $f \rightarrow_{R-R_0}^* e$, then adding $e \rightarrow f$ to R but not R_0 is *infeasible*. Adding such a rule would create a cycle of unmarked rules, which would cause deadlock. This corresponds to the situation in Figure 5.8. Table 5.1 summarizes the pruning techniques of this section.

If adding this rule causes something else that the algorithm already decided to add to become redundant, then this particular combination is redundant. For example, consider the TEL structures in Figure 5.11. In the TEL structure of Figure 5.11(a), none of the rules are redundant. However, adding the rule $x+ \rightarrow w-$ produces the TEL structure of Figure 5.11(b), causing the existing rule $w+ \rightarrow w-$

Table 5.1. Pruning opportunities.

Context	$e \rightarrow f$ added to	Result	Example
$e = f$	$R - R_0$	redundant	Figure 5.3
$e = \$$	R	equivalent to a set	Figure 5.5
$f = \$$	R	equivalent to a set	Figure 5.6
$e \rightarrow_{R-R_0}^* f$	$R - R_0$	redundant	Figure 5.7
$f \rightarrow_{R-R_0}^* e$	$R - R_0$	deadlock	Figure 5.8
oneToken(e, f, R, R_0)	R_0	redundant	Figure 5.9
$e \rightarrow_{R-R_0}^* f$	R_0	safety violation	Figure 5.10

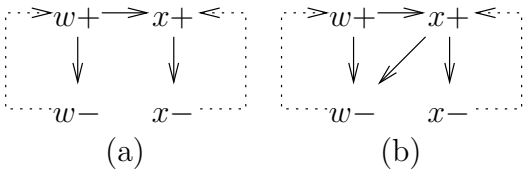

Figure 5.11. Adding a rule that makes another rule redundant. Part (a) shows the TEL structure before adding $x+ \rightarrow w-$. Part (b) shows the TEL structure after adding $x+ \rightarrow w-$.

to become redundant. The algorithm avoids adding this rule and prunes this branch of the search.

As the algorithm adds new rules into a TEL structure, it also introduces new paths. Therefore, it applies this reachability analysis recursively at each level of search tree. Adding a rule can remove multiple other candidates from consideration. The amount by which the pruning techniques of this section reduce the search space is dependent on the graph.

5.2.4 Conflicts

Conflicts provide more opportunities for pruning. Given that two conflicting events cannot both occur during the same iteration of the execution of a circuit, adding a rule for which the enabling and enabled events conflict changes the behavior and can cause deadlock. For example, starting from the TEL structure of Figure 5.12(a), adding the rule $R_0+ \rightarrow R_1+$ to produce the TEL structure of Figure 5.12(b) introduces deadlock.

5.3 Pruning Poorly-Performing Solutions

This section presents a cost metric to rate the performance of the solutions to the concurrency reduction problem. Mercer's stochastic cycle period analysis [55] can estimate the performance of a given TEL structure without actually synthesizing it. This makes use of explicit timing assumptions provided by the user in the specification. These assumptions are propagated by the compiler to the channel-level TEL structure, and from there, to the most-concurrent, signal-level TEL structure. As

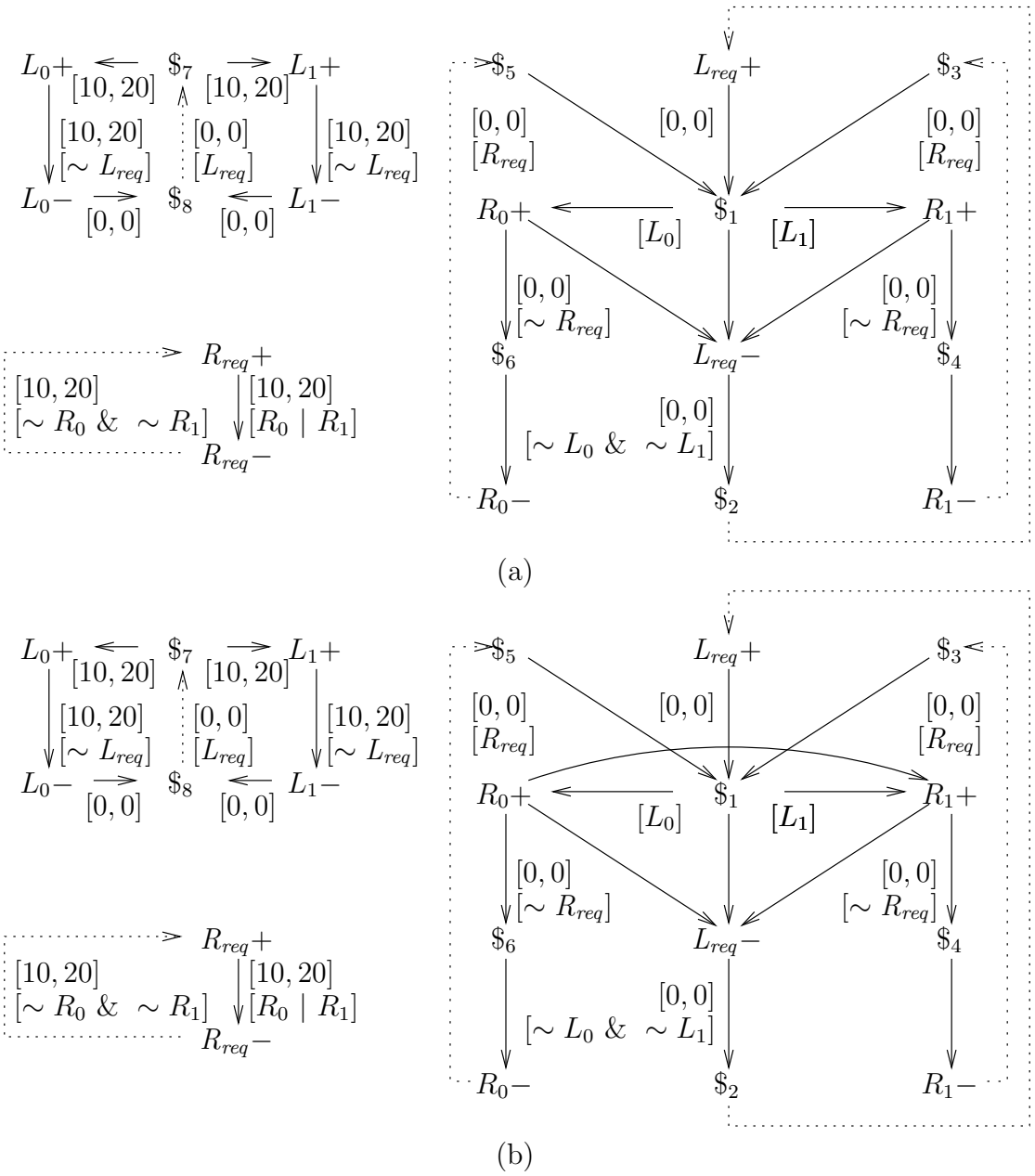

Figure 5.12. Adding a rule for which the enabling and enabled events conflict. Part (a) shows the TEL structure before adding $R_0+ \rightarrow R_1+$. Part (b) shows the TEL structure after adding $R_0+ \rightarrow R_1+$. (Conflicts: $R_0 + \pm \#_{set} R_1 + \pm$).

the concurrency-reduction search engine adds rules into the TEL structure, it uses the timing bounds *min* and *max* of Definition 5.1 on page 105. These timing bounds characterize the target gate library. Using these bounds makes sense because the extra sequencing implied by the added rules must ultimately be implemented using gates from the library. For any given TEL structure with explicit, finite timing bounds on each rule, Mercer's analysis estimates the cycle period of the design.

With cost defined in terms of the stochastic cycle period, this section presents a *branch and bound* technique [25] to prune poorly performing solutions from the search space. This requires two additional functions. The *branching* function decides, at each node of the search tree, which branch to explore first. For branch and bound to work well, this function should tend to guide the search toward the leaf that represents the least cost solution as quickly as possible. The *bounding* function should give a lower bound on the cost of any leaf reachable from a given internal node of the search tree. If the lower bound is higher than the cost of the best solution found so far, there is no need to explore the subtree rooted at that interior node.

For the branching function, this dissertation uses a generalization of the technique of the beginning of this chapter. Rules that cause the idle phase of a given communication to wait for the active phase of the next communication in series are selected as *promising* candidates. For each promising candidate, the branching function considers adding the candidate before it considers omitting the candidate. For each candidate that is not promising, the branching function considers omitting the candidate before it considers adding the candidate.

For the bounding function, this dissertation applies Mercer's stochastic cycle period analysis to the TEL structure derived from the current internal node by omitting all of the candidate rules below that node. This yields a lower bound on the cycle period of any solution reachable from this node, assuming that adding a rule to a TEL structure can only further serialize that TEL structure, and therefore, it cannot improve the cycle period of that TEL structure.

5.4 State-Variable Insertion

Sometimes even an optimal reshuffling cannot achieve complete state coding. In such cases, it is necessary to insert a state variable. The reshuffling techniques of this chapter can be extended to include state variable insertion. To add a state variable, the tool introduces a new signal, say CSC, as shown in Figure 5.13. For this new signal, the algorithm introduces its rising transition, $CSC+$, and its falling transition $CSC-$. It also adds the rule $CSC+ \rightarrow CSC-$ to R and the rule $CSC- \rightarrow CSC+$ to R_0. Initially, these are the only rules that involve these new events. However, the algorithm treats these new events as part of the output process. Starting from this initial TEL structure, the tool starts the reshuffling techniques described in this chapter. Thus, deciding where to insert the state variable becomes a special case of concurrency reduction.

5.4.1 The Search Space

In general, the search space for state variable insertion is infinite, because there is no limit on the number of state variables that can be inserted. Even inserting just one state variable can significantly increase the size of the state space compared to reshuffling alone. Adding the two events into an existing output process with event

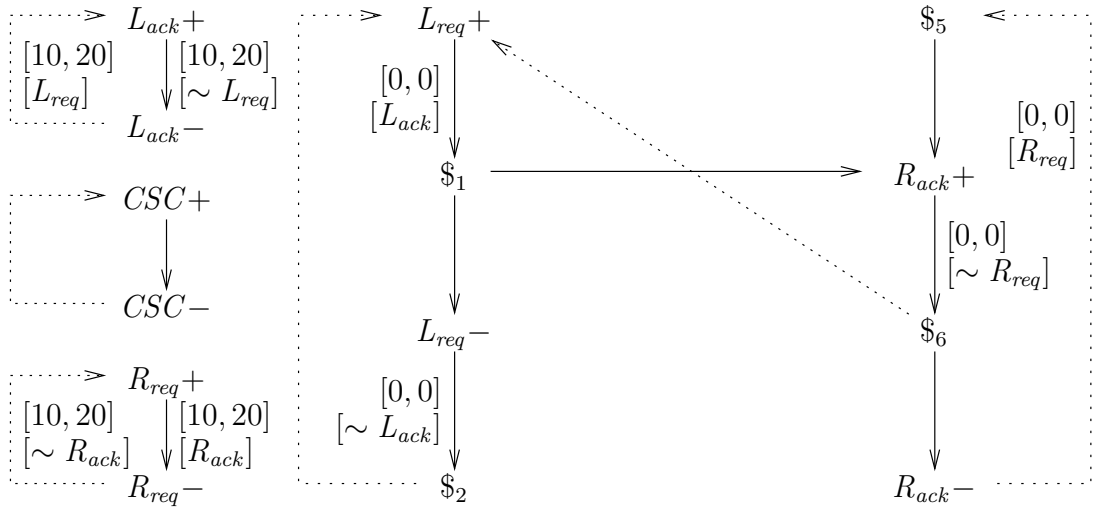

Figure 5.13. Adding an initially unconstrained state variable.

set E could introduce as many as $2\,|E|$ candidate rules. Thus, it could multiply the size of the reshuffling search space by as much as a factor of $3^{2|E|}$.

5.4.2 Pruning the Search Space

The techniques of Section 5.2 apply in the presence of the new state variable as well. Certain protocols have natural places to insert state variables. For example, inserting a state variable change between each adjacent pair of sequential communication actions ensures complete state coding. This policy can be selected by the user and used to construct the initial, concurrent TEL structure. Quality estimates can guide the state variable insertion search as well. The user can place bounds on the number of state variables to be inserted, in order to meet area requirements. Such bounds make the search space finite.

CHAPTER 6

HEURISTICS

This chapter presents pruning techniques that are known to be inexact in that they sometimes prune away synthesizable solutions. In contrast *exact* means finding the exactly optimal solution or finding the entire solution space. If a pruning technique is *inexact* it just means that it might prune away a solution that would have been synthesizable. This issue is distinct from *correctness*. A solution is correct if it still meets the constraints of the given channel-level specification and of the target protocol. Even exact pruning techniques can greatly reduce the size of the search space from its theoretical upper bound. However, for large examples, exact pruning techniques are not sufficient. Even with exact pruning techniques in place, the search space may still be too large. In such cases, it is necessary to employ *heuristic pruning techniques*. These techniques may prune away synthesizable solutions. However, they may be necessary to keep the search space manageable.

6.1 Static Heuristics

This section introduces *static heuristics*. These are heuristics that change the initial most-concurrent starting point and the list of candidate rules before concurrency reduction begins. In particular, Section 6.1.1, Section 6.1.2, and Section 6.1.3 force certain rules to be excluded from any solution. In contrast, Section 6.1.4 considers adding rules to the most-concurrent starting point, thus mandating that these rules be included in any solution. Either approach reduces the number of candidate rules and hence the number of levels in the search tree.

6.1.1 Timed Concurrency

The search engine that Chapter 5 presents attempts to reduce the concurrency in a design by adding rules to its TEL structure.

Definition 6.1 Consider two events e and f in a design. These events are *timed-concurrent*, denoted $e \parallel_t f$, if and only if e and f can be simultaneously enabled and can fire in either order, considering timing [42]. Given a reduced state graph $RSG = (I, O, T, S, \delta, \lambda_S)$, and two events $e, f \in I \cup O$:

$$e \parallel_t f \iff \exists s, s', s'' \in S \ . \ (s, e, s') \in T \wedge (s, f, s'') \in T$$

Consider a candidate rule $e \rightarrow f$. If events e and f are not concurrent, then adding $e \rightarrow f$ would not reduce the concurrency of the design, and therefore, is not a useful candidate rule. To determine whether two events are timed concurrent, the algorithm must first find the timed state space derived from the current TEL structure. Given the resulting reduced state graph, the algorithm considers only candidate rules of the form $e \rightarrow f$, where $e \parallel_t f$.

Finding the timed state space is an expensive operation. Therefore, the tool that this dissertation presents determines the timed concurrency information only for the initially, most-concurrent TEL, and uses this information to remove entire levels from the search tree. Unfortunately, this is not exact. For example, given the TEL structure of Figure 6.1(a), timed state-space exploration yields the reduced state graph of Figure 6.2(a). In this reduced state graph, $e+ \parallel_t f+$, but $e+$ and $f+$ both precede $g+$. The technique of this section would consider candidate rules $e+ \rightarrow f+$ and $f+ \rightarrow e+$, but would eliminate candidate rules $e+ \rightarrow g+$, $g+ \rightarrow e+$, $f+ \rightarrow g+$, and $g+ \rightarrow f+$ from consideration. Adding $e+ \rightarrow f+$ produces the TEL structure of Figure 6.1(b), which yields the reduced state graph of Figure 6.2(b). Adding $e+ \rightarrow f+$ delays event $f+$ to the point that it becomes timed concurrent with event $g+$, such that candidate rules $f+ \rightarrow g+$ and $g+ \rightarrow f+$ should be reconsidered. Thus, the technique of this section is a heuristic. This example also demonstrates that adding rules to a TEL structure does not always reduce the timed concurrency of that TEL structure. However, causal orderings are preserved.

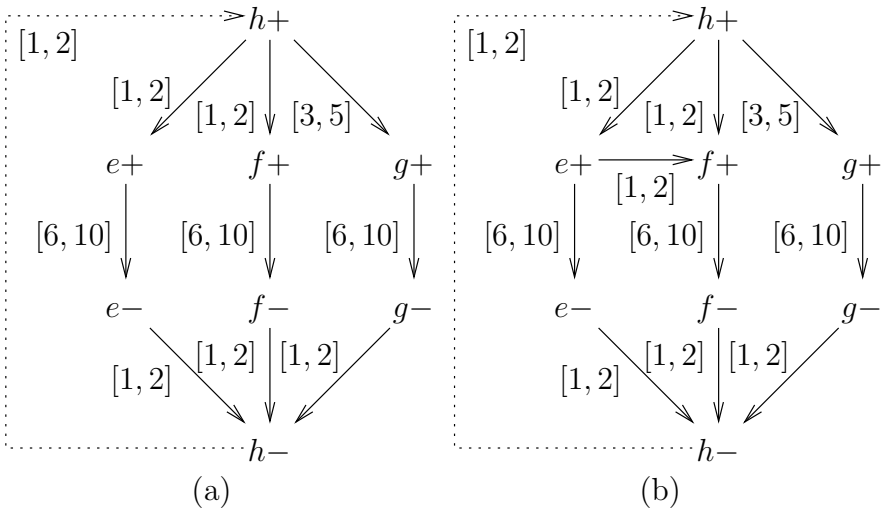

Figure 6.1. Adding a rule that shifts timed concurrency. Part (a) shows the TEL structure before adding $e+ \rightarrow f+$. Part (b) shows the TEL structure after adding $e+ \rightarrow f+$.

In the above example, if the initial TEL structure had contained a rule from $f+$ to $g+$, then the rule $e+ \rightarrow f+$ could not force $f+$ to become concurrent with $g+$.

6.1.2 Preserving User-Specified Concurrency

One possible criterion for eliminating candidate rules from consideration is to restrict the type of concurrency that may be reduced. There are two main types of concurrency: that which is inherent in the channel-level specification, and that which is an aspect of the chosen protocol. For example, if the specification executes a parallel `receive` operation on multiple channels, it is inherent in the specification that the communication on these channels takes place concurrently. In contrast, the fact that certain four-phase protocols allow certain return-to-zero events to be overlapped with events from other communications is an artifact of the protocol. In some contexts, reducing concurrency that is inherent in the channel-level specification may be considered incorrect. In such contexts, eliminating rules that would reduce such concurrency would be an exact pruning technique. Even in contexts where such concurrency reduction is permissible, choosing to avoid it is one possible heuristic to reduce the size of the search space.

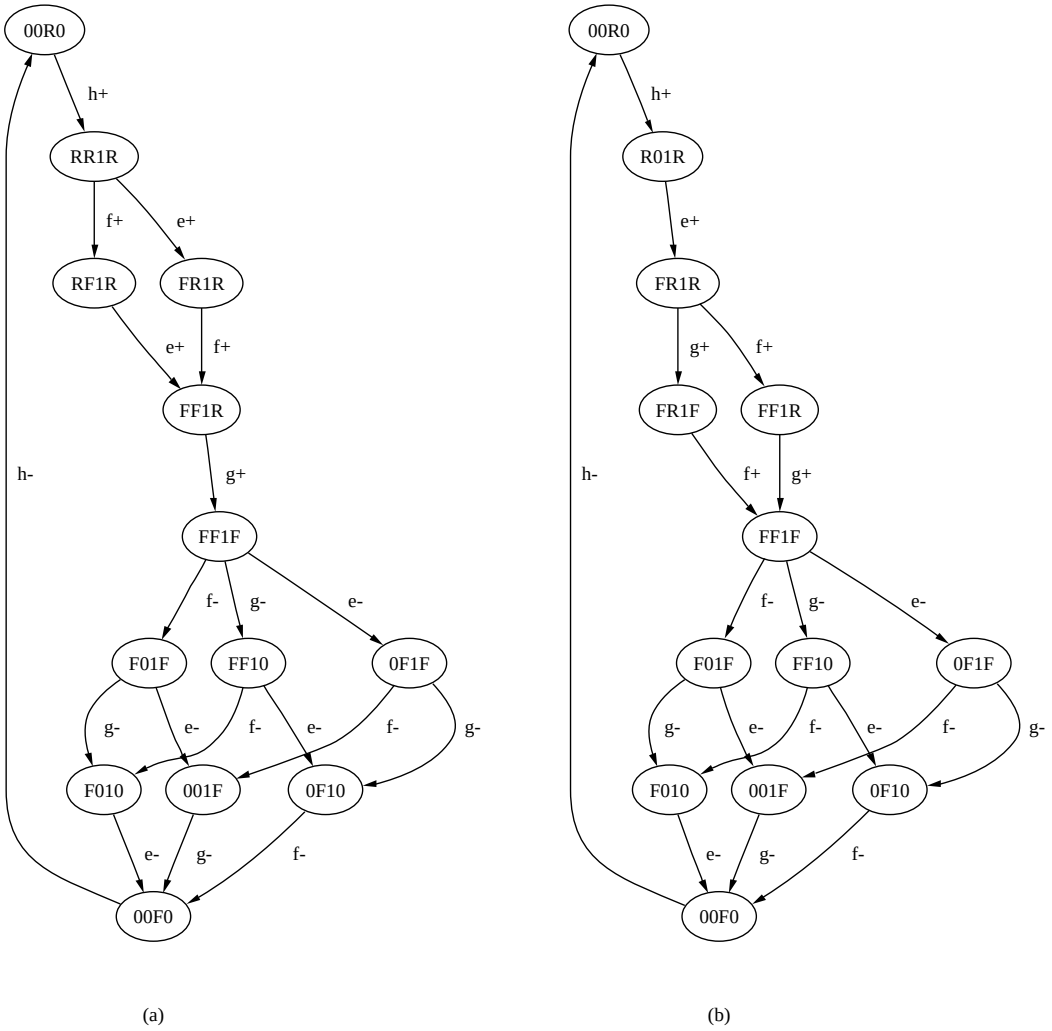

Figure 6.2. RSGs derived from TEL structures that differ only in $e+ \rightarrow f+$. Each state s labeled with $\lambda_S(s)(e)\lambda_S(s)(f)\lambda_S(s)(h)\lambda_S(s)(g)$. Part (a) was derived before adding $e+ \rightarrow f+$, and part (b) was derived after adding $e+ \rightarrow f+$.

For example, Consider Berkel's *PAR* component [10]. This component has three ports A, B and C. On each iteration, the *PAR* component waits until there is a pending communication on A. Then it communicates on the B and C channels in parallel, before finally completing the communication on A. The following channel-level VHDL code specifies this example. Recall from Section 2.1 that each channel port must be declared in `inout` mode, because control information flows in both directions.

```
entity PARsource is
  port(outgoing :  inout channel := init_channel(sender => timing(1, 2)));
end PARsource;
architecture behavior of PARsource is
  signal x :  std_logic_vector( 2 downto 0 ) := "000";
begin
  PARsource :  process
  begin
    send(outgoing, x);
    --@synthesis_off
    x <= x + 1;
    --@synthesis_on
  end process PARsource;
end behavior;
-------------------------------------------------------------
entity PARsink is
  port(incoming :  inout channel :=
       init_channel(receiver => timing(5, 10)));
end PARsink;
architecture behavior of PARsink is
  signal y :  std_logic_vector( 2 downto 0 ) := "000";
begin
  PARsink :  process
  begin
    receive(incoming, y);
  end process PARsink;
end behavior;
-------------------------------------------------------------
entity PAR is
  port(A : inout channel := init_channel(receiver => timing(1, 2));
       B, C : inout channel := init_channel(sender => timing(1, 2)));
end PAR;
architecture behavior of PAR is
  signal bb, cc :  std_logic_vector( 2 downto 0 ) := "000";
begin
  doPAR : process
  begin
    await(A);
    send(B, bb, C, cc);
    receive(A);
  end process doPAR;
```

```
end behavior;
-------------------------------------------------------------
```

From this channel-level VHDL, the compiler and the initial expander of the tool that this dissertation presents automatically derives the starting-point for concurrency reduction shown in Figure 6.3 Without preserving user-specified concurrency, there are 22 candidate rules. Preserving user-specified concurrency eliminates the rules between b and c, cutting the number of candidate rules to 12.

6.1.3 Using Only Local Concurrency Reduction

Another possible criterion is to limit the number of sequential communication actions that a candidate rule may span. This restricts the search to local concurrency reduction instead of global concurrency reduction. Local concurrency reduction is often sufficient to obtain a solution. The focus narrows the search space. For example, consider the four communications in series of Figure 4.2 on page 62. If one allows each candidate rule to span a series difference of just one communication, then candidate rules between A and C and between B and D would be not be considered.

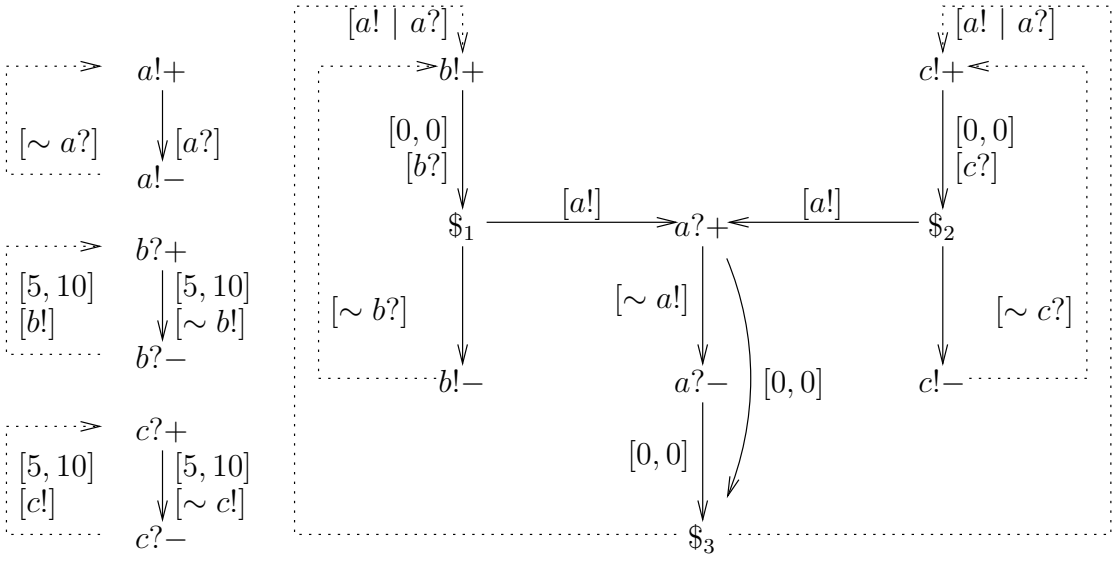

Figure 6.3. Starting point for concurrency reduction of the *PAR* example.

6.1.4 Mandating Rules

Adding rules to the initial concurrent TEL structure, beyond those that would be required for correctness, reduces the number of candidate rules and hence the number of levels in the search tree, thus dramatically reducing the size of the search space. One way to do this is by applying templates of rules that must be included whenever two communications occur in series.

For example, the interleaving technique of the beginning of Chapter 5 can be enforced in this way. For every pair of adjacent communication actions in series, this technique mandates adding rules that force the working phase of both communications to complete before the idle phase of either communication can commence. Mandating these rules greatly reduces the size of the search space.

The data constraints of Chapter 4 operate on pairs of communication actions consisting of `receive` and `send` operations using the same datum. Furthermore, those constraints are mandatory to achieve correct operation of a given data path. In contrast, the constraints used for the heuristics of this section operate on consecutive communication actions no matter what datum is involved. Furthermore, these constraints are not necessary for correct operation. In fact, they may exclude certain solutions. However, they may be selected in order to reduce the size of the search space.

The following VHDL code illustrates the different situations to which the different types of constraints apply.

```
P : process
begin
  receive(A, x);  -- communication 1
  receive(C, y);  -- communication 2
  send(B, x);  -- communication 3
  receive(C, z);  -- communication 4
end process P;
```

The sequencing constraints of this section apply to each pair of adjacent communication actions: $(1, 2)$, $(2, 3)$, $(3, 4)$, and $(4, 1)$. The data constraints of Chapter 4 apply to the `receive` and subsequent `send` that use signal x. This is the communications pair $(1, 3)$. The reuse constraint of Chapter 4 applies to the communications that share channel C. The relevant pairs are $(2, 4)$ and $(4, 2)$.

6.2 Dynamic Heuristics

This section describes *Dynamic heuristics*. These are heuristics that make pruning decisions during exploration of the search tree.

6.2.1 Choice

Conflicts provide more opportunities for pruning. Consider a rule from event e to event f. If f is in one branch of a choice, but e is not in any branch of the choice, there must also be a rule from e to some event in each other branch of the choice. Otherwise, a safety violation results. For example, starting with the TEL structure of Figure 6.4(a), adding the rule $z- \rightarrow x-$ produces the TEL structure of Figure 6.4(b), which has a safety violation. Assume that all signals are initially low. Suppose that the following sequence of events fires: $y+$, $z+$, $z-$, $y-$, $y+$, $z+$, $z-$. At this point, the rule $z- \rightarrow x-$ has two tokens on it. Hence, the TEL structure of Figure 6.4(b) is not one-safe. One solution is to also add the rule $z- \rightarrow y-$.

Definition 6.2 Given a TEL structure $T = (N, S_0, A, E, R, R_0, \#)$, and an event $e \in E$, let

$$\text{conflicts}(\#, e) = \{f \mid e \# f\}$$

Definition 6.3 Define a *clique* to be a maximal set of events that have the same conflicts. Given a TEL structure $T = (N, S_0, A, E, R, R_0, \#)$, E is partitioned into cliques. Given an event $e \in E$,

$$\text{clique}(E, \#, e) = \{f \in E \mid \text{conflicts}(\#, e) = \text{conflicts}(\#, f)\}$$

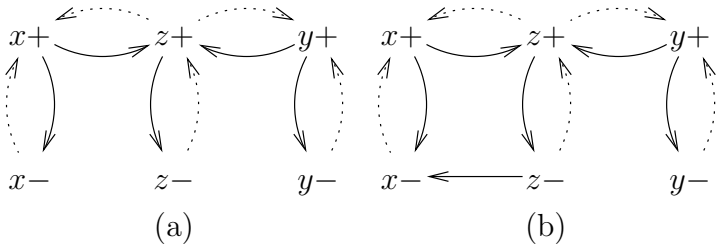

Figure 6.4. Adding a rule that causes a safety violation. Part (a) shows a one-safe TEL structure. Part (b) shows a version that is not one-safe. ($x \pm \#_{set} y \pm$).

Definition 6.4 Given a TEL structure $T = (N, S_0, A, E, R, R_0, \#)$, let the *cliques* function return the collection of cliques into which E is partitioned.

$$\text{cliques}(E, \#) = \{\text{clique}(E, \#, e) \mid e \in E\}$$

Note that $\bigcup \text{cliques}(E, \#) = E$ and $\bigcap \text{cliques}(E, \#) = \emptyset$. For example, for the TEL structure of Figure 6.5, the cliques are $\{g+, g-, h+, h-, i+, i-\}$, $\{a+, a-\}$, $\{b+, b-\}$, $\{c+, c-\}$, $\{d+, d-, e+, e-\}$, and $\{f+, f-\}$.

Definition 6.5 Given a TEL structure $T = (N, S_0, A, E, R, R_0, \#)$, and an event $e \in E$, let

$$\text{conflicting\_cliques}(E, \#, e) = \{C \in \text{cliques}(E, \#) \mid C \subseteq \text{conflicts}(\#, e)\}$$

Definition 6.6 Define the *choice* of an event to be the set of cliques that includes its own clique and those cliques whose events conflict with it. Given a TEL structure $T = (N, S_0, A, E, R, R_0, \#)$, and an event $e \in E$, let

$$\text{choice}(E, \#, e) = \{\text{clique}(E, \#, e)\} \cup \text{conflicting\_cliques}(E, \#, e)$$

Note that $\text{conflicting\_cliques}(E, \#, e) \subset \text{choice}(E, \#, e) \subseteq \text{cliques}(E, \#)$. Each member clique of a choice is also called a *branch* of the choice.

Definition 6.7 Given a TEL structure $T = (N, S_0, A, E, R, R_0, \#)$, and an event $e \in E$, let the *choices* function return the set of choices present in the TEL structure:

$$\text{choices}(E, \#) = \{\text{choice}(E, \#, e) \mid e \in E\}$$

Note that $\text{choices}(E, \#) \subseteq 2^{\text{cliques}(E, \#)}$.

For example, the TEL structure of Figure 6.5 has two choices. One of the choices is $\{\{a+, a-\}, \{b+, b-\}, \{c+, c-\}\}$. This choice has three branches. The other choice is $\{\{d+, d-, e+, e-\}, \{f+, f-\}\}$. This choice has two branches.

These definitions and observations lead to the following condition on the rules that the algorithm adds that conservatively preserves the one-safe property of the final TEL structure: From any event e that is outside of any given choice C,

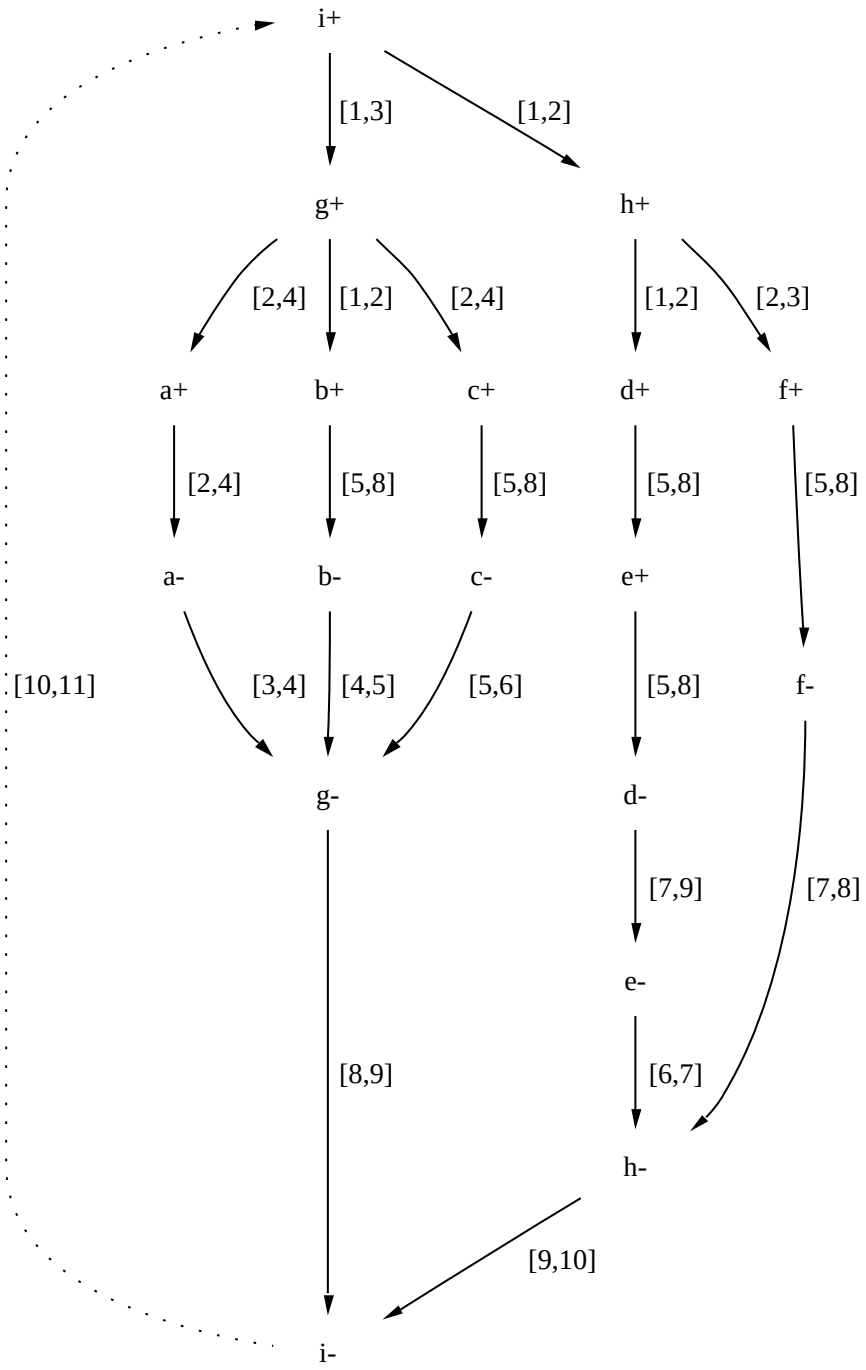

Figure 6.5. A TEL structure illustrating cliques.
$(a \pm \#_{set} b \pm \ \wedge \ a \pm \#_{set} c \pm \ \wedge \ b \pm \#_{set} c \pm \ \wedge \ d \pm \#_{set} f \pm \ \wedge \ e \pm \#_{set} f \pm)$.

there must either be rules to none of the branches of C or rules to each branch of C. Thus, the heuristic of this section assumes that for a TEL structure $T = (N, S_0, A, E, R, R_0, \#)$ to be one-safe, the following condition must hold:

$$\begin{array}{c} \forall C \in \text{choices}(E, \#), e \in E - \bigcup C \; . \\ \exists B \in C \; . \; \exists \, (e, f, l, u, B) \in R \; . \; f \in B \\ \Downarrow \\ \forall B' \in C \; . \; \exists \, (e, f', l', u', B') \in R \; . \; f' \in B' \end{array} \tag{6.1}$$

In other words, if the algorithm adds a rule from event e to any branch of a choice, then it must add a rule from event e to each branch of that choice. Therefore, as the algorithm searches the decision space, it prunes any subtrees rooted at decisions that would make it impossible to satisfy the above condition. Before deciding to add a rule from event e into a branch of a choice, it checks that it is still possible to add rules to each branch of that choice. The following definitions formalize this condition.

Definition 6.8 Given a TEL structure $T = (N, S_0, A, E, R, R_0, \#)$, and an event $e \in E$, let

$$\text{successors}(R, e) = \{f \mid \exists \, (e, f, l, u, B) \in R\}$$

At a given level of the search tree (for example, Figure 5.2), let $R'_c \subseteq R_c$ be the set candidate rules about which the algorithms has not yet made a decision. In other words, R'_c is the set of candidates corresponding to the levels of edges beneath the current node of the search tree. Before the algorithm explores a branch of the search tree that would add a candidate rule $(e, f, l, u, B) \in R'_c$ to a TEL structure $T = (N, S_0, A, E, R, R_0, \#)$, the following condition must hold.

$$\begin{array}{c} \forall C \in \text{conflicting\_cliques}(E, \#, f) \; . \\ C \cap (\text{conflicts}(\#, e) \cup \text{successors}(R \cup R'_c, e)) \neq \emptyset \end{array}$$

If the above condition is not met, the algorithm prunes the branch of the search tree that would add (e, f, l, u, B) to the TEL structure. Similarly, before the algorithm explores a branch of the search tree that would decide to omit a candidate rule $r = (e, f, l, u, B) \in R'_c$, it checks that this candidate rule is not the last remaining way to

satisfy the condition of Equation (6.1). Thus, the following is a necessary condition to consider omitting r from the TEL structure $T = (N, S_0, A, E, R, R_0, \#)$:

$$\begin{gathered}\text{successors}(R, e) \cap \text{conflicts}(\#, f) = \emptyset \\ \vee \\ \text{clique}(E, \#, f) \cap \text{successors}(R \cup R'_c, e) \neq \emptyset\end{gathered}$$

If the above condition is not met, the algorithm prunes the branch of the search tree that would omit r from the TEL structure.

Similarly, adding a rule that exits some branch of a choice without adding a rule that exits each branch of that choice causes deadlock. For example, the TEL structure of Figure 6.6(a) is deadlock free. However, the TEL structure of Figure 6.6 (b) suffers from the following deadlock condition. Assume that all signals are initially low. Then suppose that the following sequence of events occurs: $y+$,$z+$,$y-$,$y+$,$y-$. At this point no rules are enabled, and hence no further progress is possible. The system is in a state of deadlock.

These observations lead to the following condition on the rules that the algorithm adds that conservatively preserves deadlock-freedom of the final TEL structure: From any event e that is outside of any given choice C, there must either be rules to none of the branches of C or rules to each branch of C. Thus, the heuristic of this section assumes that for a TEL structure $T = (N, S_0, A, E, R, R_0, \#)$ to be deadlock-free, the following condition must hold:

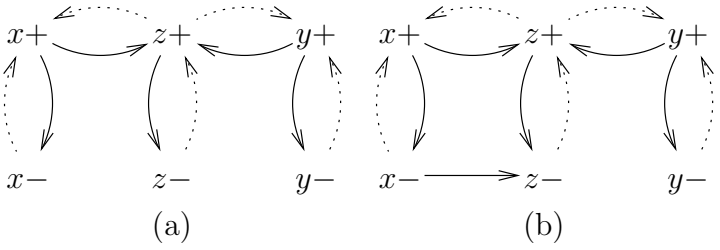

Figure 6.6. Adding a rule from only one branch of a choice. Part(a) shows the TEL structure before adding $x- \rightarrow z-$. Part(b) shows the TEL structure after adding $x- \rightarrow z-$. (Conflicts: $x \pm \#_{set} y\pm$).

$$\begin{array}{c} \forall C \in \text{choices}(E, \#), f \in E - \bigcup C \ . \\ \exists B \in C \ . \ \exists \, (e, f, l, u, B) \in R \ . \ e \in B \\ \Downarrow \\ \forall B' \in C \ . \ \exists \, (e', f, l', u', B') \in R \ . \ e' \in B' \end{array} \tag{6.2}$$

In other words, if the algorithm adds a rule to event f from any branch of a choice, then it must add a rule to event f from each branch of that choice. Therefore, as the algorithm searches the decision space, it prunes any subtrees rooted at decisions that would make it impossible to satisfy the above condition. Before deciding to add a rule to event f from a branch of a choice, it checks that it is still possible to add rules from each branch of that choice. The following definitions formalize this condition.

Definition 6.9 Given a TEL structure $T = (N, S_0, A, E, R, R_0, \#)$, and an event $f \in E$, let

$$\text{predecessors}(R, f) = \{e \mid \exists \, (e, f, l, u, B) \in R\}$$

Before the algorithm explores a branch of the search tree that would add a candidate rule $(e, f, l, u, B) \in R'_c$ to a TEL structure $T = (N, S_0, A, E, R, R_0, \#)$, the following condition must hold.

$$\begin{array}{c} \forall C \in \text{conflicting\_cliques}(E, \#, e) \ . \\ C \cap (\text{conflicts}(\#, f) \cup \text{predecessors}(R \cup R'_c, f)) \neq \emptyset \end{array}$$

If the above condition is not met, the algorithm prunes the branch of the search tree that would add (e, f, l, u, B) to the TEL structure. Similarly, before the algorithm explores a branch of the search tree that would decide to omit a candidate rule $r = (e, f, l, u, B) \in R'_c$, it checks that this candidate rule is not the last remaining way to satisfy the condition of Equation (6.2). Thus, the following is a necessary condition to consider omitting r from the TEL structure $T = (N, S_0, A, E, R, R_0, \#)$:

$$\begin{array}{c} \text{predecessors}(R, f) \cap \text{conflicts}(\#, e) = \emptyset \\ \vee \\ \text{clique}(E, \#, e) \cap \text{predecessors}(R \cup R'_c, f) \neq \emptyset \end{array}$$

If the above condition is not met, the algorithm prunes the branch of the search tree that would omit r from the TEL structure.

The technique of this section is a heuristic, because as stated, it does not take in account nested choice. For example, given the TEL structure of Figure 6.7(a), the technique of this section would not consider adding the set of rules $\{g+ \rightarrow b+, g+ \rightarrow c+\}$ to produce the TEL structure of Figure 6.7(b). The technique of this section would assume that the structure of Figure 6.7(b) would not be one-safe, because there is a rule from $g+$ to the clique $b\pm$ but no rule from $g+$ to the conflicting cliques $d\pm$ and $e\pm$. However, the TEL structure of Figure 6.7(b) is actually one-safe because the added rules cover both branches of the top-level choice.

6.2.2 Assuming Each Rule Adds No New States

Consider two events e and f in a design. Ignoring timing considerations, adding a rule $e \rightarrow f$ to a TEL structure simply has the effect of imposing the constraint $e \prec f$. Imposing the constraint $e \prec f$ on the design tends to produce a subset of the states present in the state graph without the constraint. In particular, the states

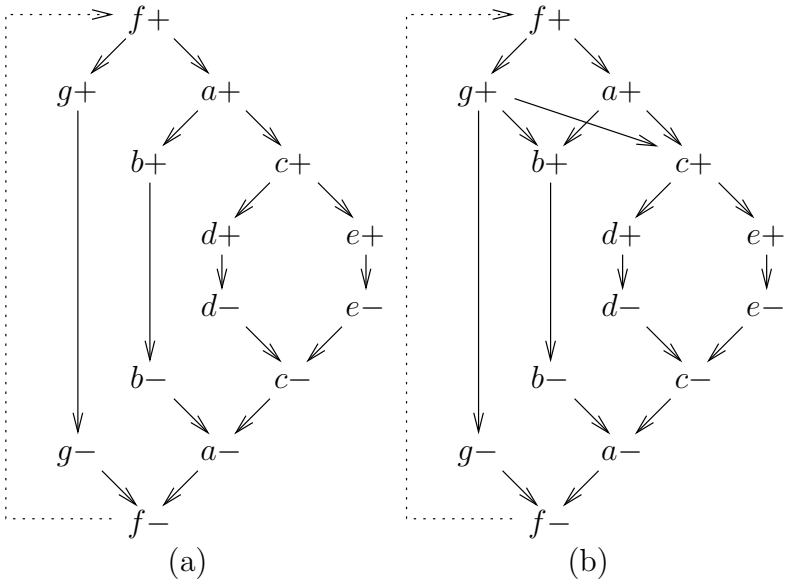

Figure 6.7. Timed event/level structure with nested choice. Part (a) shows the TEL structure before adding the rules $g+ \rightarrow b+$ and $g+ \rightarrow c+$. Part (b) shows the TEL structure after adding the rules $g+ \rightarrow b+$ and $g+ \rightarrow c+$. $(b \pm \#_{set} c \pm \ \wedge \ b \pm \#_{set} d \pm \ \wedge \ b \pm \#_{set} e \pm \ \wedge \ d \pm \#_{set} e \pm)$.

that violate $e \prec f$ are eliminated from the state graph. Other states are usually unaffected, and it is rare for new states to be introduced. Similarly, removing a constraint such as $e \prec f$ tends to add states to a state graph.

This leads us to a heuristic pruning technique. Consider two TEL structures $T = (N, S_0, A, E, R, R_0, \#)$ and $T' = (N, S_0, A, E, R', R'_0, \#)$. If T' has a subset of the rules that are either present in T or made redundant by the rules in T, then T' has a subset of the constraints of T. In this case, the heuristic of this section assumes that the state graph derived from T' has a superset of the states of the state graph derived from T.

Definition 6.10 The $\sqsubseteq$ relation is defined on TEL structures, such that given two TEL structures $T = (N, S_0, A, E, R, R_0, \#)$ and $T' = (N', S'_0, A', E', R', R'_0, \#')$:

$$\begin{gathered} T' \sqsubseteq T \\ \Updownarrow \\ N = N' \wedge S_0 = S'_0 \wedge A = A' \wedge E = E' \wedge \# = \#' \\ \wedge \\ R' \subseteq R \cup \left\{ e \rightarrow f \mid e \rightarrow^*_{R-R_0} f \right\} \wedge R'_0 \subseteq R_0 \cup \{ e \rightarrow f \mid \text{oneToken}(e, f, R, R_0) \} \end{gathered}$$

Essentially, this means that each rule in R' would be redundant if added to R, and each rule in R'_0 would be redundant if added to R_0. Therefore, $T' \sqsubseteq T$ (which is read "T' has a subset of the constraints of T"). Let $RSG = (I, O, A, S, \delta, \lambda_S)$ and $RSG' = (I, O, A, S', \delta', \lambda'_S)$ be the state graphs derived through state-space exploration from T and T', respectively. The heuristic of this section assumes that:

$$T' \sqsubseteq T \Rightarrow S' \supseteq S \wedge \delta' \supseteq \delta \wedge \forall s \in S \; . \; \lambda'_S(s) = \lambda_S(s)$$

This means that if T' has a subset of the constraints of T, then RSG' has a superset of the states of RSG. In practice, the calls to the $\rightarrow_R$ relation and the *oneToken* function can be precomputed from the rules that are marked as redundant during the redundancy checks.

The pruning technique that this section proposes is as follows. Suppose that T and T' are two leaves of the search tree. Further suppose that the algorithm encounters T before T'. If state-space exploration on T yields the state graph RSG, and RSG has a complete state coding violation, then the algorithm inserts

T into a blacklist. When the algorithm later encounters T', if $T' \sqsubseteq T$, the algorithm rejects T' without attempting state-space exploration on T'. Given a blacklist set B, the set of leaves that this technique prunes is $\{T' \mid \exists T \in B \,.\, T' \sqsubseteq T\}$.

Searching the tree of possibilities in such a way that whenever $T' \sqsubseteq T$, the algorithm encounters T before T' maximizes the number of leaves that this technique prunes. The simplest way to approximate this order is to always explore the branch that adds a rule before the branch that does not. However, this only approximates the ideal order because of the fact that adding a rule can make previously added rules redundant. Figure 5.11 illustrates this possibility. This requires the following modification to the search order. When the algorithm discovers that adding a rule would make previously-added rules redundant, it prunes the current subtree and then skips to the branch of the search tree that does not add the now redundant rules, but instead adds the rule that made them redundant. Having explored the subtree routed at that point and marked it as explored, the algorithm returns to the place in the search tree just after that which it pruned. This strategy maximizes the number of leaves that the technique of this section can prune.

Furthermore, this technique can be extended to prune interior nodes as well. Let N be the current node of the search tree. Let T' be the leaf derived from N by choosing to omit each rule that is uncommitted at N. If $\exists T \in B \,.\, T' \sqsubseteq T$, then the algorithm prunes the entire subtree rooted at N. This is because any other leaf T'' descended from N has a superset of the rules of T'. Therefore, $T'' \sqsubseteq T$.

Unfortunately, the pruning technique of this section is inexact. In particular, adding a rule to a TEL structure may introduce new states into the reduced state graph computed by timed state-space exploration. One cause of this is the conjunctive timing semantics of TEL structures. A given event is not forced to occur until all the rules that enable it have expired. Even if some of the rules that enable an event f have expired, if other rules enabling f have not yet expired, then f may not have occurred yet.

For example, consider the TEL structure of Figure 6.8(a). Without the timing bounds, this example would suffer from complete state coding violations. However,

using the timing constraints, this example achieves complete state coding. Starting from the TEL structure of Figure 6.8(a), adding the rule $(y-, z+, 0, 0, [true])$ produces the TEL structure of Figure 6.8(b). Even considering timing, the new rule results in a complete state coding violation. Figure 6.9 shows two reduced state graphs computed by timed state-space exploration. The reduced state graph of Figure 6.9(a) corresponds to the TEL structure of Figure 6.8(a). The reduced state graph of Figure 6.9(b) corresponds to the TEL structure of Figure 6.8(b). The reduced state graph of Figure 6.9(b) does have fewer states than that of Figure 6.9(a). However, the set of states in Figure 6.9(b) is not a subset of the set of states in Figure 6.9(a). The reduced state graph of Figure 6.9(b) has new states that are not present in that of Figure 6.9(a). In particular, the highlighted pair of states constitutes a complete state coding violation. Thus, choosing to avoid the TEL structure of Figure 6.9(a) just because that of Figure 6.9(b) has a complete state coding violation would overlook a solution with complete state coding. This

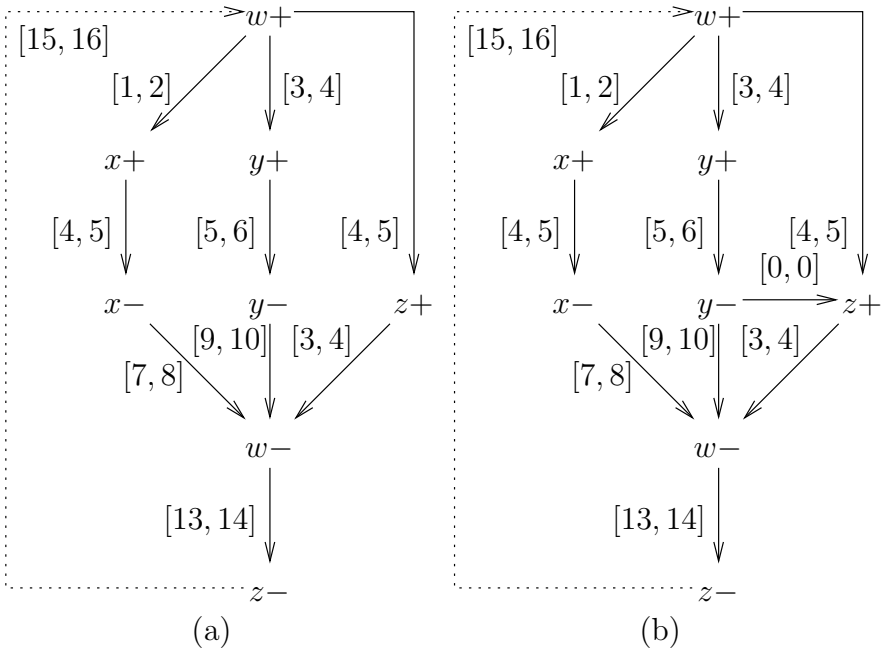

Figure 6.8. TEL structures that differ only in the rule $(y-, z+, 0, 0, [true])$. Part (a) shows the TEL structure before adding $(y-, z+, 0, 0, [true])$. Part (b) shows the TEL structure after adding $(y-, z+, 0, 0, [true])$.

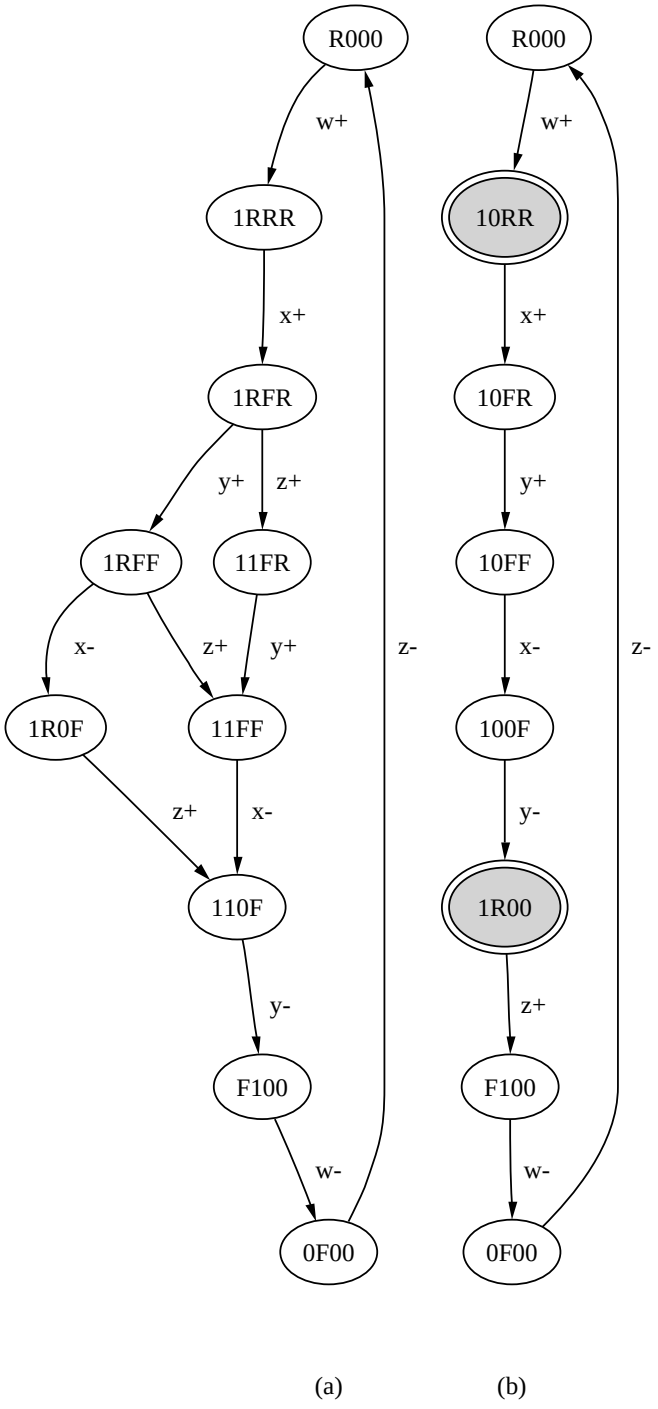

Figure 6.9. Reduced state graphs demonstrating that adding a rule can introduce a complete state coding violation. Each state s is labeled with $\lambda_S(s)(w)\lambda_S(s)(z)\lambda_S(s)(x)\lambda_S(s)(y)$. RSG (a) was derived before adding the rule from $y-$ to $z+$, and RSG (b) was derived after adding the rule from $y-$ to $z+$.

is why the technique of this section, as it is currently implemented, is a heuristic. The precise conditions under which adding a rule to a TEL structure can introduce new states needs further investigation. Note that even when attempting synthesis on a particular TEL structure leads to a complete state coding violation, the tool that this dissertation presents can use the techniques of Section 5.4 to solve the complete state coding violation. However, even in this case, the technique of this section can still be a useful way to predict when TEL structures will lead to a superset of the complete state coding violations of another TEL structure.

6.2.3 Setting Limits

Another approach to managing a large search space is to place limits on the resources consumed during the search. In particular, one could place limits on the number of solutions found. A very special case of this is to simply stop after the finding the first solution. This special case has been implemented as an option to the tool that this dissertation presents.

The remainder of this section discusses possible future extensions. For example, one might want to limit the total number of synthesis attempts made by the tool, because that is the most expensive operation. The user might also indicate requirements on the nature of the solutions found. For example, the user could place bounds on the number of rules to be added to the TEL structure. Solutions with more rules tend to be smaller but have less concurrency.

CHAPTER 7

RESULTS AND CASE STUDIES

We have implemented the techniques that this dissertation presents within the CAD tool `ATACS` [59] and used the tool to test these techniques on several example circuits. Most of these are from the literature on asynchronous circuits. On each member of this set of benchmark examples, there are several possible metrics with which to evaluate the results.

For any given example, the first aspect of success is the ability to produce a circuit that correctly implements the specification. Without this much success, the remaining metrics discussed below are moot points.

Another metric is the computing resources necessary to find and synthesize the solutions. Reducing the CPU time and memory required to compile the example frees computing resources to work on other tasks. Furthermore, these figures affect the cost of the equipment necessary to handle the example.

This chapter focuses on the run-time required to obtain each result, and on the average cycle period of that result, as determined by Mercer's stochastic cycle period analysis [55]. This allows the designer to incorporate known information about the delays of the environment (for example, the data path) and to evaluate how the solution presented will affect overall system performance.

For any metric, the validity depends strongly on the set of the benchmark examples chosen. The more representative these are of real designs, the more valid the measurement of success is. This chapter attempts to present a reasonably representative set of examples.

7.1 Exhaustive Results

This section presents the results of applying the techniques of Section 5.2. These techniques find the entire solution space for each example. This section presents various small examples. For each example, it presents the resources required to find the entire solution space, certain statistics about the solution space, and comments on the solutions found. This section presents the results of applying the concurrency reduction techniques of Chapter 5 to several example buffers from Chapter 4.

Consider the push buffer requiring no isochronic fork between control and the data path, shown in Figure 4.16(a) on page 86. Applying concurrency reduction to the constraints of Figure 4.16(b) on page 86 finds two solutions. These solutions correspond to two of the solutions that Burns found by hand for these constraints [20]. The other solutions from [20] for these constraints require a vacuous event. Vacuous events are not directly captured by the techniques of Chapter 5.

Now consider the push buffer that does require an isochronic fork between the control and data path, shown in Figure 4.18(a) on page 89. Applying concurrency reduction to the constraints in Figure 4.18(b) on page 89 finds the same solutions found for the case requiring no isochronic fork between control and data path, plus an additional two solutions. All four solutions correspond to solutions that Burns found by hand for these constraints [20]. Burns found an additional reshuffling that has a complete state coding violation, and hence would require a state variable. Introducing an initially unconstrained state variable before applying concurrency reduction yields 855 distinct solutions. Sixteen of these correspond to the 16 ways to solve Burns' reshuffling using one state variable. The remaining solutions do not actually require the state variable to achieve complete state coding. The remaining reshufflings from [20] for the push buffers would require vacuous events.

Consider the pull buffer using the *Latch* component shown in Figure 4.20(a) on page 92. In addition to the data constraints of Figure 4.20(b) on page 92, Burns [20] also applies the *constant response time* (CRT) constraint. An array of processes such as a FIFO composed of buffer stages obeys the CRT constraint if the response time at either end of the array is independent of the number of processes in the

array. For the pull buffer, this requires the following additional constraints [20]:

$$L_{req}+ \prec [R_{req}] \tag{7.1a}$$

$$L_{req}- \prec [\sim R_{req}] \tag{7.1b}$$

Adding rules for these constraints to the TEL structure of Figure 4.20 on page 92 produces that of Figure 7.1. Applying concurrency reduction to the TEL structure of Figure 7.1 produces two solutions. These correspond to two of the solutions that Burns found by hand for the pull buffer [20]. Each of the remaining solutions that [20] presents for the pull buffer requires either a state variable or a vacuous event. Introducing an initially unconstrained state variable before concurrency reduction yields 790 solutions, including all of Burns' reshufflings that need a state variable but no vacuous events.

Consider the push buffer using a normally transparent latch of Figure 4.9(a) on page 74. Applying concurrency reduction to the constraints in Figure 4.9(b) on page 74 yields 5,258 solutions. These include the simple and semi-decoupled solutions that Furber and Day present in [28]. The fully-decoupled and long-hold solutions from [28] require a state variable. Table 7.1 presents the results for this

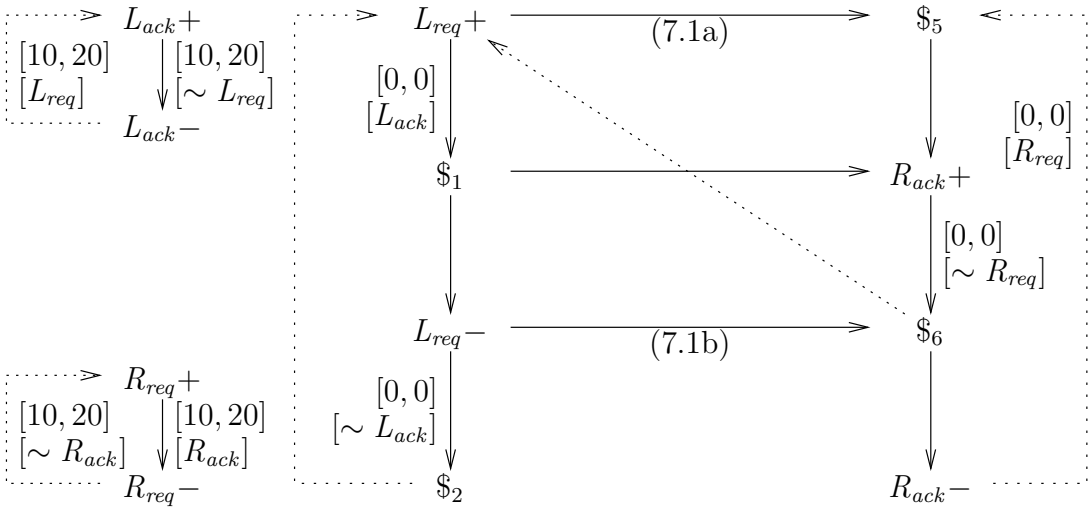

Figure 7.1. TEL structure for constraints on a pull buffer meeting the CRT constraint and using the *Latch* component.

Table 7.1. Results for myFurber.

Filters					CPU					Period	
N	P	E	O	S	time/s	levels	TELs	leaves	PRSs	max.	min.
				S	4108.63	39	13849	13849	10528	81	42
			O	S	1.91	6	16	16	3	42	42
		E		S	492.89	39	1369	1115	1	42	42
		E	O	S	3.63	6	12	8	1	42	42
	P			S	4081.28	39	13849	13849	10528	81	42
	P		O	S	2.07	6	16	16	3	42	42
	P	E		S	498.13	39	1369	1115	1	42	42
	P	E	O	S	3.62	6	12	8	1	42	42
N					12.38	39	141	141	1	42	42
N				S	12.57	39	141	141	1	42	42
N			O		0.71	6	5	5	1	42	42
N			O	S	0.73	6	5	5	1	42	42

The command `atacs -oi -oR -oq -oD -tp -G1-2 myFurber.tel` [`-P`*filter*]... `-ys` generated each row of this table.

buffer. Filter N simply stops the search after the first solution found. Filter P preserves user-specified concurrency. It corresponds to the technique of Section 6.1.2. Filter E uses a branch-and-bound technique to prune expensive solutions. That is the filter of this Section 5.3. Filter O prunes candidate rules between events that are timed-ordered in the most-concurrent TEL structure. That is the technique of Section 6.1.1. Filter S prunes possibilities that have a subset of the rules of a possibility that is known to have a complete state coding violation. That is the filter of Section 6.2.2. The *levels* column displays the number of levels in the search tree. The $TELs$ column displays the number of distinct TEL structures considered. The *leaves* column displays the number of leaves of the search tree that are encountered. The $PRSs$ column displays the number of solutions (circuits) found. The *Period* column displays the estimate of the average cycle period. Within this column, the *max.* subcolumn displays the average cycle period for the slowest circuit found using the options for its row. Each entry in the *min.* subcolumn displays the average cycle period for the fastest circuit found using the options for its row.

The footnote to the table shows the options that were given to `ATACS`. `-oi`

turns off the state-variable `insertion` method of Krieger [42]. `-oR` turns on the automatic concurrency `reduction` techniques that this dissertation presents, even though this example does not come from channel-level VHDL. `-oq` suppresses the automatic display of graphs for cases such as deadlock. `-oD` allows rule `disabling` with only a warning. `-tp` selects the *Bourne-Again Poset* (`bap`) timing analysis method of Mercer et al. [57, 56]. `-G1-2` specifies that logic gates should have a delay range of $[1, 2]$. `-lt` loads the TEL structure. `-P`*filter* turns on the specified filter. For example, the third row of Table 7.1 uses ... `-PP -PO -PS`.... Finally, `-ys` invokes synthesis using the `single`-cube algorithm of Myers [59].

In Table 7.1, the solutions range in average cycle period from 42 time units to 64 time units. The branch and bound algorithm (filter E) finds the optimal solution in about one fifth of the CPU time required without this filter. The P filter, which preserves user-specified concurrency is not useful for this example, because it has no channel-level concurrency. The N filter, which halts the search after the first solution found, reduces the run time by an order of magnitude. By itself, it does not quite find the optimal solution, but it comes close. Used in conjunction with the O filter, it does find the optimal solution. Because I do not currently have a `Petrify` specification for this example, I have not performed a comparison against `Petrify` on this example.

7.2 The PAR Component

Consider Berkel's *PAR* component [10]. This component has three ports A, B and C. On each iteration, the *PAR* component waits until there is a pending communication on A. Then it communicates on the B and C channels in parallel, before finally completing the communication on A. Figure 7.2 shows the context in which the *PAR* component operates. The following channel-level VHDL code specifies this example. Recall from Section 2.1 that each channel port must be declared in `inout` mode, because control information flows in both directions.

```
entity PARsource is
  port(outgoing :  inout channel := init_channel(sender => timing(1, 2)));
end PARsource;
architecture behavior of PARsource is
```

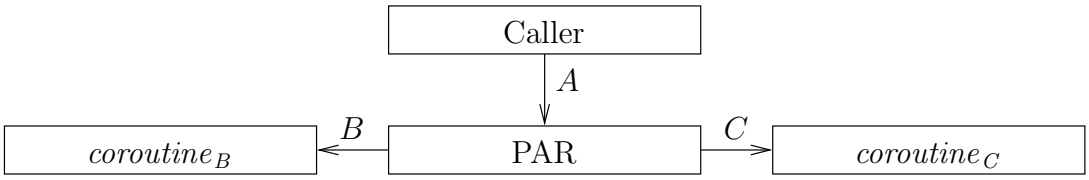

Figure 7.2. Block diagram of *PAR* and its environment.

```
  signal x :  std_logic_vector( 2 downto 0 ) := "000";
begin
  PARsource :  process
  begin
    send(outgoing, x);
    --@synthesis_off
    x <= x + 1;
    --@synthesis_on
  end process PARsource;
end behavior;
-------------------------------------------------------------
entity PARsink is
  port(incoming :  inout channel :=
       init_channel(receiver => timing(5, 10)));
end PARsink;
architecture behavior of PARsink is
  signal y :  std_logic_vector( 2 downto 0 ) := "000";
begin
  PARsink :  process
  begin
    receive(incoming, y);
  end process PARsink;
end behavior;
-------------------------------------------------------------
entity PAR is
  port(A : inout channel := init_channel(receiver => timing(1, 2));
       B, C : inout channel := init_channel(sender => timing(1, 2)));
end PAR;
architecture behavior of PAR is
  signal bb, cc :  std_logic_vector( 2 downto 0 ) := "000";
begin
  doPAR : process
  begin
    await(A);
    send(B, bb, C, cc);
    receive(A);
  end process doPAR;
end behavior;
-------------------------------------------------------------
```

Running directly from the VHDL code, the compiler and the initial expander of tool that this dissertation presents produces the starting point for concurrency reduction shown in Figure 6.3 on page 124.

Table 7.2 presents the results for synthesizing the *PAR* component using various options. The note to the table specifies the options given to `ATACS`. `-ts` specifies the `posets` timing-analysis method of Belluomini [7]. The argument `PARex` specifies using the VHDL input file `PARex.vhd`, which contains the above VHDL code. The argument `PAR` specifies that *PAR* is the component to be synthesized. Recall that each number in the *Period* column is an estimate of the average cycle period for a particular solution. The *max.* column presents the average cycle period of the

Table 7.2. Results for PARex.

Filters					CPU					Period	
N	P	E	O	S	time/s	levels	TELs	leaves	PRSs	max.	min.
					167.62	24	1382	1382	263	30	20
				S	94.25	24	806	597	215	30	20
			O		63.56	20	546	546	100	30	20
			O	S	49.12	20	394	357	100	30	20
		E			129.77	24	452	292	2	21	20
		E		S	69.45	24	251	70	2	21	20
		E	O		62.62	20	235	160	2	22	20
		E	O	S	46.72	20	175	109	2	22	20
	P				26.06	12	189	189	57	30	20
	P			S	18.53	12	142	106	49	30	20
	P		O		7.39	8	55	55	16	29	20
	P		O	S	6.73	8	53	48	16	29	20
	P	E			22.29	12	78	39	2	21	20
	P	E		S	19.42	12	73	18	2	21	20
	P	E	O		10.24	8	37	20	2	28	20
	P	E	O	S	10.21	8	37	18	2	28	20
N					0.48	24	5	1	1	21	21
N				S	0.49	24	5	1	1	21	21
N			O		0.80	20	7	5	1	22	22
N			O	S	0.73	20	7	5	1	22	22
	`Petrify`				1.26	–	–	–	1	21	21

The command `atacs -oi -oq -ts -G1-2 PARex.vhd PAR` [`-P`*filter*]... `-ys` generated each row of this table.

slowest circuit found using the options in that row. The *min.* column presents the average cycle period of the fastest circuit found using the options in that row.

7.2.1 Using the Cycle-Period Cost Function

The experiment of this section measures the impact of the branch and bound technique of Section 5.3. Comparing the rows with and without the filter E shows the effect of the branch and bound algorithm on this example. For example, compare the first and fifth rows of the table. The solutions that the branch-and-bound technique finds include the solution with the lowest average cycle period. The branch-and-bound technique reduced the number of synthesis attempts by a factor of about 4.5. However, it reduced the CPU time by only a factor of about 1.2. In some cases, the CPU time while using the branch-and-bound algorithm is actually greater than that without the filter. For example, comparing the P filter alone to the combination of the P and the E filter, the E filter still reduced the number of synthesis attempts, but the run time actually increased. The overhead is due mainly to the cost of running the simulations to compute the bounding function. Each simulation is very quick compared to each synthesis run. However, the technique runs a simulation for each internal node of the tree. There can be many more internal nodes than there are leaves of the tree, especially when the branch and bound technique succeeds in pruning away many leaves.

7.2.2 Assuming that Each Rule Adds No New States

This section determines the impact of the technique of Section 6.2.2, which assumes that if TEL structure T_1 has a superset of the rules of TEL structure T_2, and if T_1 has a complete state coding violation, then T_2 also has a complete state coding violation. Consider the S filter in Table 7.2. This filter did not affect the quality of the solutions found for the PAR component. This filter did reduce the CPU time required to find these results. For example, comparing the results using the S filter to the exhaustive results, the filter cut the number of synthesis attempts in half, and the run time by a factor of about 1.5.

7.2.3 Timed Concurrency

This section tests the impact of the technique of Section 6.1.1, which considers only candidate rules between events that are timed concurrent in the initial, most-concurrent TEL structure. Consider the *O* filter in Table 7.2. This filter does not affect the searches ability to find the optimal solution for the *PAR* example. This filter did reduce the run time required to find this optimum, often by a factor of about 3.

7.2.4 Preserving User-Specified Concurrency

This section determines the impact of the technique of Section 6.1.2, which protects user-specified concurrency. Whenever the channel-level specification contains a parallel `send` or `receive` operation, this experiment forbids adding rules between the channel communications within each parallel operation. Consider the *P* filter of Table 7.2. This filter did not affect the searches ability to find the best solution, and yet it did substantially reduce the run time. For example, relative to the exhaustive results, the *P* filter reduced run time by a factor of about 6. Thus, for an example like the *PAR* component, this filter provides significant pruning, with little overhead.

7.2.5 Stopping After The First Solution

An extremely aggressive heuristic is to simply stop after the first solution found. If the branching function of Section 5.3 succeeds in steering the search such that the first synthesis attempt succeeds, then this method is equivalent to letting the branching function determine the set of rules to include in the final TEL structure. This actually combines the techniques of Section 6.1.4, Section 6.1.2, Section 6.1.3, and Section 6.2.3. Applying this technique to the PAR example produces the last section of Table 7.2. Using this heuristic alone, the search finds a solution that is nearly optimal (within 5 percent of optimal) on its first attempt. Compared to the exhaustive approach, using this heuristic improves run time by a factor of almost 400.

7.2.6 Comparison to Petrify

The last row of Table 7.2 shows the result for this example produced by `Petrify`. The `Petrify` input is shown below.

```
.model channel
.channels a b c
.graph
a? b! c!
b! b?
c! c?
b? a!
c? a!
a! a?
.marking {<a!,a?>}
.slowenv
.end
```

For the purposes of concurrency reduction, the `.slowenv` directive serves as an approximation to the timing constraints in the `ATACS` specification. Given the above specification in the file `par.g`, the command `petrify -er -4ph -untog par.g -o par.out.4ph -redc -gc -eqn par.eqn` produced the synthesizable Petri Net. The result was input to `ATACS` and annotated with the timing assumptions of Figure 6.3 on page 124 for computation of the average cycle period and synthesis using `ATACS`. Figure 7.3 compares the result of concurrency reduction using `Petrify` to that using the tool that this dissertation presents. In particular, Figure 7.3(a) shows the TEL structure for the result from `Petrify`. Figure 7.3 shows the TEL structure best result obtained using the P and O filters. Running `ATACS` synthesis on the TEL structure of Figure 7.3(a) (the annotated Petrify result) produces the circuit of Figure 7.4(a). Running `ATACS` synthesis on the TEL structure of Figure 7.3(b) produces the circuit of Figure 7.4(b). The circuit derived from the result from `Petrify` is simpler. However, the circuit derived using the P and

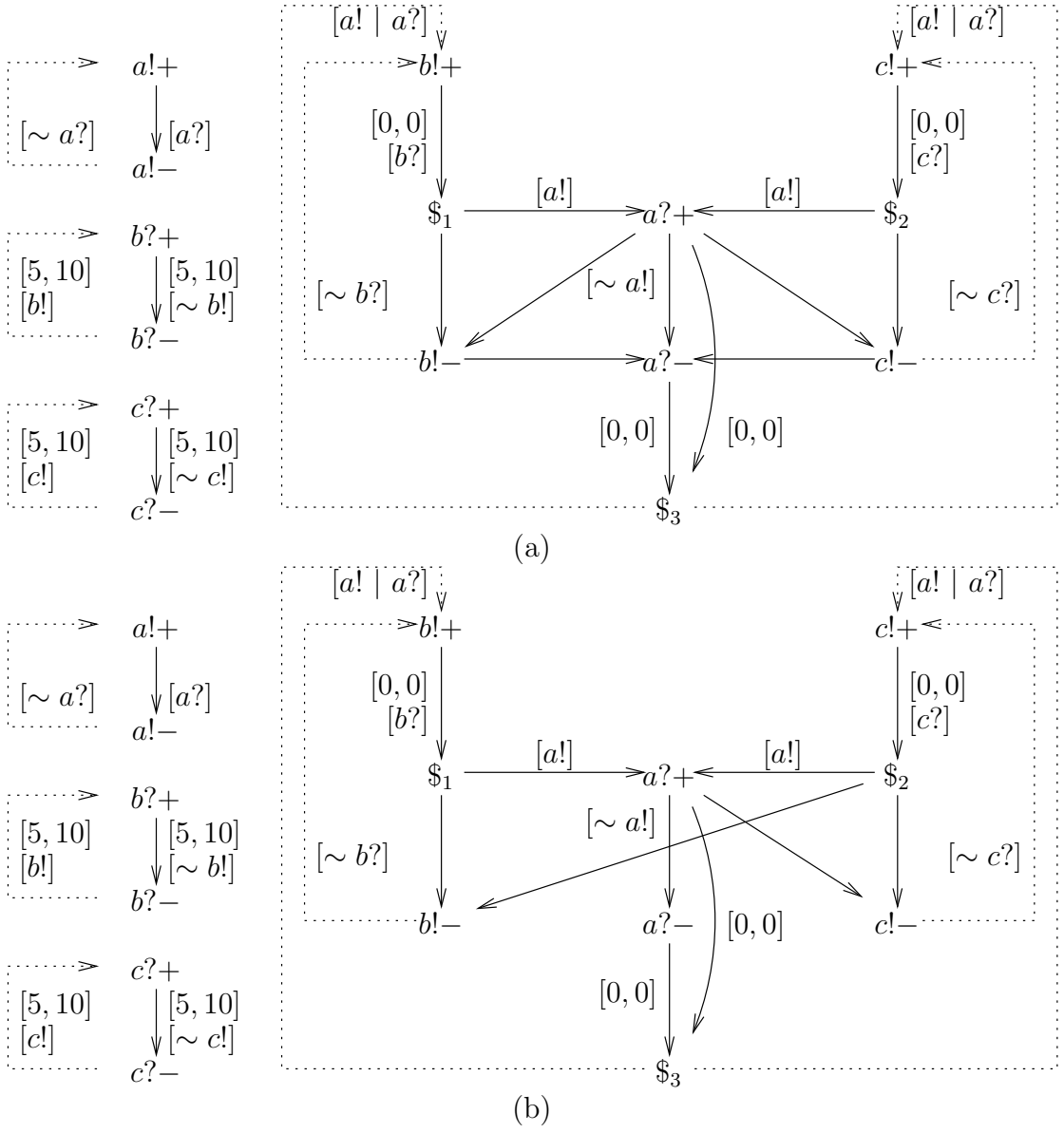

Figure 7.3. TEL structures for *PAR* example after concurrency reduction. Part (a) shows the result from `Petrify`. Part (b) shows the best result using the P and O filters.

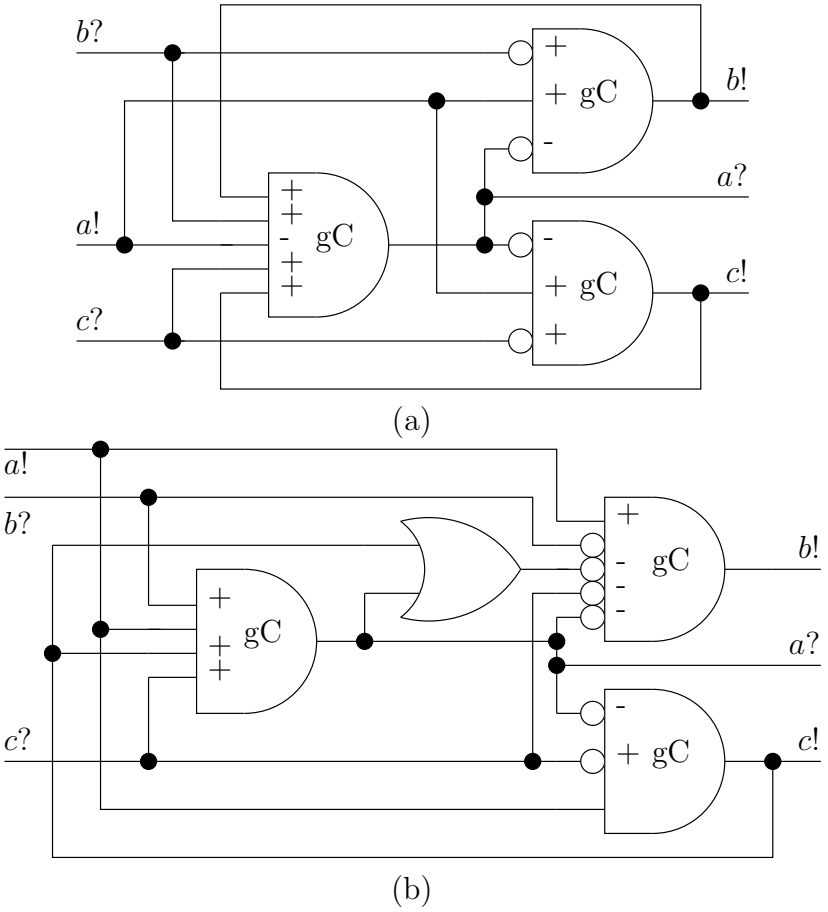

Figure 7.4. Circuit implementations of the *PAR* example. Part (a) shows the circuit derived from the `Petrify` result. Part (b) shows the best circuit derived using the P and O filters.

O filters results in a faster average cycle period when composed with the given environment, as shown in Table 7.2.

7.3 Examples Enabled by Heuristics

This section presents examples for which it would be infeasible to find the entire design space but for which heuristics can find useful solutions.

7.3.1 Shifter

Consider a systolic, n-bit shifter constructed of n modules, each of which stores one bit of the data [58]. $n-1$ of the modules are identical, but there is a special module at the end of the shifter. Each of the $n-1$ identical modules has an interface consisting of six channels named *Load*, $Shift_{in}$, $Shift_{out}$, $Done_{in}$, $Done_{out}$, and $Data_{out}$. The $Done_{in}$ and $Done_{out}$ channels are pure synchronization channels. Each other channel carries single-bit data. When a module receives a datum on its *Load* channel, it stores the received bit internally. When a communication is pending on its $Shift_{in}$ channel, the module sends the current value of its stored bit out of the $Shift_{out}$ channel, and receives a new datum on the on the $Shift_{in}$ channel. When communication is pending on the $Done_{in}$ channel, the module sends its bit on the $Data_{out}$ channel, sends a communication on the $Done_{out}$ channel, and completes the communication on the $Done_{in}$ channel. The special module on the end of the shifter does not have the $Shift_{out}$ or $Done_{out}$ channels. Figure 7.5 shows

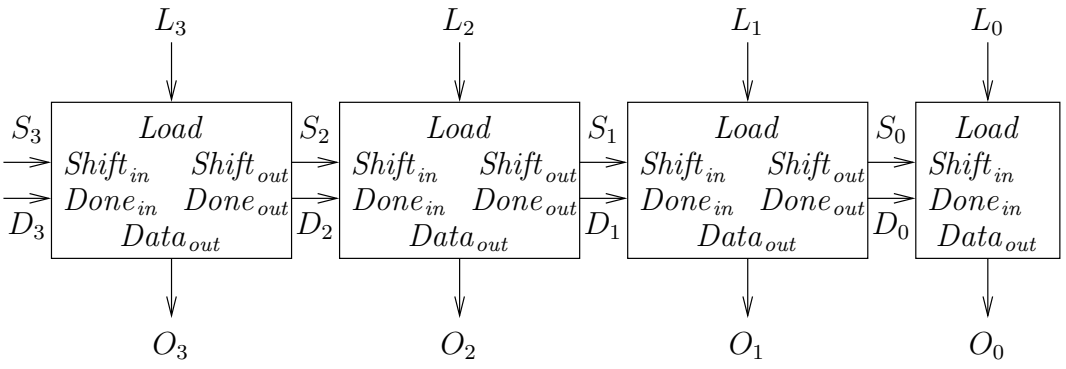

Figure 7.5. A four-bit shifter.

the interconnection of modules by channels for a four-bit shifter.

The following channel-level VHDL code specifies the behavior of each module. The *LSDOenv* process specifies the environment, the *normal* process specifies any module of the shifter except the special one on the end, and the *special* process specifies the special module at the end of the shifter.

```
entity LSDOenv is
  port(L3, L2, L1, L0 :  inout channel :=
       init_channel(sender => timing(3, 5));
       Shift, Done :  inout channel :=
       init_channel(sender => timing(3, 5));
       O3, O2, O1, O0 :  inout channel :=
       init_channel(receiver => timing(3, 5)));
end LSDOenv;
architecture behavior of LSDOenv is
  signal Ldata, Odata :  std_logic_vector( 3 downto 0 ) := "0000";
  signal Sdata :  std_logic := '0';
begin
  LSDOenv :  process
    variable z :  integer;
  begin
    --@synthesis_off
    Ldata <= Ldata + 1;
    --@synthesis_on
    wait for delay(1, 2);
    send(L3, Ldata(3), L2, Ldata(2), L1, Ldata(1), L0, Ldata(0));
    send(Done);
    receive(O3, Odata(3), O2, Odata(2), O1, Odata(1), O0, Odata(0));
    assert Ldata = Odata report "unequal data!" severity failure;
    send(Shift, Sdata);
    send(Done);
    receive(O3, Odata(3), O2, Odata(2), O1, Odata(1), O0, Odata(0));
    assert to_bitvector(Odata) = to_bitvector(Ldata) srl 1
      report "unequal data!" severity failure;
  end process LSDOenv;
end behavior;
-------------------------------------------------------------------------
entity normal is
  port(Load :  inout channel := init_channel(receiver => timing(2, 4));
       Shift_in :  inout channel := init_channel(receiver => timing(2, 4));
       Done_In :  inout channel := init_channel(receiver => timing(2, 4));
       Shift_Out :  inout channel := init_channel(sender => timing(3, 4));
       Done_Out :  inout channel := init_channel(sender => timing(3, 4));
       Data_out :  inout channel := init_channel(sender => timing(3, 4)));
end normal;
architecture behavior of normal is
  signal u :  std_logic;
begin
  normal :  process
  begin
```

```
    await_any(Load, Shift_in, Done_in);
    if (probe(Load)) then
      receive(Load, u);
    elsif (probe(Shift_in)) then
      send(Shift_out, u);
      receive(Shift_in, u);
    else
      send(Done_out);
      receive(Done_in);
      send(Data_out, u);
    end if;
  end process normal;
end behavior;
--------------------------------------------------------------------------
entity special is
  port(Load :  inout channel := init_channel(receiver => timing(2, 4));
       Shift_in :  inout channel := init_channel(receiver => timing(2, 4));
       Done_In :  inout channel := init_channel(receiver => timing(2, 4));
       Data_out :  inout channel := init_channel(sender => timing(1, 6)));
end special;
architecture behavior of special is
  signal u :  std_logic;
begin
  special :  process
  begin
    await_any(Load, Shift_in, Done_in);
    if (probe(Load)) then
      receive(Load, u);
    elsif (probe(Shift_in)) then
      receive(Shift_in, u);
    else
      receive(Done_in);
      send(Data_out, u);
    end if;
  end process special;
end behavior;
```

Table 7.3 presents results for synthesizing the *special* component of the shifter. Because of the complexity of the environment (the three instances of the *normal*) component, even one synthesis attempt is an expensive operation. Therefore, Table 7.3 presents only the results that use either the N filter or the E filter. Either filter finds the optimal solution on its first synthesis attempt. Therefore, adding the S filter has no effect, because the S filter uses only the information from failed synthesis attempts. The O filter succeeds in pruning away some rules that were unnecessary due to timing. However, this is achieved only at the expense of extra run time to compute the state space for the initial TEL structure, to find the

Table 7.3. Results for shifter.

Filters					CPU					Period	
N	P	E	O	S	time/s	levels	TELs	leaves	PRSs	max.	min.
		E			141.81	7	7	1	1	206	206
		E		S	142.71	7	7	1	1	206	206
		E	O		114.36	2	3	1	1	206	206
		E	O	S	114.03	2	3	1	1	206	206
N					72.46	7	4	1	1	206	206
N				S	74.16	7	4	1	1	206	206
N			O		86.43	2	2	1	1	206	206
N			O	S	86.98	2	2	1	1	206	206

The command `atacs -oi -oq -ts -G1-2 shifter.vhd special` [`-P`*filter*]... `-ys` generated each row of this table.

timed-concurrency information. While the O filter does help reduce run-time for the branch-and-bound algorithm, it only adds overhead to the run-time for the N filter. Because I do not currently have a `Petrify` specification for this example, I have not performed a comparison against `Petrify` on this example.

7.3.2 MMU

Consider the "MMU" from [61, 62]. The MMU converts a 16-bit memory address to a 24-bit real address by concatenating eight bits from one of two segmentation registers with the memory address. There are six possible cycles that the MMU controller can enter, depending on data from the environment. For simplicity, this section discusses the design of only one cycle, namely the memory data load cycle. Figure 7.6 shows the channel-level block diagram for part of the MMU and its environment [62].

For a memory load cycle, the MMU controller must first wait for a communication to be pending on the memory data load channel, *MDl*. Then it must receive the result of a memory address comparison on channel B while it instructs the segmentation register to put the segmentation read address on the real address bus by communicating on channel *RA*. Then the controller requests the memory load from the memory interface and waits for it to complete by communicating on the

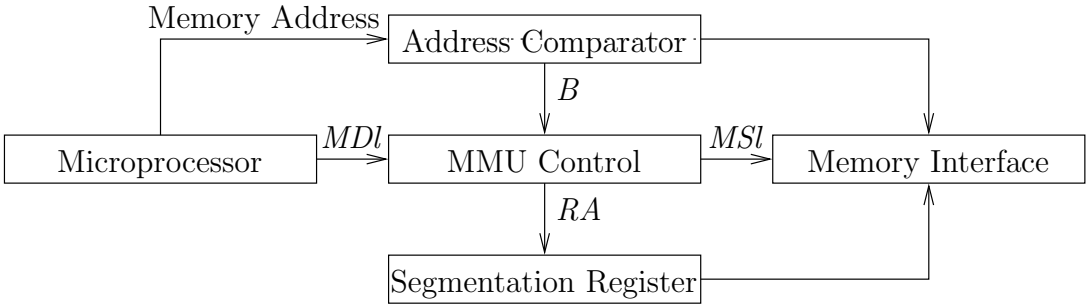

Figure 7.6. Block diagram for part of the MMU and its environment [62].

MSl channel. Finally, the controller completes the communication on the memory data load port, *MDl*. The following behavioral VHDL specifies this sequence of operations and the corresponding behavior of the environment.

```
entity environment is
  port(RA : inout channel := init_channel(sender => timing(4, 18));
       MDl, MSl :  inout channel :=
       init_channel(receiver => timing(rise_min => 60, rise_max => 100,
                                       fall_min => 10, fall_max => 60));
       B : inout channel := init_channel(sender => timing(5, 26)));
end environment;
architecture behavior of environment is
  signal bn, r :  std_logic := '0';
begin
  processor :  process
  begin
    receive(MDl);
  end process processor;
  segmentation :  process
  begin
    send(RA, r);
  end process segmentation;
  comparator :  process
  begin
    send(B, bn);
  end process comparator;
  memory :  process
  begin
    receive(MSl);
  end process memory;
end behavior;
--------------------------------------------------------------------------
entity control is
  port(MDl, MSl :  inout channel := init_channel(sender => timing(0, 2));
       RA, B : inout channel := active(receiver => timing(0, 2)));
end control;
architecture behavior of control is
```

```
  signal r, bn :  std_logic := '0';
begin
  control :  process
  begin
    await(MDl);
    receive(RA, r, B, bn);
    send(MSl);
    send(MDl);
  end process control;
end behavior;
```

From the above VHDL code in the file `MMU.vhd` the tool that this dissertation presents automatically produces the starting point for concurrency reduction show in Figure 7.7.

Table 7.4 presents results for synthesizing the *control* component of the MMU. In this case, synthesis attempts are not as expensive as for the *shifter* example. Therefore, Table 7.4 includes rows that do not use the N filter to stop after the first solution found, but do use the E filter to prune expensive solutions. Although the cycle period seems to vary slightly, this is only because it is a stochastic estimate. Each row in Table 7.4 actually finds the same one solution. Thus, for this example, the branching function steers the search such that it finds an optimal solution on

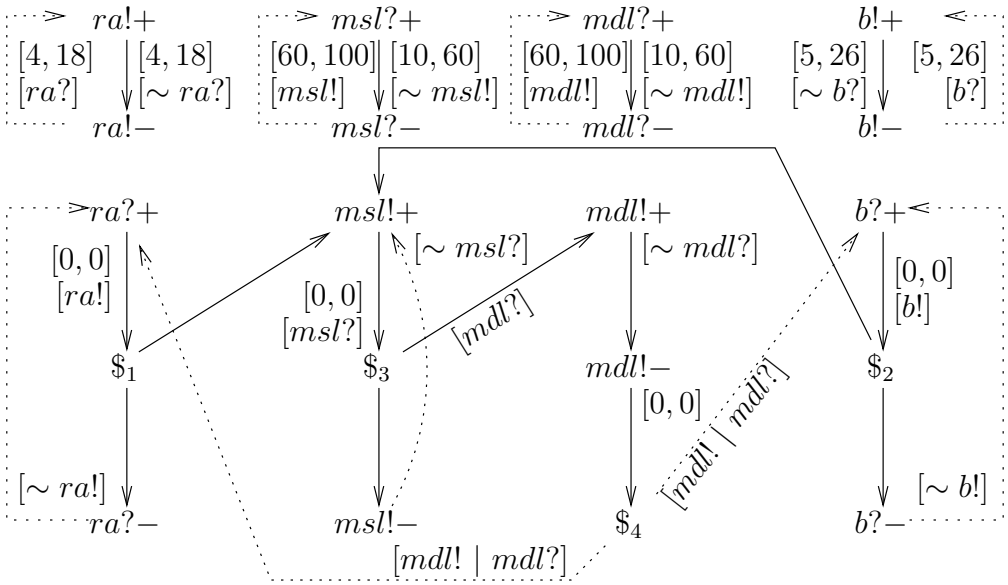

Figure 7.7. Starting point for concurrency reduction of the *MMU* example. Except where indicated, each rule has the timing bounds $[0, 2]$.

Table 7.4. Results for MMU.

Filters					CPU					Period	
N	P	E	O	S	time/s	levels	TELs	leaves	PRSs	max.	min.
		E			10.58	42	26	1	1	218	218
		E		S	10.64	42	26	1	1	218	218
		E	O		3.25	14	8	1	1	217	217
		E	O	S	3.21	14	8	1	1	217	217
N					0.67	42	7	1	1	216	216
N				S	0.66	42	7	1	1	216	216
N			O		0.63	14	4	1	1	215	215
N			O	S	0.62	14	4	1	1	215	215
	`Petrify`				16.51	–	–	–	1	225	225

The command `atacs -oi -oq -ts -G0-2 MMU.vhd control` [`-P`*filter*]... `-ys` generated each row of this table.

its first attempt. The cost function is implemented such that when two average cycle periods are within an average deviation of each other, they are considered equal. In such cases, the bounding function prunes the search tree. Thus, there may actually be several solutions that have approximately the same cost as the one solution listed in Table 7.4. However, there are no solutions that have significantly lower cost.

The last row of Table 7.4 shows the result for this example produced by `Petrify`. The `Petrify` input is shown below.

```
.name mmu
.channels mdl ra b msl
.graph
mdl? ra!
ra! ra?
ra? msl!
mdl? b!
b! b?
b? msl!
msl! msl?
```

```
msl? mdl!
mdl! mdl?
msl! ra!
msl! b!
mdl! msl!

.marking {<mdl!,mdl? > <msl!,ra!> <msl!,b!> <mdl!,msl!> }
.slowenv
.end
```

For the purposes of concurrency reduction, the `.slowenv` directive serves as an approximation to the timing constraints in the `ATACS` specification. Given the above specification in the file `MMU.g`, the command `petrify -er -4ph -untog MMU.g -o MMU.out.4ph -redc -gc -eqn MMU.eqn` produced the synthesizable Petri Net. The result was input to `ATACS` and annotated with the timing assumptions of Figure 7.7 for computation of the average cycle period and synthesis using `ATACS`. Figure 7.8 compares the result of concurrency reduction using `Petrify` to that using the tool that this dissertation presents. In particular, Figure 7.8(a) shows the TEL structure for the result from `Petrify`. Figure 7.8(b) shows the TEL structure for the result using the N and O filters. `Petrify` inserted a state variable, while the tool that this dissertation presents used only reshuffling to achieve complete state coding. Running `ATACS` synthesis on the TEL structure of Figure 7.8(a) (the annotated Petrify result) produces the circuit of Figure 7.9(a). Running `ATACS` synthesis on the TEL structure of Figure 7.8(b) produces the circuit of Figure 7.9(b). In this case, the circuit derived using the N and O filters is both smaller and faster (in terms average cycle period when composed with the given environment, as shown in Table 7.4).

As with the *shifter* example, the S filter has no effect, because there were no failed synthesis attempts to consider. Again, the O filter pruned away candidate rules between events that were already timed-ordered in the initial TEL structure. For this example, this is about two-thirds of the candidate rules, and the O filter

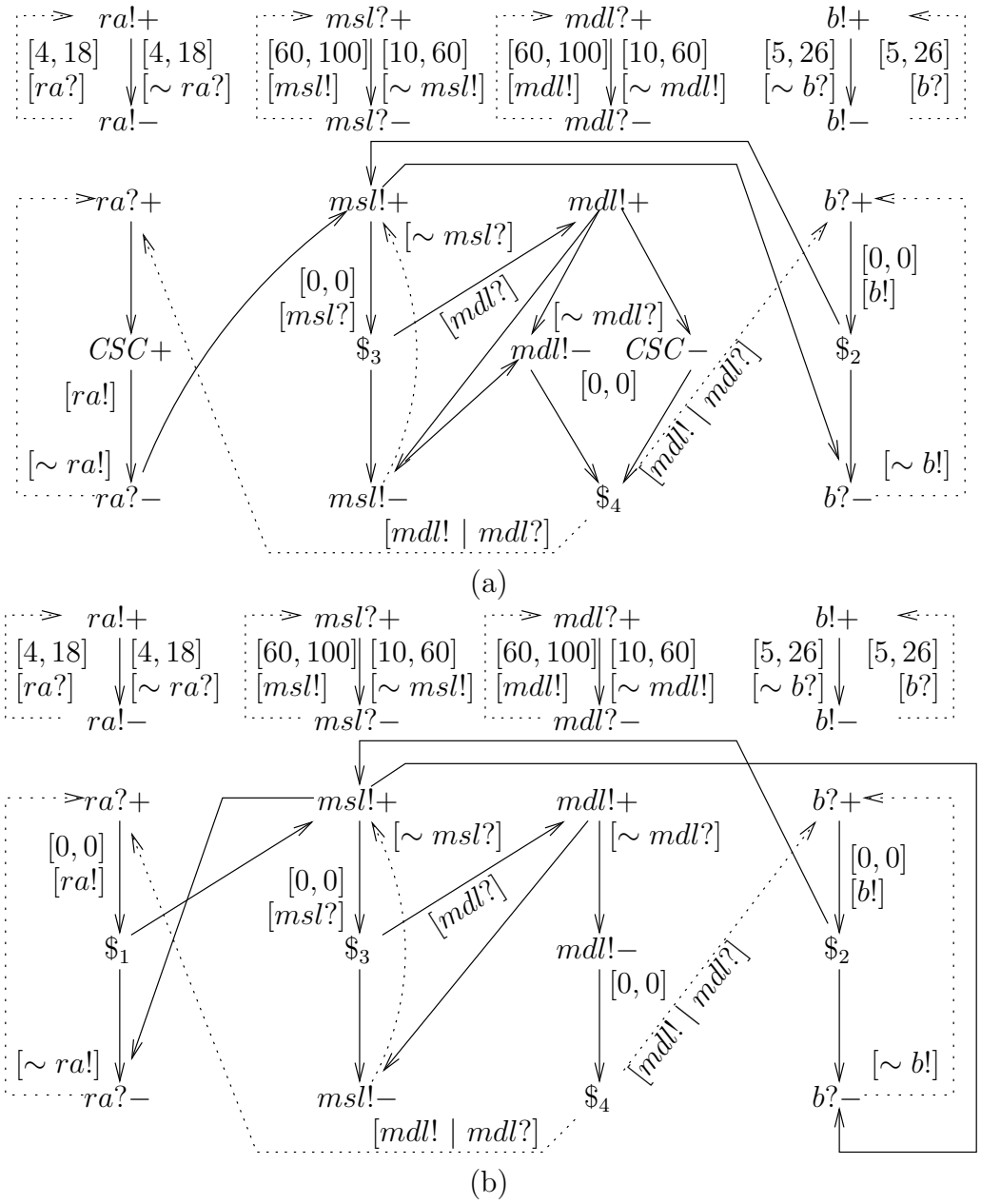

Figure 7.8. TEL structures for *MMU* example after concurrency reduction. Part (a) shows the result from `Petrify`. Part (b) shows the best result using the N and O filters.

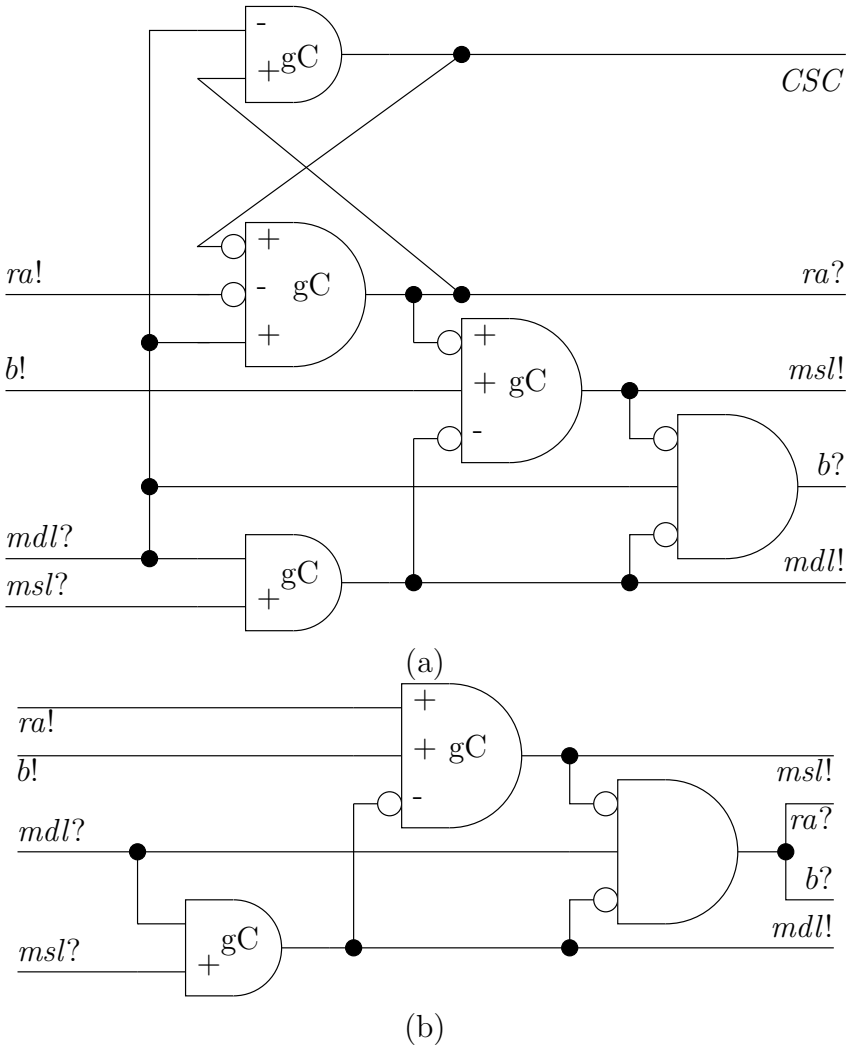

Figure 7.9. Circuit implementations of the *MMU* example. Part (a) shows the circuit derived from the `Petrify` result. Part (b) shows the best circuit derived using the N and O filters.

does reduce the run time.

7.3.3 MPEG

Consider a decoder for the *Motion Picture Expert Group* (MPEG) format. One of the important operations within the decoder is the dithering operation. Zhao [73] is developing a hardware/software co-design to implement the dithering unit for an MPEG decoder. Figure 7.10 shows the channel-level block diagram for the dithering unit design. The following VHDL code (adapted from [73]) specifies the behavior of the dithering unit.

```
--------------------------------------------------------------------------
--
-- Title       :  dpCal
```

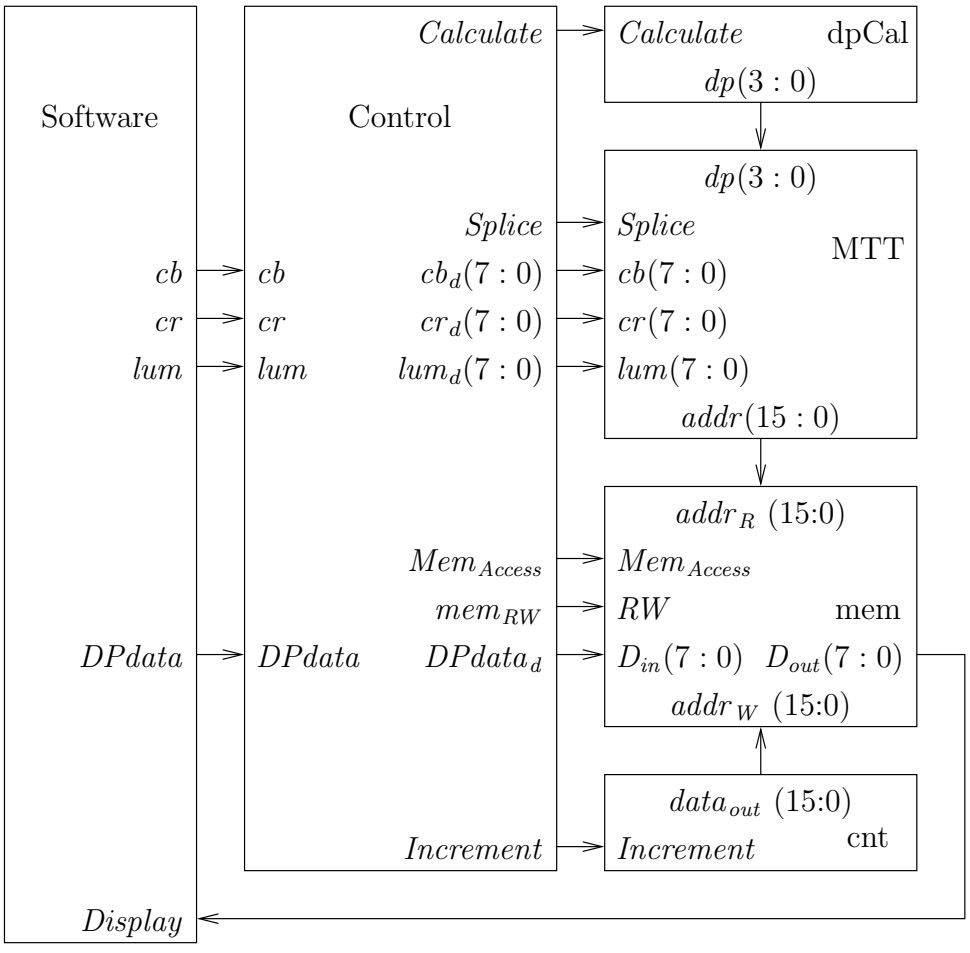

Figure 7.10. Block diagram for an MPEG dithering unit and its environment.

```
-- Design      :  datapath
-- Author      :  yy
-- date        :  6-8-2002
-------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.nondeterminism.all;
use work.channel.all;
entity dpCal is
  generic(
    fwidth :  integer := 15
    );
  port(
    reset :  in std_logic;
    Calculate :  inout channel := init_channel(receiver => timing(1, 3));
    dp :  out std_logic_vector(3 downto 0)
    );
end dpCal;
architecture dpCal of dpCal is
  type NumType is array (0 to 15) of integer range 0 to 15;
  constant dpi :  NumType := (0,8,12,4,2,10,14,6,3,11,15,7,1,9,13,5);
begin
  dpCalculate :  process
    variable cnt :  integer := 0;
    variable index :  integer;
    variable w :  integer := fwidth*16;
  begin
    await(Calculate);
    --@synthesis_off
    index := ((cnt/w/2) mod 2)*8 +cnt mod 8;
    dp <= conv_std_logic_vector(dpi(index), 4);
    cnt := cnt+1;
    --@synthesis_on
    wait for delay (3, 5);
    receive(Calculate);
  end process;
end dpCal;
-------------------------------------------------------------------------
--
-- Title       :  mtt
-- Design      :  datapath
-- Author      :  yy
-- date        :  6/8/2002
-------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.nondeterminism.all;
use work.channel.all;
use std.textio.all;
```

```
entity mtt is
  generic(
    fheight :  integer := 11;
    fwidth :  integer := 15;
    pagesize :  integer := 64
    );
  port(
    Splice :  inout channel := init_channel(receiver => timing(1, 3));
    cr :  in std_logic_vector(7 downto 0);
    cb :  in std_logic_vector(7 downto 0);
    lum :  in std_logic_vector(7 downto 0);
    addr :  out std_logic_vector(15 downto 0);
    dp :  in std_logic_vector(3 downto 0)
    );
end MTT;
architecture mtt of mtt is
begin
  ttb_cal :  process
    variable pos :  integer := 0;
    variable w :  integer := fwidth*16;
    variable idp :  integer;
    --for MAKETTable
    type ntype is array (0 to 2) of integer;
    constant ttb :  ntype := (33, 97, 161);
    variable ttx :  integer := 0;
    variable tty :  integer := 0;
    variable ix :  integer := 0;
    variable iy :  integer := 0;
  begin
    await(Splice);
    --@synthesis_off
    ix := conv_integer(cb);
    iy := conv_integer(cr);
    idp := conv_integer(dp);
    ttx := 3;
    while ix < ttb(ttx-1)+idp*4 loop
      ttx := ttx-1;
    end loop;
    tty := 3;
    while iy < ttb(tty-1)+idp*4 loop
      tty := tty-1;
    end loop;
    addr <= dp & conv_std_logic_vector(ttx, 2) &
            conv_std_logic_vector(tty, 2) & lum;
    wait for delay(3, 5);
    --@synthesis_on
    receive(Splice);
  end process;
end mtt;
--------------------------------------------------------------------------
--
-- Title       :  mem
-- Design      :  datapath
```

```
-- Author      :  YY
-- date        :  6-8-2002
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.nondeterminism.all;
use work.channel.all;
entity mem is
  port(
    Mem_Access :  inout channel := init_channel(receiver=>timing(1, 3));
    addr_r :  in std_logic_vector(15 downto 0);
    addr_w :  in std_logic_vector(15 downto 0);
    Din :  in std_logic_vector(7 downto 0);
    RW : in std_logic;
    Dout :  out std_logic_vector(7 downto 0)
    );
end mem;
architecture mem of mem is
  constant ttMemorySize :  integer := 16*4*4*256;
  type ttype is array (0 to ttMemorySize - 1)
    of std_logic_vector(7 downto 0);
  signal tt :  ttype;
begin
  memory :  process
  begin
    await(Mem_Access);
    --read delay is less that write
    --@synthesis_off
    if (RW = '0') then
      tt(conv_integer(addr_w)) <= Din;
      wait for delay(3, 5);
    else
      Dout <= tt(conv_integer(addr_r));
      wait for delay(1, 3);
    end if;
    --@synthesis_on
    receive(Mem_Access);
  end process;
end mem;
--------------------------------------------------------------------------------
--
-- Title       :  cnt
-- Design      :  datapath
-- Author      :  yy
-- date        :  6-8-2002,
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.nondeterminism.all;
```

```
use work.channel.all;
entity cnt is
  port(
    Increment :  inout channel := init_channel(receiver => timing(1, 3));
    DataOut :  buffer std_logic_vector(15 downto 0) := (others => '0')
    );
end cnt;
architecture cnt of cnt is
begin
  cnt :  process
  begin
    await(Increment);
    --@synthesis_off
    DataOut <= DataOut + 1;
    --@synthesis_on
    wait for delay (3, 5);
    receive(Increment);
  end process;
end cnt;
--------------------------------------------------------------------------------
--
-- Title       :  datapath
-- Design      :  datapath
-- Author      :  yy
--------------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use work.channel.all;
entity datapath is
  port(
    Reset :  in std_logic;
    Increment :  inout channel := init_channel;
    Calculate :  inout channel := init_channel;
    mem_RW : in std_logic;
    Mem_Access :  inout channel := init_channel;
    Splice :  inout channel := init_channel;
    cb :  in std_logic_vector(7 downto 0);
    cr :  in std_logic_vector(7 downto 0);
    lum :  in std_logic_vector(7 downto 0);
    mem_Din :  in std_logic_vector(7 downto 0);
    mem_Dout :  out std_logic_vector(7 downto 0)
    );
end datapath;
architecture datapath of datapath is
  component cnt
    port(
      Increment :  inout channel := init_channel(receiver => timing(1, 3));
      DataOut :  buffer std_logic_vector(15 downto 0) := (others => '0')
      );
  end component;
  component dpcal
    port(
      reset :  in std_logic;
```

```
      Calculate :  inout channel := init_channel(receiver => timing(1, 3));
      dp :  out std_logic_vector(3 downto 0)
      );
  end component;
  component mem
    port(
      Mem_Access :  inout channel := init_channel(receiver=>timing(1, 3));
      addr_r :  in std_logic_vector(15 downto 0);
      addr_w :  in std_logic_vector(15 downto 0);
      Din :  in std_logic_vector(7 downto 0);
      RW : in std_logic;
      Dout :  out std_logic_vector(7 downto 0)
      );
  end component;
  component mtt
    port(
      Splice :  inout channel := init_channel(receiver => timing(1, 3));
      cr :  in std_logic_vector(7 downto 0);
      cb :  in std_logic_vector(7 downto 0);
      lum :  in std_logic_vector(7 downto 0);
      addr :  out std_logic_vector(15 downto 0);
      dp :  in std_logic_vector(3 downto 0)
      );
  end component;
  signal cbCal :  std_logic_vector (7 downto 0);
  signal cnt_addr :  std_logic_vector (15 downto 0);
  signal crCal :  std_logic_vector (7 downto 0);
  signal dpIn :  std_logic_vector (3 downto 0);
  signal lumCal :  std_logic_vector (7 downto 0);
  signal mtt_addr :  std_logic_vector (15 downto 0);
begin
  addr :  mtt
    port map(
      Splice => Splice,
      addr => mtt_addr,
      cb => cb,
      cr => cr,
      dp => dpIn,
      lum => lum
      );
  dp :  dpcal
    port map(
      Calculate => Calculate,
      dp => dpIn,
      reset => reset
      );
  memo :  mem
    port map(
      Din => mem_Din,
      Dout => mem_Dout,
      RW => mem_RW,
      Mem_Access => Mem_Access,
      addr_r => mtt_addr,
```

```
      addr_w => cnt_addr
      );
  pos :  cnt
    port map(
      Increment => Increment,
      DataOut => cnt_addr
      );
end datapath;
--------------------------------------------------------------------------
--
-- Title       :  ctrl
-- Design      :  control
-- Author      :  yy
-- date        :  6-8-2002
--------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use std.textio.all;
use work.nondeterminism.all;
use work.channel.all;
use work.handshake.all;
entity ctrl is
  port(
    DPdata :  inout channel := init_channel(receiver => timing(1, 2));
    cr :  inout channel := init_channel(receiver => timing(1, 2));
    cb :  inout channel := init_channel(receiver => timing(1, 2));
    lum :  inout channel := init_channel(receiver => timing(1, 2));
    Increment :  inout channel := init_channel(sender => timing(1, 3));
    Mem_Access :  inout channel := init_channel(sender => timing(3, 5));
    Calculate :  inout channel := init_channel(sender => timing(3, 5));
    Splice :  inout channel := init_channel(sender => timing(3, 5));
    cr_d :  buffer std_logic_vector(7 downto 0);
    cb_d :  buffer std_logic_vector(7 downto 0);
    lum_d :  buffer std_logic_vector(7 downto 0);
    DPdata_d :  buffer std_logic_vector(7 downto 0);
    reset :  out std_logic := '0';
    mem_RW : buffer std_logic := '0'
    );
end ctrl;
architecture ctrl of ctrl is
begin
  Decoder :  process
  begin
    await_any (DPdata, cb, cr, lum);
    if (probe (DPdata))then
      vassign(mem_RW, '0', 1, 2);
      receive(DPdata, DPdata_d);
      send(Mem_Access);
      send(Increment);
    elsif (probe (cb)) then
      receive(cb, cb_d);
```

```
    elsif (probe (cr)) then
      receive(cr, cr_d);
    elsif (probe(lum))then
      vassign(mem_RW, '1', 1, 3);
      receive(lum, lum_d);
      send(Calculate);
      send(Splice);
      send(Mem_Access);
    end if;
    wait for delay(3, 5);
  end process;
end ctrl;
--------------------------------------------------------------------------------
--
-- Title       :  decoder
-- Design      :  datapath
-- Author      :  yy :p
--------------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use work.channel.all;
entity decoder is
  port(
    sampling :  out std_logic;
    Display :  inout std_logic_vector(7 downto 0);
    DPdata :  inout channel := init_channel;
    cb :  inout channel := init_channel;
    cr :  inout channel := init_channel;
    lum :  inout channel := init_channel
    );
end decoder;
architecture decoder of decoder is
  component ctrl
    port(
      DPdata :  inout channel := init_channel(receiver => timing(1, 2));
      cr :  inout channel := init_channel(receiver => timing(1, 2));
      cb :  inout channel := init_channel(receiver => timing(1, 2));
      lum :  inout channel := init_channel(receiver => timing(1, 2));
      Increment :  inout channel := init_channel(sender => timing(1, 3));
      Mem_Access :  inout channel := init_channel(sender => timing(3, 5));
      Calculate :  inout channel := init_channel(sender => timing(3, 5));
      Splice :  inout channel := init_channel(sender => timing(3, 5));
      cr_d :  buffer std_logic_vector(7 downto 0);
      cb_d :  buffer std_logic_vector(7 downto 0);
      lum_d :  buffer std_logic_vector(7 downto 0);
      DPdata_d :  buffer std_logic_vector(7 downto 0);
      reset :  out std_logic := '0';
      mem_RW : buffer std_logic := '0'
      );
  end component;
  component datapath
    port(
      Reset :  in std_logic;
```

```
      Increment :  inout channel := init_channel;
      Calculate :  inout channel := init_channel;
      mem_RW : in std_logic;
      Mem_Access :  inout channel := init_channel;
      Splice :  inout channel := init_channel;
      cb :  in std_logic_vector(7 downto 0);
      cr :  in std_logic_vector(7 downto 0);
      lum :  in std_logic_vector(7 downto 0);
      mem_Din :  in std_logic_vector(7 downto 0);
      mem_Dout :  out std_logic_vector(7 downto 0)
      );
  end component;
  signal Increment :  channel := init_channel;  -- tells counter to count
  signal Calculate :  channel := init_channel;  -- tells dpCalc to do it
  signal Mem_Access :  channel := init_channel;  -- tells memory to do it
  signal mem_RW : std_logic;
  signal Splice :  channel := init_channel;  -- tells MTT to splice address
  signal Reset :  std_logic;
  signal n_cb :  std_logic_vector (7 downto 0);
  signal n_cr :  std_logic_vector (7 downto 0);
  signal n_DPdata :  std_logic_vector (7 downto 0);
  signal n_lum :  std_logic_vector (7 downto 0);
begin
  control :  ctrl
    port map(
      DPdata => DPdata,
      DPdata_d => n_DPdata,
      cb => cb,
      cb_d => n_cb,
      Increment => Increment,
      cr => cr,
      cr_d => n_cr,
      Calculate => Calculate,
      lum => lum,
      lum_d => n_lum,
      mem_RW => mem_RW,
      Mem_Access => Mem_Access,
      Splice => Splice,
      reset => Reset
      );
  datap :  datapath
    port map(
      Reset => Reset,
      cb => n_cb,
      Increment => Increment,
      cr => n_cr,
      Calculate => Calculate,
      lum => n_lum,
      mem_Din => n_DPdata,
      mem_Dout => Display,
      mem_RW => mem_RW,
      Mem_Access => Mem_Access,
      Splice => Splice
```

```
      );
--@synthesis_off
  sampling <= mem_RW when probe(Mem_Access) else '0';
--@synthesis_on
end decoder;
-------------------------------------------------------------------------
--
-- Title       :  environment for decoder
-- Design      :  datapath
-- Author      :  yy
-- date        :  6-8-2002
-------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use std.textio.all;
use work.nondeterminism.all;
use work.channel.all;
use work.handshake.all;
entity MPEG is
  generic(
    fheight :  integer := 11;
    fwidth :  integer := 15
    );
end MPEG;
architecture TB_ARCHITECTURE of MPEG is
  file fh_cbx :  text;
  file fh_crx :  text;
  file fh_lum :  text;
  file fh_dp :  text;
  file fh_out :  text;
  signal s_cb :  std_logic_vector(7 downto 0);
  signal s_cr :  std_logic_vector(7 downto 0);
  signal s_lum :  std_logic_vector(7 downto 0);
  signal s_DPdata :  std_logic_vector(7 downto 0);
  signal s_output :  std_logic_vector(7 downto 0);
  signal s_display :  std_logic_vector(7 downto 0);
  component decoder
    port(
      sampling :  out std_logic;
      Display :  inout std_logic_vector(7 downto 0);
      DPdata :  inout channel;
      cb :  inout channel;
      cr :  inout channel;
      lum :  inout channel );
  end component;
  signal reset :  std_logic := '1';
  signal failed :  std_logic := '0';
  signal sampling :  std_logic;
  signal DPdata :  channel := init_channel(sender => timing(1, 2));
  signal cr, cb, lum :  channel := init_channel(sender => timing(1, 2));
  signal Display :  std_logic_vector(7 downto 0);
begin
```

```
read_file :  process
  variable v_line :  line;
  variable good :  boolean;
  variable v_int :  integer;
begin
  wait for delay (3, 5);
  --@synthesis_off
  reset <= '0';
  --@synthesis_on
  wait for delay (3, 5);
  -- open/read/send translation table to decoder
  --
  file_open(fh_dp, "dp.dat", read_mode);
  --@synthesis_off
  while not endfile(fh_dp) loop
    --@synthesis_on
    --read data from dp.dat
    readline(fh_dp, v_line);
    read(v_line, v_int, good);
    assert good
      report " dp.dat Text I/O read error!"
      severity error;
    s_DPdata <= conv_std_logic_vector(v_int, 8);
    wait for 1 ns;
    send(DPdata, s_DPdata);
    --@synthesis_off
  end loop;
  --@synthesis_on
  file_close(fh_dp);
  report "dp_file is closed successfully!";
  file_open(fh_out, "out.dat", read_mode);
  file_open(fh_cbx, "cbx.dat", read_mode);
  file_open(fh_crx, "crx.dat", read_mode);
  file_open(fh_lum, "lum.dat", read_mode);
  --@synthesis_off
  while not endfile (fh_cbx) loop
    --@synthesis_on
    --read data from cbx.dat
    readline(fh_cbx, v_line);
    read(v_line, v_int, good);
    assert good
      report " cbx.dat Text I/O read error!"
      severity error;
    s_cb <= conv_std_logic_vector(v_int, 8);
    wait for delay (3, 5);
    --read data from crx.dat
    readline(fh_crx, v_line);
    read(v_line, v_int, good);
    assert good
      report " crx.dat Text I/O read error!"
      severity error;
    s_cr <= conv_std_logic_vector(v_int, 8);
    wait for delay (3, 5);
```

```
    send(cb, s_cb);
    send(cr, s_cr);
    --read 4 data from lum.dat
    --@synthesis_off
    for dummy in 1 to 4 loop
      --@synthesis_on
      readline(fh_lum, v_line);
      read(v_line, v_int, good);
      assert good
        report " lum.dat Text I/O read error!"
        severity error;
      s_lum <= conv_std_logic_vector(v_int, 8);
      wait for delay (3, 5);
      send(lum, s_lum);
      --  Testing
      --read file from out.dat
      --@synthesis_off
      readline(fh_out, v_line);
      read(v_line, v_int, good);
      assert good
        report " out.dat Text I/O read error!"
        severity error;
      s_display <= conv_std_logic_vector(v_int, 8);
      wait for delay(1, 3);
      guard(sampling, '1');
      assert Display = s_display
        report "Miss-match!  "
        severity warning;
      if Display/=s_display then
        failed <= '1';
      end if;
      wait for delay (1, 1);
      guard(sampling, '0');
      --@synthesis_on
      --@synthesis_off
    end loop;
  end loop;
  --@synthesis_on
  file_close(fh_out);
  report "out_file is closed successfully!";
  file_close(fh_cbx);
  report "cbx_file is closed successfully!";
  file_close(fh_crx);
  report "crx_file is closed successfully!";
  file_close(fh_lum);
  report "lum_file is closed successfully!";
  wait;
end process;
please :  decoder
  port map (
    sampling => sampling,
    Display => Display,
    DPdata => DPdata,
```

```
        cb => cb,
        cr => cr,
        lum => lum
        );
end TB_ARCHITECTURE;
```

From the above VHDL code in the file the tool that this dissertation presents automatically produces the starting point for concurrency reduction show in Figure 7.11.

Table 7.5 presents results for synthesizing the *control* component of the MPEG. In this case, synthesis attempts are not as expensive as for the *shifter* example. Therefore, Table 7.5 includes a row that does not use the N filter to stop after the first solution found, but does use the E filter to prune expensive solutions. Each filter finds the same solution. However, the N filter does so in about one fifth of the CPU time required for the E filter. Figure 7.12 presents the TEL structure for this solution. Figure 7.13 shows the resulting circuit. Because I do not currently have a `Petrify` specification for this example, I have not performed a comparison against `Petrify` on this example.

7.4 Comparison to Existing Approaches

This section compares the effectiveness of the techniques that this dissertation presents to that of existing approaches. This section measures effectiveness by considering both the resources required to process a given example as well as the quality of the solutions found. The experiments of this section take a set of several examples, and run each example through the tool that this dissertation presents as well as other, existing CAD tools. This section presents and compares the results.

Table 7.5. Results for MPEG.

Filters					CPU					Period	
N	P	E	O	S	time/s	levels	TELs	leaves	PRSs	max.	min.
		E			47.14	105	39	1	1	18	18
N					9.28	105	13	1	1	18	18

The command `atacs -oi -oq -oD -tp -G1-2 cnt mem MTT dpCal control datapath decoder MPEG decoder.ctrl` [`-P`*filter*]... `-ys` generated each row of this table.

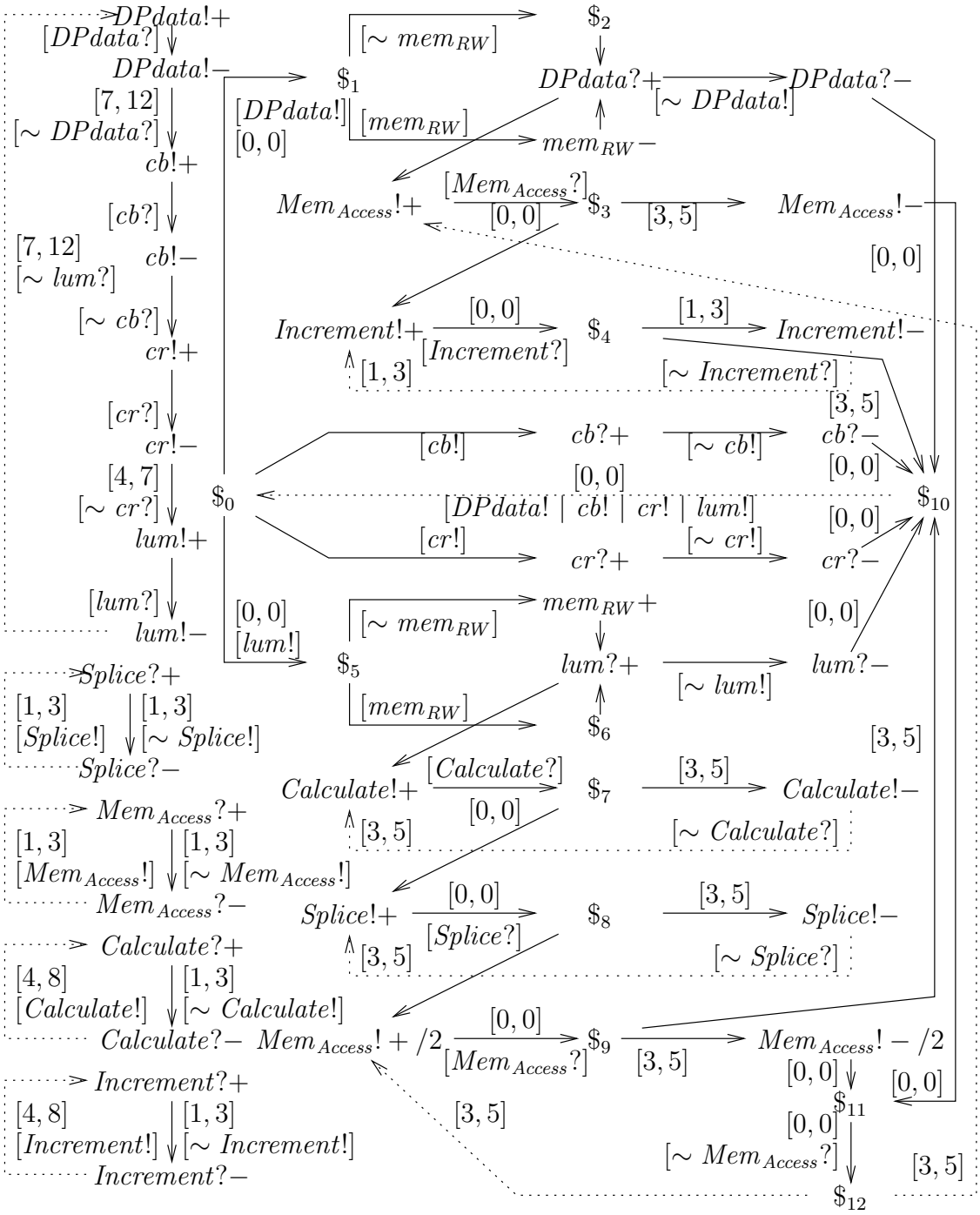

Figure 7.11. Starting point for concurrency reduction of the *MPEG* example. Except where indicated, each rule has the timing bounds $[1, 2]$. Conflicts: $\{\$_1, \$_2, \$_3, \$_4, mem_{RW}-\} \cup DPdata? \pm \cup Mem_{Access} \pm \cup Increment! \pm \#_{set} cb? \pm \#_{set}\ cr? \pm \#_{set} (\{\$_5, \$_6, \$_7, \$_8, \$_9, \$_{11}, \$_{12}, mem_{RW}+\} \cup lum? \pm \cup Mem_{Access} \pm /2) \wedge \{\$_{11}, \$_{12}\} \#_{set} cb? \pm \wedge \{\$_{11}, \$_{12}\} \#_{set} cr? \pm \wedge \$_2 \# mem_{RW} - \wedge \$_6 \# mem_{RW}+$.

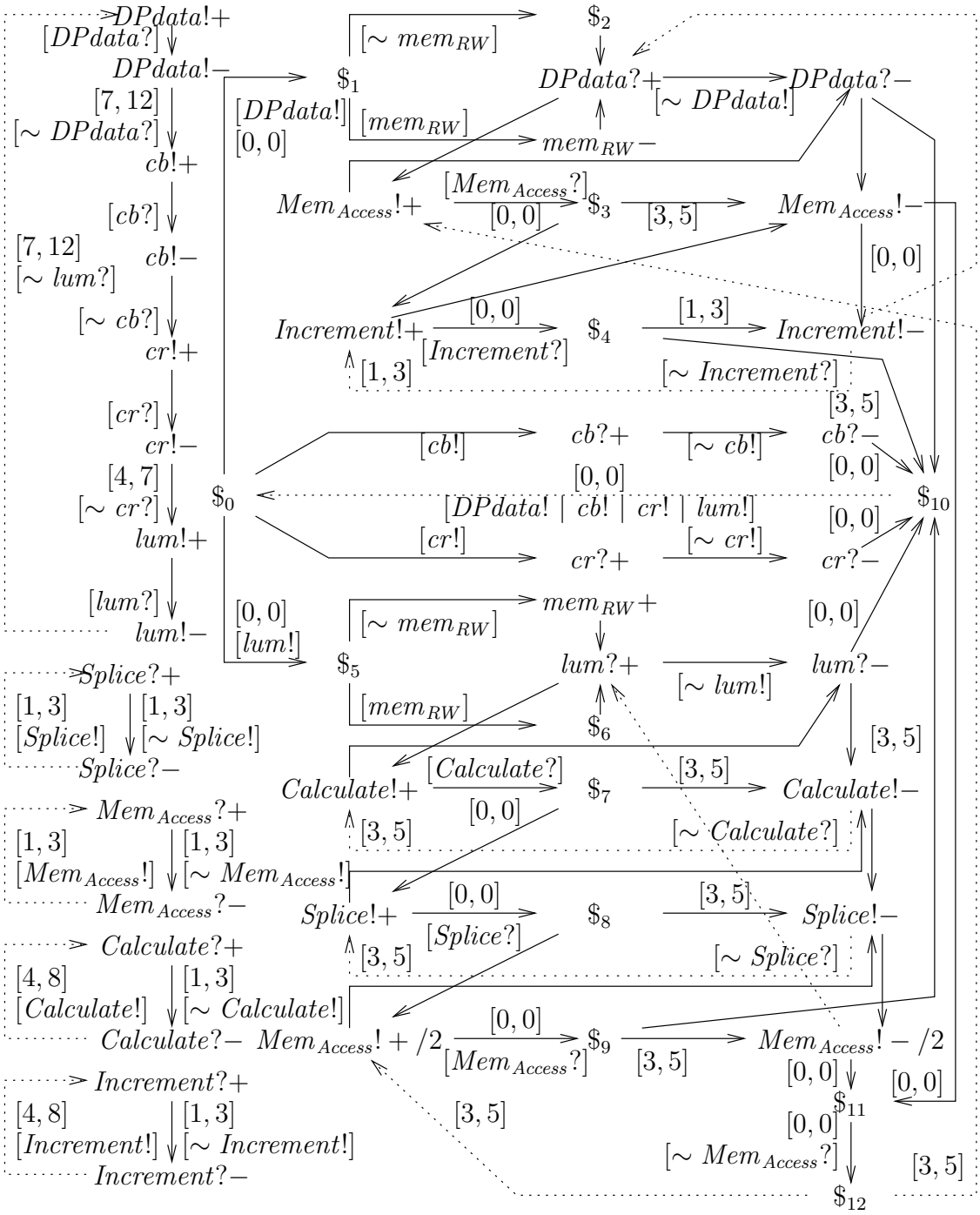

Figure 7.12. TEL structure for the *MPEG* example after concurrency reduction. Except where indicated, each rule has the timing bounds $[1, 2]$. Conflicts: $\{\$_1, \$_2, \$_3, \$_4, mem_{RW}-\} \cup DPdata? \pm \cup Mem_{Access} \pm \cup Increment! \pm \#_{set} cb? \pm \#_{set}\ cr? \pm \#_{set} (\{\$_5, \$_6, \$_7, \$_8, \$_9, \$_{11}, \$_{12}, mem_{RW}+\} \cup lum? \pm \cup Mem_{Access} \pm /2) \wedge \{\$_{11}, \$_{12}\} \#_{set} cb? \pm \wedge \{\$_{11}, \$_{12}\} \#_{set} cr? \pm \wedge \$_2 \# mem_{RW} - \wedge \$_6 \# mem_{RW}+$.

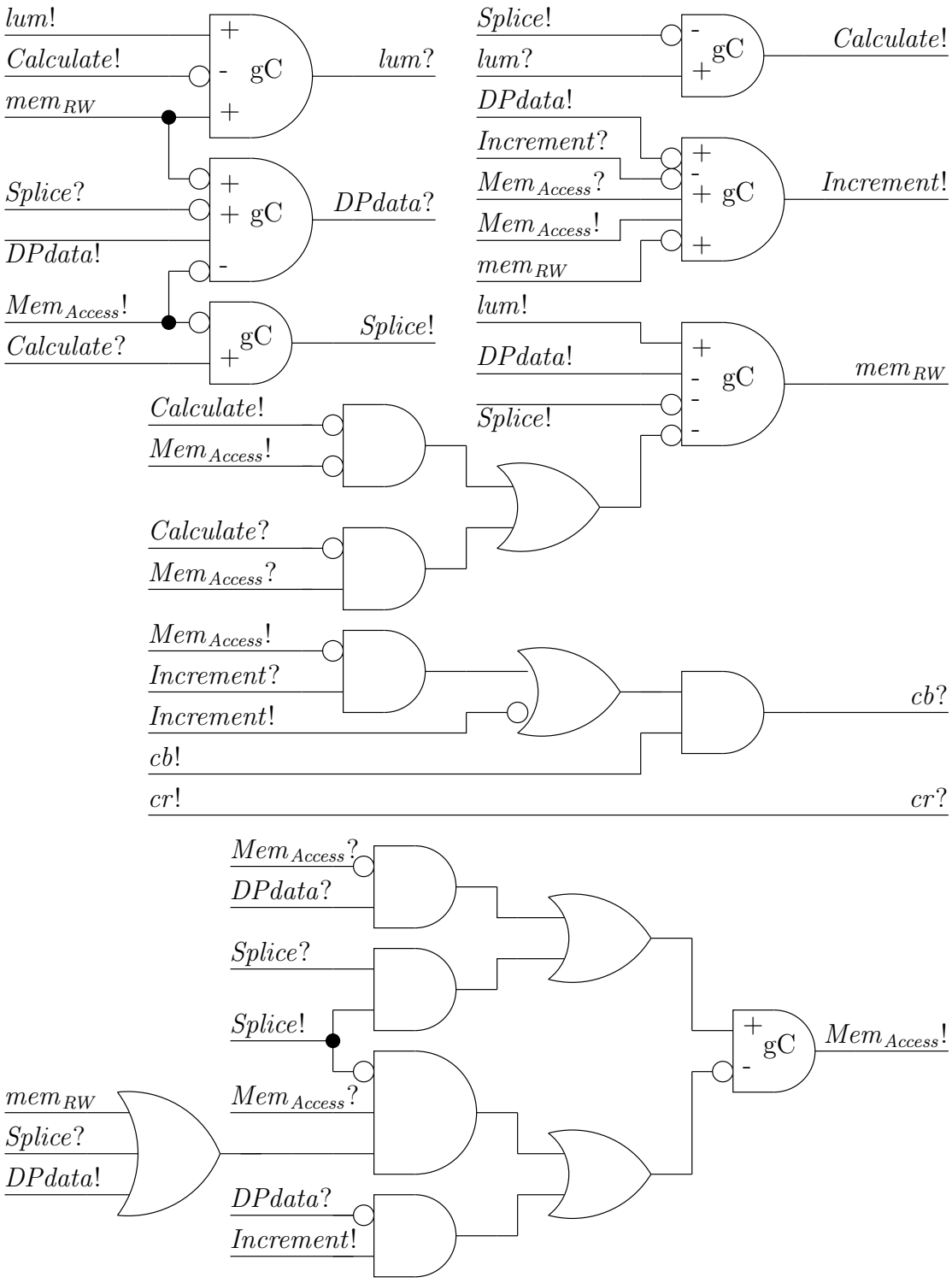

Figure 7.13. Circuit found using concurrency reduction on the *MPEG* example.

As Section 1.1 points out, the CAD tool `Petrify` uses a reshuffling approach that is very similar to that which this dissertation presents. In particular, both approaches treat reshuffling as a special case of concurrency reduction. However, there are also several differences. The techniques of this dissertation support quantitative timing assumptions and a different mechanism for specifying which transitions are important for data integrity. This section attempts to quantify the similarities and differences by comparing the two tools on a set of examples.

Each example of this section uses a channel-level Petri-Net specification as the input to `Petrify`. The command `petrify -er -4ph -untog` *example*`.g -o` *example*`.out.4ph -redc -gc -eqn` *example*`.eqn` produced the synthesizable Petri Net. The result was input to `ATACS` and annotated with the same timing assumptions used in the `ATACS` specification to compute the average cycle period.

For example, consider Table 7.2. In this case, `Petrify` finds the same solution as the tool that this dissertation presents does when it uses the N filter alone to stop after it finds the first solution. However, `Petrify` requires about three times the run time to find this solution.

Now consider Table 7.4. In this case, the N filter that this dissertation presents finds the optimal solution on its first attempt. However, `Petrify` did not quite find the optimal solution, even though it took nearly 40 times the run time to find its solution. Furthermore, it took about twice the run time of the branch-and-bound algorithm (filter E) that this dissertation presents.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

This dissertation presents new algorithms that accept channel-level specifications and find signal-level implementations that are correct, synthesizable, and efficient. There are several constraints on this problem. Some of these constraints are inherent in the channel-level specification. Other constraints are required by a particular choice of protocol or data path. This dissertation shows that a TEL structure can be used to represent all of these constraints.

Starting from a TEL structure that contains only these constraints, the algorithms that this dissertation presents must find implementations that have complete state coding. The usual approaches to this problem include reshuffling and state-variable insertion. This dissertation shows that starting from the most concurrent TEL structure that satisfies the initial constraints, concurrency reduction can find any possible reshuffled handshaking expansion of the original specification. Furthermore, state variables can be inserted by introducing initially unconstrained events into the TEL structure, and then proceeding with concurrency reduction. Thus, this dissertation treats both reshuffling and state-variable insertion as special cases of the general operation of concurrency reduction. This approach is similar to that taken by the CAD tool `Petrify` [23], but this dissertation applies it to specifications that contain quantitative timing assumptions.

Supporting quantitative timing assumptions allows optimizations at several levels of the design process. Section 6.1.1 and Section 7.2.3 show that timing information can reduce the size of the search space for concurrency reduction itself. This is because events that would otherwise be concurrent may be ordered under the given timing assumptions. Exactly determining all timed-concurrency information

requires timed-state-space exploration. This is an expensive operation. Therefore the heuristic of Section 6.1.1 computes this information just once on the initial TEL structure, and uses this information as an approximation to the timed-concurrency information for the derived TEL structures. This approximation allows the cost of the timed-state-space exploration to be amortized over many pruning decisions. Section 7.2.3 shows that this does reduce total run time, and for the examples considered, the approximation does not impair the ability of the search to find the optimal solution. Furthermore, some handshaking expansions that would not be synthesizable without timing assumptions are synthesizable under appropriate timing assumptions. Finally, prior work by Myers et al. [60, 59, 7, 58] has shown that quantitative timing information can speed up the synthesis process and result in better circuits.

Finding all possible ways to reduce concurrency in a TEL structure is a fundamentally exponential problem. However, this dissertation presents techniques to dramatically prune the search space. Even the pruning techniques of Chapter 5 can reduce the number of possibilities to be considered by many orders of magnitude compared to the theoretical upper bound. This dissertation also presents heuristics that may not find all solutions, but reduce the size of the search space even further. This dissertation demonstrates that these heuristics are capable of reducing the search space by an additional two orders of magnitude beyond the techniques of Chapter 5 — and by one order of magnitude beyond existing techniques — without significantly impacting the quality of the solutions found.

Furthermore, this dissertation shows that reducing concurrency in a TEL structure is simply a matter of adding rules to the TEL structure. This enables the algorithms to do significant work at the TEL-structure level instead of the state-graph level. This enables significant pruning to be done before state-space exploration.

These algorithms are adjustable, so that the user can decide the level of optimization required. For the most optimization, these algorithms must still search large solution spaces, but better circuits result. With more heuristics, the solutions are suboptimal, but require far less time to find.

The tool that this dissertation presents operates within a framework that supports hierarchy at multiple levels of the design process. The specification that Section 2.1 introduces is an extension of the *Very High-Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL)*, which contains ample support for hierarchical design. Furthermore, the extensions that Section 2.1 presents support channel communications and hence the Communicating Sequential Processes [35, 36] paradigm, which encourages modular design. This specification level, as well as the graphical, intermediate forms that this dissertation uses (Belluomini's *timed event/level (TEL) structures* [8, 7, 9]) support specifications that depend on signal levels. This allows the graphical intermediate forms to retain the modularity present in the specification. Each process results in a distinct connected component in the graphical representation. The abstraction techniques of Zheng et al. [75] and the modular synthesis techniques of Mercer et al. [57, 56] allow the synthesis engine to exploit this hierarchy information.

8.1 Future Work

We want to extend the example set to more realistic examples. Often the limiting factor is the synthesis engine itself. Once the specification becomes large, synthesizing even one signal-level solution becomes prohibitively expensive. Thus, more research is also needed on the synthesis engine, to enable it to handle larger examples.

More theory needs to be developed to support the techniques of this dissertation. This dissertation conjectures that certain techniques will not prune away any synthesizable solutions. While these conjectures are reasonable and consistent with our experimental findings, they have not yet been proven. For each technique, work needs to be done to either prove or disprove its exactness. In this case, *exact* means finding the exactly optimal solution or finding the entire solution space. If a pruning technique is *inexact* it just means that it might prune away a solution that would have been synthesizable. This issue is distinct from *correctness*. A solution is correct if it still meets the constraints of the given channel-level specification and

of the target protocol. Proving correctness of the concurrency reduction techniques is another important direction for future work.

At the other end of the spectrum, more work needs to be done on implementation. Much of Section 4.5 and Section 6.2.3 are speculative, in the sense that the techniques of these sections have not yet been implemented and tested. The current tool deals only with a four-phase protocol with no data and narrow sequencing. We want to generalize this to multiple, user-specifiable protocols. We also want to support data.

There are also many communication topologies for channels beyond what is currently implemented in the tool that this dissertation presents. For example, as discussed in Section 1.1 `SHILPA` supports multicast and broadcast channel with one sender and multiple receivers. There are also several ways to implement bidirectional channels. The simplest way still uses separate data buses for the two directions, but shares the handshake wires. For example, a *request* signal from a processor to a memory could indicate that a new read address is available on the address bus, and the corresponding *acknowledge* signal from the memory back to the processor could indicate that the data at that address are now present on the data bus. Burns [21] unifies this approach for small data values using a *unary* data encoding scheme. In this scheme, a channel can have multiple *request* wires and multiple *acknowledge* wires. Exactly one *request* wire and one *acknowledge* wire is used in any given communication. Which wire gets used determines the value of the datum being sent and received. In this scheme a unidirectional channel is just the special case in which one of the sets (either *request* or *acknowledge*) contains just one wire. If both sets contain just one wire, the channel is a pure synchronization channel. It is also possible to share a signal data wire for communication in both directions. In this case, care must be taken to ensure that multiple processes do not drive the same wire at the same time, and the keepers are used to drive the wire if no process is actively driving it. None of these communication topologies for individual channels are currently implemented in the tool that this dissertation presents. However, they are worth investigating in the future.

As the metric for evaluating the quality of the solutions found, the tool that this dissertation presents, as well as the discussion in Chapter 7, uses the average cycle period. There is more that could be done with this metric alone. Mercer's stochastic cycle period analysis can also provide a profile of the relative impact of each signal transition on the performance of the whole circuit. This could help to focus optimization efforts on the performance-critical communication actions.

There are many other possible metrics. The cycle period metric is most directly related to the *throughput* of the circuit. Throughput measures the circuit's ability to handle a high volume of transactions. Specifically, throughput is the number of transactions completed per unit time. However, in many applications, *Latency* is important. Latency measures the time it takes the circuit to perform any given transaction, from start to finish.

Finally, one must consider the costs of the final circuit. *Area* affects the cost of producing (or buying) the circuit. For an extremely rough, first-cut estimate of area, heuristics could count the state variables necessary for a given alternative. Note that this estimate may be misleading. Although each state variable requires circuitry, its presence may simplify the circuitry required for other, existing state variables. Hence, adding a state variable can actually reduce the total area of the final implementation. However, even this initial estimate can be useful for bounding the searches performed by the techniques that this dissertation presents. For example, it is not practical to consider adding so many state variables that the area for the storage elements for these state variables alone would exceed the area budget for the entire design.

Power consumption can also affect costs, as it affects the cost of the package and equipment necessary to deliver enough power and dissipate enough heat. Power consumption also affects the cost of using the circuit. For an extremely rough, first-cut estimate of power consumption, heuristics could count the number of transitions necessary to complete any given communication action. Again, this may be insufficient, because it ignores the amount of capacitance that each transition must drive. However, if driving even just minimal inverters through the required

number of transitions would exceed the energy budget for the communication action, then that implementation is not worth considering.

In some applications, it is not the average power consumption, but rather the peak power consumption that is critical. Future work could guide handshaking expansion with the goal of minimizing or limiting peak power. The framework developed by this dissertation could still be used. However, the branch-and-bound algorithm of Section 5.3 would need to use an estimate of peak power instead of an estimate of cycle time. It is significant that the framework that this dissertation presents is modular in this sense. Different estimators could be used in place of Mercer's stochastic cycle period analysis in order to optimize different metrics.

Once the estimates have narrowed the alternatives down to a few good candidates, the tool would attempt to synthesize each such candidate and compare synthesis results. The tool would reject any candidate that fails to synthesize. The analysis of the remaining synthesis results could be more detailed than the initial estimates. For example, since synthesis produces a transistor net-list, postsynthesis analysis could use the number of transistors (instead of the number of state variables) to estimate area. Similarly, postsynthesis analysis would have access to fanin and fanout information, which could be included in the cost function as well.

The tool could then present an annotated list of the possibilities found to the user. If none of these met the design objectives, the user could direct the tool to repeat the process using adjusted heuristics.

This dissertation uses TEL structures as the graphical intermediate format. This format could also benefit from some extensions and refinements. For example, currently TEL structures have actions that raise signals and other actions that lower signals. It could be useful to add an action that toggles a given signal, such as those available in `Petrify`. Whether this corresponds to raising or lowering the signal would depend on the current state. Such a toggle action would make implementing two-phase protocols more natural.

We want to add support for automatically evaluating many different protocols on each given example to find the best fit. This shortens the design time necessary

to find the first protocol that works. It also helps the designer to find a protocol that results in an efficient implementation. This would involve a new search problem in which the techniques that this dissertation has implemented would become the inner loop that tries a given protocol. The outer loop of the new optimization problem would try different protocols to find the best fit. For this to be practical, the inner loop must be efficient. That is the main significance of the results on run-time in Chapter 7, especially Section 7.2.5 and Section 7.4. While this dissertation implements a solution only to the subproblem of optimizing for one protocol, the efficiency of this implementation lays the foundation for the next challenge of optimizing over multiple protocols.

REFERENCES

[1] Akella, V., and Gopalakrishnan, G. SHILPA: A high-level synthesis system for self-timed circuits. In *Proc. International Conf. Computer-Aided Design (ICCAD)* (Nov. 1992), IEEE Computer Society Press, pp. 587–591.

[2] Akella, V., and Gopalakrishnan, G. Specification and validation of control-intensive IC's in hopCP. *IEEE Transactions on Software Engineering 20*, 6 (1994), 405–423.

[3] Armstrong, D. B., Friedman, A. D., and Menon, P. R. Design of asynchronous circuits assuming unbounded gate delays. *IEEE Transactions on Computers C-18*, 12 (Dec. 1969), 1110–1120.

[4] Bachman, B. M. Architectural-level synthesis of asynchronous systems. Master's thesis, The University of Utah, Dec. 1998.

[5] Bachman, B. M. Architectural synthesis of timed asynchronous systems. In *Proc. International Conf. Computer Design (ICCD)* (1999), pp. 354–363.

[6] Bardsley, A., and Edwards, D. A. The Balsa asynchronous circuit synthesis system. In *Forum on Design Languages* (Sept. 2000).

[7] Belluomini, W. *Algorithms for Synthesis and Verification of Timed Circuits and Systems*. PhD thesis, The University of Utah, Sept. 1999.

[8] Belluomini, W., and Myers, C. J. Timed event-level structures. In *Proc. International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (TAU)* (Austin, Texas, USA, Dec. 1997).

[9] Belluomini, W., and Myers, C. J. Timed circuit verification using tel structures. *IEEE Transactions on Computer-Aided Design 20*, 1 (Jan. 2001), 129–146.

[10] Berkel, K. v. *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*, vol. 5 of *International Series on Parallel Computation*. Cambridge University Press, 1993.

[11] Berkel, K. v. Introduction to VLSI programming. Lecture notes 2L760, Eindhoven University of Technology, July 1997.

[12] Berkel, K. v., and Bink, A. Single-track handshaking signaling with application to micropipelines and handshake circuits. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems* (Mar. 1996), IEEE Computer Society Press, pp. 122–133.

[13] Berkel, K. v., Kessels, J., Roncken, M., Saeijs, R., and Schalij, F. The VLSI-programming language Tangram and its translation into handshake circuits. In *Proc. European Conference on Design Automation (EDAC)* (1991), pp. 384–389.

[14] Blunno, I., and Lavagno, L. Automated synthesis of micro-pipelines from behavioral Verilog HDL. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems* (Apr. 2000), IEEE Computer Society Press, pp. 84–92.

[15] Bruno, J., and Altman, S. M. A theory of asynchronous control networks. *IEEE Transactions on Computers 20*, 6 (June 1971), 629–638.

[16] Brunvand, E. *Translating Concurrent Communicating Programs into Asynchronous Circuits*. PhD thesis, Carnegie Mellon University, 1991.

[17] Brunvand, E. Designing self-timed systems using concurrent programs. *Journal of VLSI Signal Processing 7*, 1/2 (Feb. 1994), 47–59.

[18] Brunvand, E., and Sproull, R. F. Translating concurrent programs into delay-insensitive circuits. In *Proc. International Conf. Computer-Aided Design (ICCAD)* (Nov. 1989), IEEE Computer Society Press, pp. 262–265.

[19] Burns, S. M. Automated compilation of concurrent programs into self-timed circuits. Master's thesis, California Institute of Technology, 1988.

[20] Burns, S. M. *Performance Analysis and Optimization of Asynchronous Circuits*. PhD thesis, California Institute of Technology, 1991.

[21] Burns, S. M., and Martin, A. J. Syntax-directed translation of concurrent programs into self-timed circuits. In *Advanced Research in VLSI* (1988), J. Allen and F. Leighton, Eds., MIT Press, pp. 35–50.

[22] Cortadella, J., Kishinevsky, M., Kondratyev, A., Lavagno, L., and Yakovlev, A. A region-based theory for state assignment in speed-independent circuits. *IEEE Transactions on Computer-Aided Design 16*, 8 (Aug. 1997), 793–812.

[23] Cortadella, J., Kishinevsky, M., Kondratyev, A., Lavagno, L., and Yakovlev, A. Automatic handshake expansion and reshuffling using concurrency reduction. In *Proc. of the Workshop Hardware Design and Petri Nets (within the International Conference on Application and Theory of Petri Nets)* (June 1998), pp. 86–110.

[24] Cummings, U., Lines, A., and Martin, A. An asynchronous pipelined lattice structure filter. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems* (Nov. 1994), pp. 126–133.

[25] De Micheli, G. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, Inc., New York, New York, 1994.

[26] Friedman, A. D., and Menon, P. R. Synthesis of asynchronous sequential circuits with multiple-input changes. *IEEE Transactions on Computers C-17*, 6 (June 1968), 559–566.

[27] Friedman, A. D., and Menon, P. R. Systems of asynchronously operating modules. *IEEE Transactions on Computers 20* (1971), 100–104.

[28] Furber, S. B., and Day, P. Four-phase micropipeline latch control circuits. *IEEE Transactions on VLSI Systems 4*, 2 (June 1996), 247–253.

[29] Genrich, H. J., and Lautenbach, K. The analysis of distributed systems by means of predicate/ transition-nets. *Lecture Notes in Computer Science: Semantics of Concurrent Computation 70* (1979), 123–146.

[30] Genrich, H. J., and Lautenbach, K. System modelling with high-level petri nets. *Theoretical Computer Science 13* (1981), 109–136.

[31] Genrich, H. J., and Lautenbach, K. S-invariance in predicate/transition nets. In *Informatik-Fachberichte 66: Application and Theory of Petri Nets — Selected Papers from the Third European Workshop on Application and Theory of Petri Nets, Varenna, Italy, September 27–30, 1982* (1983), Pagnoni, A. and Rozenberg, G., Eds., Springer-Verlag, pp. 98–111.

[32] Gopalakrishnan, G., and Akella, V. A transformational approach to asynchronous high-level synthesis. In *Proceedings of VLSI 93* (Sept. 1993), T. Yanagawa and P. A. Ivey, Eds., pp. 5.3.1–5.3.10.

[33] Gopalakrishnan, G., and Akella, V. High level optimizations in compiling process descriptions to asynchronous circuits. *Journal of VLSI Signal Processing 7*, 1/2 (Feb. 1994), 33–45.

[34] Gu, J., and Puri, R. Asynchronous circuit synthesis with boolean satisfiability. *IEEE Transactions on Computer-Aided Design 14*, 8 (Aug. 1995), 961–973.

[35] Hoare, C. A. R. Communicating sequential processes. *Communications of the ACM 21*, 8 (Aug. 1978), 666–677.

[36] Hoare, C. A. R. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[37] Jacobs, G. M., and Brodersen, R. W. A fully asynchronous digital signal processor using self-timed circuits. *IEEE Journal of Solid-State Circuits 25*, 6 (Dec. 1990), 1526–1537.

[38] Jacobson, H., Brunvand, E., Gopalakrishnan, G., and Kudva, P. High-level asynchronous system design using the ACK framework. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems* (Apr. 2000), IEEE Computer Society Press, pp. 93–103.

[39] Jump, J. R., and Thiagarajan, P. S. On the interconnection of asynchronous control structures. *Journal of the ACM 22* (Oct. 1975), 596–612.

[40] Kessels, J., Peeters, A., Kramer, T., Feuser, M., and Ully, K. Designing an asynchronous bus interface. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems* (Mar. 2001), IEEE Computer Society Press, pp. 108–117.

[41] Kim, E., Lee, J.-G., and Lee, D.-I. Automatic process-oriented control circuit generation for asynchronous high-level synthesis. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems* (Apr. 2000), IEEE Computer Society Press, pp. 104–113.

[42] Krieger, C. Solving state coding problems in timed asynchronous circuits. Master's thesis, University of Utah, 1999.

[43] Lavagno, L., Moon, C. W., Brayton, R. K., and Sangiovanni-Vincentelli, A. An efficient heuristic procedure for solving the state assignment problem for event-based specifications. *IEEE Transactions on Computer-Aided Design 14*, 1 (Jan. 1995), 45–60.

[44] Lin, B., Ykman-Couvreur, C., and Vanbekbergen, P. A general state graph transformation framework for asynchronous synthesis. In *Proc. European Design Automation Conference (EURO-DAC)* (Sept. 1994), IEEE Computer Society Press, pp. 448–453.

[45] Lines, A. M. Pipelined asynchronous circuits. Technical Report 1998.cs-tr-95-21, California Institute of Technology, June, 1998.

[46] Manohar, R. An analysis of reshuffled handshaking expansions. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems* (Mar. 2001), IEEE Computer Society Press, pp. 96–105.

[47] Manohar, R., and Tierno, J. A. Asynchronous parallel prefix computation. *IEEE Transactions on Computer-Aided Design 47*, 11 (Nov. 1998), 1244–1252. An earlier version is available as Caltech technical report CS-TR-96-20.

[48] Martin, A. J. The probe: An addition to communication primitives. *Information Processing Letters 20*, 3 (1985), 125–130. Erratum: IPL 21(2):107, 1985.

[49] Martin, A. J. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing 1*, 4 (1986), 226–234.

[50] Martin, A. J. Formal program transformations for VLSI circuit synthesis. In *Formal Development of Programs and Proofs* (1989), E. W. Dijkstra, Ed., UT Year of Programming Series, Addison-Wesley, pp. 59–80.

[51] Martin, A. J. Programming in VLSI: From communicating processes to delay-insensitive circuits. In *Developments in Concurrency and Communication* (1990), C. A. R. Hoare, Ed., UT Year of Programming Series, Addison-Wesley, pp. 1–64.

[52] Martin, A. J. Synthesis of asynchronous VLSI circuits. In *Formal Methods for VLSI Design*, J. Straunstrup, Ed. North-Holland, 1990, ch. 6, pp. 237–283.

[53] Martin, A. J., Lines, A., Manohar, R., Nystroem, M., Penzes, P., Southworth, R., and Cummings, U. The design of an asynchronous MIPS R3000 microprocessor. In *Advanced Research in VLSI* (Sept. 1997), pp. 164–181.

[54] Meng, T. H.-Y., Brodersen, R. W., and Messerschmitt, D. G. Asynchronous design for programmable digital signal processors. *IEEE Transactions on Signal Processing 39*, 4 (Apr. 1991), 939–952.

[55] Mercer, E. G., and Myers, C. J. Stochastic cycle period analysis in timed circuits. In *Proc. International Symposium on Circuits and Systems* (2000), pp. 172–175.

[56] Mercer, E. G., Myers, C. J., and Yoneda, T. Improved POSET timing analysis in Timed Petri Nets. In *Proceedings of International Workshop on Synthesis and System Integration of Mixed Technologies* (October 2001).

[57] Mercer, E. G., Myers, C. J., and Yoneda, T. Modular synthesis of timed circuits using partial orders on LPNs. In *Electronic Notes in Theoretical Computer Science* (April 2002), U. Nestmann and B. C. Pierce, Eds., vol. 65, Elsevier Science Publishers. Proc. Theory and Practice of Timed Systems.

[58] Myers, C. *Asynchronous Circuit Design*. John Wiley & Sons, 2001.

[59] Myers, C. J. *Computer-Aided Synthesis and Verification of Gate-Level Timed Circuits*. PhD thesis, Dept. of Elec. Eng., Stanford University, Oct. 1995.

[60] Myers, C. J., Belluomini, W., Killpack, K., Mercer, E., Peskin, E., and Zheng, H. Timed circuits: A new paradigm for high-speed design. In *Proc. of Asia and South Pacific Design Automation Conference* (Feb. 2001), pp. 335–340.

[61] Myers, C. J., and Martin, A. J. The design of an asynchronous memory management unit. Tech. Rep. CS-TR-93-30, California Institute of Technology, 1993.

[62] Myers, C. J., and Meng, T. H.-Y. Synthesis of timed asynchronous circuits. *IEEE Transactions on VLSI Systems 1*, 2 (June 1993), 106–119.

[63] Prosser, F., Winkel, D., and Brunvand, E. Sequencing in modular self-timed asynchronous control. Technical Report TR-420, Indiana University Computer Science Department, Oct. 1994.

[64] Rotem, S., Stevens, K., Ginosar, R., Beerel, P., Myers, C., Yun, K., Kol, R., Dike, C., Roncken, M., and Agapiev, B. RAPPID: An asynchronous instruction length decoder. In *Proc. International Symposium*

on Advanced Research in Asynchronous Circuits and Systems (Apr. 1999), pp. 60–70.

[65] Stevens, K. S., Rotem, S., Ginosar, R., Beerel, P., Myers, C. J., Yun, K. Y., Koi, R., Dike, C., and Roncken, M. An asynchronous instruction length decoder. *IEEE Journal of Solid-State Circuits 36*, 2 (Feb. 2001), 217–228.

[66] Sutherland, I. E. Micropipelines. *Communications of the ACM 32*, 6 (June 1989), 720–738.

[67] Vanbekbergen, P., Catthoor, F., Goossens, G., and Man, H. D. Optimized synthesis of asynchronous control circuits form graph-theoretic specifications. In *Proc. International Conf. Computer-Aided Design (ICCAD)* (1990), IEEE Computer Society Press, pp. 184–187.

[68] Vanbekbergen, P., Lin, B., Goossens, G., and de Man, H. A generalized state assignment theory for transformations on signal transition graphs. In *Proc. International Conf. Computer-Aided Design (ICCAD)* (Nov. 1992), IEEE Computer Society Press, pp. 112–117.

[69] Ykman-Couvreur, C., and Lin, B. Efficient state assignment framework for asynchronous state graphs. In *Proc. International Conf. Computer Design (ICCD)* (1995), IEEE Computer Society Press, pp. 692–697.

[70] Ykman-Couvreur, C., and Lin, B. Optimised state assignment for asynchronous circuit synthesis. In *Asynchronous Design Methodologies* (May 1995), IEEE Computer Society Press, pp. 118–127.

[71] Ykman-Couvreur, C., Lin, B., Goossens, G., and Man, H. D. Synthesis and optimization of asynchronous controllers based on extended lock graph theory. In *Proc. European Conference on Design Automation (EDAC)* (Feb. 1993), IEEE Computer Society Press, pp. 512–517.

[72] Ykman-Couvreur, C., Vanbekbergen, P., and Lin, B. Concurrency reduction transformations on state graphs for asynchronous circuit synthesis. In *Proc. International Workshop on Logic Synthesis* (May 1993).

[73] Zhao, Y. Hardware/software co-design for asynchronous systems. Bachelor's thesis, University of Utah, to be published.

[74] Zheng, H. *Modular Synthesis and Verification of Timed Circuits Using Automatic Abstraction*. PhD thesis, The University of Utah, Aug. 2001.

[75] Zheng, H., Mercer, E., and Myers, C. Automatic abstraction for verification of timed circuits and systems. *Lecture Notes in Computer Science 2102* (2001), 182–193.

[76] Zheng, H., and Myers, C. J. Automatic abstraction for synthesis and verification of deterministic timed systems. In collection of papers from TAU'00 available from http://www.async.ece.utah.edu.