Correct Compilation of Specifications to Deterministic Asynchronous Circuits

Scott F. Smith and Amy E. Zwarico

Department of Computer Science, The Johns Hopkins University, Baltimore, MD 21218 USA.

1 Introduction

In the past few years, researchers have developed powerful methods to aid in the construction of large asynchronous circuits [Mar90a, MBL+89, vB92, BS89, MBM89]. These methods are a significant departure from the traditional design methodologies used in circuit development in they are automatic or semi-automatic techniques for synthesizing asynchronous circuits from high-level specifications.

All of these projects excepting [MBM89] use the same basic methodology. The circuit is specified as a set of concurrently executing processes that can communicate via fixed channels. Each process is constructed from simple programming language constructs that include variables x and assignments $x := a$, conditional branching, looping, and sequencing. The specification then undergoes a series of transformations to produce a circuit.

While these methods have been around for the better part of a decade, there has been recent work that makes progress toward showing the correctness of such a methodology may be rigorously established by formal means [SZ92, WBB92, vB92]. Thus, using these methods it is possible to design and implement provably correct asynchronous circuits.

The asynchronous design method our work is based on is that of Martin *et al.* [Mar90a, MBL+89]. Burns has described and implemented a *circuit compiler* [BM88] that uses this method to automatically translate specifications into circuits. Preliminary results showing the correctness of this general methodology were reported by us in [SZ92].

The results in this paper rely on a standard model of asynchronous circuit behavior that is *speed-independent*, *fair* and exhibits *hazards* under certain conditions. By speed independence we mean that gates may delay arbitrarily but wire values propagate instantly from source to destination, and a wire is considered to have only one value at a time. In the literature, this is also known as the *isochronic forks* assumption, where a forked signal arrives at all destinations simultaneously. Unlike other formal models for circuits in the literature, we make explicit assumptions that gate delay *cannot* be infinite: if a gate is continuously enabled to switch, it will eventually switch (weak fairness assumption). Finally, we assume that a gate may ignore a spike on an input if the output of the gate does not depend on that particular input. Any other spike will produce a hazard. Hazards should provably never occur in circuits produced. The weaknesses of the

model are the zero wire-delay assumption and allowance of arbitrary fan-in and fan-out.

The paper is structured as follows. We begin by defining C-CSP (Circuit CSP) in Section 2. In Section 3, we give C-CSP meaning via an operational semantics. Two distinguishing features of our presentation are that we allow only fair computations and that we formalize violations of mutual exclusion on reading, writing, and synchronization. We define equivalence in Section 4 based on ideas of testing equivalence, and formalize what it means for the transformations to be semantics-preserving. In Section 5, we formalize compilation as a 6-phase rewrite system for translating a high-level C-CSP specification into an asynchronous circuit implementation and show that each rewriting phase is semantics preserving. A more complete version of this paper is available as [SZ93].

2 The Circuit Language — C-CSP

In this section we introduce C-CSP (Circuit-CSP), a language for specifying asynchronous circuits loosely based on [Hoa85, BM88]. We use the same language as the specification language, the intermediate language, and to express circuits. This decreases the overhead brought about by performing explicit language translations. C-CSP is specified by the following grammar.

$e ::= \mathbf{true} \mid \mathbf{false} \mid x \mid \bar{P}? \mid (e \wedge e) \mid (e \vee e) \mid \neg e$

$c ::= \mathbf{skip} \mid x := e \mid c; c \mid [e \rightarrow c] \| \ldots \| [e \rightarrow c] \mid *[c] \mid c \| c \mid P! \mid P? \mid \mathbf{with}\ d\ \mathbf{do}\ c\ \mathbf{en}$

$d ::= \mathbf{r}\ x\ d \mid \mathbf{w}\ x\ d \mid P!\ d \mid P?\ d \mid \epsilon$

We use e, c and d to range over boolean expressions, commands (also referred to as terms or processes), and declaration lists, respectively. $\mathcal{V}$ is the set of C-CSP variables; $x, y, z \ldots \in \mathcal{V}$ range over variables, and $!x, !y, !z, ?y, ?y, ?z \ldots \in \mathcal{V}$ range over *handshaking* variables. The handshaking variables are only used in the implementation of handshaking protocols. $\mathcal{P}_a$ is the set of active port names; $S!, P!, C!, D! \ldots$ range over $\mathcal{P}_a$. Similarly, $\mathcal{P}_p$ is the set of passive port names, and $S?, P?, C?, D? \ldots$ range over $\mathcal{P}_p$.

The boolean expressions e are the usual ones plus Martin's probe $\bar{P}?$, by which a passive port may test to see if the corresponding active port is enabled without causing a synchronization to occur.

The commands are similar to those of CSP. **skip** does nothing. Assignment, $x := e$, assigns the value of e to the boolean variable x. As a shorthand we represent $x :=$ **true** and $x :=$ **false** by $x \uparrow$ and $x \downarrow$, respectively. Sequential composition is denoted by ";". Choice among guarded commands is designated using $[]$ and infinite repetition of c is designated by $*[c]$. Parallel composition is represented by $||$. As in CSP, processes synchronize with one another through ports $P!$ and $P?$. An active port $P!$ and its corresponding passive port $P?$ form a *channel* we informally named P.

The declarations d in **with** d **do** c **end** bind variables and port names as follows. Declaration $P!$ in d binds occurrences of $P!$ in c. Declaration $P?$ binds

occurrences of $\bar{P}?$ an $P?$, **w** x binds all write occurrences of x and **r** x binds all read occurrences of x. All boolean variables may be declared twice: once by a declaration in which they may *only* be read (**r**), and once where they may be both read and written (**w**). These dual declarations are critical for proving the transformation correct. Also, corresponding active and passive parts of a channel P are declared separately as $P!$ and $P?$. Outside of the declaration of $P?$, for instance, $P?$ may not be used. It is not possible to declare the same port name or use of a variable more than once. A C-CSP term c is said to be *syntactically well-formed* if this property and some other technical restrictions hold. Hereafter C-CSP terms are taken to be the syntactically well-formed terms.

In order to formally describe compilation, we need to define three types of C-CSP terms — *modules, components* and *closed terms.* The class of C-CSP terms that may be separately compiled are called *modules.* All ports and write variables used in a module are declared in that module. The terms in the language corresponding to physical silicon devices (chips) are *components.* They may communicate with the outside world via ports, but, unlike modules, may not share boolean variables with the outside world. Finally, we have *closed terms* that share no variables or ports with the outside world.

C-CSP spans the expressibility gamut from high-level to gate-level circuit descriptions. Two sublanguages of C-CSP are S-CSP, a "pure" specification part of the language, and H-CSP, used to describe silicon devices. S-CSP (Specification CSP) terms are defined to be C-CSP terms without instances of handshake variables $!x$, $?x$. H-CSP (Hardware CSP) terms consist of the parallel execution of C-CSP processes that represent gates. Letting ℓ range over literals of the form x or $\neg x$, these processes are of the form $*[x := \ell_1 \ \mathbf{op} \ \ldots \ \mathbf{op} \ \ell_n]$ where **op** is $\wedge$ or $\vee$; or C-elements, $*[x := (\ell_1 \vee \ell_2) \wedge (x \vee (\ell_1 \wedge \ell_2))]$, abbreviated $*[x := (\ell_1 \ \mathbf{C} \ \ell_2)]$.

3 Operational Semantics of C-CSP

We define the operational semantics of C-CSP, by defining a relation $\rightarrow$ that represents a single step of the computation. Each configuration consists of a closed C-CSP term and a state σ containing the current values of ports and variables.

There are a number of challenges to giving semantics for C-CSP. Most importantly, mutual exclusion must be enforced on certain parts of circuits. We cannot properly realize circuits which violate mutual exclusion on writing variables or synchronization. We thus construct our C-CSP semantics so an **ERROR** is yielded if mutual exclusion is violated. The translations in turn guarantee that well-formed processes stay well-formed, resulting in a circuit that does not have two simultaneous requests for the same resource. One of the key features of our work is making these ideas rigorous; the other formalizations [WBB92, vB92] choose to weaken the expressiveness of the language and translation scheme so such errors may never occur.

State, initial state, and configurations are defined as follows. A *state* σ is a finite mapping from $\mathcal{V} \cup \mathcal{P}_a$ to **Bool**. $\iota(c)$ is the initial state for c, with all

variables set to **false**. A *configuration* $\langle c, \sigma \rangle$ consists of a closed term c and a state σ that represents a point in the computation.

Augmenting or changing the state function σ is abbreviated $\sigma[x = b]$. $\mathcal{P}_a$ is part of the domain of σ and is used to define the semantics of the probe: $\sigma(\bar{P}?) = \mathbf{true}$ iff $P!$ is waiting to synchronize.

One important notational convenience is the *context*, a term with a hole "$\bullet$" poked in it where another term may be placed. A *context* C is a term containing numbered holes "$\bullet_i$", $i \in N$. $C[c_1] \ldots [c_n]$ is the result of syntactically replacing all occurrences of $\bullet_i$ in C with terms c_i for each $1 \leq i \leq n$. A *closing context* for a term c is a context C such that $C[c]$ is closed.

We define a subclass of contexts, the *reduction contexts* to simplify the presentation of operational semantics. A reduction context is a syntactic means of isolating the next computation step to be performed. A *reduction context* R is a context constrained to be of the form

$$R = \bullet_i \text{ or } R; c \text{ or } R \| c \text{ or } c \| R \text{ or } R \| R \text{ or } \mathbf{with}\ d\ \mathbf{do}\ R\ \mathbf{end},$$

Often contexts with only one distinct hole are used, in which case $\bullet_k$ for the single present value of k may be abbreviated $\bullet$.

Evaluation is now defined. First, all boolean expressions are evaluated with respect to a state σ by homomorphically extending the domain of σ to all boolean expressions.

Second, the semantics of commands are defined by the single-step computation relation $\rightarrow$ mapping configurations to configurations as follows.

Definition 1.

(Assignment)
$$\langle R[x := e], \sigma \rangle \rightarrow \langle R[\mathbf{skip}], \sigma[x = \sigma(e)] \rangle$$
(Sequencing)
$$\langle R[\mathbf{skip}; c], \sigma \rangle \rightarrow \langle R[c], \sigma \rangle$$
(Selection)
$$\langle R[[e_1 \longrightarrow c_1 [\!] \ldots [\!] e_i \longrightarrow c_i [\!] \ldots [\!] e_n \longrightarrow c_n]], \sigma \rangle \rightarrow \langle R[c_i], \sigma \rangle$$
where $\sigma(e_i) = \mathbf{true}$ and $\forall j \neq i . \sigma(e_j) = \mathbf{false}$

(Repetition)
$$\langle R[*[c]], \sigma \rangle \rightarrow \langle R[c; *[c]], \sigma \rangle$$
(Parallelism)

(1) $$\langle R[P!], \sigma[P! = \mathbf{false}] \rangle \rightarrow \langle R[P!], \sigma[P! = \mathbf{true}] \rangle$$

(2) $$\langle R[P!][P?], \sigma[P! = \mathbf{true}] \rangle \rightarrow \langle R[\mathbf{skip}][\mathbf{skip}], \sigma[P! = \mathbf{false}] \rangle$$

(3) $$\langle R[\mathbf{skip} \| \mathbf{skip}], \sigma \rangle \rightarrow \langle R[\mathbf{skip}], \sigma \rangle$$

We next want to identify the bad configurations that violate mutual exclusion principles. Leading up to this we define those computation steps that change some expression value and those that depend on some expression value. A computation step *changes* an expression, $changes(e, \langle c, \sigma \rangle \rightarrow \langle c', \sigma' \rangle)$, if the value of e changes as a result of that computation step. A computation *depends* on the value of e, $depends(e, \langle c, \sigma \rangle \rightarrow \langle c', \sigma' \rangle)$, if e must be **true** in order for the step to occur or if the step assigns the value of e to a variable x.

Definition 2. $\langle c, \sigma \rangle \rightarrow$ **ERROR** (the configuration is in error) iff either

1. *changes*$(e, \langle c, \sigma \rangle \rightarrow \langle c', \sigma' \rangle)$ and *depends*$(e, \langle c, \sigma \rangle \rightarrow \langle c'', \sigma'' \rangle)$, and $c' \neq c''$, or
2. *changes*$(e, \langle c, \sigma \rangle \rightarrow \langle c', \sigma' \rangle)$ and *changes*$(e, \langle c, \sigma \rangle \rightarrow \langle c'', \sigma'' \rangle)$, and $c' \neq c''$, or
3. $c = R[[e_1 \longrightarrow c_1] \| \ldots \| e_i \longrightarrow c_i] \| \ldots \| e_n \longrightarrow c_n]]$ and $\sigma(e_i) = \sigma(e_j) =$ **true** for $j \neq i$

Definition 3. $\xrightarrow{*}$ is the transitive, reflexive closure of single-step computation $\rightarrow$.

We define the notion of a *semantically well-formed* component as a component that never causes an **ERROR** to arise internally (technical details are omitted). It is then an obligation of the circuit designer to show component specifications well-formed, and our translation process then guarantees that the resulting circuit is well-formed.

There are many computation paths possible, since at a given point multiple processes (or gates, at the hardware level) may be running and the next step could be performed by any one of those processes. Certain computation paths are *unfair* because processes that are able to execute are kept from doing so forever because all the steps are taken by other active processes.

Since circuits execute fairly (gates do not delay infinitely), our proofs of correctness will depend on the *fair behaviors* of processes. Specifically, we only concern ourselves with the *weakly fair* computations. That is, if a process is continuously enabled to execute, it will eventually execute. Although the informal idea is simple, the formal definition is not. For the full definitions, see [SZ93].

4 Circuit Testing and Equivalence

The equivalence we define is a variation on the *testing* equivalence of [Hen88]. This is a precise formalization of exhaustive testing, so if two processes are testing-equivalent, no difference will be ever be able to be ascertained between the two by a tester.

We add a new distinguished *success variable*, x_{success}, to the existing C-CSP variables, resulting in an extended language C-CSP*, the language of *testers*. The testing process indicates success by setting x_{success} to **true**. A *testing context* is a C-CSP* context.

Definition 4. Let c_0 be a closed C-CSP* term. A fair computation $\langle c_0, \iota(c_0) \rangle \rightarrow \langle c_1, \sigma_1 \rangle \rightarrow \ldots \rightarrow \langle c_n, \sigma_n \rangle \rightarrow \ldots$, is *successful* iff for some i, $\sigma_i(x_{\text{success}}) =$ **true**. It is *failing* if it is not successful.

We prove the transformations equivalence-preserving by showing that each transformation preserves the testing behavior. There is a technical complication that some transformations may actually *reduce* the possibility of **ERROR**s

arising in our circuits. This obviously causes no problems, but complicates the mathematics because something with more errors is not equal to something with less errors. This small complication will be ignored in this abbreviated presentation. The most important definition is the following.

Definition 5 Observation Equivalence. Let c and c' be closed semantically well-formed C-CSP* terms. $c \cong_{obs} c'$ iff there exists a successful computation of c iff there exists a successful computation of c', and there exists a failing computation of c iff there exists a failing computation of c'.

One property we may prove that simplifies our work considerably is the following.

Lemma 6 Determinism. *For all well-formed closed C-CSP* terms c, either all fair computations are successful or all fair computations are failing.*

Proof. The proof hinges on a **Bubbling Lemma**. If there is both a successful and failing computation of some c, the failing one can be translated (bubbled) into a successful one, contradicting the fact that it was failing. Fairness is critical to the proof.

A corollary of this theorem is all hazard-free circuits constructed of and, or, not, and C-elements must be observably deterministic, since H-CSP is a sublanguage of C-CSP. We plan to present this in full detail in a future paper, it is out of the scope of our present task. This theorem allows us to remove the second clause from the definition of $\cong_{obs}$, making it easier to show equivalence.

Theorem 7. *Let c and c' be closed semantically well-formed C-CSP* terms. $c \cong_{obs} c'$ iff there exists a successful computation of c iff there exists a successful computation of c'.*

Another complexity in defining equivalence for our system arises because certain of the transformation rules change the way processes interact with their environment, by for instance replacing a port P with an explicit handshaking protocol. These two processes will not be "equal" in the standard sense. We thus define a notion of equivalence tied to the compilation process, details of which will soon be given.

4.1 Rewriting and Equivalences

As described in detail in the next section, we divide the translation process into six phases and implement each phase using a distinct term rewriting system. For instance, if c; **skip** $\triangleright_R c$ is a rule in some rewrite system R, this rule allows any subexpression of the form c; **skip** to be replaced with c, for arbitrary terms c.

In Section 5, we define the six rewrite systems $\Rightarrow_1$–$\Rightarrow_6$ by giving six sets of rules $\triangleright_1 \ldots \triangleright_6$. A specification m_0 is compiled to a circuit m_6 by the rewriting

$$m_0 \Rightarrow_1^N m_1 \Rightarrow_2^N m_2 \Rightarrow_3^N m_3 \Rightarrow_4^N m_4 \Rightarrow_5^N m_5 \Rightarrow_6^N m_6.$$

For this we use the abbreviation $m_0 \Rightarrow_{1-6} m_6$ (m_0 compiles to m_6). To refer to modules that are the result of translating an initial specification through i levels we write $\Rightarrow_i m$, for $i \in \{1, \ldots, 6\}$.

4.2 Transformation Equivalence

The notion of "equivalence" under transformation is a difficult issue, because the method by which the specification/circuit interacts with the environment changes during compilation. Our solution is to change the *tests* in a similar fashion to the process being tested, and show the outcome of tests is the same. With this intuitive idea in mind, we now define the equality we use in proving the transformations correct. We note this equality $\cong_i$ for each level i of rewriting.

Definition 8. $m_k \cong_{k+1} m_{k+1}$ iff $\Rightarrow_k^N m_k$, $m_k \Rightarrow_{k+1}^N m_{k+1}$ and for any module m_k^t in C-CSP* such that $\Rightarrow_k^N m_k^t$ and $m_k^t || m_k$ is closed, if $m_k^t \Rightarrow_{k+1}^N m_{k+1}^t$, then $(m_k^t || m_k) \cong_{obs} (m_{k+1}^t || m_{k+1})$.

Intuitively given a specification module m_0 compiled to a digital circuit m_6, this compilation is *correctness-preserving* if and only if for any tester of the original system, m_0^t, this tester gets the same results on m_0 as its compiled counterpart m_6^t does on m_6.

5 Compilation of C-CSP Specifications to Circuits

Our system for incrementally translating C-CSP process specifications to circuit implementations consists of six rewrite systems that are applied sequentially, translating a specification m_0 to m_1, to m_2, ..., and finally to a circuit module m_6. Each of these rewrite systems is defined with respect to a set of equations, SCA, that equates certain terms that differ only slightly in their scoping structure, the way they commute guard order and parallel composition, and the associativity of parallel and sequential composition.

We begin by adding a "start channel" S to the term being compiled. We then begin the transformation. Phase 1 produces a separate process for each constructor of the original term, be it guard, loop, active or passive communication, assignment, or parallelism. Phase 2 expands the high-level synchronization of C-CSP into a 4-phase handshaking protocol. Phase 3 simplifies guarded commands so each guard is evaluated in parallel. Phase 4 modularizes the specification by giving each use of a port name a new, distinct, name. Phase 5 reshuffles the handshake protocols in order to make efficient circuit implementations more feasible. Finally, Phase 6 translates each of the small modules that remain into digital circuitry consisting of and, or, not gates, C-elements, and wires.

5.1 Phase 1: Syntax-Directed Rewriting

This phase transforms the original specification into many small processes of the form $*[[\tilde{S} \longrightarrow c; S?]]$, where c is a node of the original syntax tree. It also creates a

single "assignment process" corresponding to each boolean variable to physically isolate its storage location (**1 : ASSIGN 1**). All assignments synchronize with this process to assign a new value to the variable (**1 : ASSIGN 2**). (**1 : SEP**) creates a separate process for each part of the expression. In this rule, **OP** is metanotation for one of sequential composition, parallel composition, looping or choice. If it is choice, then **OP** also incorporates the guarding expressions $e_1, \ldots, e_n$. Top-down application of these rules produces a set of processes, one for each node in the abstract syntax tree The rewrite rules for phase 1 appear in Figure 1.

(**1 : ASSIGN 1**) **with w** x **do** c **end** $\triangleright_1$ **with w** x **do**
with $S_0?, S_1?$ **do**
$*[[\bar{S}_0? \longrightarrow x\downarrow; S_0? \| \bar{S}_1? \longrightarrow x\uparrow; S_1?]]$
end ||
with $S_0!, S_1!$ **do** c **end**
end

(**1 : ASSIGN 2**) **with w** x **do**
with $S_0?, S_1?$ **do** $*[[\bar{S}_0? \longrightarrow x\downarrow; S_0? \| \bar{S}_1? \longrightarrow x\uparrow; S_1?]]$ **end** ||
with $S_0!, S_1!$ **do** $C[x := e]$ **end**
end $\triangleright_1$
with w x **do**
with $S_0?, S_1?$ **do** $*[[\bar{S}_0? \longrightarrow x\downarrow; S_0? \| \bar{S}_1? \longrightarrow x\uparrow; S_1?]]$ **end** ||
with $S_0!, S_1!$ **do** $C[[\neg e \longrightarrow S_0! \| e \longrightarrow S_1!]]$ **end**
end

(**1 : SEP**) $*[[\bar{S}? \longrightarrow \mathbf{OP}(c_1, \ldots, c_n); S?]] \triangleright_1$
with $S_1!, \ldots, S_n!$ **do** $*[[\bar{S}? \longrightarrow \mathbf{OP}(S_1!, \ldots, S_n!); S?]]$ **end**||
$(||_{i=1}^{n}$ **with** $S_i?$ **do** $*[[\bar{S}_i? \longrightarrow c_i; S_i?]]$ **end**

Fig. 1. Rewrite rules for Phase 1

5.2 Phase 2: Handshaking Expansion

Handshaking expansion replaces the C-CSP synchronization constructs with boolean handshaking variables implementing the four-phase handshaking protocol. Since the active and passive ports need not be declared in the same scope, we must introduce two rules to carry out this rewriting. The rules appear in Figure 2, where we simplify the notation by letting $\mathbf{AHS}(!p, ?p) = !p\uparrow; [?p \longrightarrow \mathbf{skip}]; !p\downarrow; [\neg?p \longrightarrow \mathbf{skip}]$ and $\mathbf{PHS}(!p, ?p) = [!p \longrightarrow \mathbf{skip}]; ?p\uparrow; [\neg!p \longrightarrow ?p\downarrow]$, the active and passive handshaking protocols.

The left- and right-hand sides of the handshaking expansion rules do not have the same testing behavior when tested with the same test, because the tester is a fixed process, but to communicate with m_1 it must use ports and to communicate

(2 : HS 1) with $P!$ **do** $C[P!]$ **end** $\triangleright_2$ **with w** $!p$, **r** $?p$ **do** $C[\mathbf{AHS}(!p, ?p)]$ **end**
where C contains no occurrences of $P!$.

(2 : HS 2) with $P?$ **do** $C[\bar{P}?][P?]$ **end** $\triangleright_2$ **with r** $!p$, **w** $?p$ **do** $C[!p][\mathbf{PHS}(!p, ?p)]$ **end**
where C contains no occurrences of $P?$ or $\bar{P}$.

Fig. 2. Rewrite rules for Phase 2: Handshaking Expansion

with m_2 must use handshaking. Therefore the two will look different to almost all testers. However, the two rules are transformationally equivalent because all occurrences of ports are uniformly replaced with handshaking variables and handshaking variables are *exclusively* used in the handshaking protocols. It is then straightforward to show that the observable behavior in presence of ports (behavior of $m_1||m_1^t$) is the same as in the presence of handshaking (behavior of $m_2||m_2^t$ for $m_1^t \Rightarrow_2 m_2^t$).

5.3 Phase 3: Guard Simplification

This phase first separates guarded processes into parallel processes, **(3 : GUARD 1)** and then further separates each guarded process into its own collection of mutually exclusive guards that are conjunctions of literals **(3 : GUARD 2)**. These rules appear in Figure 2. **(3 : GUARD 1)** preserves the mutual exclusivity of each of the guarded processes by reshuffling the passive handshake on $!s, ?s$ so that the process activated by $\mathbf{AHS}(!s_i, ?s_i)$ waits for $!s$ to become false, deactivating all guards, before actually executing the process.

(3 : GUARD 1) $*[[!s \longrightarrow [e_1 \longrightarrow \mathbf{AHS}(!s_1, ?s_1)] \| \ldots \| e_n \longrightarrow \mathbf{AHS}(!s_n, ?s_n)]; \mathbf{PHS}(!s, ?s)]] \triangleright$
$*[[!s \wedge e_1 \longrightarrow ?s\uparrow; [\neg!s \longrightarrow \mathbf{AHS}(!s_1, ?s_1); ?s\downarrow]]] || \ldots ||$
$*[[!s \wedge e_n \longrightarrow ?s\uparrow; [\neg!s \longrightarrow \mathbf{AHS}(!s_n, ?s_n); ?s\downarrow]]]$

(3 : GUARD 2) with w $?s$ **r** $!s$ **do** $*[[!s \wedge e \longrightarrow ?s\uparrow; [\neg!s \longrightarrow \mathbf{AHS}(!s', ?s'); ?s\downarrow]]]$ **end** $\triangleright_3$
with w $?s$ **r** $!s$ **do**
$*[[!s \wedge e_1 \longrightarrow ?s\uparrow; [\neg!s \longrightarrow \mathbf{AHS}(!s', ?s'); ?s\downarrow]]] || \ldots ||$
$*[[!s \wedge e_n \longrightarrow ?s\uparrow; [\neg!s \longrightarrow \mathbf{AHS}(!s', ?s'); ?s\downarrow]]]$ **end**
where $e_1 \vee \ldots \vee e_n$ is the result of placing
e in disjoint disjunctive normal form.

Fig. 3. Rewrite rules for Phase 3.

5.4 Phase 4: Modularization

In this phase, each of the processes created by phases 1–3 is transformed into a module in order to localize all write scopes of variables. It should be noted that the only processes that are not already modules are those implementing atomic active and passive synchronization on non-distinguished ports, and the individual guarded command processes. The synchronization processes may fail to be modules because several distinct processes may use the same non-distinguished port (at this point handshaking variables). Thus the declarations of the "write" handshaking variables cannot be made local to a single process. Similarly, with guarded processes, there may be many guarded processes that wait for a start signal from the same active handshake. In the interest of space, we only present the rule for modularizing an active handshake variable shared by two processes.

$$
\begin{array}{l}
\textbf{with w } !p, \textbf{r } ?p \textbf{ do } C[\textbf{AHS}(!p, ?p)][\textbf{AHS}(!p, ?p)] \textbf{ end} \triangleright_4 \\
\textbf{with r } ?p, \textbf{r } !p_1, \textbf{r } !p_2, \textbf{w } !p, \textbf{w } ?p_1, \textbf{w } ?p_2 \textbf{ do} \\
\quad *[!p :=!p_1 \vee !p_2] \parallel *[?p_1 :=!p_1 \textbf{ C } ?p] \parallel *[?p_2 :=!p_2 \textbf{ C } ?p] \\
\quad C[\textbf{with w } !p_1 \textbf{ r } ?p_1 \textbf{ do AHS}(!p_1, ?p_1) \textbf{ end}] \\
\quad [\textbf{with w } !p_2 \textbf{ r } ?p_2 \textbf{ do AHS}(!p_2, ?p_2) \textbf{ end}] \\
\textbf{end}
\end{array}
$$

5.5 Phase 5: Reshuffling

Upon entering this phase, each module is of the form $[!s \longrightarrow c; \textbf{PHS}(!s, ?s)]$ for some c (ignoring declarations). The hardware implementation is simpler if the initial $[!s \longrightarrow \textbf{skip}]$ of the passive protocol is eliminated, and if some of the response $?s\uparrow; [\neg !s \longrightarrow \textbf{skip}]; ?s\downarrow$ is interleaved with the execution of c. Although each type of module requires a different form of reshuffling, the general principle is that a reshuffling may occur when the active communication $\textbf{AHS}(!s, ?s)$ corresponding to the passive communication $\textbf{PHS}(!s, ?s)$ above has not yet had anything reshuffled into it. As an example we show the reshuffling rule for active handshaking.

$$
\begin{array}{l}
(\textbf{5 : ACT}) \textbf{ with r } !s, \textbf{r } ?a, \textbf{w } ?s, \textbf{w } !a \textbf{ do } *[[!s \longrightarrow \textbf{AHS}(!a, ?a); \textbf{PHS}(!s, ?s)]] \textbf{ end} \\
\quad \triangleright_5 \\
\quad \textbf{with r } !s, \textbf{r } ?a, \textbf{w } ?s, \textbf{w } !a \textbf{ do} \\
\qquad *[[!s \longrightarrow \textbf{skip}]; !a\uparrow; [?a \longrightarrow \textbf{skip}]; ?s\uparrow; \\
\qquad [\neg !s \longrightarrow \textbf{skip}]; !a\downarrow; [\neg ?a \longrightarrow \textbf{skip}]; ?s\downarrow] \\
\quad \textbf{end}
\end{array}
$$

5.6 Phase 6: Final Compilation into circuits

This phase takes individual processes representing the various atomic actions, and transforms each into a circuit representation. The rules appear in Figure 4 (scoping information has been removed for brevity). The correctness of each of these rules relies critically on the fact that the handshaking protocol is obeyed by all parts of the circuit.

$$
\begin{aligned}
\textbf{(6 : ASSIGN)}\ & *[[!s_1 \longrightarrow x\uparrow;\ \mathbf{PHS}(s_1)[]!s_0 \longrightarrow x\downarrow;\ \mathbf{PHS}(s_0)]]\ \triangleright_6 \\
& *[x :=!s_1\ \mathbf{C}\ \neg!s_0]] * [?s_1 :=!s_1 \wedge x]] * [?s_0 :=!s_0 \wedge \neg x] \\
\textbf{(6 : SEQ)}\ & *[[!s \longrightarrow \mathbf{skip}];!s_1\uparrow;[?s_1 \longrightarrow \mathbf{skip}], ?s\uparrow;[\neg!s \longrightarrow \mathbf{skip}]; \\
& !s_1\downarrow;[\neg?s_1 \longrightarrow \mathbf{skip}];\mathbf{AHS}(!s_2, ?s_2); ?s\downarrow]\ \triangleright_6 \\
& *[?s_1 := s]] * [x :=?s_1\ \mathbf{C}\ \neg?s_2]] * [?s := x \vee ?s_2]] * [!s_2 := x \wedge \neg?s_1] \\
\textbf{(6 : GUARD)}\ & *[[!s \wedge e \longrightarrow ?s\uparrow;[\neg!s \longrightarrow \mathbf{AHS}(!s_1, ?s_1); ?s\downarrow]]]\ \triangleright_6 \\
& *[t_1 := s \wedge e]] * [t_2 :=?s_1 \wedge \neg!s]] * [x := t_1\ \mathbf{C}\ \neg t_2]] \\
& *[?s := x \vee ?s_1]] * [!s_1 := x \wedge \neg!d] \\
\textbf{(6 : ACT/PAR)}\ & *[[!s \longrightarrow \mathbf{skip}];!a\uparrow;[?a \longrightarrow \mathbf{skip}];?s\uparrow; \\
& [\neg!s \longrightarrow \mathbf{skip}];!a\downarrow;[\neg?a \longrightarrow \mathbf{skip}];?s\downarrow]\ \triangleright_6 \\
& *[!a :=!s]] * [?s :=?a] \\
\textbf{(6 : PASS)}\ & *[[!s \wedge !p \longrightarrow \mathbf{skip}];(?p\uparrow ||?s\uparrow);[\neg!p \wedge \neg!s \longrightarrow \mathbf{skip}];(?p\downarrow ||?s\downarrow)]\ \triangleright_6 \\
& *[x :=!s\ \mathbf{C}\ !p]] * [?s := x]] * [?p := x] \\
\textbf{(6 : LOOP)}\ & *[[!s \longrightarrow \mathbf{skip}];*[\mathbf{AHS}(!a, ?a)];\mathbf{PHS}(!s, ?s)]\ \triangleright_6 \\
& *[x :=!s\ \mathbf{C}\ !s]] * [!c := x \wedge \neg?c] \\
\textbf{(6 : SKIP)}\ & *[[!s \longrightarrow \mathbf{skip}];\mathbf{skip};\mathbf{PHS}(!s, ?s)]\ \triangleright_6 *[?s :=!s]
\end{aligned}
$$

Fig. 4. Rewrite rules for Phase 6: Circuit Generation

5.7 The Correctness of the Translation Process

The translation system presented above is correct in the following two senses. First, the compilation always produces a circuit and second, the final circuit is transformationally equivalent to the original specification.

Theorem 9. *For all $m \in$ S-CSP,*

1. *$m \Rightarrow_{1-6} m_6$ for some $m_6 \in$ H-CSP; and*
2. *if $m \Rightarrow_{1-6} m_6$ then for any closing testing module $m^t \in$ S-CSP* such that $m^t \Rightarrow_{1-6} m_6^t$, $(m||m^t) \cong_{obs} (m_6||m_6^t)$.*

6 Conclusions

We have shown that Martin *et al.*'s methodology can be made more rigorous. In order to accomplish this the new concepts of partial declarations, module and component, mutual exclusion violations, fairness, handshaking variables, distinguished ports, equational rewriting, separate compilation and observable determinism were introduced.

Two recent papers address the same general problem of proving correctness of asynchronous circuit compilation [WBB92, vB92]. These two systems are more closely related to each other than either are to our work. In brief, the main advantage over our work is lack of the mutual exclusion issue and complications introduced thereby, and the disadvantages are expressiveness of the specification language and speed of resulting circuits.

In a prelimary report [WBB92], Weber, Bloom, and Brown define a process language Joy and its compilation to asynchronous circuitry. Joy has a number of syntactic restrictions including the restriction that no processes may share variables or passive port names. Their correctness proof is based on a bisimulation equivalence, which does not incorporate fairness.

van Berkel gives a correctness proof for compiling the CSP-based specification language Tangram to circuits. Tangram can only have single uses of each port, and disallows concurrent reads. His proof of correctness is based on a trace equivalence that does not take fairness into account.

References

[BM88] Steven M. Burns and Alain J. Martin. Synthesis of self-timed circuits by program transformation. In G.J. Milne, editor, *The Fusion of Hardware Design and Verification*, pages 99–116. Elsevier Science Publishers B.V. (North-Holland), 1988.

[BS89] Erik Brunvand and Robert F. Sproull. Translating concurrent programs into delay-insensitive circuits. In *Proceedings of ICCAD-89*, pages 262–265. IEEE Computer Society Press, 1989.

[Hen88] M. Hennessy. *Algebraic Theory of Processes.* MIT Press, 1988.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[Mar90] Alain J. Martin. Programming in VLSI: From communicating processes to delay-insensitive circuits. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication.* Addison-Wesley, 1990. UT Year of Programming Institute on Concurrent Programming.

[MBL+89] Alain J. Martin, Steven M. Burns, T.K. Lee, Drazen Borkovic, and Pieter J. Hazewindus. The design of an asynchronous microprocessor. In Charles L. Seitz, editor, *Advanced Research in VLSI: Proc. of the Decennial Caltech Conference on VLSI*, pages 351–373, 1989.

[MBM89] Teresa H.-Y. Meng, Robert W. Brodersen, and David G. Messerschmitt. Automatic synthesis of asynchronous circuits from high-level specifications. *IEEE Trans. on CAD*, 8(11):1185–1205, November 1989.

[SZ92] Scott F. Smith and Amy E. Zwarico. Provably correct synthesis of asynchronous circuits. In Jørgen Staunstrup and Robin Sharp, editors, *2nd Workshop on Designing Correct Circuits, Lyngby*, pages 237–260. Elsevier, North Holland, 1992.

[SZ93] Scott F. Smith and Amy E. Zwarico. Correct compilation of specifications to deterministic asynchronous circuits. Technical Report 05-93, The Johns Hopkins University, Department of Computer Science, Baltimore, MD 21218 USA., 1993. Anonymous ftp: ftp.cs.jhu.edu.

[vB92] Kees van Berkel. *Handshake Circuits: an intermediary between communicating processes and VLSI.* PhD thesis, Eindhoven U., May 1992. Oxford University Press, to appear.

[WBB92] S. Weber, B. Bloom, and G. Brown. Compiling Joy to silicon. In *Advanced research in VLSI and parallel systems : proceedings of the 1992 Brown/MIT conference.* MIT Press, 1992.