Correct Compilation of Specifications to Deterministic Asynchronous Circuits

Scott F. Smith and Amy E. Zwarico[*]

December 11, 1992

1 Introduction

In the past few years, researchers have developed powerful methods to aid in the construction of large asynchronous circuits [Mar90b, Mar90c, vBNRS88, vBKR$^+$91, 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 language usually is similar to Hoare's CSP (Communicating Sequential Processes) [Hoa85]. The specification then undergoes a series of transformations to produce a circuit.

The asynchronous design method our work is based on is that of Martin *et al.* [Mar90b, Mar90c, Mar85, Mar86, MBL$^+$89]. Burns has described and implemented a *circuit compiler* [BM88, Bur88] that uses this method to automatically translate specifications into circuits. The first microprocessor ever constructed using this methodology has reasonable execution times [MBL$^+$89].

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. A proof of the correctness of such a methodology should has the following components: a definition of a circuit model, a formal syntactic and semantic definition of a high-level specification language, a definition of equivalence between the language and circuit models, and a proof that a specification s translated to some circuit c has the property that s is equivalent in behavior to c.

As hardware verification, what we present is only one half of a verification effort. We show that given a high-level CSP-style description, an equivalent circuit may be produced. The other half should be a logic in which high-level properties of the CSP-style specifications can be proven. Numerous such logics have been constructed [Hoa85, Mil89, Hen83, Hen88, BK84], so this is an eminently feasible task. The advantage of this approach is the relative simplicity, readability and abstractness of the high-level specification in contrast to reasoning directly about the circuit.

This paper presents a clarified presentation of the preliminary work described in [SZ92]. Contrasts of our approach with the others above may be found in the concluding section.

[*]Authors' addresses: Department of Computer Science, The Johns Hopkins University, Baltimore, MD 21218 USA. Email: `scott@cs.jhu.edu`, `amy@cs.jhu.edu`

The results in this paper rely on a circuit model that is *speed-independent*, *fair* and exhibits *hazards* under certain conditions. Hazards should provably never occur in circuits produced. This model is a standard model of asynchronous circuit behavior. The weaknesses of the model are the zero wire-delay assumption and allowance of arbitrary fan-in and fan-out.

Speed-Independent We work under the standard assumptions of *speed-independence*. That is, 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. An equivalent statement of this model is some forks in wires may be assumed to be *isochronic*, namely a forked signal arrives at all destinations simultaneously. Isochronic forks are imaginary objects, so it is up to the circuit layout and fabrication process to guarantee isochronicity [vB91, Mar90a].

Fairness 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).

Hazards 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.

1.1 Overview

We begin by defining the process algebra C-CSP (Circuit CSP) in Section 2. C-CSP is the specification language, the language in which each phase of the translation is carried out, *and* the language in which circuits are described. This use of one language greatly simplifies the presentation, and comes at little expense, since the definition of gates in the model can be seen from the semantics to exactly correspond to the appropriate gate behavior.

In Section 3, we give the language meaning via an operational semantics. Defining the semantics presents several challenges. First, we need to define executions so only fair computations are allowed. Since we are using only one language, this means the specifications are fair, intermediate states in the translation are fair, and the final circuits are also fair. Second, the semantics must formalize violations of mutual exclusion.

We define equivalence in Section 4 based on ideas of testing equivalence, and formalize what it means for the transformations to be semantics-preserving. One useful property we prove is all semantically well-formed terms (no mutual exclusion violations) are observably deterministic, meaning any test will always give the same result. A corollary is all circuits constructed of only and, or, and not gates and C-elements that are free of hazards are observably deterministic, assuming no bound is placed on gate delay.

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. Each phase of the rewrite system performs a particular type of translation. We show that meaning as formally defined is not changed by each phase of rewriting.

2 The Circuit Language — C-CSP

In this section we introduce C-CSP (Circuit-CSP), a variation of the CSP language [Hoa85] based on the version of CSP designed by Martin and Burns [Bur88] for specifying asynchronous circuits. We remove some syntactic sugar and add constructs needed to guarantee correctness. 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. In

Section 2.2 we describe two sublanguages, S-CSP and H-CSP, used to represent specifications and actual hardware devices respectively. The full C-CSP language is defined by the following grammar

$$\begin{aligned} e &::= \mathbf{true} \mid \mathbf{false} \mid x \mid \bar{P}? \mid (e \wedge e) \mid (e \vee e) \mid \neg e \\ c &::= \mathbf{skip} \mid x := e \mid c; c \mid [e \rightarrow c \,\|\, \ldots \,\|\, e \rightarrow c] \mid *[c] \mid c \| c \mid P! \mid P? \mid \\ &\ \mathbf{with}\ d\ \mathbf{do}\ c\ \mathbf{end} \\ d &::= \mathbf{r}\ x\ d \mid \mathbf{w}\ x\ d \mid P!\ d \mid P?\ d \mid \epsilon \end{aligned}$$

We will use e to range over boolean expressions, c to range over commands (also referred to as terms or processes), and d to range over declaration lists. $\mathcal{V}$ 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 take on special significance 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 the probe $\bar{P}?$ [Mar85], 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 $x := \mathbf{true}$ and $x := \mathbf{false}$ by $x \uparrow$ and $x \downarrow$, respectively. Sequential composition is denoted by ";". Choice is written using guarded commands 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. We call any occurrence of a variable on the left-hand side of an assignment a *write occurrence*. Any variable occurring in a guard, or the right-hand side of an assignment is a *read occurrence*.

Definition 1 (Binding) 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 $P?$, $\mathbf{w}\ x$ binds all write occurrences of x and $\mathbf{r}\ x$ binds all read occurrences of x.

All boolean variables may be declared (scoped) twice: once by a declaration in which they may *only* be read ($\mathbf{r}$), and once where they may be both read and written ($\mathbf{w}$). These dual declarations are important 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. Scoping restrictions are made explicit by the following.

Definition 2 A C-CSP term c is *syntactically well-formed* if and only if it can be generated by the above grammar and

1. each send $P!$ or receive port $P?$ occurring in c is declared at most once in c;
2. each variable x is declared at most once $\mathbf{w}\ x$ at most once $\mathbf{r}\ x$ in c;
3. if a variable x occurs inside a declaration $\mathbf{r}\ x$ but not inside a declaration $\mathbf{w}\ x$, that variable is not in a write occurrence; and
4. if c contains a declaration $P?$, no occurrences of $\bar{P}?$ or $P?$ in c are outside the scope of that declaration, if c contains a declaration $P!$, no occurrences of $P!$ in c are outside that declaration, if c contains a declaration $\mathbf{w}\ x$, x is not written outside this declaration, and if c contains both $\mathbf{r}\ x$ and $\mathbf{w}\ x$ declarations, x does not occur in c outside both of these declarations.

Hereafter C-CSP terms are taken to be the syntactically well-formed terms, and the composition of smaller terms to make larger ones must implicitly satisfy these properties. Define *strip*(c) to be c with all declarations removed, but all else left intact.

2.1 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. However, a module may interact with an external environment via ports and variables, indicated by declaring only the write variable, active or passive half of a channel. We formally define a module as follows.

Definition 3 A *module* m is a C-CSP term such that if $x := e$ occurs in m then this subterm occurs inside a declaration $\mathbf{w}\ x$, and all uses of ports $P!$ and $P?$ occur inside declarations of $P!$ and $P?$ respectively.

Modules here are useful in the same sense they are useful in programming languages: if a large specification is divided up into modules, each part may be compiled separately.

The terms in the language corresponding to physical silicon devices (chips) are components. They may communicate with the outside world via ports, but may not share boolean variables with the outside world. A module is not a component because modules may share variables, but variables are local to components. Components are generally composed of a collection of modules, that may be re-used in other components. Formally,

Definition 4 A *component* k is a module where any use of a non-handshaking variable x implies declarations $\mathbf{w}\ x$ and $\mathbf{r}\ x$ are both present in k.

Lastly we define closed terms. Closed terms are useless as hardware because they cannot communicate with the outside world, but are necessary in proving correctness.

Definition 5 A C-CSP term c is *closed* if and only if it is a component, all channels P used in c have both active and passive halves $P!, P?$ declared in c, and all variables used are both read and write scoped.

Before presenting the semantics, we describe two sublanguages of C-CSP which will be important in defining circuit compilation.

2.2 Specification and Hardware Sublanguages: S-CSP and H-CSP

Some of the constructs of C-CSP are only used in the final description of circuits. This arises because we wish to keep the entire translation process in one language, but it is a very large gap for one language to span. We need to isolate the sublanguage of C-CSP that is the "pure" specification part of the language, S-CSP, and the sublanguage that describes silicon devices, H-CSP. Terms in S-CSP all may be compiled to circuits.

Definition 6 S-CSP (Specification CSP) modules are C-CSP modules without instances of handshake variables $!x$, $?x$.

S-CSP forces specifications to abstract from the actual implementation of the synchronization between components.

H-CSP (Hardware CSP) modules are a small subclass of the C-CSP modules that represent a collection of gates. Let ℓ range over literals of the form x or $\neg x$ for variable x.

Definition 7 (gate processes) and, or, not/wire and C-element *gate processes* are defined as follows.

(and gate)	$*[x := \ell_1 \wedge \ldots \wedge \ell_n]$
(or gate)	$*[x := \ell_1 \vee \ldots \vee \ell_n]$
(not gate/wire)	$*[x := \ell_1]$
(C element)	$*[x := (\ell_1 \vee \ell_2) \wedge (x \vee (\ell_1 \wedge \ell_2))]$, abbreviated $*[x := (\ell_1 \mathbf{C}\, \ell_2)]$

Definition 8 H-CSP modules are modules m such that

$$strip(m) = c_1 || c_2 \ldots || c_n,$$

where each c_i is a gate process and no two gate processes c_i, c_j, $i \neq j$ may assign to the same variable.

3 Operational Semantics of C-CSP

An operational semantics describes the execution of a program or process in terms of the operations it can perform. Each operation takes the process from one configuration to another, where a configuration consists of a process and some internal state of the computation. In this way, computation is seen as a sequence of transitions involving simple data manipulations. 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. Side effects are necessary because state-holding elements are one of the fundamental structures of modern digital circuits. Almost all of the process algebra work in the literature is restricted to languages that have no side effects. Another challenge to overcome is the need to enforce mutual exclusion on certain parts of circuits. In asynchronous circuit design, there is often the need to have shared resources. However, using our translation method, we cannot properly realize circuits which violate mutual exclusion. Thus we construct our C-CSP semantics so that 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. So, if we begin with a well-formed component we will end with one. Martin also emphasizes the importance of mutual exclusion, but he argues informally about the well-formedness of a circuit description, where here requirements for mutual exclusion are completely rigorous.

State, initial state, and configurations the computation passes through are formally defined as follows.

Definition 9

- A *state* σ is a finite mapping from $\mathcal{V} \cup \mathcal{P}_a$ to **Bool**.
- ι :C-CSP $\rightarrow$ States maps a term c to an initial state σ_0 such that the domain of σ_0 is all variables x and active ports $P!$ occurring in c, and for all x and $P!$ in the domain of σ_0, $\sigma(x) = \mathbf{false}$ and $\sigma(P!) = \mathbf{false}$.
- 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 $\sigma[x = b]$, where $b \in \{\mathbf{true}, \mathbf{false}\}$. $\mathcal{P}_a$ 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. We let v (a general variable) range over $\mathcal{V} \cup \mathcal{P}_a$. Configurations are defined to be closed because computations are restricted to closed terms only.

One important notational convenience is the *context*, a term with a hole "$\bullet$" poked in it where another term may be placed. We define a subclass of contexts, the *reduction contexts*. This notion comes from [FFK87] (and is called there an evaluation context) to simplify the presentation of operational semantics. A reduction context is a syntactic means of isolating the next computation step to be performed.

Definition 10

- A *context* C is a term containing numbered holes "$\bullet_i$", $i \in N$. There may be multiple occurrences of $\bullet_i$ for some i and no occurrences for other values of i. $C[c_1]\dots[c_n]$ 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.

- A *reduction context* R is a context constrained to be of the form

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

  where each $\bullet_i$ for $i \in \mathbf{Nat}$ occurs at most once in R and c is a term.

Often contexts with only one distinct hole are used, in which case $\bullet_k$ for the single present value of k may be abbreviated $\bullet$. Also, we sometimes wish to denote an arbitrary expression that could either be of the form $R[c]$ or of the form $R[c][c']$. We write this as the latter, and if the hole $\bullet_2$ does not occur in R, $R[c][c'] = R[c]$. We first define evaluation of boolean expressions, and then define the operational semantics for closed C-CSP terms.

3.1 Semantics of Expressions

All boolean expressions are evaluated with respect to a state σ by homomorphically extending the domain of σ to all boolean expressions.

3.2 Semantics of Commands

The semantics of commands are defined by the single-step computation relation $\rightarrow$ mapping configurations to configurations. Most of the rules are straightforward. For instance, the assignment rule takes a configuration $\langle R[x := e], \sigma\rangle$ to one in which the command has finished execution and the state σ has been augmented with the value of e assigned to x, $\langle R[\mathbf{skip}], \sigma[x = \sigma(e)]\rangle$.

Definition 11 The one-step computation relation on configurations, $\rightarrow$, is the least relation such that

(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)
$$\langle R[[e_1 \longrightarrow c_1 [\!] \ldots [\!] e_i \longrightarrow c_i [\!] \ldots [\!] e_n \longrightarrow c_n]], \sigma\rangle \rightarrow \langle R[c_i], \sigma\rangle$$
$$\text{where } \sigma(e_i) = \mathbf{true} \text{ and } \forall j \neq i . \sigma(e_j) = \mathbf{false}$$
(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 those 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 if the value of e changes as a result of the computations. A computation depends on the value of e if e must be true in order for the step to occur or if the step only assigns the value of e to a variable x.

Definition 12

1. *changes*$(e, \langle c, \sigma\rangle \rightarrow \langle c', \sigma'\rangle)$ iff $\sigma(e) \neq \sigma'(e)$.
2. *depends*$(e, \langle R[c_1][c_2], \sigma\rangle \rightarrow \langle R[c_1'][c_2'], \sigma'\rangle)$ iff either
   (a) $\sigma(e) = \mathbf{true}$ and for all σ'', σ''', if $\sigma''(e) = \mathbf{false}$ $\langle R[c_1][c_2], \sigma''\rangle \not\rightarrow \langle R[c_1'][c_2'], \sigma'''\rangle$; or
   (b) $c_1 = x := e$ and $\bullet_2$ does not appear in R.

Definition 13 $\varepsilon(\langle c, \sigma\rangle)$ (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 $\sigma(e_i) = \sigma(e_j) = \mathbf{true}$ for $j \neq i$

We will informally write $\langle c, \sigma\rangle \rightarrow \mathbf{ERROR}$ to mean $\varepsilon(\langle c, \sigma\rangle)$.

Definition 14 $\overset{*}{\rightarrow}$ is the transitive, reflexive closure of single-step computation $\rightarrow$.

3.3 Semantic Well-Formedness

At the level of components, we require that no mutual exclusion errors may occur for a well-formed component. It is then an obligation that all component specifications be shown well-formed. The translation process then guarantees that the resulting circuit is well-formed.

Definition 15 A component k is *semantically well-formed* iff for all components k' such that $k\|k'$ is closed and all computations

$$\langle k\|k', \iota(k\|k')\rangle \xrightarrow{*} \langle k_n\|k'_n, \sigma_n\rangle \to \textbf{ERROR}$$

implies $\neg\varepsilon(\langle k_n, \sigma_n\rangle)$, and furthermore if $\neg\varepsilon(\langle k'_n, \sigma_n\rangle)$ then $k_n = R[C!]$ and $k'_n = R[C?][C?]$, or $k_n = R[C?]$ and $k'_n = R[C!][C!]$.

This means that any error may be traced to something outside the component, never internal to the component.

Definition 16 Configuration $\langle c, \sigma\rangle$ is *(semantically) well-formed* iff there is no computation $\langle c, \sigma\rangle \xrightarrow{*}$ **ERROR**. A closed term c is well-formed if $\langle c, \iota(c)\rangle$ is well-formed.

3.4 Computations and Fairness

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. For instance, the term

$$*[x := \neg x]\| * [\bar{P} \longrightarrow Q!; P?]\|P!$$

has an unfair infinite computation that starves out the synchronization on channel P by repeatedly and without interruption executing $x := \neg x$.

Since circuits execute fairly (gates to 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 this reason, the definition of fairness and properties about fair computations are relegated to Appendix A.

4 Circuit Testing and Equivalence

The equivalence we define is a variation on the *testing* equivalence of [NH83, Hen88], extended to include some notions from Morris/Plotkin operational equivalence [Plo77]. It 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. Testing is an internal or self-consistent notion of equivalence, processes are tested by other processes only. As long as we believe that these tests are rich enough, testing has the advantage that it is the strongest (most things equal) nontrivial equivalence that is a congruence.

We add a new distinguished *success variable*, x_{success}, to the existing C-CSP variables $\mathcal{V}$, 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 17 Let c_0 be a closed C-CSP* term. A fair computation $\langle c_0, \iota(c_0)\rangle \to \langle c_1, \sigma_1\rangle \to \ldots \to \langle c_n, \sigma_n\rangle \to \ldots$, is *successful* iff for some i, $\sigma_i(x_{\text{success}}) = \textbf{true}$. It is *failing* if it is not successful.

We prove the transformations equivalence-preserving by showing that each transformation preserves the testing behavior. The technical definition is complicated by a number of issues. The first complexity arises because a transformation may actually decrease the number of errors that may occur. Namely, if a component has an error, it may be compiled to a circuit that has no errors. This is no problem, but it means that the transformations are not completely equivalence-preserving. To account for this, the definition of equality must first ensure that each transformation decreases the possibility for error; and then, assuming no errors are present, ensure that meaning is preserved. Two relations are defined for this purpose.

Definition 18 (Error ordering) Let c and c' be closed C-CSP* terms. Define $c \geq_{err} c'$ iff if c' is semantically well-formed then c is semantically well-formed;

Definition 19 (Observation Equivalence) Let c and c' be closed semantically well-formed C-CSP* terms. Define $c \cong_{obs} c'$ iff

- There exists a successful computation of c iff there exists a successful computation of c'.
- There exists a failing computation of c iff there exists a failing computation of c'.

The testing equivalence we use is the conjunction of these two definitions, abbreviated $c \cong_{o,e} c'$. Although technically this is not an equivalence ($\geq_{err}$ is not symmetric), we write it as such because it will be an equivalence over semantically well-formed components.

Lemma 20 (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 the Bubbling Lemma (38), found in Appendix A. 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. $\square$

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. It gives an elegant theoretical characterization of the arbiter-free speed-independent circuits. This also allows us to remove the second clause from the definition of $\cong_{obs}$, making it easier to show equivalence.

Theorem 21 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 are presented after the general structure of that process is 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 (see [DJ90] for background and references on rewriting).

Definition 22 A rewrite system R/E over C-CSP consists of a finite set of rules of the form $\epsilon_0 \rhd_R \epsilon_1$ and a finite set of equations of the form $\epsilon_0 =_E \epsilon_1$, where the ϵ_i are C-CSP metametavariables, i.e. they are terms which may themselves contain metavariables.

We use the following four relations. $m =_E m'$ indicates that m is equivalent by one of the equational rules to m'. $=_E^*$ is its transitive reflexive closure. $m \Rightarrow_{R/E} m'$ whenever m rewrites in one step modulo the equations E to m'. $\Rightarrow_{R/E}^*$ is its transitive reflexive closure and $m \Rightarrow_{R/E}^N m'$ if m' cannot be further rewritten.

In Section 5, we define the scoping equations, and six rewrite systems $\Rightarrow_1$–$\Rightarrow_6$ by giving six sets of rules $\rhd_1 \ldots \rhd_6$ and a (fixed for all systems) set of scope, commutativity, and associativity equations SCA. Since these equations are fixed for all systems, we will not explicitly mention "SCA" and notate the six rewrite relations $\Rightarrow_i$. 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

As noted above, the changes introduced by certain of the transformations are too great to be equivalence preserving with respect to the above definition of testing equivalence. However, by changing the tests in a similar fashion to the process being tested, the behavior remains the same.

This intuitive notion of equivalence is similar to that used by hardware designers: a circuit using a four-phase handshaking protocol for synchronizing with the outside world would not be expected to behave sensibly if tested using a tester that used a different synchronization protocol. Hence, we should not expect our circuit implementations that synchronize using a handshaking protocol to behave correctly if tested by a tester that uses the handshaking protocol incorrectly or expects to synchronize using the high-level port commands. 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 23 $m_k \cong_{k+1} m_{k+1}$ 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 $(m_k^t \| m_k) \cong_{o,e} (m_{k+1}^t \| m_{k+1})$.

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

We now define a system for incrementally translating C-CSP process specifications to circuit implementations. Our translation roughly follows that of [Bur88], though many changes were necessary to make the translation provably correct That is not to say we found any significant *errors* in their work, only ambiguities. A specification m_0 is compiled using the rewrite systems by applying the six rewrite systems in turn, translating 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.

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

(**SCA : SCOPE 1**)	$\textbf{with } d_1, d_2 \textbf{ do } c \textbf{ end} = \textbf{with } d_1 \textbf{ do with } d_2 \textbf{ do } c \textbf{ end end}$
(**SCA : SCOPE 2**)	$\textbf{with } d_1 \textbf{ do with } d_2 \textbf{ do } c \textbf{ end end} = \textbf{with } d_2 \textbf{ do with } d_1 \textbf{ do } c \textbf{ end end}$
(**SCA : SCOPE 3**)	$C[\textbf{with } d \textbf{ do } c \textbf{ end}] = \textbf{with } d \textbf{ do } C[c] \textbf{ end}$ where declarations d bind nothing in context C
(**SCA : PAR COM**)	$C_1 \| C_2 = C_2 \| C_1$
(**SCA : PAR ASSOC**)	$(C_1 \| C_2) \| C_3 = C_1 \| (C_2 \| C_3)$
(**SCA : SEQ ASSOC**)	$(C_1; C_2); C_3 = C_1; (C_2; C_3)$
(**SCA : GUARD PERM**)	$[e_1 \longrightarrow C_1 [\!] \ldots [\!] e_i \longrightarrow c_i [\!] \ldots [\!] e_j \longrightarrow c_j [\!] \ldots [\!] e_n \longrightarrow c_n] =$ $[e_1 \longrightarrow C_1 [\!] \ldots [\!] e_j \longrightarrow c_j [\!] \ldots [\!] e_i \longrightarrow c_i [\!] \ldots [\!] e_n \longrightarrow c_n]$

Figure 1: SCA Equations

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 Scope, Commutativity and Associativity Equations

For each of the rewrite systems $\Rightarrow_i$, $1 \leq i \leq 6$, we rewrite with respect to the same fixed set of equations, the scope, commutativity and associativity (SCA) equations. These equations provide sound means for moving declarations and commuting and associating parallel and sequential composition. By rewriting with respect to this set of equations, the number and complexity of rewrite rules is reduced. The set of equivalences SCA appears in Table 1. (**SCA : SCOPE 1**) equates a single list of variable and port declarations with a nested declaration of the same variables and ports. (**SCA : SCOPE 2**) swaps two tightly nested scopes. (**SCA : SCOPE 3**) allows the movement of scoping information in and out of parallel, sequencing, guard, and looping commands. The commutativity and associativity equations are self-explanatory.

5.2 Phase 1: Syntax-Directed Rewriting

The first phase separates the original specification into many small processes by transforming each node c of the syntax tree into a separate process of the form $*[[\bar{S} \longrightarrow c; S?]]$ In addition a single "assignment process" is made for each boolean variable to physically isolate its storage location. All assignments synchronize with this process to assign a new value to the variable.

$S! \in \mathcal{P}_a$ and $S? \in \mathcal{P}_p$ are *distinguished* active and passive port names. These port names are used to distinguish those ports added in the translation process from those that are not, ensuring termination of the rewrite system.

Initially, a "start channel" S is added to the term being compiled. Execution of the term begins with a synchronization on this channel. This is a global operation, performed once on the entire module being compiled. After the initialization operation, the phase 1 rewrite rules may be applied. The following translation accomplishes this.

$$(\mathbf{1 : INIT}) \quad m \quad \Rightarrow_1 \quad \textbf{with } S! \textbf{ do } S! \textbf{ end } \| \textbf{with } S? \textbf{ do } *[[\bar{S} \longrightarrow m; S?]] \textbf{ end}$$

The rewrite rules for phase 1 appear in Figure 2. (**1 : SEQ**), (**1 : GUARD**), (**1 : LOOP**), and (**1 : PAR**) rules each introduce a separate process for each part of the expression. Rules (**1 : ASSIGN 1**) and (**1 : ASSIGN 2**) are used to isolate all assignments to a particular variable into a single assignment process. (**1 : ASSIGN 1**) creates an assignment process, a guarded command with two guards: one for assigning **true** ($\uparrow$), and the other for assigning **false** ($\downarrow$). (**1 : ASSIGN 2**) replaces all assignment statements by synchronizations with this new process. Top-down application of these rules produces a set of processes representing the whole syntax tree.

The correctness of each of the rules follows from the observation that although the transformations add new processes, these processes are activated in the same order as the subprocesses of the original. For example, three applications of (**1 : SEQ**) transforms $*[[\bar{S}? \longrightarrow x\uparrow; y\downarrow; z := x \vee y; S?]]$ into $*[[\bar{S}? \longrightarrow S_1!; S_2!; S_3!; S?]] || *[[\bar{S}_1? \longrightarrow x\uparrow; S_1?]] || *[[\bar{S}_2? \longrightarrow y\downarrow; S_2?]] || *[[\bar{S}_3? \longrightarrow z := x \vee y; S_3?]]$ Although the new process consists of four subprocesses executing in parallel, in effect the three subprocesses guarded by $\bar{S}_i?$ execute their bodies in the same sequential order as in the original. The correctness of (**1 : ASSIGN 1**) and (**1 : ASSIGN 2**) follows from the observation that if no errors arose from the assignment to x in the original, then it is not possible for two distinct synchronizations to the assignment cell for x to occur simultaneously in the transformed process. The actual proofs of all the rules consist of rather long inductive arguments.

Lemma 24 If m_0 is a S-CSP module and $m_0 \Rightarrow_1^N m_1$ then $m_0 \cong_1 m_1$.

5.3 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. Each rule eliminates a port scope construct by simultaneously substituting a term that implements the handshaking protocol for each occurrence of the port.

To simplify notation, we let $\mathbf{AHS}(!p, ?p)$ abbreviate the active handshaking protocol

$$!p\uparrow; [?p \longrightarrow \mathbf{skip}]; !p\downarrow; [\neg?p \longrightarrow \mathbf{skip}],$$

and let $\mathbf{PHS}(!p, ?p)$ abbreviate the passive handshaking protocol

$$[!p \longrightarrow \mathbf{skip}]; ?p\uparrow; [\neg!p \longrightarrow ?p\downarrow]$$

The rules appear in Figure 3.

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 with m_2 must use handshaking. Therefore the two will look different to almost all testers. This is one of the main motivations behind the use of transformation equivalence, which this phase preserves.

Lemma 25 If $\Rightarrow_1 m_1$ and $m_1 \Rightarrow_2^N m_2$ then $m_1 \cong_2 m_2$.

A sketch of the proof is as follows. Since the phase uniformly replaces all occurrences of ports with handshaking variables, it only must be demonstrated 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 $m_1^t \Rightarrow_2 m_2^t$). Critical to this is the fact that handshaking variables are *exclusively* used in the handshaking protocols, $\mathbf{AHS}(!p, ?p)$ and $\mathbf{PHS}(!p, ?p)$. This means all uses of these variables will

```
(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 : SEQ)       $*[[\bar{S}? \longrightarrow c_1; c_2; S?]]$   $\triangleright_1$   with $S_1!, S_2!$ do $*[[\bar{S}? \longrightarrow S_1!; S_2!; S?]]$ end$||$
                                                             with $S_1?$ do $*[[\bar{S}_1? \longrightarrow c_1; S_1?]]$ end$||$
                                                             with $S_2?$ do $*[[\bar{S}_2? \longrightarrow c_2; S_2?]]$ end
                    where $c_1 \neq c_3; c_4$, $c_1 \neq S_1!$, and $c_2 \neq S_2!$
(1 : GUARD)     $*[[\bar{S}? \longrightarrow [e_1 \longrightarrow c_1 [] \ldots [] e_n \longrightarrow c_n]; S?]]$
                $\triangleright_1$
                with $S_1!, \ldots, S_n!$ do $*[[\bar{S}? \longrightarrow [e_1 \longrightarrow S_1! [] \ldots [] e_n \longrightarrow S_n!]; S?]]$ end $||$
                with $S_1?$ do $*[[\bar{S}_1? \longrightarrow c_1; S_1?]]$ end
                $|| \ldots ||$
                with $S_n?$ do $*[[\bar{S}_n? \longrightarrow c_n; S_n?]]$ end
                    where $c_1, \ldots, c_n$ are not distinguished active synchronizations
(1 : LOOP)      $*[[\bar{S}? \longrightarrow *[c]; S?]]$   $\triangleright_1$   with $S'!$ do $*[[\bar{S}? \longrightarrow *[S'!]; S?]]$ end $||$
                                                             with $S'?$ do $*[[\bar{S}'? \longrightarrow c; S'?]]$ end
                    where $c$ is not a distinguished active synchronization
(1 : PAR)       $*[[\bar{S}? \longrightarrow (c_1! || \ldots || c_n); S?]]$  $\triangleright_1$
                with $S_1!, \ldots, S_n!$ do $*[[\bar{S}? \longrightarrow (S_1! || \ldots || S_n!); S?]]$ end $||$
                with $S_1?$ do $*[[\bar{S}_1? \longrightarrow c_1; S_1?]]$ end
                $|| \ldots ||$
                with $S_n?$ do $*[[\bar{S}_n? \longrightarrow c_n; S_n?]]$ end
                    where $c_1, \ldots, c_n$ are not distinguished active synchronizations
```

Figure 2: Rewrite rules for Phase 1

$(\mathbf{2} : \mathbf{HS\ 1})$ $\mathbf{with}\ P!\ \mathbf{do}\ C[P!]\ \mathbf{end} \rhd_2$
$\mathbf{with\ w}\ !p, \mathbf{r}\ ?p\ \mathbf{do}\ C[\mathbf{AHS}(!p, ?p)]\ \mathbf{end}$
where C contains no occurrences of $P!$.

$(\mathbf{2} : \mathbf{HS\ 2})$ $\mathbf{with}\ P?\ \mathbf{do}\ C[\bar{P}?][P?]\ \mathbf{end} \rhd_2$
$\mathbf{with\ r}\ !p, \mathbf{w}\ ?p\ \mathbf{do}\ C[!p][\mathbf{PHS}(!p, ?p)]\ \mathbf{end}$
where C contains no occurrences of $P?$ or $\bar{P}$.

Figure 3: Rewrite rules for Phase 2: Handshaking Expansion

$(\mathbf{3} : \mathbf{GUARD\ 1})$ $*[[!s \longrightarrow [e_1 \longrightarrow \mathbf{AHS}(!s_1, ?s_1)] \ldots]e_n \longrightarrow \mathbf{AHS}(!s_n, ?s_n)]; \mathbf{PHS}(!s, ?s)]] \rhd_3$
$*[[!s \wedge e_1 \longrightarrow ?s \uparrow; [\neg !s \longrightarrow \mathbf{AHS}(!s_1, ?s_1); ?s \downarrow]]] \| \ldots \|$
$*[[!s \wedge e_n \longrightarrow ?s \uparrow; [\neg !s \longrightarrow \mathbf{AHS}(!s_n, ?s_n); ?s \downarrow]]]$

$(\mathbf{3} : \mathbf{GUARD\ 2})$ $\mathbf{with\ w}\ ?s\ \mathbf{r}\ !s\ \mathbf{do}\ *[[!s \wedge e \longrightarrow ?s \uparrow; [\neg !s \longrightarrow \mathbf{AHS}(!s', ?s'); ?s \downarrow]]]\mathbf{end} \rhd_3$
$\mathbf{with\ w}\ ?s\ \mathbf{r}\ !s\ \mathbf{do}$
$*[[!s \wedge e_1 \longrightarrow ?s \uparrow; [\neg !s \longrightarrow \mathbf{AHS}(!s', ?s'); ?s \downarrow]]] \| \ldots \|$
$*[[!s \wedge e_n \longrightarrow ?s \uparrow; [\neg !s \longrightarrow \mathbf{AHS}(!s', ?s'); ?s \downarrow]]]\mathbf{end}$
where $e_1 \vee \ldots \vee e_n$ is the result of applying
disjoint-guards to the disjunctive normal form of e.

Figure 4: Rewrite rules for Phase 3.

obey the protocol. From the inspection of the rules for parallelism, (1) directly corresponds to execution of $!p \uparrow$, the initial step of the active protocol. (2) has the exact function as the remainder of the protocol if executed in isolation. All that remains is to show nothing can intervene in the middle of the handshaking protocol to cause its completion to fail. The only possibility is if some other process assigned to the handshaking variables, but if this were the case, this would violate the mutual exclusion on synchronization.

5.4 Phase 3: Guard Simplification

Guard simplification introduces further parallelism by first separating guarded processes into parallel processes, and then further separating each guarded process into its own collection of mutually exclusive guards that furthermore have guards that are conjunctions of literals only. These rules appear in Figure 3.

Since guards are guaranteed to be mutually exclusive, each one can be evaluated by a separate process. $(\mathbf{3} : \mathbf{GUARD\ 1})$ separates the component processes of a selection into distinct parallel processes. In order to prevent a guard e_i from evaluating to true while executing the process guarded by e_j (the process triggered by $!c_i$ could theoretically change the value of some variable and cause e_j to become true), we reshuffle the passive handshake on $!s, ?s$ so that the process activated by $\mathbf{AHS}(!s_i, ?s_i)$ waits for $!s$ to become false, de-activating all guards, before actually executing the process.

$(\mathbf{3} : \mathbf{GUARD\ 2})$ simplifies each guard into disjunctive normal form, strengthens the disjuncts so that they are mutually exclusive, and then further separates this guarded process into a set of concurrently executing processes each guarded by one of the disjuncts. The algorithm used (disjoint-guards) to strengthen the disjuncts so as to make them mutually exclusive is presented in [Bur88].

We conclude this section with the lemma showing rewriting in this phase preserves meaning.

LEMMA 26 If $\Rightarrow_2 m_2$ and $m_2 \Rightarrow_3^N m_3$ then $m_2 \cong_3 m_3$.

The correctness of the two rules depends upon the fact that the guards $e_1, \ldots, e_n$ in the original term are mutually exclusive. (**3 : GUARD 1**) separates each of the guarded terms into a concurrently executing process. The guards are prevented from changing value by preceding the execution of the command with part of the synchronization on $!s, ?s$. Since (**3 : GUARD 2**) simplifies each guarded process into a set of mutually exclusive guarded processes, its correctness follows almost immediately.

5.5 Phase 4: Modularization

The module that results from the phase 3 transformations is a collection of processes executing in parallel. In order to transform this module into a circuit, each of the processes must itself be a module. This is because circuits can write a variable in only one location (the gate that has that named wire as output), so all write scopes of variables must be localized. In this phase, each process that is not already a module is transformed into one. 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.

LEMMA 27 If $\Rightarrow_3 m_3$ and $m_3 \Rightarrow_4^N m_4$ then $m_3 \cong_4 m_4$.

The other important result is after this phase of the translation, all processes that comprise the module are themselves modules.

LEMMA 28 If $m_0 \Rightarrow_{1234}^N m_4$ then $m_4 = m_1' || \ldots || m_n'$, where each m_i' is in turn a module.

These two lemmas are proved by showing a strong correspondence exists between the computations before and after the translation process; critical is the fact that all handshaking synchronizations are "pure," meaning both active and passive sides step through the protocol without intervening activity.

5.6 Phase 5:Reshuffling

Before producing circuitry for each module that results from modularization, we reshuffle some of the handshake protocols. The purpose of this action is to make modules that are more easily implemented in circuitry. All of the reshufflings involve interleaving the final passive handshake with the preceding handshake.

Upon entering this phase, each module is of the form $[!s \longrightarrow c; \mathbf{PHS}(!s, ?s)]$ for some c (ignoring declarations). The hardware implementation is simpler if the initial $[!s \longrightarrow \mathbf{skip}]$ of the passive protocol is eliminated, and if some of the response $?s \uparrow; [\neg !s \longrightarrow \mathbf{skip}]; ?s \downarrow$ is interleaved with the execution of c. Although each type of module requires a different form of reshuffling, the general

```
(4 : MOD 1)  with w $!p$, r $?p$ do $C$[AHS($!p, ?p$)]...[AHS($!p, ?p$)] end
             $\rhd_4$
             with r $?p$ do
                 with r $!p_1, \ldots,$ r $!p_n$ do
                     with w $!p$ do $*[!p := !p_1 \vee \ldots \vee !p_n]$ end ||
                     with w $?p_1$ do $*[?p_1 := !p_1$ C $?p]$ end ||
                         ⋮
                     with w $?p_n$ do $*[?p_n := !p_n$ C $?p]$ end
                 end ||
                 $C$[with w $!p_1$ r $?p_1$ do AHS($!p_1, ?p_1$) end]
                         ⋮
                 [with w $!p_n$ r $?p_n$ do AHS($!p_n, ?p_n$) end]
             end
             where $n > 1$, each hole $\bullet_i$ occurs at most once in $C$, and
             $?p$, $!p$ do not occur in $C$
(4 : MOD 2)  with r $!p$, w $?p$ do
                 $C[!p]$[PHS($!p, ?p$)]...[PHS($!p, ?p$)]
             end
             $\rhd_4$
             with r $?p_1, \ldots,$ r $?p_n$, w $?p$ do $*[?p := ?p_1 \vee \ldots \vee ?p_n]$ end ||
             with r $!p$ do
                 $C[!p]$
                 [with w $?p_1$ do PHS($!p, ?p_1$) end]
                     ⋮
                 [with w $?p_n$ do PHS($!p, ?p_n$) end]
             end
             where $n > 1$, each hole $\bullet_i$ for $i > 1$ occurs at most once in $C$, and
             $?p$, $!p$ do not occur in $C$
```

Figure 5: Rewrite rules for Phase 4: Modularization

principle is that a reshuffling may occur when the active communication $\mathbf{AHS}(!p, ?p)$ corresponding to the passive communication $\mathbf{PHS}(!p, ?p)$ has not yet had anything reshuffled into it. The rules appear in Figure 6.

The following lemma is the key lemma, showing a passive handshake may be reshuffled, provided its corresponding active handshake is pure. It allows us to prove the correctness of this phase, by applying the lemma in a bottom-up order to the tree of processes induced by apllying phase 1 rules to the original syntax tree.

LEMMA 29 (RESHUFFLING PRINCIPLE)

$$\begin{array}{ll} C' & [\mathbf{with\ r}\ ?d,\ \mathbf{w}\ !d\ \mathbf{do}\ C[\mathbf{AHS}(!d, ?d)]\mathbf{end}] \\ & [\mathbf{with\ w}\ ?d,\ r\ !d\ \mathbf{do}\ [!d \longrightarrow \mathbf{skip}]; c_0; c_1; c_2; \mathbf{PHS}(!d, ?d)\ \mathbf{end}] \\ = & \\ C' & [\mathbf{with\ r}\ ?d,\ \mathbf{w}\ !d\ \mathbf{do}\ C[\mathbf{AHS}(!d, ?d)]\mathbf{end}] \\ & [\mathbf{with\ w}\ ?d,\ \mathbf{r}\ !d\ \mathbf{do}\ [!d \longrightarrow \mathbf{skip}]; c_0; ?d\uparrow; c_1; [\neg !d \longrightarrow \mathbf{skip}]; c_2; ?d\downarrow\ \mathbf{end}] \end{array}$$

provided there is no occurrences of $?d$ or $!d$ in C, c_0, c_1, or c_2.

LEMMA 30 If $\Rightarrow_4 m_4$ and $m_4 \Rightarrow_5^N m_5$, then $m_4 \cong_5 m_5$.

5.7 Phase 6: Final Compilation into circuits

The last part of the translation takes the individual processes representing atomic assignment, sequencing, guard, active and passive communication, loop, skip, and parallel execution, and transforms each into a circuit representation. The rules appear in Figure 7.

Each circuit process below executes the same handshaking sequence as the reshuffled versions they come from, for each case that is not too difficult to establish. The problem, however, is *robustness*: the reshuffled processes execute actions in a certain order, and only in that order. Ordering is difficult to impose on circuits, and many of the circuits in fact will function differently than the reshuffled processes if placed in an evironment that does not obey the handshake protocol. For instance, if an active circuit, implemented as just two wires, has its passive input $?a$ set high before it has set output $!a$ high (a violation of handshake protocol), the circuit will set $?d$ high. However, the reshuffled process is guarded by $!d$, so will ignore any value of $?a$ until $!d$ is high. Thus, important to correctness of each process is the assumption that all neighbor processes this process interacts with also obey the handshake protocol. This is summed up in the following lemma.

LEMMA 31 Given $\Rightarrow_6 m$ and $\Rightarrow_6 m^t$, the handshake protocol is never violated in $m \| m^t$. In particular, for any pair of handshake variables $!d, ?d$ occurring in m and m^t, it is not the case that, in some computation sequence $\langle m \| m^t, \iota(m \| m^t) \rangle$,

1. $!d$ is set to **false** when $?d =$ **false**,
2. $!d$ is set to **true** when $?d =$ **true**,
3. $?d$ is set to **false** when $!d =$ **true**,
4. $?$ is set to **true** when $!d =$ **false**,

Proof: The proof proceeds by counterexample, suppose there in fact was a violation of protocol, then there must have been an earliest step number in the computation at which a violation of protocol occurred. But, by case analysis on each of the circuit processes, none of them could have

```
(5 : SEQ)   with  r $!s$, r $?s_1$, r $?s_2$, w $?s$, r $!s_1$, w $!s_2$ do
                  $*[[!s \longrightarrow$ AHS$(!s_1, ?s_1)$; AHS$(!s_2, ?s_2)$; PHS$(!s, ?s)]]$
            end
             $\triangleright_5$
            with  r $!s$, r $?s_1$, r $?s_2$, w $?s$, r $!s_1$, w $!s_2$ do
                  $*[[!s \longrightarrow$ skip$]; !s_1 \uparrow; [?s_1 \longrightarrow$ skip$], ?s \uparrow; [\neg !s \longrightarrow$ skip$];$
                   $!s_1 \downarrow; [\neg ?s_1 \longrightarrow$ skip$]; !s_2 \uparrow; [?s_2 \longrightarrow$ skip$]; !s_2 \downarrow; [\neg ?s_2 \longrightarrow$ skip$]; ?s_2 \downarrow]$
            end
(5 : PAR)   with  r $!s$, r $?s_1, \ldots, ?s_n$, w $?s$, w $!s_1 \ldots, !s_n$ do
                  $*[[!s \longrightarrow ($AHS$(!s_1, ?s_1) \| \ldots \|$AHS$(!s_n, ?s_n))$; PHS$(!s, ?s)]]$
            end
             $\triangleright_5$
            with  r $!s$, r $?s'$, w $?s$, w $!s'$ do
                  $*[[!s \longrightarrow$ skip$]; !s' \uparrow; [?s' \longrightarrow$ skip$]; ?s \uparrow;$
                   $[\neg !s \longrightarrow$ skip$]; !s' \downarrow; [\neg ?s' \longrightarrow$ skip$]; ?s \downarrow]$
           end$\|$
            with  r $!s'$, w $!s_1, \ldots, !s_n$ do $*[!s_1 := !s'] \| \ldots \| * [!s_n := !s']$end$\|$
            with  r $?s_1 \ldots, ?s_n$, w $?s'$ do $?s' := ?s_1 \wedge \ldots ?s_n$end
(5 : ACT)   with  r $!s$, r $?a$, w $?s$, w $!a$ do
                  $*[[!s \longrightarrow$ AHS$(!a, ?a)$; PHS$(!s, ?s)]]$
            end
             $\triangleright_5$
            with  r $!s$, r $?a$, w $?s$, w $!a$ do
                  $*[[!s \longrightarrow$ skip$]; !a \uparrow; [?a \longrightarrow$ skip$]; ?s \uparrow;$
                   $[\neg !s \longrightarrow$ skip$]; !a \downarrow; [\neg ?a \longrightarrow$ skip$]; ?s \downarrow]$
            end
(5 : PASS)  with  r $!s$, r $!p$, w $?s$, w $?p$ do
                  $*[[!s \longrightarrow$ PHS$(!p, ?p)$; PHS$(!s, ?s)]]$
            end
             $\triangleright_5$
            with  r $!s$, r $!p$, w $?s$, w $?p$ do
                  $*[[!s \wedge !p \longrightarrow$ skip$]; (?p \uparrow \| ?s \uparrow);$
                   $[\neg !p \wedge \neg !s \longrightarrow$ skip$]; (?p \downarrow \| ?s \downarrow)]$
            end
```

Figure 6: Rewrite rules for Phase 5: Reshuffling

initiated the first violation. All cases are straightforward except the case of the active circuit. If there are multiple active processes, one of them may be synchronizing with a passive process while the other active processes are idle. This means the idle active processes could in theory receive $?a$ signals from the passive process, as alluded to in the example above. However, the C-element inserted by modularization guarantees this will never be the case. The only active process that receives $?a$ is the one that initiated the synchronization. □

With this lemma we may prove correctness of this phase of translation.

LEMMA 32 If $\Rightarrow_5 m_5$ and $m_5 \Rightarrow_6^N m_6$, then $m_5 \cong_6 m_6$.

5.8 Summarizing the Translation Process

We have presented six rewriting systems for compiling S-CSP circuit specifications into H-CSP circuit realizations. We now justify that compilation of a specification always produces a circuit, and meaning is preserved. First, we state without proof that the compilation process is total.

LEMMA 33 If $m \in$ S-CSP, then $m \Rightarrow_{1-6} m_6$ for some $m_6 \in$ H-CSP.

The correctness of the entire transformation process follows immediately from the correctness of each of the phases.

THEOREM 34 (CORRECTNESS) For all $m \in$ S-CSP, if $m \Rightarrow_{1-6} m_6$ then for any closing testing module $m^t \in$ S-CSP* such that $m^t \Rightarrow_{1-6} m_6^t$, $(m \| m^t) \cong_{o,\epsilon} (m_6 \| m_6^t)$.

Proof: The proof follows immediately from Lemmas 24, 25, 26, 27, 30, and 32. □

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.

6.1 Related Work

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 a prelimary report [WBB92], Weber, Bloom, and Brown define a process language Joy and its compilation to asynchronous circuitry. In contrast to C-CSP, Joy has a number of additional syntactic restrictions including the restriction that no processes may share variables or passive port names. This restriction is critical to the proof of correctness of the compilation of Joy specifications into circuits. Additionally, because handshaking is required to read variables, the resulting circuits are slower than those we generate. Another minor difference is that they make no fairness assumption. A benefit of their method over ours is that the transformation process guarantees isochronic forks to be isolated in small parts of the circuit. The primary difference, though, is their use of a bisimulation ordering rather than the testing ordering we use. Bisimulation is a stronger equivalence than testing (fewer things are related by bisimulation). Our choice to use testing rather than bisimulation was based on our belief that bisimulation is too strong a relation to use if we want to incorporate (and prove correct) optimizations into our translation.

(6 : ASSIGN) **with** $\mathbf{w}\ x,\ \mathbf{w}\ ?s_0, \mathbf{w}\ ?s_1, \mathbf{r}\ !s_1, \mathbf{r}\ !s_0$ **do**
$*[[!s_1 \longrightarrow x\uparrow;\ \mathbf{PHS}(s_1) [\!] !s_0 \longrightarrow x\downarrow;\ \mathbf{PHS}(s_0)]]$ **end**

$\rhd_6$

with $\mathbf{w}\ x,\ \mathbf{w}\ ?s_0, \mathbf{w}\ ?s_1, \mathbf{r}\ !s_1, \mathbf{r}\ !s_0$ **do**
$*[x :=!s_1\ \mathbf{C}\ \neg !s_0] \| *[?s_1 :=!s_1 \wedge x] \| *[?s_0 :=!s_0 \wedge \neg x]$ **end**

(6 : SEQ) **with** $\mathbf{r}\ !s, \mathbf{r}\ ?s_1, \mathbf{r}\ ?s_2, \mathbf{w}\ ?s, \mathbf{w}\ !s_1, \mathbf{w}\ !s_2$ **do**
$*[[!s \longrightarrow \mathbf{skip}]; !s_1\uparrow; [?s_1 \longrightarrow \mathbf{skip}], ?s\uparrow; [\neg !s \longrightarrow \mathbf{skip}];$
$!s_1\downarrow; [\neg ?s_1 \longrightarrow \mathbf{skip}]; \mathbf{AHS}(!s_2, ?s_2); ?s\downarrow]$ **end**

$\rhd_6$

with $\mathbf{w}\ x, \mathbf{r}\ x, \mathbf{r}\ !s, \mathbf{r}\ ?s_1, \mathbf{r}\ ?s_2, \mathbf{w}\ ?s, \mathbf{w}\ !s_1, \mathbf{w}\ !s_2$ **do**
$*[?s_1 := s] \| *[x :=?s_1\ \mathbf{C}\ \neg ?s_2] \| *[?s := x \vee ?s_2] \| *[!s_2 := x \wedge \neg ?s_1])$ **end**

(6 : GUARD) **with** $\mathbf{r}\ !s, \mathbf{w}\ ?s, \mathbf{r}\ ?s_1, \mathbf{w}\ !s_1$ **do**
$*[[!s \wedge e \longrightarrow ?s\uparrow; [\neg !s \longrightarrow \mathbf{AHS}(!s_1, ?s_1); ?s\downarrow]]]$ **end**

$\rhd_6$

with $\mathbf{r}\ !s, \mathbf{w}\ ?s, \mathbf{r}\ ?s_1, \mathbf{w}\ !s_1, \mathbf{r}\ x, \mathbf{w}\ x, \mathbf{r}\ t_1, \mathbf{w}\ t_1, \mathbf{r}\ t_2, \mathbf{w}\ t_2$ **do**
$*[t_1 := s \wedge e] \| *[t_2 :=?s_1 \wedge \neg !s] \| *[x := t_1\ \mathbf{C}\ \neg t_2] \|$
$*[?s := x \vee ?s_1] \| *[!s_1 := x \wedge \neg !d]$ **end**

(6 : ACT/PAR) **with** $\mathbf{r}\ !s, \mathbf{r}\ ?a, \mathbf{w}\ ?s, \mathbf{w}\ !a$ **do**
$*[[!s \longrightarrow \mathbf{skip}]; !a\uparrow; [?a \longrightarrow \mathbf{skip}]; ?s\uparrow;$
$[\neg !s \longrightarrow \mathbf{skip}]; !a\downarrow; [\neg ?a \longrightarrow \mathbf{skip}]; ?s\downarrow]$ **end**

$\rhd_6$

with $\mathbf{r}\ !s\ \mathbf{r}\ ?a\ \mathbf{w}\ ?s\ \mathbf{w}\ !a$ **do**
$*[!a :=!s] \| *[?s :=?a]$ **end**

(6 : PASS) **with** $\mathbf{r}\ !s, \mathbf{r}\ !p, \mathbf{w}\ ?s, \mathbf{w}\ ?p$ **do**
$*[[!s \wedge !p \longrightarrow \mathbf{skip}]; (?p\uparrow \| ?s\uparrow);$
$[\neg !p \wedge \neg !s \longrightarrow \mathbf{skip}]; (?p\downarrow \| ?s\downarrow)]$ **end**

$\rhd_6$

with $\mathbf{w}\ x, \mathbf{r}\ x, \mathbf{r}\ !s, \mathbf{r}\ !p, \mathbf{w}\ ?s, \mathbf{w}\ ?p$ **do**
$*[x :=!s\ \mathbf{C}\ !p] \| *[?s := x] \| *[?p := x]$ **end**

(6 : LOOP) **with** $\mathbf{r}\ !s, \mathbf{r}\ ?a, \mathbf{w}\ ?s, \mathbf{w}\ !a$ **do**
$*[[!s \longrightarrow \mathbf{skip}]; *[\mathbf{AHS}(!a, ?a)]; \mathbf{PHS}(!s, ?s)]$ **end**

$\rhd_6$

with $\mathbf{r}\ !s, \mathbf{r}\ ?a, \mathbf{w}\ ?s, \mathbf{w}\ !a, \mathbf{r}\ x, \mathbf{w}\ x$ **do**
$*[x :=!s\ \mathbf{C}\ !s] \| *[!c := x \wedge \neg ?c]$ **end**

(6 : SKIP) **with** $\mathbf{r}\ !s, \mathbf{w}\ ?s$ **do** $*[[!s \longrightarrow \mathbf{skip}]; \mathbf{skip}; \mathbf{PHS}(!s, ?s)]$ **end**

$\rhd_6$

with $\mathbf{r}\ !s, \mathbf{w}\ ?s$ **do** $*[?s :=!s]$ **end**

Figure 7: Rewrite rules for Phase 6: Circuit Generation

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, but in principle allows concurrent read/write via an arbiter, something we disallow. The compilation process goes through an intermediate language, *handshake circuits*. He uses trace equivalence, a weaker equivalence than testing, but has no notion of fairness. His presentation is very thorough, and addresses many other important issues such as initialization and optimization. Unfortunately, the correctness proof stops at the handshake circuit level and does not exist for the circuit level.

References

[BK84] J.A. Bergstra and J.W. Klop. Process Algebra for Synchronous Communication. *Information and Control*, 60:109–137, 1984.

[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.

[Bur88] Steven M. Burns. Automated compilation of concurrent programs into self-timed circuits. Technical Report Caltech-CS-TR-88-2, California Institute of Technology, 1988.

[DJ90] N. Dershowitz and J.-P. Jouannaud. Rewriting systems. In *Handbook of theoretical computer science*. MIT/Elsevier, 1990.

[FFK87] M. Felleisen, D. Friedman, and E. Kohlbecker. A syntactic theory of sequential control. *Theoretical Computer Science*, 52:205–237, 1987.

[Hen83] Matthew Hennessy. Synchronous and asynchronous experiments on processes. *Inform. and Control*, 59:36–83, 1983.

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

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

[Mar85] Alain J. Martin. The design of a self-timed circuit for distributed mutual exclusion. In *1985 Chapel Nill Conference on VLSI*, pages 245–260, 1985.

[Mar86] Alain J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1(4):226–234, 1986.

[Mar90a] Alain J. Martin. The limitations to delay-insensitivity in asynchronous circuits. In William J. Dally, editor, *Sixth MIT Conference on Advanced Research in VLSI*, pages 263–278. MIT Press, 1990.

[Mar90b] 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.

[Mar90c] Alain J. Martin. Synthesis of asynchronous VLSI circuits. In J. Straunstrup, editor, *Formal Methods for VLSI Design*, pages 237–283. North-Holland, 1990.

[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.

[Mil89] Robin Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[NH83] R. De Nicola and M.C.B. Hennessy. Testing Equivalences for Processes. *Theoretical Computer Science*, 34:83–133, 1983.

[Plo77] G. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.

[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.

[vB91] C. H. van Berkel. Beware the isochronic fork. Nat. Lab. Unclassified Report UR 003/91, Philips Research Lab., Eindhoven, The Netherlands, 1991.

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

[vBKR+91] Kees van Berkel, Joep Kessels, Marly Roncken, Ronald Saeijs, and Frits Schalij. The VLSI-programming language Tangram and its translation into handshake circuits. In *Proceedings of the European Design Automation Conference*, pages 384–389, 1991.

[vBNRS88] C. H. (Kees) van Berkel, Cees Niessen, Martin Rem, and Ronald W. J. J. Saeijs. VLSI programming and silicon compilation. In *Proceedings ICCD '88*, pages 150–166, Rye Brook, New York, 1988. IEEE Computer Society Press.

[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.

A Fairness and Determinicity

In this appendix we present details of the definition of fairness and the Lemmas leading up to the proof of determinicity, Lemma 20.

Definition 35

- A finite computation path

$$\langle c_0, \sigma_0\rangle \rightarrow \langle c_1, \sigma_1\rangle \rightarrow \ldots \rightarrow \langle c_n, \sigma_n\rangle$$

is *fair* iff $\langle c_n, \sigma_n\rangle \not\rightarrow \langle c_{n+1}, \sigma_{n+1}\rangle$ for any c_{n+1}, σ_{n+1}.

- An infinite computation path

$$\langle c_0, \sigma_0\rangle \rightarrow \langle c_1, \sigma_1\rangle \rightarrow \ldots \rightarrow \langle c_i, \sigma_i\rangle \rightarrow \ldots,$$

  is *fair* iff it is not unfair. It is *unfair* iff

$$\begin{array}{ll}
\exists c_a, c_b. & \exists\{c_{1i}, c_{2i}, c'_{1i}, c'_{2i} \mid i \in \mathbf{Nat}\}.\forall i. \\
& c_i = R_i[c_a][c_b][c_{1i}][c_{2i}] \\
& \wedge \\
& \langle R_i[c_a][c_b][c_{1i}][c_{2i}], \sigma_i\rangle \rightarrow \langle R_i[c_a][c_b][c'_{1i}][c'_{2i}], \sigma_{i+1}\rangle \text{ (never executed)} \\
& \wedge \\
& R_i[\bullet_1][\bullet_2][c'_{1i}][c'_{2i}] = R_{i+1}[\bullet_1][\bullet_2][c_{1(i+1)}][c_{2(i+1)}] \\
& \wedge \\
& \exists\sigma', c'_a, c'_b.\langle R_i[c_a][c_b][c_{1i}][c_{2i}], \sigma_i\rangle \rightarrow \langle R_i[c'_a][c'_b][c_{1i}][c_{2i}], \sigma'\rangle \text{ (continuously enabled)}
\end{array}$$

The most important consequence of fairness for this system is progress. That is, once a reduction step is enabled, it will remain enabled and eventually be executed.

Lemma 36 (Progress) Given semantically well-formed $\langle c_0, \sigma_0\rangle$ and any potentially infinite fair computation sequence

$$\langle c_0, \sigma_0\rangle \rightarrow \langle c_1, \sigma_1\rangle \rightarrow \ldots \rightarrow \langle c_i, \sigma_i\rangle \rightarrow \ldots,$$

if $c_0 = R_0[c_a][c_b][c_{10}][c_{20}]$ and

$$\langle R_0[c_a][c_b][c_{10}][c_{20}], \sigma_0\rangle \rightarrow \langle R_0[c'_a][c'_b][c_{10}][c_{20}], \sigma'_1\rangle$$

then there exists some n such that

$$\begin{array}{l}
\exists\{c_{1i}, c_{2i}, c'_{1i}, c'_{2i} \mid i \in \mathbf{Nat} \wedge i < n\}.\forall i < n. \\
c_{i+1} = R_{i+1}[c_a][c_b][c_{1(i+1)}][c_{2(i+1)}] \\
\wedge \\
\langle R_i[c_a][c_b][c_{1i}][c_{2i}], \sigma_i\rangle \rightarrow \langle R_i[c_a][c_b][c'_{1i}][c'_{2i}], \sigma_{i+1}\rangle \text{ (not executed for } n \text{ steps)} \\
\wedge \\
R_i[\bullet_1][\bullet_2][c'_{1i}][c'_{2i}] = R_{i+1}[\bullet_1][\bullet_2][c_{1(i+1)}][c_{2(i+1)}] \\
\wedge \\
\exists\sigma', c'_a, c'_b.\langle R_i[c_a][c_b][c_{1i}][c_{2i}], \sigma_i\rangle \rightarrow \langle R_i[c'_a][c'_b][c_{1i}][c_{2i}], \sigma'\rangle \text{ (continuously enabled)} \\
\wedge \\
\langle R_n[c_a][c_b][c_{1n}][c_{2n}], \sigma_n\rangle \rightarrow \langle R_n[c'_a][c'_b][c_{1n}][c_{2n}], \sigma_{n+1}\rangle \text{ (executed on } n+1\text{-st step)}
\end{array}$$

Proof: This is almost a direct consequence of fairness. If $R_i[c_a][c_b]$ were always enabled to reduce, the result would follow by fairness. So, we must demonstrate that no enabled reduction is disabled. For the rules (Assignment), (Sequencing), (Repetition) and (Parallelism)(3), enabled redices can easily be shown to stay enabled by inspection of the other rules. For the other cases, (Selection), and (Parallelism)(1)&(2), the only possibility of a redex becoming disabled is the fact that some other intervening step changes the value of a critical boolean expression e, but this is disallowed by error rule 1. □

Lemma 37 (Strong Diamond) Given well-formed $\langle R[c_a][c_b][c_c][c_d], \sigma\rangle$, a strong diamond property holds. Namely, if

$$\langle R[c_a][c_b][c_c][c_d], \sigma\rangle \rightarrow \langle R[c'_a][c'_b][c_c][c_d], \sigma[v_{01} = e_{01}]\rangle$$

and

$$\langle R[c_a][c_b][c_c][c_d], \sigma\rangle \rightarrow \langle R[c_a][c_b][c'_c][c'_d], \sigma[v_{10} = e_{10}]\rangle,$$

then $v_{10} \neq v_{01}$ and

$$\langle R[c'_a][c'_b][c_c][c_d], \sigma[v_{01} = e_{01}]\rangle \rightarrow \langle R[c'_a][c'_b][c'_c][c'_d], \sigma[v_{01} = e_{01}, v_{10} = e_{10}]\rangle$$

and

$$\langle R[c_a][c_b][c'_c][c'_d], \sigma[v_{10} = e_{10}]\rangle \rightarrow \langle R[c'_a][c'_b][c'_c][c'_d], \sigma[v_{01} = e_{01}, v_{10} = e_{10}]\rangle,$$

where to admit the possibility that σ was in fact not changed by one or both steps, v_{01}, and/or v_{10} may not be in the domain of σ, in which case no modification of σ takes place.

Proof: The proof proceeds by analysis of the 28 different cases the two initial rules could have been (there are 7 single-step rules, both steps may have been a different use of the same rule). □

The next Lemma uses the Strong Diamond Lemma (Lemma 37) to show any enabled step executed sometime in the future can be bubbled up to occur as the next step, without altering the remaining computation.

LEMMA 38 (BUBBLING) Given semantically well-formed $\langle c_0, \sigma_0\rangle$ and a potentially infinite fair computation sequence

$$\langle c_0, \sigma_0\rangle \rightarrow \langle c_1, \sigma_1\rangle \rightarrow \ldots \rightarrow \langle c_i, \sigma_i\rangle \rightarrow \ldots,$$

with $c_0 = R_0[c_a][c_b][c_{10}][c_{20}]$ and

$$\langle R_0[c_a][c_b][c_{10}][c_{20}], \sigma_0\rangle \rightarrow \langle R_0[c'_a][c'_b][c_{10}][c_{20}], \sigma'_1\rangle$$

i.e. $R_0[c_a][c_b]$ is enabled to reduce, and furthermore, there exists some n such that

$$\begin{array}{l}
\exists\{c_{1i}, c_{2i}, c'_{1i}, c'_{2i} \mid i \in \mathbf{Nat} \wedge i < n\}.\forall i < n.\\
c_{i+1} = R_{i+1}[c_a][c_b][c_{1(i+1)}][c_{2(i+1)}]\\
\wedge\\
\langle R_i[c_a][c_b][c_{1i}][c_{2i}], \sigma_i\rangle \rightarrow \langle R_i[c_a][c_b][c'_{1i}][c'_{2i}], \sigma_{i+1}\rangle \text{ (not executed for } n \text{ steps)}\\
\wedge\\
R_i[\bullet_1][\bullet_2][c'_{1i}][c'_{2i}] = R_{i+1}[\bullet_1][\bullet_2][c_{1(i+1)}][c_{2(i+1)}]\\
\wedge\\
\langle R_n[c_a][c_b][c_{1n}][c_{2n}], \sigma_n\rangle \rightarrow \langle R_n[c'_a][c'_b][c_{1n}][c_{2n}], \sigma_{n+1}\rangle\text{(executed on } n+1\text{-st step)}
\end{array}$$

then, the reduction of c_a/c_b can be bubbled up to be the first step of computation without changing the end result, namely,

$$\langle R_0[c_a][c_b][c_{10}][c_{20}], \sigma_0\rangle \rightarrow \langle R_0[c'_a][c'_b][c_{10}][c_{20}], \sigma'_1\rangle \xrightarrow{*} \langle R_n[c'_a][c'_b][c_{1n}][c_{2n}], \sigma_{n+1}\rangle.$$

Proof: Proceed by induction on n. For the case $n = 1$, the desired result is exactly the Strong Diamond Property (37). Assume the result is true for values smaller than n, show the result holds for n. Consider the computation starting at the second step of the original computation,

$$\langle R_0[c_a][c_b][c'_{10}][c'_{20}], \sigma_1\rangle$$

By the continual enabledness of redices from the Progress Lemma (36),

$$\langle R_0[c_a][c_b][c'_{10}][c'_{20}], \sigma_1\rangle \rightarrow \langle R_0[c'_a][c'_b][c'_{10}][c'_{20}], \sigma'_2\rangle$$

By the induction hypothesis, we then have

$$(1) \quad \langle R_0[c'_a][c'_b][c'_{10}][c'_{20}], \sigma'_2\rangle \xrightarrow{*} \langle R_n[c'_a][c'_b][c_{1n}][c_{2n}], \sigma_{n+1}\rangle.$$

So, by the Strong Diamond Property (37), the first two steps

$$\langle R_0[c_a][c_b][c_{10}][c_{20}], \sigma_0\rangle \rightarrow \langle R_0[c_a][c_b][c'_{10}][c'_{20}], \sigma_1\rangle \rightarrow \langle R_0[c'_a][c'_b][c'_{10}][c'_{20}], \sigma'_2\rangle$$

may be swapped to give

$$\langle R_0[c_a][c_b][c_{10}][c_{20}], \sigma_0\rangle \rightarrow \langle R_0[c'_a][c'_b][c_{10}][c_{20}], \sigma'_1\rangle \rightarrow \langle R_0[c'_a][c'_b][c'_{10}][c'_{20}], \sigma'_2\rangle,$$

So by (1) above, the proof is complete. □