A Burst-Mode Oriented Back-End for the Balsa Synthesis System[†]

Tiberiu Chelcea[*] Andrew Bardsley[**] Doug Edwards[**] Steven M. Nowick[*]

[*]*Department of Computer Science*
Columbia University
1214 Amsterdam Av., New York, NY 10027, USA
{tibi,nowick}@cs.columbia.edu

[**]*Department of Computer Science*
The University of Manchester
Oxford Road, Manchester M13 9PL, UK
{bardslea,doug}@cs.man.ac.uk

Abstract

This paper introduces several new component clustering techniques for the optimization of asynchronous systems. In particular, novel "Burst-Mode aware" restrictions are imposed to limit the cluster sizes and to ensure synthesizability. A new control specification language, CH, is also introduced which facilitates the manipulation and optimization of handshake control components. The new method has been fully integrated into a comprehensive asynchronous synthesis package, Balsa. Experimental results on several substantial design examples, including an 32-bit microprocessor core, indicate significant performance improvements for the optimized circuits.

1. Introduction

Several approaches have been proposed for synthesizing asynchronous circuits from high-level specification languages [1,8,5,4,12]. These approaches generally fall into two categories: algebraic calculi [5] and CSP-like language compilation [1,4,8,12].

The focus of this paper is on synthesis methods in the latter category, which are used in such well-known compilers as Balsa [1] and Tangram [4]. In these methods, a high-level specification is compiled through syntax-directed translation into an unoptimized intermediate representation, which is a netlist of *handshake components*. Each handshake component is then synthesized into a corresponding circuit using a template-based approach. While such compilation methods have the advantage of "transparency" (the programmer is controlling the final results from the high-level program), they also have the disadvantage of excluding aggressive transformations, generally using only simple and local peephole optimizations. In an alternative approach [8], a high-level specification is compiled into intermediate components through "process decomposition" and "handshake expansion", but this method is not entirely syntax-directed.

Generally, two types of optimizations have been proposed to improve the intermediate circuit representation: *peephole* and *resynthesis*. Peephole optimizations replace a fixed pattern of components by an optimized configuration of existing components. In contrast, resynthesis methods manipulate one or more components and produce new specifications which do not correspond to existing components; these specifications are then directly synthesized. Much of Balsa's and Tangram's optimization techniques [1,4,12] are peephole style. In Martin's approach [8], optimizations such as "guard symmetrization" can be regarded as a simple form of peephole, while "handshake reshuffling" is a limited form of resynthesis: taking one component and producing another one with potentially more optimized behavior.

The optimizations proposed in this paper fall into a separate category of resynthesis transformations: *clustering* [7,11]. The goal is to cluster several controllers into a single larger controller. Since the channels connecting the merged components are eliminated, so potentially are the overheads introduced by communications on those channels. The new controllers are then directly translated into Burst-Mode asynchronous controller specifications [9,10], and then synthesized using existing tools.

A small channel-based description language, CH, is introduced both to specify each handshake component's behavior and to facilitate the optimizations. There are two advantages to this approach: (a) optimizations can be regarded as abstract language manipulations through pattern-matching and template transformations, and (b) the resulting controller descriptions are independent of the synthesizable low-level specifications into which they are translated. Burst-Mode specifications are currently employed, but they are just one of several possible low-level specification styles.

In particular, the contributions of this paper are as follows:

- *new controller specification language*: The paper introduces a formal controller description language, CH (an abbreviation of "channel"), which is intermediate between an existing high-level description language (Balsa [1]) and low-level asynchronous controller specifications. The CH language is especially suitable to model handshake control components [1,4] and directly facilitates the manipulation and optimization of these components. With restrictions imposed on its syntax, CH translates to correct-by-construction Burst-Mode specifications [9,10].
- *system-level control optimization*: The paper introduces and formalizes several control optimizations as language manipulation procedures. The optimizations use clustering to eliminate internal channels, thus merging the components connected by these channels. Unlike other approaches, which target area, these optimizations are targeted to improving speed. The optimizations have all been formally verified using the trace theory verifier, AVER [17].
- *CAD package*: The paper introduces a new automated back-end for the Balsa asynchronous synthesis system. The back-end incorporates a Balsa-to-CH translator, the new controller optimization algorithm, a CH-to-BM translator, and a new interface with the Synopsys' Design Compiler for technology mapping.

The integrated CAD back-end is applied to four substantial asynchronous design examples, including a small 32-bit microprocessor core. The complete synthesis flow starts with a high-level Balsa system description and includes all steps through technology mapping. Pre-layout back-annotated Verilog simulations indicate up to 21% speed improvement over the unoptimized implementations.

The paper is organized as follows. Section 2 presents an overview of the proposed synthesis approach. Section 3 introduces the CH language, while Section 4 presents the two new resynthesis optimizations. Section 5 briefly discusses the technology mapping process. The last two sections present experimental results as well as conclusions and directions for future work.

Related Work. The CH description language is first compared with some existing asynchronous description languages. Then, our optimization techniques are compared with well-known peephole optimizations, as well as with resynthesis approaches closer to ours.

CH is an intermediate-level description language for handshake components, and it naturally captures channel behaviors and communication protocols between components. In contrast, both Balsa [1] and Tangram [4] are higher-level languages more suitable for system-level description rather than for individual component manipulation. At the opposite extreme, the Caltech handshake expansion language [8] is much more low-level: it explicitly indicates *every* individual signal transition, and thus is more cumbersome for modelling channel-based optimizations.

Closer to our work, there exist three intermediate-level languages for formally specifying individual handshake components [3,4,12]. In each approach, a specification defines both the channel interface of the component as well as the component's operation and interface protocol. Of these, van Berkel's [4] uses "generic operators" to describe a component's behavior. A drawback is that, in effect, it defines abstract

[†]This work was supported by NSF ITR Award No. NSF-CCR-0086036 and by EPSRC Grant No. GR/N19618.

classes of related handshake components; therefore, the language cannot distinguish subtle differences in protocol between related controllers. In contrast, our approach is able to model a variety of different protocols for a given operation by using a set of concrete "interleaving operators". van Berkel also uses a separate language to describe concrete handshake behaviors for each component; however, unlike our approach, this modelling loses information about component ports and is at a much lower-level (modelling individual signal transitions rather than channels).

In an alternative approach, Bardsley [3] adopts all of van Berkel's generic operators, but, unlike van Berkel, each operator is assigned *exactly* one associated behavior, i.e. handshake expansion. In contrast, CH defines more language operators, which specify a variety of feasible interleaving protocols, thus allowing for modelling multiple related components. Finally, Peeters [12] uses much lower-level descriptions (traces of individual signal transitions) to specify a component's operation. All three languages are capable of specifying both control and datapath handshake components; however CH currently can only specify control components.

Peephole optimizations have been proposed by both van Berkel [4] and Peeters [12]. There are two main types: higher-level ones (on handshake components) and gate-level ones (on netlists of gates). Higher-level optimizations include parallel compositions, sharing, reordering, and multi-channel optimizations; they typically consider small windows of 2-4 components and transform them to one or several other library handshake components. Gate-level optimizations form a region around a netlist of gates, and transform them into an optimized netlist of gates. In contrast, our proposed optimizations operate on much larger windows of components, and do not operate at the gate level.

Finally, closer to our work, clustering optimizations for asynchronous control components have been proposed by both Pena et al. [11] and Kolks et al. [7]. These approaches use behavioral [7] and structural [7,11] composition to obtain a single specification for the entire control part. The behavioral methods use trace structures [15] to perform composition, while the structural ones manipulate Petri nets to weave together controllers' behaviors. In each case, Petri nets are used as low-level specifications for the clustered controllers, which are then synthesized using Petrify [16].

There are three main differences between these approaches and ours. First, we use an intermediate-level description language, CH, for modelling and manipulating control components, while they use only low-level language models (Petri nets). Second, their approaches are targeted towards *unlimited* clustering, which may result in synthesis run-time problems. In contrast, our approach results in smaller clusters, and shows improvements in synthesis run time while still allowing for optimizing medium to large systems. Also, our approach is targeted towards Burst Mode [9] controllers, which have been shown to have good performance, while neither one of the previous papers have reported figures for speed improvement. Finally, the new back-end has been integrated into a comprehensive environment for asynchronous synthesis (Balsa), which provides an entire design flow from a high-level system specification down to layout.

2. Overview of the Approach

The proposed new flow for the Balsa system back-end is shown in Fig. 1. The shaded boxes indicate research contributions.

The new back-end splits the control and datapath handshake components synthesized by Balsa. The control part is then optimized, synthesized using a Burst-Mode CAD package, and technology mapped using a commercial tool. Datapath components are synthesized using the existing Balsa system technology mapper.

Control optimization is performed for an entire Balsa procedure (i.e. process), which describes a system or component to be synthesized. The idea is to eliminate internal channels between control handshake components and collapse them into larger controllers (see Fig. 2). The result of collapsing two components is a new controller with the same interface as the initial two controllers, minus the eliminated channels, and with the same observable behavior on the remaining

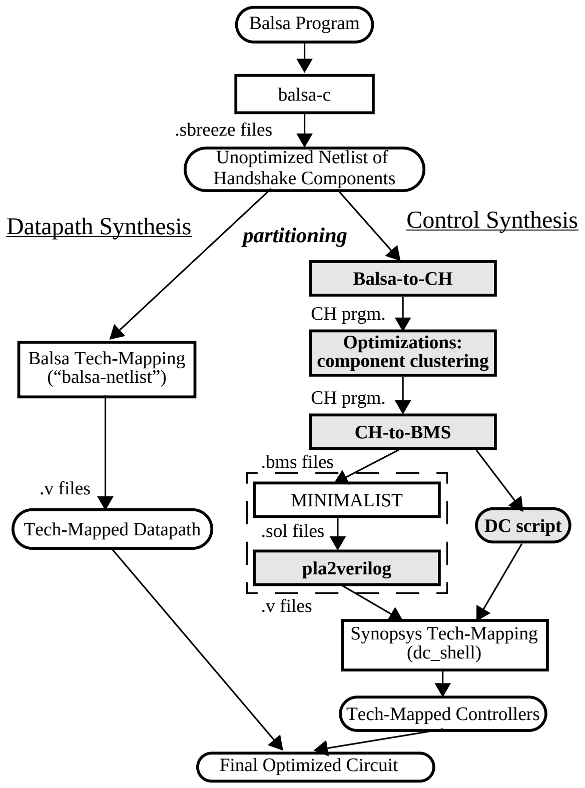

Figure 1: Balsa System Back-End

channels. In the end, the whole control part is reduced to a network of resynthesized controllers that communicate with each other, as well as with the datapath and other processes.

A small language (called CH) is introduced to allow for the description of control handshake components. The language also facilitates the new optimizations, which are treated as simple language manipulation procedures. The CH language contains a small set of channel types, as well as looping and interleaving operators. The operators were chosen with two aims: first, to be expressive enough to capture a number of useful component behaviors; and second, to be sufficiently constrained such that the specifications written in this language can be translated to correct-by-construction Burst-Mode specifications, the intended medium for controller synthesis.

The Minimalist package [6,10] is used to synthesize the optimized hazard-free controllers. The package enables the designer to direct the synthesis towards improvements in speed, area or power. Minimalist

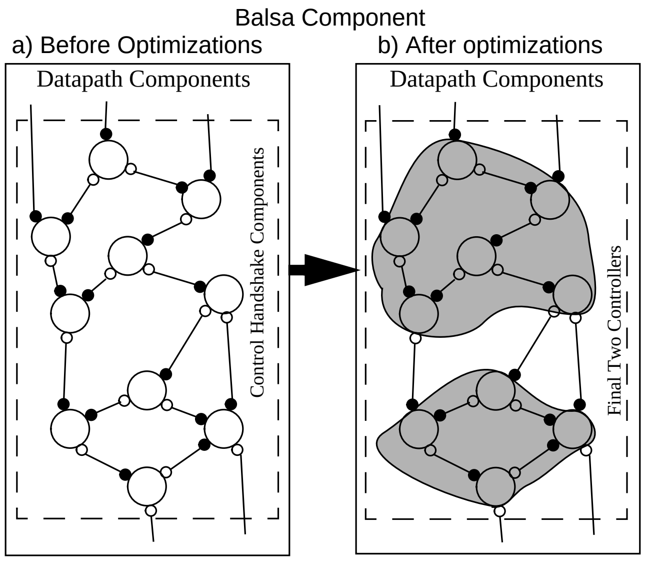

Figure 2: Control Optimization

synthesizes the controllers into two-level circuits containing generic gates; these circuits are then technology mapped using Synopsys' Design Compiler [14]. Finally, the technology-mapped datapath and control components are merged together to obtain the final optimized circuit implementation of the Balsa process.

3. The CH Language

The CH language is an intermediate control specification language between a high-level description language and a low-level controller specification language.

A CH program can be used to model both the interfaces and behavior of a *single* controller. The language has two types of expressions: *channel declarations* and *operators*. The language treats the two types uniformly: both channels and operators have an associated four-phase expansion and an "activity" type (passive or active), as discussed below.

A CH channel declaration defines the four-phase handshake expansion on a given channel. In practice, a channel is used to connect a controller with other controllers, datapath components or external interfaces. Connections can be either point-to-point or multi-way. A channel connecting various controllers is identified by using the same channel name in the CH program of each of the connected controllers. Channels are always assumed to be dataless (in Balsa, simple datapath components are used for data transfer, and are not associated with the control part). For example, a *passive* point-to-point channel A is described by the CH expression:

`(p-to-p passive A)`

and its four-phase handshake expansion is:

`[(i a_r +)] [(o a_a +)] [(i a_r -)] [(o a_a -)]`

where `i` indicates an input, `o` indicates an output, and the handshaking is initiated by an input request. A transition is denoted by the signal type (input or output), the signal name, and the transition type: + for rising, and - for falling, transitions. In contrast, an *active* channel has a handshake expansion that is initiated by an output request (see below).

Unlike channels, CH operators have up to two arguments. Each argument can be either a channel declaration or another operator. Just like channels, each operator is considered as having four distinct events, which follow either a passive or active protocol. In particular, the result of applying an operator to its arguments is a four-phase expansion which depends on both the operator's type and the type of its arguments (passive or active). The operator effectively defines, and returns, an *interleaving* of the events of its arguments, which are also grouped into four "higher-level" atomic events. As an example, given two channels, A passive and B active, which have four-phase handshake expansions, respectively:

`[(i a_r +)] [(o a_a +)] [(i a_r -)] [(o a_a -)]`

and

`[(o b_r +)] [(i b_a -)] [(o b_r -)] [(i b_a -)]`,

a CH expression which interleaves them using an *early-enclosure operator* is:

`(enc-early (p-to-p passive A) (p-to-p active B))`

which represents the following interleaving:

```
[(i a_r +)(o b_r +)(i b_a -)(o b_r -)(i b_a -)]
[(o a_a +)] [(i a_r -)] [(o a_a -)],
```

In this interleaving, immediately after the initial input request `a_r+` on channel A, the entire four-phase handshake on B is completed, followed by the remaining three events on channel A. Note that the operator effectively groups the input request and the entire four-phase handshaking on B into a single new event.

The reminder of this section now formally describes the syntax and the semantics of CH channel declarations and operators. It concludes with several examples of modelling handshake components and a discussion of compiling CH programs into Burst-Mode specifications.

3.1 Channel Declarations

There are six types of channels. The term channel is used in a general sense, as a means of synchronization on a group of wires, between a controller described by a CH expression and one or more other controllers and datapath components. Each channel type is introduced by its corresponding language keyword.

- **p-to-p** (point-to-point) channels connect two components; they are the most common type of handshake channels. They consist of two wires: a request and an acknowledge. The syntax is: `(p-to-p activity name)`, where `activity` is either *passive* or *active*. If the channel is active, its expansion is `[(o name_r +)] [(i name_a +)] [(o name_r -)] [(i name_a -)]`; if it is passive, the channel expands to `[(i name_r +)] [(o name_a +)] [(i name_r -)] [(o name_a -)]`, where the square brackets enclose events, and the round ones enclose a transition.
- **mult-ack** channels connect one component to several, and consist of a single request wire and multiple acknowledge wires. A transition on the request wire is followed by synchronized transitions on all acknowledge wires. The syntax is: `(mult-req activity name n)`, where `activity` is either passive or active, and `n` is the number of acknowledge wires. The resulting channel activity is given by the `activity` parameter. All of the acknowledge transitions can be regarded as grouped into a single event. For example, `(mult-req active c 2)` expands to `[(o c_r +)] [(i c_a1 +) (i c_a2 +)] [(o c_r -)] [(i c_a1 -) (i c_a2 -)]`.
- **mult-req** channels connect one component to several, and consist of multiple request wires and a single acknowledge wire. Transitions on all request wires are then followed by a transition on the acknowledge wire. The syntax is: `(mult-ack activity name n)`, where `activity` is either passive or active, and `n` is the number of request wires. The resulting channel activity is given by the activity parameter. All transitions on request wires are grouped into a single event.
- **mux-ack** channels connect one component to several, and consist of a single request wire and multiple acknowledge wires. Unlike a mult-ack channel, a request transition is followed by a transition on *exactly* one acknowledgment wire; if the ith acknowledgment is received, then the ith CH expression in a list is executed. This channel type is *always* active. For example:

  `(mux_ack a (`*`op1 arg1`*`) (`*`op2 arg2`*`))`

  expands to:

  ```
  [(o a_r +) choice
     (op1 [][(o a_a1 +)][(i a_r -)][(o a_a1-)]
          arg1)
     (op2 [][(o a_a2 +)][(i a_r -)][(o a_a2-)]
          arg2)
  ][][][]
  ```

  where channel `a` is the mux-ack channel, *arg1* and *arg2* are the two CH expressions in the list, *op1* and *op2* are interleaving operators which define the interleaving of the events on the mux-ack channel and the respective CH expression to be executed (*arg1* or *arg2*), `choice` is a keyword which indicates mutual exclusion between two events, and the four events returned by the mux-ack expansion are a single complex interleaving and three null events.
- **mux-req** channels connect several components to one, and consist of multiple request wires and a single acknowledge wire. Unlike a mult-req channel, a transition occurs on *exactly* one request wire; if the ith request is received, then the ith CH expression in a list is executed, including signaling on the acknowledge wire. This channel type is *always* passive. For example:

  `(mux_req a (`*`op1 arg1`*`) (`*`op2 arg2`*`))`

  expands to:

  ```
  [choice
     (op1 [(i a_r1 +)][(o a_a +)][(i a_r1 -)]
          [(o a_a-)] arg1)
     (op2 [(i a_r2 +)][(o a_a +)][(i a_r2 -)]
          [(o a_a-)] arg2)
  ][][][]
  ```
- **void** channels are channels whose all events are empty and are neither passive nor active. These channels are only used internally during the optimization process.

- **verb** channels are channels whose events and activity are entirely specified by the user. The activity of this channel is given by the first transition.

3.2 Looping Operators

There are two categories of CH operators: *looping* operators and *interleaving* operators. The operators of the former type provide a means for repeating the behavior of CH expressions.

- **rep** operators have only one argument. The expansion given by each such argument is repeated forever, unless interrupted by a break (discussed below). The syntax is: `(rep expr)`. If `expr` is passive then the `rep` expression is passive, otherwise it is active. The result of the `rep` expression is one non-empty event (containing the four-phase expansion of `expr`), followed by three empty events; in effect, the four-phase handshake expansion is degenerate. For example, the expansion of `(rep c)` is `[label c (goto label)][][][]`, where `c` is the expansion of `rep`'s argument, and `label` and `goto` are keywords, as generated by the parsing algorithm.
- **break** operators end the inner-most loop. The syntax is `(break)`. This operator is neither passive nor active. The operator expands to `[(bgoto label)][][][]`, where `bgoto` and `label` are keywords generated by the parsing algorithm. `bgoto` is similar to `goto`, but is handled differently by the parsing algorithm.

3.3 Interleaving Operators

The CH language has six interleaving operators: three for enclosures, two for sequencing and one for mutual exclusion. An enclosure-type interleaving on two channels is initiated and terminated by events on the first channel, and the events on the second channel are enclosed within the first handshake. A sequencing-type interleaving of two channels is initiated by events on the first channel and terminated by events on the second one. The mutual-exclusion interleaving of two channels places the events on each channel one after the other. However, this interleaving is more complex than a straight sequencing of the events, because the order of the two handshakes is determined by the environment. All interleaving operators take exactly two arguments.[1] (The expansions are presented in more detail in Table 2.)

- **enc-early**: this operator models the enclosure of the second argument's handshake expansion between the first and second events of the first argument's expansion. The syntax is `(enc-early a1 a2)`. If the first argument is active, then the operator is active, otherwise it is passive.
- **enc-middle**: this operator models the interleaving of the two arguments as follows: events 1 and 2 (3 and 4) of the second argument's expansion are enclosed between events 1 and 2 (3 and 4) of the first argument's expansion. The syntax is: `(enc-middle expr1 expr2)`. If the first argument is active, then the operator is active, otherwise it is passive. This is a powerful operator that can model C-element-like synchronization of channels as well as fork components.
- **enc-late**: this operator models the enclosure of the entire second argument's handshake expansion between the third and fourth events of the first argument. The syntax is: `(enc-late expr1 expr2)`. If the first argument is active, then the operator is active, otherwise it is passive.
- **seq** operator models the sequencing of its two arguments. The syntax is `(seq expr1 expr2)`. If the first argument is active, then the operator is active, otherwise it is passive. If more than two channels are to be sequenced, then the extra channels can be recursively embedded in the second argument: for example `(seq c1 c2 c3)=(seq c1 (seq c2 c3))`.
- **seq-ov** operator models the overlapped sequencing of its two arguments. The syntax is `(seq_ov expr1 expr2)`. It is defined only for active-active arguments; for all other activity pairs, it degenerates into other interleaving operators. This operator is useful in modelling transferrers [4].
- **mutex** operator indicates the mutually-exclusive execution of its two arguments. The syntax is `(mutex expr1 expr2)`. This operator is defined only for passive arguments, since it models external input choices. Therefore, this operator is always passive.The returned expansion of `(mutex c1 c2)` is `[(choice c1 c2)] [][][]`; the first event contains the arguments' expansions plus the keyword `choice` to indicate input choices, while the last three events are null. If the component allows mutual exclusion between n>2 expressions, then alternatives 2..n can be recursively embedded in the second argument: for example `(mutex c1 c2 c3)=(mutex c1 (mutex c2 c3))`.

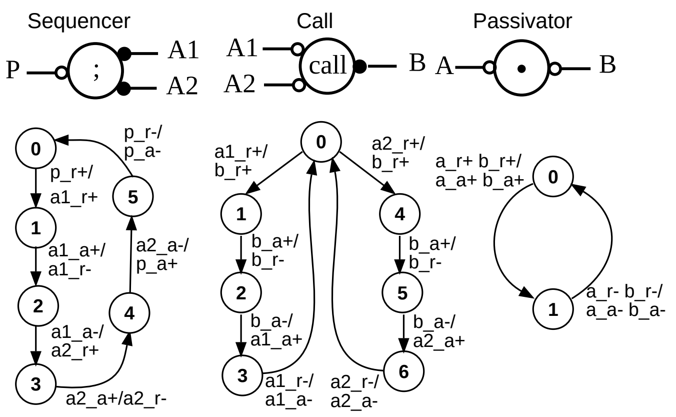

Figure 3: Symbols and Burst-Mode Specification for Three Handshake Components

3.4 Handshake Components: Modelling Examples

This subsection illustrates the CH language by modelling three common handshake components: a sequencer, a call module and a passivator.

A sequencer [1,4] has three channels: one passive one, P, and two active ones, A1 and A2 (see Fig. 3). The component is activated on its passive channel, and then completes handshakes on each of the two active channels before completing the handshake on the passive one. The CH program for the sequencer is:

```
(rep (enc-early (p-to-p passive P)
                (seq (p-to-p active A1)
                     (p-to-p active A2))))
```

A call module [1,4] has two passive channels (A1 and A2) and one active one (B). The environment starts a handshake on either A1 or A2; the call module then performs a full handshake on B before finishing the handshake on A1 or A2 (whichever was selected by the environment). The CH expression for the sequencer is:

```
(rep (mutex
        (enc-early (p-to-p passive A1)
                   (p-to-p active B))
        (enc-early (p-to-p passive A2)
                   (p-to-p active B))))
```

The final example is a passivator [1,4]. This component has two passive channels (A and B); it waits for the environment to start handshakes on both A and B, and then acknowledges on both channels. It then waits for the environment to start the return-to-zero phase on both A and B, and then finishes the two handshakes concurrently. The CH expression is:

```
(rep (enc-middle  (p-to-p passive A)
                  (p-to-p passive B)))
```

3.5 Burst-Mode Aware Restrictions on CH

An important practical issue in modelling asynchronous controllers using the CH language is to ensure they can be compiled into specifications that are easily synthesizable. Since our focus is on Burst-Mode controllers, restrictions on CH operator usage must be imposed to

1. The only exceptions are when defining `mux-ack` and `mux-req` channels, as discussed in Section 3.1. In these cases, the enclosure and sequencing operators only have one explicit argument, but, in each case, the implicit argument is the mux-ack or mux-req channel, respectively.

Table 1: Legal Combinations of Operands and Arguments

Operator	active/ active	active/ passive	passive/ active	passive/ passive
enc-early	Yes	No	Yes	Yes
enc-late	No	No	Yes	Yes
enc-middle	Yes	No	Yes	Yes
seq	Yes	No	Yes	Yes
seq-ov	Yes	No	No	No
mutex	No	No	No	Yes

guarantee that Burst-Mode specifications are always produced.

In particular, only certain combinations of interleaving operators and argument types are allowed; these combinations are called “Burst-Mode aware”. Table 1 summarizes the *exact* combinations of operators and argument types which result in correct-by-construction Burst-Mode specifications during the CH-to-BM translation. The rows represent the operator types, while the columns indicate the possible combinations of argument types (passive or active). If an entry corresponds to a “no”, then no valid Burst-Mode specification can be obtained; if it is a “yes”, then a correct specification can be obtained. The translation into Burst-Mode specifications for each “yes”/“no” combination has been manually verified for correctness.

Table 2 indicates the resulting four-phase expansion of each interleaving operator and combination of argument types. Only Burst-Mode aware entries are included. The four events of the first argument’s expansion are named a1 ... a4, and the four events of the second argument’s expansion are named b1 ... b4. As before, the square brackets enclose an event in the returned result.

3.6 Compiling CH into Burst-Mode Specifications

Once a control component is modelled in CH, it can be translated into a lower-level asynchronous controller specification. This subsection briefly introduces Burst-Mode (BM) specifications, the target of our synthesis path, and then presents the new compilation algorithm.

Burst-Mode is a commonly-used Mealy-type of specification for asynchronous controllers [9,10] (see Fig. 3). A specification consists of a set of states and a set of arcs. An arc is labelled with an input burst followed by an output burst, and connects two states. An input (output) burst describes a set of input (output) events or transitions: rising transitions are marked with a ‘+’, and falling ones are marked with a ‘-’. A BM machine waits for an input burst to arrive; transitions may come in any order and at any time. Once the complete input burst has arrived, the output burst is generated and the machine moves to the next specification state.

The new compilation algorithm, CH-to-BMS, translates CH programs into Burst-Mode specifications. The compilation algorithm has two steps: first, the CH program is translated into an intermediate form; then the Burst-Mode specification is obtained from the intermediate form.

The intermediate form is a list of signal transitions, with inserted labels for states, goto statements, and keywords indicating external input choices. The intermediate form is obtained by traversing the CH program recursively. As intermediate nodes (operators) are encountered, the traversal expands the operator hierarchically. When a terminal or leaf node (channel) is finally encountered, the recursion terminates and the four-phase expansion of the channel is returned. As recursion unwinds through the intermediate nodes, the partial results are combined into “higher-level” handshake expansions, according to the rules presented in Table 2, and Sections 3.2 and 3.3. If the intermediate node contains a control flow construct (rep, break, mutex), Section 3.2 and 3.3 indicate the additional keywords that are inserted into the handshaking expansion. In the end, a straight line four-phase handshake expansion for the entire CH program is obtained.

The resulting intermediate form is then translated into a Burst-Mode specification as follows. The list is traversed in linear order. Initially, a start state is created for the Burst-Mode specification. An input burst is created from the appropriate signal transitions at the start of the list, followed by a subsequent output burst. If a new input burst is encountered, the output burst is complete, so a new state is generated as the next state, as well as an arc connecting the two states and annotated with the corresponding input/output burst. If, instead, a goto statement is encountered, the program checks the existing state associated with the goto, and similarly makes it the next state; an input/output burst is created on an arc connecting the two states. The translation terminates when the list is empty.

Fig. 3 shows the Burst-Mode specifications of the three handshake components described above. For example, while in state zero, the call component waits for a rising transition on either `a1_r` or `a2_r`. If `a1_r+` has arrived (the environment starts a communication on the a1 channel), then `b_r+` is generated and the machine moves to state 1. The machine then waits for the next input burst (`b_a+`), and, after generating b_r- it moves to state 2. The machine returns to the starting state (zero) after two more transitions (`b_a-`/`a1_a+` and `a1_r-`/`a1_a-`). The behavior is similar if the environment starts a handshake on the a2 channel.

4. Optimizations

In this section, new optimizations are introduced which attempt to improve the speed of the control part of the circuits. To do so, the handshake components are clustered into larger controllers. The potential for improvements comes from the elimination of handshaking on the internal channels.

Currently, two new optimizations are proposed: *activation channel removal* and *call distribution*. The first optimization eliminates activation channels between handshake components, while the second one eliminates call modules by inlining them directly into their calling sites. Each optimization is discussed in turn in the next two subsections. The section ends with a discussion of the formal verification of these optimizations, and also with a summary of the optimization algorithms.

4.1 Activation Channel Removal

The idea behind this optimization lies in a fundamental characteristic of several control handshake components: in order to start their behavior, the components are first “activated” along a passive channel. If the channel that activates them is eliminated, then the handshaking overhead on that channel is also eliminated.

The following terminology will be used below: an *activation channel* is the handshake channel that connects the activating and activated components (it is an active channel for the first component and passive for the second one), and which encloses the “useful” behavior of the activated component within its handshake expansion. The *body* of a component is the “useful” part of that component, enclosed within a handshake on an activation channel (if present).

The optimization proceeds in two steps. It first hides the activation channel in the activated component (by replacing it with a *void* channel), and then inlines this component’s body directly into the CH expression of the activating component.

To illustrate the optimization, let us take two components: a deci-

Table 2: The Four-Phase Expansion of CH Operators

Operator	active/active	active/passive	passive/active	passive/passive
enc-early	$[a_1][a_2b_1b_2b_3b_4][a_3][a_4]$	-	$[a_1b_1b_2b_3b_4][a_2][a_3][a_4]$	$[a_1b_1b_2b_3b_4][a_2][a_3][a_4]$
enc-late	-	-	$[a_1][a_2][a_3][b_1b_2b_3b_4a_4]$	$[a_1][a_2][a_3][b_1b_2b_3b_4a_4]$
enc-middle	$[a_1b_1][b_2a_2][a_3b_3][b4a_4]$	-	$[a_1b_1][b_2a_2][a_3b_3][b4a_4]$	$[a_1b_1][b_2a_2][a_3b_3][b4a_4]$
seq	$[a_1a_2a_3a_4b_1][b_2][b_3][b_4]$	-	$[a_1a_2a_3a_4b_1][b_2][b_3][b_4]$	$[a_1a_2a_3a_4b_1][b_2][b_3][b_4]$
seq-ov	$[a_1a_2][b_1b_2][a_3a_4][b_3b_4]$	-	-	-

sion-wait and a sequencer. A decision-wait component [3,4] is used to sample passive channels. It consists of an activation channel and several pairs of passive-active channels. When the component is activated and a handshake is initiated on exactly one of its passive channels, the component performs a full handshake on the corresponding active channel, before completing the handshakes on both the passive and activation channels. The sequencer is activated on its passive channel, and then performs handshakes in order on each of its active channels. The CH expression for decision-wait (the activating component) is:

```
(rep (enc-early (p-to-p passive a1)
        (mutex
           (enc-early (p-to-p passive i1)
                      (p-to-p active o1))
           (enc-early (p-to-p passive i2)
                      (p-to-p active o2))))))
```

and for the sequencer (the activated component) is:

```
(rep (enc-early (p-to-p passive o2)
                (seq (p-to-p active c1)
                     (p-to-p active c2))))
```

Channel o2 is an activation channel for the sequencer, so it may be optimized out. The activation point is highlighted in bold in the CH description of decision-wait, and the body of the sequencer is also highlighted in bold.

First, a hide operation is performed on channel o2 in the sequencer:

```
(rep (enc-early void
                (seq (p-to-p active c1)
                     (p-to-p active c2))))
```

Then, the body of the new component is directly inlined into the decision-wait component's expression to replace the o2 channel:

```
(rep (enc-early (p-to-p passive a1)
     (mutex
        (enc-early (p-to-p passive i1)
                   (p-to-p active o1))
        (enc-early (p-to-p passive i2)
                   (enc-early void
                      (seq (p-to-p active c1)
                      (p-to-p active c2)))))))
```

The above optimization can also be illustrated on the corresponding handshake circuits, as shown in Fig. 4. The activation channel between the original components is highlighted in bold, and the two corresponding BM specifications are shown. On the right, the final merged component is given, after optimization, along with its BM specification.

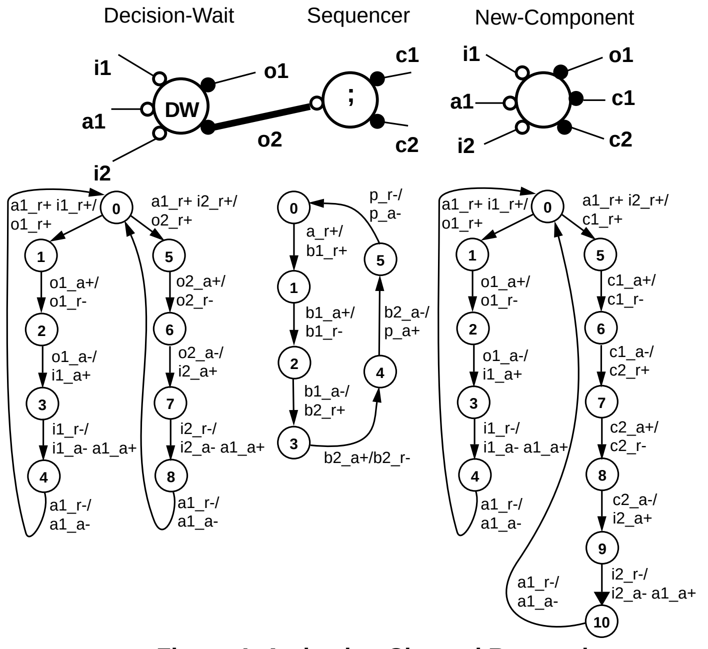

Figure 4: Activation Channel Removal

An important restriction in applying this optimization is that the resulting Burst-Mode specification should still be synthesizable, as discussed in Section 3.5. The CH language introduces a set of "Burst-Mode aware" restrictions on operators usage. Before the optimization is applied the activating component is BM aware, but during the optimization the activating channel disappears, and the body of the activated component is inserted in its place. Therefore, we must check again, after clustering, that the operator in the activating component and its two arguments (one of which is now the inlined code, with its own passive/active type) are still legal according to Table 1. If it is, the optimization succeeds; otherwise it fails, and the optimization cannot be performed.

4.2 Call Distribution

The final clustering optimization is to eliminate call handshake components, by folding the individual call statements into their call sites.

An n-way call component [1,4] has n passive channels and one active one; it receives mutually exclusive activations on the n channels, and, for each activation, performs a single handshake on its active interface before completing the handshake on the activating channel.

The new optimization works as follows. It first breaks an n-way call component into n simple fragments, each of which has a single passive and single active channel. The active channel is a replication of the original one, and the passive channel is one of the original n passive channels of the call component. The behavior of each call fragment is to enclose the handshake on the active channel within the handshake on the passive one. During optimization, these simple fragments are inlined into other controllers. If all n fragments are inlined into the same controller, then the optimization has succeeded; otherwise, the original call component is restored. In case of success, the n activation channels of the initial call component are thus eliminated.

To illustrate this optimization, an example taken from one of the simulated circuits (the systolic counter) is presented. The example consists of two handshake components, a sequencer and a call module. Both branches of the sequencer activate the call module:

```
SEQ: (rep (enc-early (p-to-p passive a)
        (seq (p-to-p active b1)
             (p-to-p active b2))))
CALL:(rep (mutex
        (enc-early (p-to-p passive b1)
                   (p-to-p active c))
        (enc-early (p-to-p passive b2)
                   (p-to-p active c))))
```

Initially, the call module is split into two fragments:

```
CALL1: (rep (enc-early (p-to-p passive b1)
                       (p-to-p active c)))
CALL2: (rep (enc-early (p-to-p passive b2)
                       (p-to-p active c)))
```

The first optimization can then be applied to both CALL1 and CALL2, and the new controller has the behavior:

```
(rep (enc-early (p-to-p passive a)
     (seq (enc-early void (p-to-p active c))
          (enc-early void (p-to-p active c))))
```

This optimization is also illustrated in Fig. 5. The optimization can be applied because the two call fragments are inlined into the *same* final controller. If they cannot be inlined together, the optimization would not be applied and the sequencer and the call would remain as two separate modules.

4.3 Formal Verification

The optimizations described above were proved to be correct using a trace theory verifier, Aver [15,17]. To do so, we have checked the behavior of two optimized controllers (using Activation Channel Removal) against the combined behavior of the same controllers (obtained using the "compose" and "hide" operators from trace theory [15]) for "conformation equivalence".

The verification procedure is as follows. Two CH simple programs, containing just one operator each (one in the activating component and one in the activated one), and sharing an activation channel, are manu-

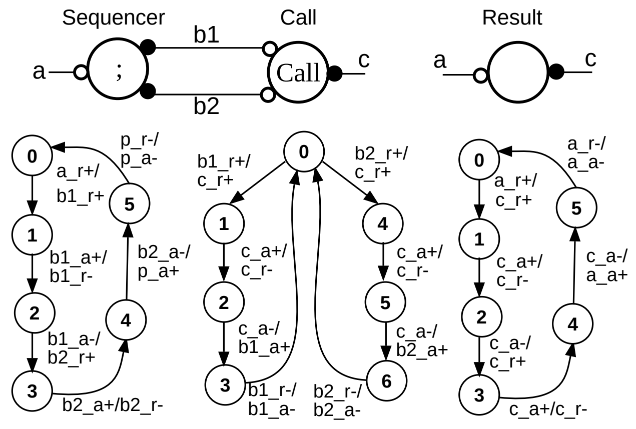

Figure 5: Call Distribution Example

ally translated into Petri nets. Then, using Aver, the two Petri nets are transformed into trace structures, composed [15], and the activation channel is hidden [15]. The result is a single trace structure that describes the external behavior of the two CH programs. In a separate step, the two CH programs are also optimized using Activation Channel Removal. The resulting program is then manually translated into a Petri net, read into Aver, and translated into a trace structure. The two trace structures are then checked for conformation equivalence. The process is repeated for all possible combinations of operators in the activating and activated CH programs.

The experiment has succeeded for all operator combinations. This indicates that the Activation Channel Removal optimization is behavior preserving, and therefore it is correct.

4.4 Optimization Algorithms

The optimization algorithms are introduced as systematic and automatic methods of applying the two new optimizations of Sections 4.1 and 4.2. They receive as an input a collection of CH programs (describing the handshake components) and a list of point-to-point channels connecting the control components (currently, only point-to-point channels are considered for optimization). The algorithms return a set of CH programs modelling the clustered components.

Intuitively, the Activation Channel Removal algorithm forms as large clusters as possible, according to the pseudo-code shown below (procedure T1_clustering). The algorithm starts with the netlist of control components. In its inner loop, the algorithm picks a point-to-point channel connecting two components, and merges their behaviors. If the composed behavior is still Burst-Mode synthesizable, the new component is returned to the current netlist of control components; otherwise the netlist is not updated. The algorithm iterates until all point-to-point channels have been inspected. At this point, the algorithm returns the netlist of clustered control components.

```
procedure T1_clustering (N);
            // N = netlist of control components
  for c = 1 ... (# point-to-point channels in N)
    (x, y) = components connected by c;
            // create the clustered component of x & y
    clustered_comp = activation_channel_removal (c, x, y);
            // if clustered component still synthesizable,
            // update netlist and continue
    if (BM_synthesizable? clustered_comp) then
      N = update (N, clustered_comp);
  end_for
  return (N);
end_procedure
```

The Call Distribution algorithm (procedure T2_clustering, shown below) has three steps. Starting with the initial netlist of control components, all call components are split, as explained above. Then, the Activation Channel Removal algorithm (T1_clustering, shown above) is called as a subroutine to obtain the netlist of clustered components. Finally, the algorithm checks that all call fragments obtained from the same original call component have been inlined into the same final controller. If that is not the case, the call component (the CH program describing it) is restored.

```
procedure T2_clustering (N);
            // N = netlist of control components
  C = get_call_components (N); // list of call components
            // Split each call component into a set of call
            // fragments. Group all sets into a list C'.
  C' = split_call_components (C);
            // update the netlist of components
  N' = update (N, C');
            // apply Activation Channel Removal to new netlist
  N" = call (T1_clustering (N'));
            // check for successful Call Distribution
  foreach c in C'  // for each set of call fragments
    if (all call fragments in c clustered together) then
            // Success. Do not update the netlist
    else
      N" = restore_call_component (N", c);
  end_for
  return (N");
end_procedure
```

Note that in our approach, using T1_clustering and T2_clustering, the size of the clustered components is not directly controllable by the designer. Instead, the inherent restrictions on applying “Activation Channel Removal” and “Call Distribution”, as well as the “Burst-Mode aware” restrictions on the CH language, determine how many components can be clustered together. In practice, the algorithms tend to yield netlists of several clustered components, as opposed to a single, monolithic controllers.

5. Technology Mapping

Before technology mapping, the Minimalist tool [6] synthesizes the Burst-Mode specifications into hazard-free two-level logic implementations. The technology mapping flow is then as follows (more details are explained below). First, the two-level logic implementations are modelled in Verilog HDL. Next, Synopsys’ Design Compiler is used to technology map these controllers. In general, note that this method is *not necessarily* sound, since the internals of Synopsys’ Design Compiler are not published and, therefore, it has the potential to introduce hazards. As a consequence, the technology-mapped circuits are formally analysed for hazard-freedom conditions. In all examples on which we have applied this method, the controllers are hazard-free.

The individual steps are now considered in more detail. A simple solution is used for structural modelling of the two-level logic implementations in Verilog HDL. The two-level nand-nand implementation of the controllers is divided into three separate Verilog modules: one for each of the two logic levels and a hierarchical module for the entire controller. Each module contains only inverters and nand functions. Next, each module is technology mapped separately with the Synopsys’ Design Compiler.

In the final step, each technology-mapped controller is formally verified for hazard freedom by analysing its decomposition into library gates. In particular, to verify hazard freedom, the transformations introduced by Synopsys’ Design Compiler were analysed to determine if all were *hazard-non-increasing*. In the literature, a number of common algebraic transformations have been identified as not introducing new hazards and therefore their application is safe; these include DeMorgan’s laws, factoring, associativity law, etc. [18].

The above technology mapping method is fairly unoptimized and may not yield the best possible results. Therefore, further research is be needed to take full advantage of the existing technology-mapping algorithms and improve the quality of the circuits.

6. Results

The entire synthesis flow has been tested for a number of examples. Each example is described using the Balsa language, and synthesized with `balsa-c`, to obtain the initial netlist of control and datapath handshake components. These components are then read by the optimization program to obtain the BM descriptions of the new controllers.

Table 3: Experimental Results

	Speed (ns)			Area (mm^2)		
	Unoptimized	**Optimized**	**Improvement**	**Unoptimized**	**Optimized**	**Overhead**
Systolic counter	51.29	40.43	**21.16%**	39.68	50.43	27.09%
Wagging register	49.82	42.43	**14.83%**	228.93	283.71	23.92%
Stack	121.58	107.70	**11.41%**	282.48	335.19	18.66%
Microprocessor core	66.48	60.65	**8.76%**	453.76	563.47	24.17%

Each Burst-Mode controller is synthesized with Minimalist [6], running the speed optimization scripts. The results are then passed to Synopsys' Design Compiler for technology mapping using the AMS 0.35 micron library. Datapath components are technology-mapped separately into the same library using Balsa tools. Finally, the control and datapath are merged together into the final circuit implementation, which is back-annotated using `pearl` and simulated with Cadence Verilog-XL.

The new optimization software consists of over 2500 lines of Scheme and Tcl code. The code is divided into several tools: a Balsa-to-CH translator, the new optimization algorithms, a CH-to-BM translator, interfaces with Minimalist and Synopsys' Design Compiler, and `pla2verilog` which translates Minimalist results into Verilog.

Four substantial examples have been optimized using the presented tools. These examples include an 8-handshake systolic counter [4], an 8-place 8-bit word wagging register [4], an 8-place 8-bit word stack, and a small 32-bit non-pipelined RISC-like microprocessor core. The microprocessor core (called SSEM) was described in [2]. Each example was simulated in a benchmark run. The systolic counter was simulated for an entire 8-handshake cycle, the wagging register has been simulated for forward latency, and the stack was simulated for a cycle of three push operations followed by three pop operations. The microprocessor core was simulated for a small program that writes consecutive memory locations with numbers 0 through 4.

Table 3 shows, for each of the four examples, the speed improvements and the area overheads compared to the unoptimized Balsa circuits. The optimizations presented in this paper attempt to improve only the control parts. Therefore, the overall speed improvement depends on the ratio between the control and the datapath: if the circuit is control dominated (for example, the systolic counter) then larger improvements can be expected; if the circuits are datapath dominated (for example, the microprocessor core), then the speed improvements tend to be smaller. Performance improvements range up to 21.16%.

In all four examples, the optimized circuits show area overheads over the unoptimized ones. There are several reasons for this result. The controllers were synthesized using Minimalist's speed scripts, so they were minimized using single-output optimization. The area of the synthesized circuit is often adversely affected because this Minimalist mode usually duplicates gates in order to decrease critical paths.

Our technology mapping method may also introduce area overheads. The two-level logic implementation of the controllers is split into three modules, each one being technology-mapped separately. This prohibits the Synopsys' Design Compiler from finding an optimal implementation *across* the two levels of logic. For example, an "and" gate in the first level of logic, followed by an "inverter" in the second one could be techology-mapped to a single "nand" gate, with much smaller area. However, our method currently prohibits this kind of simple optimization since the two levels are tech-mapped separately. In contrast, the original Balsa control components are manually designed and they have highly-optimized implementations.

7. Conclusions and Future Work

This paper discusses the implementation of a new back-end for the Balsa system. A channel description language has been introduced to facilitate control modelling and optimization. The proposed flow optimizes the control handshake components through clustering, and uses Minimalist and the Synopsys' Design Compiler to synthesize and tech-map the controllers. The entire flow has been validated through a number of examples.

The CH language may be extended to use other low-level specification styles (Extended Burst-Mode or Petri nets). Each of these styles may enable more optimizations, by removing some of the proposed Burst-Mode aware restrictions. However, removing restrictions may result in unlimited clustering, and the synthesis run-time may be affected [7,11]. There are two ways to alleviate this problem: either follow clustering by a decomposition step, or, more interestingly, elaborate a set of restrictions such that the synthesis step becomes manageable.

Relative timing [13] has been proved successful in a number of circuits. As it is, the CH language is not suited to handle relative-timing assumptions. Further research in modifying the language is needed to handle this type of optimization.

References

[1] A. Bardsley and D. Edwards, "Compiling the Language Balsa to Delay-Insensitive Hardware", *Hardware Description Languages and their Applications (CHDL)*, April 1997, pp. 89-91.

[2] A. Bardsley, "Balsa: An Asynchronous Circuit Synthesis System", MPhil Thesis, Department of Computer Science, University of Manchester, 1998.

[3] A. Bardsley, "Implementing Balsa Handshake Circuits", PhD Thesis, Department of Computer Science, University of Manchester, 2000.

[4] K. van Berkel, "Handshake Circuits: an Asynchronous Architecture for VLSI Programming", *International Series on Parallel Computation*, Vol. 5, Cambridge University Press, 1993.

[5] J. Ebergen, "Arbiters: an exercise in specifying and decomposing asynchronously communicating components", *Science of Computer Programming*, 18(3), pp. 223-245, 1992.

[6] R.M. Fuhrer, S.M. Nowick, M. Theobald, N.K. Jha, B. Lin, and L. Plana, "Minimalist: An Environment for the Synthesis, Verification and Testability of Burst-Mode Asynchronous Machines", *Technical Report CUCS-020-99*, Columbia University, July 1999.

[7] T. Kolks, S. Vercauteren, and B. Lin, "Control Resynthesis for Control-Dominated Asynchronous Design", *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async96)*, IEEE Computer Society Press, November 1996, pp. 233-243.

[8] A.J. Martin, "Programming in VLSI: From Communicating Processes to Delay-Insensitive Circuits", in *Developments in Concurrency and Communication*, UT Year of Programming Institute on Concurrent Programming, Addison-Wesley, 1990, pp. 1-64.

[9] S.M. Nowick, "Automatic Synthesis of Burst-Mode Asynchronous Controllers", *Technical Report CSL-TR-95-686*, Stanford University, March 1993.

[10] R.M. Fuhrer and S.M. Nowick, "Sequential Optimization of Asynchronous and Synchronous Finite-State Machines: Algorithms and Tools", Kluwer Academic Press, 2001.

[11] M.A. Pena and J. Cortadella, "Combining Process Algebras and Petri Nets for the Specification and Synthesis of Asynchronous Circuits", *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems (Async96)*, IEEE Computer Society Press, November 1996, pp. 222-232.

[12] A.M.G. Peeters, "Single-Rail Handshake Circuits", PhD Thesis, Eindhoven University of Technology, 1996.

[13] K. Stevens, S. Rotem, S.M. Burns, J. Cortadella, R. Ginosar, M. Kishinevsky, and M. Roncken, "CAD Directions for High Performance Asynchronous Circuits", *Proceedings of the ACM/IEEE Design Automation Conference*, IEEE Computer Society, 1999, pp 116-121.

[14] "Design Compiler Family Datasheet", http://www.synopsys.com/products/logic/design_comp_ds.html.

[15] D. L. Dill, "Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits", ACM Distinguished Dissertations, MIT Press, 1989.

[16] J. Cortadella, M. Kishinevsky, A. Kondratyev, L. Lavagno, and A. Yakovlev, "Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers", *Proceedings of XI Conference on Design of Integrated Circuits and Systems*, November 1996.

[17] D. Dill, S.M. Nowick, and R. Sproull, "Specification and Automatic Verification of Self-Timed Queues", *Formal Methods in System Design*, v1, pp. 29-60, 1992.

[18] D.S. Kung, "Hazard-non-increasing gate-level optimization algorithms", IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 1992, P, 1992, pp. 631-634.