Formal Methods in System Design, 9, 1–73 (1995)
© 1995 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Correct Compilation of Specifications to Deterministic Asynchronous Circuits

SCOTT F. SMITH AND AMY E. ZWARICO {scott, amy}@cs.jhu.edu
Department of Computer Science, The Johns Hopkins University, Baltimore, MD 21218 USA

Abstract. Powerful methods have been developed by A. Martin and others whereby asynchronous circuits may be automatically constructed by starting from high-level specifications and incrementally transforming them into asynchronous circuits. In this paper we make the informal arguments for the correctness of this compilation process mathematically rigorous. With rigorously justified transformations, specifications may be translated into circuits that provably meet their specification. A full proof of the correctness of the circuit compiler is given. Other results of independent interest include: the process model takes fairness of gates into account, hazard-freeness is formally defined, and all hazard-free circuits constructed solely of and, or, not gates and C elements are proven to behave deterministically to any outside observer. A novel notion of equivalence is used to justify the correctness of the compiler.

Keywords: Asynchronous circuits, speed-independent circuits, verification, concurrency, compilation

1. Introduction

A large number of research projects have developed methods for the automatic synthesis of asynchronous circuits from high-level specifications [17], [18], [28], [2], [20]. 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 the above projects excepting [20] use the same basic methodology, described as follows. The circuit is specified in a CSP-like language [13] as a set of concurrently executing processes that can communicate via fixed channels. A specification is constructed from simple programming language constructs that include variables x and assignments $x := a$, conditional branching, looping, parallel execution, and sequencing. The specification then undergoes a series of transformations that in the end results in an asynchronous circuit.

Our work is based on the asynchronous design method of Martin *et al.* [17], [18], [14], [15], [19]. Burns and Martin have described and implemented a *circuit compiler* [6], [5] that uses this method to automatically translate specifications into circuits. This paper is concerned with rigorously establishing the correctness of asynchronous circuit compilers. There have been multiple efforts in this area, including [25], [29], [27]. This paper is a complete presentation of preliminary work described in [24], [25]. We first present an overview of our method, and conclude this section by contrasting with related work.

1.1. Our method

The results in this paper are based on a mathematical circuit model described as follows.

Speed-Independent We work under the assumption 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 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. There has been some debate about the appropriateness of the isochronic forks assumption [26], [16].

Fairness Unlike most 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). Since gates are in practice fair, this is an important assumption.

Hazards If a gate is enabled to switch and one of its input changes to disable the gate from firing, a *hazard* results. Constructed circuits should be provably hazard-free.

Weaknesses of this model include the isochronic forks assumption, and allowance of arbitrary fan-in and fan-out. The model further assumes each wire has only one value, and there are no values allowed between 0 and 1.

We define our specification language C-CSP (Circuit CSP) in Section 2. C-CSP is modeled loosely on Hoare's CSP and Occam. C-CSP is the only language we use, it describes specifications and circuits and all intermediate forms between the two. In this sense our approach is different from the literature. Circuits are collections of atomic processes running in parallel performing simple boolean assignments, and can be expressed in a sublanguage of C-CSP.

We give C-CSP meaning via an operational semantics, in Section 3. Defining the semantics presents several challenges. First, we need to define executions so only *fair* computations are allowed. Real asynchronous circuits do not contain gates that starve, so this assumption is critical in accurately modeling asynchronous circuits. Second, the semantics provides a mechanism for reasoning about potential mutual exclusion problems. This differs from other researchers [29], [27] who prevent all violations of mutual exclusion from occurring by placing syntactic restrictions on the sharing of variables. One such restriction others impose is to disallow two processes executing in parallel to share a variable. We believe such restrictions are too drastic, because many programs, realizable as circuits, have specifications in which concurrently executing processes share variables but nonetheless have no simultaneous assignment to the variable. For example, a CPU may contain a register that may be written by several concurrently executing processes, but in a mutually exclusive manner. This CPU is implementable if we can show that at no time during its execution is the register simultaneously assigned. If such direct variable access is disallowed, a slower circuit will by necessity be constructed. We indicate violations of mutual exclusion by having the executing process enter a special **ERROR** state. The disadvantage of our approach is the need to *prove* that no **ERROR** states arise during execution, for each specification.

Key to proving the correctness of the transformation process is defining a reasonable notion of "equivalence" between processes. Many useful notions of equivalence have been defined. Trace equivalence [27], [8], [9] and bisimulation equivalence [29] have both been effectively used for reasoning about asynchronous circuits. Our approach is based on a third variety, *testing equivalence.* Two processes are testing equivalent if they pass the same set of infinitely many tests. Testing equivalence can thus be viewed as a notion of passing a "complete" test suite for a device. We define testing equivalence in Section 4, and formalize what it means for transformations to be semantics-preserving.

One key property we prove for our language is all semantically well-formed (hazard-free) processes are *observably deterministic.* This means multiple actions can occur in parallel in an undeterminted order, but to an observer the process is deterministic because the output sequence resulting from a given input sequence is always the same. Observable determinism holds because the language has no explicit means for nondetermisism via arbiters, and if nondeterminism could be observed it would mean the circuit sometimes would have to exhibit hazards. From this a fundamental Corrolary concerning the behavior of asynchronous circuits may be derived: hazard-free circuits constructed solely of and, or, not gates and C-elements are all observably deterministic. The difficulty of establishing correctness is lessened in the presence of the observable determinism property, for it suffices to consider any particular run of the circuit rather than all possible runs.

The compilation process proceeds in five phases, as described in Section 5. Breaking down the process into a number of phases means the individual transformations are smaller, and also simplifies the proof of the correctness of the compiler. Each phase is specified by a rewriting system. The first phase breaks each construct of the original specification into its own process, so each guard, assignment, and sequencing for instance is now a separate (and of relatively small size) process. The second phase replaces the atomic synchronization action with a 4-phase handshaking protocol. The remaining three phases translate the remaining processes into circuitry. Putting the five phases together we have a collection of transformations that allow all specifications to be automatically transformed to a circuit. A sample translation appears in Section 5.7.

Correctness of the compiler is established in Section 6. Intuitively, correctness means the specification has "the same behavior" as the circuit. This then leads one to believe we should prove the specification *equivalent* to the circuit. This is not sensible, though, for one main reason: specifications and circuits must interact with their environment, but the two differ in the way they interact with the environment. In particular, specifications communicate with their environments by synchronization, and circuits communicate by 4-phase handshaking. The equivalence must then take into account the change of environment. We define a notion of *transformation equivalence* specifically for this purpose. Transformation equivalence is a "closed-world" condition, specifications compile to correct circuits in the sense that they will operate properly when connected to other circuits compiled by our method. This may seem like too restrictive a notion of correctnss, but we will argue that it in fact is the most appropriate one. From this correctness result we can derive that the final circuit contains no hazards provided the initial specification was semantically well-formed. Lengthy definitions and proofs have been relegated to Appendices.

1.2. Contrasts with Martin and Burns

We derive our method from that of Martin and Burns. A number of changes have been made to make it more precise and complete and thus feasible to rigorously prove correctness. Differences are summarized here for readers familiar with their work.

The language We define a somewhat simpler language, with only single-bit variables and mutually exclusive guards. No data may be sent across channels. These restrictions are solely to make correctness proofs more manageable, we do not think it would be difficult to extend our results to a full language. We use a more precise notion of variable scoping, important in establishing correctness. We use one language for the compilation process as opposed to their three (high-level, production rules, gates). We have no direct analogue of production rules, as this was not needed for our purposes.

Components and mutual exclusion We have a formal notion of a semantically well-formed component, free of mutual exclusion violations, and it is only these components for which the compilation process will produce a correct circuit. Martin also uses a notion of mutual exclusion, our work can be viewed as a rigorous reformulation of his ideas. We also define a precise notion of *guard stability*.

The translation process We use the same general technique for asynchronous circuit compilation, but with significantly different particulars. We use one language for the whole translation process. Our translation scheme has separate phases to modularize descriptions and reshuffle handshakes. There is no production rule phase. A somewhat different compilation method is used for guards and synchronizations.

1.3. Related Work on Correctness of Asynchronous Circuit Compilers

Two papers address the same general problem of proving correctness of asynchronous circuit compilers [29], [27]. These two papers are more closely related to each other than either are to our work. Some comparisons are as follows. Weber, Bloom, and Brown define a process language Joy and show how it may be compiled to asynchronous circuitry. Joy imposes a number of syntactic restrictions not found in our C-CSP language, including the restriction that no processes may share variables or passive port names. The proof of correctness of the compiler is heavily dependent on these restrictions. The language restriction necessitates the use of explicit handshaking to read variables, so the resulting circuits are slower than those we generate. A benefit of their method is the transformation process guarantees isochronic forks will be isolated in small sections of the final circuit. They use a bisimulation ordering to establish correctness, whereas we use testing. Bisimulation is a stronger equivalence than testing (fewer processes are related by bisimulation). The two different equivalences give rise to two significantly different proof techniques. It is not clear if bisimulation could be used to prove our compilation method correct, the more liberal nature of C-CSP requires a more "context-dependent" analysis achievable through testing but not bisimulation.

Kees van Berkel [27] gives a correctness proof for compiling the CSP-based specification language Tangram to handshake circuits. Tangram can only have single uses of each port. The compilation process goes through an intermediate language, *handshake circuits*. His book focuses on many other important issues such as initialization and optimization. The correctness proof is based on trace theory and stops at the handshake circuit level, thus mostly avoiding the problems of hazards and mutual exclusion violation. Neither of these works incorporates a fairness assumption as we do, fairness is notoriously difficult to deal with and is generally not taken into account for this reason.

2. The Circuit Language—C-CSP

In this section we introduce C-CSP (Circuit-CSP), a variation of the CSP language [13] based on the version of CSP designed by Martin [17] for specifying asynchronous circuits. We have removed some syntactic sugar to make the correctness task more feasible and added refined scoping constructs needed to guarantee correctness. Unlike Martin, 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 translations from one language to another, and giving semantics to all the languages. In Section 2.1 we define modules, components and closed terms, concepts that will be important in proving correctness. In Section 2.2 we define two sublanguages of C-CSP: S-CSP represents specifications and H-CSP represents actual hardware devices, respectively.

DEFINITION 1 C-CSP boolean expressions e, commands c, and declarations d are defined by the following grammar

$$\begin{aligned} e &::= \mathbf{t} \mid \mathbf{f} \mid x \mid \bar{P}? \mid (e \wedge e) \mid (e \vee e) \mid \neg e \\ c &::= \mathbf{skip} \mid x := e \mid c ; c \mid [e \rightarrow c \rrbracket \ldots \llbracket e \rightarrow c] \mid \; *[c] \mid c \| c \mid P! \mid P? \mid \end{aligned}$$

$$\begin{array}{rl} & \textbf{with } d \textbf{ do } c \textbf{ end} \\ d ::= & \mathbf{r}\ x,\ d \mid \mathbf{w}\ x,\ d \mid P!,\ d \mid P?,\ d \mid \epsilon \end{array}$$

Commands will also be referred to as terms or processes. $\mathcal{V}$ is the set of C-CSP variables where $x, y, z \ldots \in \mathcal{V}$ range over variables, and $!a, !p, !s, !t, ?a, ?p, ?s, ?t \ldots \in \mathcal{V}$ range over *handshaking* variables. The handshaking variables are boolean variables that take on special significance in the implementation of handshaking protocols. $\mathcal{P}_a$ is the set of active port names and is ranged over by $S!, P!, \ldots$. Similarly, $\mathcal{P}_p$ is the set of passive port names, and $S?, P?, \ldots$ range over $\mathcal{P}_p$.

The boolean expressions e are the usual ones plus the probe $\bar{P}?$ [14], 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 $x := \mathbf{t}$ and $x := \mathbf{f}$ by $x \uparrow$ and $x \downarrow$, respectively. Sequential composition is denoted by ";". Choice among guarded commands is designated by $[\!]$ 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* 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*.

Various syntactic constraints must be placed on the above grammar. Declarations bind the occurrences of variables and ports, and a variable or port name can be used *only* within the scope of its declaration. All boolean variables may be declared (scoped) twice: once by a declaration in which they may be read ($\mathbf{r}$), and once where they may be both read and written ($\mathbf{w}$). This form of declaration is useful in proofs of correctness. For instance, the declaration of $\mathbf{w}\ x$ in **with** $\mathbf{w}\ x$ **do** c **end** allows us to reason about c knowing x is written *only* in c, but could be read elsewhere. It should be noted that in the term **with** $\mathbf{w}\ x$ **do with** $\mathbf{r}\ x$ **do** c **end end**, x may be written in c: a "$\mathbf{r}\ x$" declaration is not interpreted as "read-only", only as "readable." Corresponding active and passive parts of a channel P are declared separately as $P!$ and $P?$, for the same reasons: in **with** $P!$ **do** c **end** it can be assumed that $P!$ is used *only* in c, but $P?$ could in theory occur both in c and externally. It is also not possible to declare the same port name more than once, nor is it possible to declare a variable either $\mathbf{r}\ x$ or $\mathbf{w}\ x$ more than once. Another syntactic restriction required for circuits to behave properly is it is impossible to negate a probe or use probes in assignments. These restrictions are made precise in the following definition.

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. All probes $\bar{P}?$ occurring in guard expressions e occur *positively*, namely embedded within an *even* number of negations.

2. Probes do not occur in the expression part of any assignment statement.
3. each send $P!$ or receive port $P?$ occurring in c is declared at most once in c;
4. each variable x is declared at most once $\mathbf{w}\ x$ at most once $\mathbf{r}\ x$ in c;
5. If both $\mathbf{w}\ x$ and $\mathbf{r}\ x$ declarations occur in c and x occurs within the scope of the latter but not the former, it cannot be a write occurrence.
6. 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 implicitly taken to be syntactically well-formed, and the composition of smaller terms to make larger ones must be syntactically well-formed.

Finally, we may define *strip*(c) to be c with all declarations removed. A small example of a C-CSP expression is

```
with $P?,\ Q!,\ R!,\ \mathbf{r}\ x, \mathbf{w}\ x,\ \mathbf{r}\ y$ do
    $*[[\bar{P}? \longrightarrow\ x := \neg x;$
                    $[x \vee y \longrightarrow Q! [\!] \neg x \wedge \neg y \longrightarrow R!]; P?]]$
end,
```

a process that alternates synchronizations on P with synchronizations on Q and R, respectively, with y an external override in favor of Q.

2.1. Modules, Components and Closed Terms

In order to formally describe compilation, we need to define three classes of C-CSP terms—*modules, components* and *closed terms.* The C-CSP terms that may be separately compiled are *modules.* All ports used and variables written in a module must be declared in that module. A module may interact with its external environment via ports and variables, indicated by declaring a variable only written or only read, or declaring only one of the active and passive ports of a channel. Modules 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. They also will prove to be important in correctness arguments.

The terms in the language corresponding to physical silicon devices (chips and their specifications) are *components.* They may communicate with the outside world via ports, but, unlike modules, may not share boolean variables with the outside world. Components are generally composed of a collection of modules that may be re-used in other components. The main purpose of components for our purposes is they may be shown to be *semantically well-formed*, meaning lacking in mutual exclusion errors, a notion defined in the next section. The notions of component and module cannot be conflated for this reason: it is not possible to define what a semantically well-formed module is—since modules can share variables, one module may be semantically well-formed when hooked up with some module but ill-formed when hooked up with another. Since components share no variables, they are well- or ill-formed in and of themselves.

Closed terms share no variables or ports with the outside world. Formally, these three classes of terms are defined as follows.

Definition 3

1. A C-CSP term m is a *module* exactly when 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.
2. A C-CSP term k is a *component* if it 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. Additionally, for any handshaking variable $!s/?s$ read or written, it must be appropriately declared as $\mathbf{r}\ !s/\mathbf{r}\ ?s$ or $\mathbf{w}\ !s/\mathbf{w}\ ?s$, respectively.
3. 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 ports $P!$ and $P?$ declared in c, and all variables occurring in c (including handshaking variables) have both read and write declarations.

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

C-CSP spans the expressibility gamut from high-level specifications 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 hardware devices.

S-CSP forces specifications to abstract from the actual implementation of the synchronization between components by prohibiting any use of handshaking variables. All terms in S-CSP may be compiled to circuits.

Definition 4 S-CSP (Specification CSP) terms are C-CSP terms with no instances of handshake variables $!s$, $?s$.

H-CSP (Hardware CSP) terms represent a collection of gates. Let ℓ range over literals of the form x or $\neg x$ for variable x.

Definition 5 (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 6 H-CSP terms are terms c such that

$$strip(c) = \mathbf{AHS}(!s, ?s) || c_1 || c_2 \ldots || c_n,$$

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.

$\mathbf{AHS}(!s, ?s)$ (defined in section 5.3) is a single active handshaking protocol to initiate the execution of the entire circuit. This could have been be compiled to hardware, but since initialization methods are more technology-dependent, we have left it abstract.

These gate processes in fact serve well as a mathematical model of gate behavior. Each atomic execution of a gate process updates its output value x. Isochronicity of forks is implicit in that wire states are variables, and once a variable is written, all locations that read that variable will get the new value.

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 certain mutual exclusion constraints. 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 7

1. A *state* σ is a finite mapping from $\mathcal{V} \cup \mathcal{P}_a$ to **Bool**. We denote the set of states by States.

2. ι :C-CSP $\longrightarrow$ 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, $\sigma_0(x) = \mathbf{f}$ and $\sigma_0(P!) = \mathbf{f}$.

3. 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 $b \in \{\mathbf{t}, \mathbf{f}\}$. $\mathcal{P}_a$ is part of the domain of σ and is used to define the semantics of the probe: $\sigma(\bar{P}?) = \mathbf{t}$ 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 [10] (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 8

1. 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. $C[c_1] \ldots [c_n]$ is the result of syntactically replacing all occurrences of $\bullet_i$ in C with terms c_i, for each $1 \leq i \leq n$.

2. A *closing context* for a term c is a context C such that $C[c]$ is closed.

3. 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, $R[c][c'] = R[c]$. The following subsections define the evaluation of boolean expressions, and 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.

DEFINITION 9 Let e, e_1, e_2 be expressions.

$$\begin{aligned}
\sigma(\mathbf{t}) &= \mathbf{t} \\
\sigma(\mathbf{f}) &= \mathbf{f} \\
\sigma(\bar{P}?) &= \sigma(P!) \\
\sigma(e_1 \wedge e_2) &= \sigma(e_1) \wedge \sigma(e_2) \\
\sigma(e_1 \vee e_2) &= \sigma(e_1) \vee \sigma(e_2) \\
\sigma(\neg e) &= \neg\sigma(e)
\end{aligned}$$

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 10 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)

$$\langle R[[e_1 \longrightarrow c_1 \talloblong \dots \talloblong e_i \longrightarrow c_i \talloblong \dots \talloblong e_n \longrightarrow c_n]], \sigma\rangle \rightarrow \langle R[c_i], \sigma\rangle$$
$$\text{where } \sigma(e_i) = \mathbf{t} \text{ and } \forall j \neq i.\sigma(e_j) = \mathbf{f}$$

(Repetition)

$$\langle R[*[c]], \sigma\rangle \rightarrow \langle R[c; *[c]], \sigma\rangle$$

(Parallelism)

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

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

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

It should be noted that except for synchronization on $P!/P?$, parallel execution is approximated by interleaving. This is a standard approach taken by process algebra researchers [13], [21]. Since the circuits we are synthesizing are asynchronous, this approximation cannot lead to any timing errors in the final circuit.

We next define those configurations that violate mutual exclusion principles, and thus should not arise in computing. Proper specications and circutis thus should never under any conditions exhibit these errors.

DEFINITION 11 (ERROR CONDITIONS) $\varepsilon(\langle c, \sigma\rangle)$ is defined as the union of the following clauses:

(reading while writing) $\varepsilon(\langle R[x := e][y := e'], \sigma\rangle)$ if $\sigma(e') \neq (\sigma[x = \sigma(e)])(e')$.

(writing while writing) $\varepsilon(\langle R[x := e_1][x := e_2], \sigma\rangle)$

(multiple enabled active ports) $\varepsilon(\langle R[P!][P!][P?], \sigma\rangle)$.

(multiple enabled passive ports) $\varepsilon(\langle R[P!][P?][P?], \sigma\rangle)$.

(multiple true guards) $\varepsilon(\langle R[[e_1 \longrightarrow c_1 [\!] \dots [\!] e_i \longrightarrow c_i [\!] \dots [\!] e_n \longrightarrow c_n]], \sigma\rangle)$ if $\sigma(e_i) = \sigma(e_j) = \mathbf{t}$ for some $j \neq i$

(non-stable guard)
$\varepsilon(\langle R[x := e][[e_1 \longrightarrow c_1 [\!] \dots [\!] e_i \longrightarrow c_i [\!] \dots [\!] e_n \longrightarrow c_n]], \sigma\rangle)$ if $\sigma(e_i) = \mathbf{t}$ and $(\sigma[x = \sigma(e)])(e_i) = \mathbf{f}$ for some i.

(probing guard while synchronizing)
$\varepsilon(\langle R[P!][P?][[e_1 \longrightarrow c_1 [\!] \dots [\!] e_i \longrightarrow c_i [\!] \dots [\!] e_n \longrightarrow c_n]], \sigma[P! = \mathbf{t}]\rangle)$ if $(\sigma[P! = \mathbf{t}])(e_i) = \mathbf{t}$ and $(\sigma[P! = \mathbf{f}])(e_i) = \mathbf{f}$ for some i.

Note the presence of reduction contexts R mean the above errors only happen when the particular statements are enabled to execute. An idle guarded command can thus have multiple true guards. Guards must be *stable* in the sense that if one of the boolean guards becomes true while the guard is enabled to execute, this guard must stay true. The **(non-stable guard)** condition captures when this fails. We will informally write $\langle c, \sigma\rangle \rightarrow$ **ERROR** to mean $\varepsilon(\langle c, \sigma\rangle)$.

DEFINITION 12 $\overset{*}{\rightarrow}$ is the transitive, reflexive closure of single-step computation $\rightarrow$.

3.3. Semantic Well-Formedness

Only those specifications that lead to no mutual exclusion **ERROR**s, as defined above, can be compiled into hazard-free circuits. These are the *semantically well-formed* specifications. Hazard-freeness for circuits is defined as semantic well-formedness: semantic well-formedness means no mutual exclusion errors occur, and for gates a hazard is a **(reading while writing)** mutual exclusion error: a gate that was enabled to fire becomes disabled by a change to its inputs.

It is not possible to sensibly define well-formedness for modules since modules may read external variables, and there is no condition on when external variables may be written. Thus the need for the components defined earler—they are modules with all variables declared locally. A component k is semantically well-formed if when executed concurrently with any other component k', any error that arises is “caused by” k', not by k.

DEFINITION 13

1. A 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 semantically well-formed if $\langle c, \iota(c) \rangle$ is semantically well-formed.

2. A component k is *semantically well-formed* iff for all components k' such that $k||k'$ is closed, for all computations

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

it is the case that $\neg\varepsilon(\langle k_n, \sigma_n \rangle)$, and furthermore if $\neg\varepsilon(\langle k'_n, \sigma_n \rangle)$ then the error was caused by one of **(multiply enabled active ports)**, **(multiply enabled passive ports)**, or **(probing guard while synchronizing)** cases, and two of the three processes producing the error were in k'_n.

The last part of 2. assigns blame in the special case that three processes were involved in an error: the component with two processes involved gets the blame. If only two processes were involved in the error, the first part of 2. guarantees the error was not in k_n (note that by the fact that k_n and k'_n do not share variables, both of the processes involved in a two-process error must be in one of k_n and k'_n, and not spread between the two). This definition of semantic well-formedness is only intended to apply to two varieties of configurations: closed configurations, and S-CSP specifications. It turns out that there is no need to define semantic well-formedness for other varieties of configuration, avoiding a more complex definition.

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

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

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 do not delay infinitely), we make a fairness assumption part of our semantics. Specifically, we hereafter restrict ourselves to the *weakly fair* computations. A weakly fair computation path is roughly defined as follows: if a process is continuously enabled to execute a particular step, the step will eventually execute later in the computation path. In the above example it means a synchronization on channel P must eventually occur because we have ruled out the unfair computations that starve all but the first process. The full definition of fairness is found in Appendix A as Definition 28.

4. Circuit Testing and Equivalence

Having defined the operational semantics and semantic well-formedness, we can now define notions of equivalence on C-CSP terms that will be used to prove the compiler correct. The equivalences we define are in the spirit of the *testing* equivalence of [22], [11], with some ideas taken from Morris/Plotkin operational equivalence [23]. Testing equivalence 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.

We begin in Section 4.1 by defining the basic framework for testing equivalence. Next we present the Observable Determinicity Theorem, which shows all processes in our language have no nondeterministic behavior. In Section 4.2 we review the basics of rewriting systems and in Section 4.3 we define a notion of testing equivalence over rewriting that is used to establish correctness of the compiler, *transformational equivalence.* As discussed in the introduction, the translation process does not preserve standard notions of equivalence because the interaction with the environment is changed when synchronization is replaced by handshaking.

4.1. Testing

The basic idea of testing equivalence is a process c is tested by running it in parallel with a testing process c', $c||c'$. c' can communicate with c to test its behavior. Two processes are equivalent if they behave the same when tested by all tests.

We define a language of testers, C-CSP*, by adding a new distinguished *success variable*, x_{success} to the variable set $\mathcal{V}$. A testing process indicates success by setting x_{success} to $\mathbf{t}$. We then define the notion of a *successful* and a *failing* computation.

Definition 14 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, $\sigma_i(x_{\text{success}}) = \mathbf{t}$. It is *failing* if it is not successful.

Two processes are testing equivalent if when tested by any test (i.e., run together with the test process) they have the same behavior. The question, then, is exactly what this "behavior" is to be. The classical definition is that they have the same successful or unsuccessful outcomes. That idea is captured in the following definition (c and c' here are terms that are combinations of the processes being tested and the tester).

Definition 15 (Observation Equivalence) Let c and c' be closed semantically well-formed C-CSP* terms. Then, $c \cong_{obs} c'$ iff

1. There exists a successful computation of c iff there exists a successful computation of c'.

2. There exists a failing computation of c iff there exists a failing computation of c'.

In our setting, we in addition need to take errors into account, and not equate an error-free process with one that may error upon execution. But, some of the transformations we define actually *decrease* the number of errors that occur. Namely, if a component has an error, there is a chance that it is compiled to a circuit that has no errors. To account for this, an error ordering $c \geq_{err} c'$ is defined to indicate a decrease in the number of errors.

DEFINITION 16 (ERROR ORDERING) Let c and c' be closed C-CSP* terms. Define $c \geq_{err} c'$ iff c is semantically well-formed implies c' is semantically well-formed.

The desired observation that takes errors into account may now be defined.

DEFINITION 17 (OBSERVATION/ERROR EQUIVALENCE) For c and c' closed terms, $c \overset{\Rightarrow}{\cong}_{obserr} c'$ iff

1. $c \geq_{err} c'$, and
2. if c and c' are semantically well-formed, then $c \cong_{obs} c'$.

An important property of C-CSP is all terms are *observably deterministic*, that is, either all fair computations are successful or all fair computations are failing. There may be significant parallelism, but none of the parallelism can be detected by an outside observer. This property simplifies the definition of observation equivalence above. Since we deal only with deterministic processes, it suffices to have only the first clause of the definition of $c \cong_{obs} c'$, simplifying the proofs of correctness.

THEOREM 1 (OBSERVATIONAL DETERMINISM) For any semantically well-formed closed C-CSP* terms c, either all fair computations of c are successful or all fair computations are failing.

A complete proof of this theorem is found in Appendix B. We provide a sketch here. First a local notion of determinicity, the Strong Diamond Property (Lemma 11), is proven. If two different computation steps are possible from a given configuration, this Lemma shows the order the two are executed in does not matter since the two different configurations can merge again by executing a single step. More precisely, there are 49 possible parallel combinations of single step reductions (there are 7 single step reductions). By symmetry, we can reduce this to 28 distinct cases. We then show that if two different single-step reductions are enabled at the same time, then the two steps can execute in either order and reach the same state. Consider for instance the case of two concurrently executing assignment statements, $x := e_1$ and $x := e_2$. If executing e_1 before e_2 changes the value of e_2, or executing e_2 before e_1 changes the value of e_1, then the term has an error condition. Thus executing $x := e_1$ does not change the value of e_2. Similarly, executing $x := e_2$ does not change the value of e_1. Thus executing the two in either order will lead to the same configuration. By chaining applications of the Strong Diamond property (via the Bubbling Lemma, Lemma 13), any fair failing path can be turned into a successful one provided at least one successful path exists. This is because the succesful and failing path really must just be permutations of the "same" computation by the above Strong Diamond Property. Thus, there cannot be present both a successful and a failing path, proving the Theorem. Note that unfair failing paths cannot necessarily be turned into successful paths because the testing process could starve. So, this property fails without a fairness assumption.

Corollary 1 All closed, hazard-free (i.e., semantically well-formed) H-CSP circuits c (constructed solely of and, or, not gates, and C-elements) are observably deterministic.

Proof: This follows directly from the previous Theorem and the observation that H-CSP is a sublanguage of C-CSP. ■

This Corollary gives an elegant theoretical characterization of the arbiter-free speed-independent circuits.

The practical utility of observational determinism is the definition of $\cong_{obs}$ can be considerably simplified.

Corollary 2 In the definition of $\cong_{obs}$, Definition 15, case 2. may be removed without changing the meaning of the definition.

Proof: It suffices to show case 2. follows from case 1. Suppose there exists a failing computation c (the opposite direction follows by symmetry). Then by determinism all computations are failing, so there exists no successful computation of c. Thus, by assumption all computations of c' are also failing. ■

Now that observations have been defined, we may define exactly how the tests are performed. Define a *testing context* to be a C-CSP* context. To test a term c we place it in a closing testing context C, forming $C[c]$.

Definition 18 (Testing Equivalence) Let c, c' be C-CSP processes. $c \overset{\rightrightarrows}{\cong} c'$ if for all closing C-CSP* testing contexts C, $C[c] \overset{\rightrightarrows}{\cong}_{obserr} C[c']$.

Theorem 2 (Congruence) For any context C, $c \overset{\rightrightarrows}{\cong} c'$ implies $C[c] \overset{\rightrightarrows}{\cong} C[c']$.

Proof: Suppose $c \overset{\rightrightarrows}{\cong} c'$; show $C[c] \overset{\rightrightarrows}{\cong} C[c']$, i.e. show for any closing context C' that $C'[C[c]]$ and $C'[C[c']]$ have the same success and failure behavior. This is trivial by assumption, since $C'[C]$ is a particular context our assumption holds for. ■

This congruence principle is useful in showing the correctness of the compilation process: it allows us to substitute equals for equals, and justifies the use of local rewriting rules.

Unfortunately, the general notion of testing equivalence defined above is too strong to use in proving the transformations correct. For instance, $\overset{\Rightarrow}{\cong}$ is not preserved by the compilation step that replaces a high level synchronization with a handshaking protocol because a process that communicates with its environment by communication channels is not equivalent to the process with the channels replaced by handshaking wires. In addition, the actual circuits need to make a number of assumptions about the environment in which they operate. Even when a particular process is idle, certain inputs cannot be changed because they may in fact cause the circuit to execute when it should be idle. In order to adequately handle these problems, we define a more refined notion of equivalence, called *transformation equivalence*, that restricts the set of tests used to make them dependent on the translation process itself. This results in a "closed-world" notion of correctness: modules are compiled correctly in the sense that they will work correctly when connected to other modules compiled by our method. This may seem too weak a notion of correctness, but a full-scale version of C-CSP would include a specific mechanism for interfacing with other devices, and through this interface the devices connected would not have to be compiled by our method. There is a strong analogy here with programming lanaguages: modules produced by a C compiler cannot be expected to be linkable with modules produced by a Pascal compiler. By working under this closed-world assumption, our language can be more liberal, and the compilation method more efficient. For this reason we believe this approach is the best one.

Before we define transformation equivalence, a digression into the general structure of the translation process is given.

4.2. Rewriting and Equivalences

We divide the translation process into five phases and implement each phase using a distinct term rewriting system (see [7] for background and references on rewriting). These systems are presented in Section 5. Rewrite systems are simple rule-based systems for replacing one subterm by another. One additional feature we use is to rewrite with respect to a set of equations (*equational rewriting*). This allows simple equivalence-preserving transformations such as commutativity and associativity to be performed implicitly.

DEFINITION 19 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 five 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.

DEFINITION 20 Given a rewrite system R/E, the 2-place relations $=_E$, $=^*_E$, $\Rightarrow_{R/E}$, $\Rightarrow^*_{R/E}$, and $\Rightarrow^N_{R/E}$ on modules are defined as follows.

1. $m_0 =_E m_1$ (m_0 is equivalent by one of the equational rules to m_1) iff $m_0 = C'[c_0]$, $m_1 = C'[c_1]$, $\epsilon_0 =_E \epsilon_1$ is a rule in E, and the c_i are derived from the ϵ_i by uniformly substituting C-CSP terms for metavariables.

2. $=^*_E$ is the transitive reflexive closure of $=_E$.

3. $m_0 \Rightarrow_{R/E} m_1$ (m_0 rewrites in one step modulo the equations E to m_1) iff $m_0 = C'[c_0]$, $m_1 = C'[c_1]$, $m_0 =^*_E C'[c'_0]$, $C'[c'_1] =^*_E m_1$, $\epsilon_0 \rhd_R \epsilon_1$, and the c'_i are derived from the ϵ_i by uniformly substituting C-CSP terms for metavariables. That is, any of the equations may be applied before and after the application of the rewriting rule.

4. $\Rightarrow^*_{R/E}$ is the transitive reflexive closure of $\Rightarrow_{R/E}$.

5. $m_0 \Rightarrow^N_{R/E} m_1$ (Normalizing rewrite) iff $m_0 \Rightarrow^*_{R/E} m_1$, and there is no m_2 such that $m_1 \Rightarrow_{R/E} m_2$.

In Section 5, we define five rewrite systems $\Rightarrow_1$–$\Rightarrow_5$ by giving five sets of rules $\rhd_1 \ldots \rhd_5$, and a set of scope, commutativity, and associativity equations, SCA, that is the same for all systems. We will not explicitly mention "SCA" and notate the five rewrite relations as $\Rightarrow_i$. A specification m_0 is compiled to a circuit m_5 by the rewriting

$$m_0 \Rightarrow^N_1 m_1 \Rightarrow^N_2 m_2 \Rightarrow^N_3 m_3 \Rightarrow^N_4 m_4 \Rightarrow^N_5 m_5,$$

abbreviated $m_0 \Rightarrow_{1-5} m_5$ (m_0 compiles to m_5). Modules that are the result of translating an initial specification through i levels are defined as follows.

DEFINITION 21 For $i \in \{1, \ldots, 5\}$, $\Rightarrow_i m$ iff $m_0 \Rightarrow^N_1 m_1 \ldots \Rightarrow^N_i m$ and $m_0 \in$ S-CSP, $\Rightarrow_0 m$ holds if $m \in$ S-CSP.

The two important facts to establish about each rewrite system are normalization, so a terminating translator can be written; and semantics preservation, so the rewritten term has the same behavior as the original.

4.3. Transformation Equivalence

Having reviewed the basic concepts of rewriting systems, we can now identify the set of tests that will allow us to define *transformation equivalence*, the equivalence used to prove the synthesis method correct. The intuition behind transformation equivalence is that in order to prove a transformed process is equivalent to the original, the transformed process can only be tested by tests that are obtained by transforming the tests used to test the original process. That is, one phase of the translation process preserves transformation equivalence if the behavior remains the same when both the process and its test are translated and have the same testing outcome.

The tests take the form of *testing modules* m^t running in parallel with the module m to be tested, $m^t \| m$. This gives a sensible notion of how a specification is tested, because m^t serves as a complete description of the environment of the specification. Transformation equivalence is then defined as follows.

Definition 22 (Transformation Equivalence) $m_i \overset{\Rightarrow}{\cong}_{i+1} m_{i+1}$, for $0 \leq i \leq 4$, iff

1. $\Rightarrow_i^N m_i$,
2. $m_i \Rightarrow_{i+1}^N m_{i+1}$,
3. for any testing module m_i^t such that $\Rightarrow_i^N m_i^t$ and $m_i^t \| m_i$ is closed, if $m_i^t \Rightarrow_{i+1}^N m_{i+1}^t$, then $(m_i^t \| m_i) \overset{\Rightarrow}{\cong}_{obserr} (m_{i+1}^t \| m_{i+1})$.

Note $\overset{\Rightarrow}{\cong}_i$ is the transformational equivalence for rewrite phase i. We let $m_0 \overset{\Rightarrow}{\cong}_c m_5$ abbreviate $m_0 \overset{\Rightarrow}{\cong}_1 m_1 \ldots \overset{\Rightarrow}{\cong}_5 m_5$, the correctness of of a complete compilation from specification to circuit. This definition means given a specification module m_0 compiled to a digital circuit m_5, the compilation process is *correctness-preserving* if and only if for any tester of the original system, m_0^t, this tester makes the same observations on m_0 as its compiled counterpart m_5^t does on m_5.

5. Compilation of C-CSP Specifications to Circuits

We now define a family of five rewrite systems for incrementally translating C-CSP process specifications to circuit implementations. Our translation roughly follows that of [5], though many changes were necessary to define a provably correct translation process. To their credit we did not find any significant *errors* in their work, only ambiguities.

A specification m_0 is compiled by applying the five rewrite systems in turn, translating m_0 to m_1, to m_2, ..., and finally to a circuit module m_5. Each of these rewrite systems is defined with respect to a set of equations, SCA, that equates certain terms that differ in trivial ways.

Phase 1, process decomposition, 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 modularizes the specification by giving each use of a port a new, distinct, name. Phase 4 reshuffles the handshake protocols in order to make efficient circuit implementations more feasible. Finally, Phase 5 translates each of the small modules that remain into digital circuitry consisting of and, or, not gates, C-elements, and wires.

(**SCA : SCOPE 1**)	**with** d_1, d_2 **do** c **end** $=$ **with** d_1 **do with** d_2 **do** c **end end**
(**SCA : SCOPE 2**)	**with** d_1 **do with** d_2 **do** c **end end** $=$ **with** d_2 **do with** d_1 **do** c **end end**
(**SCA : SCOPE 3**)	$C[$**with** d **do** c **end**$]$ $=$ **with** d **do** $C[c]$ **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 : GD 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

5.1. Scope, Commutativity and Associativity Equations

For each of the rewrite systems $\Rightarrow_i$, $1 \leq i \leq 5$, we rewrite with respect to the same fixed set of equations, the scope, commutativity and associativity (SCA) equations. These equations provide a sound means for moving declarations, commuting and associating parallel and sequential composition, and permuting guarded command order in a choice construct. By rewriting with respect to this set of equations, the number and complexity of rewrite rules in each phase is reduced. The set of SCA equivalences 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, associativity and permutation equations are self-explanatory.

LEMMA 1 The SCA equations preserve testing equivalence in both directions. That is, for any SCA equation $p = p'$, $p \overset{\Rightarrow}{\cong} p'$ and $p' \overset{\Rightarrow}{\cong} p$.

5.2. Phase 1: Process Decomposition

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. Before any other transformations can be applied, a distinguished "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.

We use *distinguished* active and passive port names $S!$ and $S?$ in this phase. These names are used to distinguish those ports added in the translation process from those that were in the original specification. Since all the new processes are guarded by distinguished port names, the rewriting is guaranteed to terminate.

The rewrite rules for phase 1 appear in Figure 2. Rule (**1 : INIT**) adds a "starting" channel to the process. Rules (**1 : ASSN1**) and (**1 : ASSN2**) isolate all assignments to a particular variable into a single assignment process by creating an assignment process (**1 : ASSN1**) and replacing all assignment statements by synchronizations with this new process (**1 : ASSN 2**). An assignment processes consists of a choice between two a guarded commands: one for assigning **t** ($\uparrow$), and the other for assigning **f** ($\downarrow$). (**1 : SEQ**), (**1 : GD**), (**1 : LOOP**), and (**1 : PAR**) rules each introduce a separate process for each part of the expression. Additionally, (**1 : GD**) simplifies each guard into disjunctive normal form, strengthens the disjuncts so that they are mutually exclusive, and creates a separate guarded process for each of the strengthened disjuncts. The algorithm used to strengthen the disjuncts is identical to that described in [5] and is presented below.

```
algorithm: disjoint-guards
Let $F = \bigvee_{1 \leq i \leq n} f_i$ (where each $f_i$ is a conjunction of literals)
Let $C = \{f_i \mid 1 \leq i \leq n\}$
while $\exists i \geq 1, j \leq n, i \neq j$ such that $f_i \wedge f_j$ is satisfiable do
    $C := C - \{f_i\}$;
    $F' := \textit{disjunctive-normal-form}(f_i \wedge \neg f_j)$;
    $C' := \{f' \mid f'$ is a disjunct of $F'$ and $f'$ is satisfiable$\}$;
    $C := C \cup C'$;
end
```

After exhaustively applying these rules to a semantically well-formed S-CSP term, we obtain an equivalent term in phase 1 normal form.

DEFINITION 23 A C-CSP module m is in *phase 1 normal form* if *strip*(m) conforms to the following grammar.

$$\begin{aligned}
nf \;&::=\; S!\|nf'\|\ldots\|nf' \\
nf' \;&::=\; *[[\bar{S}_0? \longrightarrow x\downarrow; S_0?[\!]\bar{S}_1? \longrightarrow x\uparrow; S_1?]] \mid \\
&\quad\; *[[\bar{S}? \longrightarrow [e_{1_1} \longrightarrow S_1!; S?[\!] \ldots [\!]e_{1_{m_1}} \longrightarrow S_1!; S?[\!] \ldots \\
&\qquad\quad [\!]e_{n_1} \longrightarrow S_n!; S?[\!] \ldots [\!]e_{n_{m_n}} \longrightarrow S_n!; S?]]] \mid \\
&\quad\; *[[\bar{S}? \longrightarrow (S_1!\|\ldots\|S_n!); S?]] \mid *[[\bar{S}? \longrightarrow *[S'!]; S?]] \mid \\
&\quad\; *[[\bar{S}? \longrightarrow S_1!; S_2!; S?]] \mid *[[\bar{S}? \longrightarrow \textbf{skip}; S?]] \mid *[[\bar{S}? \longrightarrow P\$; S?]]
\end{aligned}$$

where e is any boolean expression, $S!, S?, S'!, S_1, ! \ldots, S_n!$ are distinguished port names, P is a non-distinguished port name, $\$ \in \{!, ?\}$, and for all $1 \leq i \leq n, 1 \leq$

```
(1 : INIT)   $c \triangleright_1 \mathbf{with}\ S!\ \mathbf{do}\ S!\ \mathbf{end}\ \| \mathbf{with}\ S?\ \mathbf{do}\ *[[\bar{S}? \longrightarrow c; S?]]\ \mathbf{end}$
                 where $c$ contains no occurrences of distinguished variables.
(1 : ASSN1)  $\mathbf{with\ w}\ x\ \mathbf{do}\ *[c]\ \mathbf{end} \triangleright_1$
             $\mathbf{with\ w}\ x\ \mathbf{do}$
                 $\mathbf{with}\ S_0?, S_1?\ \mathbf{do} *[[\bar{S}_0? \longrightarrow x\downarrow; S_0? [\!] \bar{S}_1? \longrightarrow x\uparrow; S_1?]]\ \mathbf{end}\ \|$
                 $\mathbf{with}\ S_0!, S_1!\ \mathbf{do}\ *[c]\ \mathbf{end}$
             $\mathbf{end}$
(1 : ASSN2)  $\mathbf{with\ w}\ x\ \mathbf{do}$
                 $\mathbf{with}\ S_0?, S_1?\ \mathbf{do}\ *[[\bar{S}_0? \longrightarrow x\downarrow; S_0? [\!] \bar{S}_1? \longrightarrow x\uparrow; S_1?]]\ \mathbf{end}\ \|$
                 $\mathbf{with}\ S?, S_0!, S_1!\ \mathbf{do}\ C[\,*[[\bar{S}? \longrightarrow x := e; S?]]]\ \mathbf{end}$
             $\mathbf{end}$
             $\triangleright_1$
             $\mathbf{with\ w}\ x\ \mathbf{do}$
                 $\mathbf{with}\ S_0?, S_1?\ \mathbf{do}\ *[[\bar{S}_0? \longrightarrow x\downarrow; S_0? [\!] \bar{S}_1? \longrightarrow x\uparrow; S_1?]]\ \mathbf{end}\ \|$
                 $\mathbf{with}\ S?, S_0!, S_1!\ \mathbf{do}$
                     $C[\,*[[\bar{S}? \longrightarrow [\neg e \longrightarrow S_0!; S? [\!] e \longrightarrow S_1!; S?]]]]\ \mathbf{end}$
             $\mathbf{end}$
(1 : SEQ)    $*[[\bar{S}? \longrightarrow c_1; c_2; S?]] \triangleright_1$
             $\mathbf{with}\ S_1!, S_2!\ \mathbf{do}\ *[[\bar{S}? \longrightarrow S_1!; S_2!; S?]]\ \mathbf{end}\ \|$
             $\mathbf{with}\ S_1?\ \mathbf{do}\ *[[\bar{S}_1? \longrightarrow c_1; S_1?]]\ \mathbf{end}\ \|$
             $\mathbf{with}\ S_2?\ \mathbf{do}\ *[[\bar{S}_2? \longrightarrow c_2; S_2?]]\ \mathbf{end}$
                 where $c_1 \neq c_3; c_4$, $c_1 \neq S_1!$, and $c_2 \neq S_2!$
(1 : GD)     $*[[\bar{S}? \longrightarrow [e_1 \longrightarrow c_1 [\!] \ldots [\!] e_n \longrightarrow c_n]; S?]] \triangleright_1$
             $\mathbf{with}\ S_1!, \ldots, S_n!\ \mathbf{do}$
                 $*[[\bar{S}? \longrightarrow [e_{1_1} \longrightarrow S_1!; S? [\!] \ldots [\!] e_{1_{m_1}} \longrightarrow S_1!; S? [\!] \ldots [\!]$
                           $e_{n_1} \longrightarrow S_n!; S? [\!] \ldots [\!] e_{n_{m_n}} \longrightarrow S_n!; S?]]]\ \mathbf{end}\ \|$
             $\mathbf{with}\ S_1?\ \mathbf{do}\ *[[\bar{S}_1? \longrightarrow c_1; S_1?]]\ \mathbf{end}\ \| \ldots \|$
             $\mathbf{with}\ S_n?\ \mathbf{do}\ *[[\bar{S}_n? \longrightarrow c_n; S_n?]]\ \mathbf{end}$
                 where for $1 \leq i \leq n$ $c_i$ is not a distinguished active
                 synchronization and $e_{i_1} \vee \ldots \vee e_{i_{m_i}}$ is the result of applying
                 the disjoint guards algorithm to the disjunctive normal form of $e_i$
(1 : LOOP)   $*[[\bar{S}? \longrightarrow *[c]; S?]] \triangleright_1$
             $\mathbf{with}\ S'!\ \mathbf{do}\ *[[\bar{S}? \longrightarrow *[S'!]; S?]]\ \mathbf{end}\ \|$
             $\mathbf{with}\ S'?\ \mathbf{do}\ *[[\bar{S}'? \longrightarrow c; S'?]]\ \mathbf{end}$
                 where $c$ is not a distinguished active synchronization
(1 : PAR)    $*[[\bar{S}? \longrightarrow (c_1 \| \ldots \| c_n); S?]]\ \triangleright_1$
             $\mathbf{with}\ S_1!, \ldots, S_n!\ \mathbf{do}\ *[[\bar{S}? \longrightarrow (S_1! \| \ldots \| S_n!); S?]]\ \mathbf{end}\ \|$
             $\mathbf{with}\ S_1?\ \mathbf{do}\ *[[\bar{S}_1? \longrightarrow c_1; S_1?]]\ \mathbf{end}\ \| \ldots \|$
             $\mathbf{with}\ S_n?\ \mathbf{do}\ *[[\bar{S}_n? \longrightarrow c_n; S_n?]]\ \mathbf{end}$
                 where $c_1, \ldots, c_n$ are not distinguished active
                 synchronizations and $n > 1$
```

Figure 2. Rewrite rules for Phase 1

$$(\mathbf{2:HS\ ACT})\quad \mathbf{with}\ P!\ \mathbf{do}\ C[P!]\ \mathbf{end} \rhd_2$$
$$\mathbf{with\ w}\ !p, \mathbf{r}\ ?p\ \mathbf{do}\ C[\mathbf{AHS}(!p, ?p)]\ \mathbf{end}$$

where C contains no occurrences of $P!$.

$$(\mathbf{2:HS\ PAS})\quad \mathbf{with}\ P?\ \mathbf{do}\ C[\bar{P}?][P?]\ \mathbf{end} \rhd_2$$
$$\mathbf{with\ r}\ !p, \mathbf{w}\ ?p\ \mathbf{do}\ C[!p][\mathbf{PHS}(!p, ?p)]\ \mathbf{end}$$

where C contains no occurrences of $P?$ or $\bar{P}?$.

Figure 3. Rewrite rules for Phase 2: Handshaking Expansion

$j \leq m_i$ each e_{i_j} is a disjunct and for all $1 \leq k \leq m_i$ if $j \neq k$ then e_j and e_k are disjoint.

The normalization proof of phase 1 is found in Appendix C. A concrete example of the translation process is found in section 5.7.

LEMMA 2 If m_0 is a S-CSP module and $m_0 \Rightarrow_1^N m_1$ then $m_0 \overset{\Rightarrow}{\cong}_1 m_1$.

The proof of this Lemma appears in Appendix D.2. Intuitively, the correctness of (**1 : ASSN1**) and (**1 : ASSN2**) follow from the observation that if no mutual exclusion 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 correctness of the other rules follows from the observation that although these transformations add new processes, the new processes are activated in the same order as the subprocesses of the original.

5.3. Phase 2: Handshaking Expansion

Handshaking expansion replaces the C-CSP synchronization constructs with boolean handshaking variables implementing a four-phase handshaking protocol. Since the active and passive ports need not be declared in the same scope and in fact could be external, 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

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

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

$$[!p \longrightarrow ?p \uparrow]; [\neg !p \longrightarrow ?p \downarrow]$$

The rules appear in Figure 3.

We now define the normal form produced by handshaking expansion.

Definition 24 A C-CSP term m is in *phase 2 normal form* if $strip(m)$ conforms to the following grammar.

$$
\begin{aligned}
nf \; &::= \; \mathbf{AHS}(!s, ?s) \| nf' \| \ldots \| nf' \\
nf' \; &::= \; *[[!s_0 \longrightarrow x \downarrow; \mathbf{PHS}(!s_0, ?s_0) [\!] !s_1 \longrightarrow x \uparrow; \mathbf{PHS}(!s_1, ?s_1)]] \mid \\
&\quad\; *[[!s \longrightarrow [e_1 \longrightarrow \mathbf{AHS}(!s_1, ?s_1); \mathbf{PHS}(!s, ?s) [\!] \ldots \\
&\qquad\qquad [\!] e_n \longrightarrow \mathbf{AHS}(!s_n, ?s_n); \mathbf{PHS}(!s, ?s)]]] \mid \\
&\quad\; *[[!s \longrightarrow (\mathbf{AHS}(!s_1, ?s_1) \| \ldots \| \mathbf{AHS}(!s_n, ?s_n)); \mathbf{PHS}(!s, ?s)]] \mid \\
&\quad\; *[[!s \longrightarrow *[\mathbf{AHS}(!s', ?s')]; \mathbf{PHS}(!s, ?s)]] \mid \\
&\quad\; *[[!s \longrightarrow \mathbf{AHS}(!s_1, ?s_1); \mathbf{AHS}(!s_2, ?s_2); \mathbf{PHS}(!s, ?s)]] \mid \\
&\quad\; *[[!s \longrightarrow \mathbf{skip}; \mathbf{PHS}(!s, ?s)]] \mid \\
&\quad\; *[[!s \longrightarrow \mathbf{HS}(!p, ?p); \mathbf{PHS}(!s, ?s)]]
\end{aligned}
$$

where e is any boolean expression, $!s, ?s, !s', ?s', !s_1, ?s_1, \ldots, !s_n, ?s_n$ are distinguished handshaking variables, $!p, ?p$ are non-distinguished handshaking variables, and **HS** is either **AHS** or **PHS**.

Lemma 3 If $\Rightarrow_1 m_1$ and $m_1 \Rightarrow_2^N m_2$ then $m_1 \overset{\Rightarrow}{\cong}_2 m_2$.

The proof is found in Appendix D.1. In order to prove this phase correct, we first observe that the handshaking protocol is well behaved with respect to the high-level synchronization constructs, that is all steps in the protocol can be correlated to steps in the high-level synchronization. Further, we show that any error in the protocol that arises in the transformed term will also cause an error at the high level.

5.4. Phase 3: Modularization

The module that results from the phase 2 transformations is a collection of processes executing in parallel. In order to transform this module into a circuit, we next must *modularize* the module: each of the parallel processes is transformed into a module, so we now have a large module consisting of a series of small modules all running in parallel. This transformation is required because circuits can write a variable in only one location (by the gate that has that named wire as output), so all write scopes of variables thus must be localized; this is accomplished by making each process a module. The only processes that are not already modules upon entering this phase are those implementing atomic active and passive synchronization on non-distinguished ports, and the individual guarded command processes. Synchronization processes will fail to be modules at this point if there were multiple occurrences of either $P!$ or $P?$ in the original specification. With guarded processes, there may be many guarded processes that wait for a start signal from the same active handshake, and several guard processes may activate the same subprocess. Both of these guard cases are illustrated by the sample translation in section 5.7.

```
(3 : MOD ACT)  with w !p, r ?p do C[AHS(!p, ?p)] ... [AHS(!p, ?p)] end
               ⊳_3
               with r ?p do
                    with r !p_1, ..., r !p_n do
                         with w !p do ∗[!p :=!p_1 ∨ ... ∨!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 •_i occurs at most once in C, and
               ?p, !p do not occur in C
(3 : MOD PAS)  with r !p, w ?p do
                    C[!p][PHS(!p, ?p)] ... [PHS(!p, ?p)]
               end
               ⊳_3
               with r ?p_1, ..., r ?p_n, w ?p do ∗[?p :=?p_1 ∨ ... ∨?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 •_i for i > 1 occurs at most once in C,
               and ?p, !p do not occur in C
```

Figure 4. Rewrite rules for Phase 3: Modularization

The modularization rules appear in Figure 4. (**3 : MOD ACT**) replaces the i-th occurrence of an active handshake $\mathbf{AHS}(!p, ?p)$ with the handshake $\mathbf{AHS}(!p_i, ?p_i)$. Then, circuitry is added to merge the resulting $!p_1, \ldots, !p_n$ values into one (via an or-gate), and to fan out the $?p$ value to each process. The latter requires the addition of a C-element that remembers which active process initiated the synchronization, thus preventing idle active processes from receiving the reply signal $?p$ (if they did receive this signal anomalous behavior would result at the circuit level). (**3 : MOD PAS**) is the analogous rule for multiply-occurring passive handshakes $\mathbf{PHS}(!p_i, ?p_i)$. There is no need to insert C-elements here, however, because the circuitry to be produced will be robust with respect to having $!p$ high even when the process is inactive.

The normal form for phase 3 extends nf' in the definition of the normal form for phase 2 with the following three clauses:

$$*[!p = !p_1 \vee \ldots \vee !p_n] \mid *[?p_i = !p_i \ \mathbf{C} \ ?p] \mid *[?p = ?p_1 \vee \ldots \vee ?p_n]$$

Phase 3 is also provably normalizing.

The important property after this phase is completed is all processes that comprise the module are themselves modules.

Lemma 4 If $m_0 \Rightarrow^{N}_{123} m_3$ then $m_3 = m'_1 || \ldots || m'_n$, where each m'_i is a module.

We establish correctness of this phase.

Lemma 5 If $\Rightarrow_2 m_2$ and $m_2 \Rightarrow^{N}_{3} m_3$ then $m_2 \overset{\Rightarrow}{\cong}_3 m_3$.

The proof is found in Appendix D.3. The proof hinges on the fact that modularization does not alter the steps in the handshaking protocol even though the transformed process must go through extra steps in order to set the distinct wires. During these extra steps (executing an "or" gate or a "C" element), the other side of the protocol cannot proceed.

5.5. Phase 4: Reshuffling

Before producing circuitry for each module resulting from modularization, we reshuffle some of the handshake protocols to simplify the hardware implementation. All of the reshufflings involve interleaving the final passive handshake with handshakes that directly precede it. This particular form of reshuffling is used because it allows any functional block to be conected to any other. An optimizing compiler would use a wider array of reshuffling techniques.

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 test of $!s$ in the passive protocol $\mathbf{PHS}(!s, ?s)$ is eliminated, and if some of the response $?s \uparrow; [\neg !s \longrightarrow ?s \downarrow]$ is interleaved with the execution of c. Each type of module requires a different form of reshuffling to achieve the most efficient implementation, so the transformations of this phase are meant to set-up the translations to circuitry in the next phase. The transformation rules appear in Figure 5.

```
(4 : SEQ)  with  $\mathbf{r}\ !s, \mathbf{r}\ ?s_1, \mathbf{r}\ ?s_2, \mathbf{w}\ ?s, \mathbf{r}\ !s_1, \mathbf{w}\ !s_2$ do
                 $*[[!s \longrightarrow \mathbf{AHS}(!s_1, ?s_1); \mathbf{AHS}(!s_2, ?s_2); \mathbf{PHS}(!s, ?s)]]$ end
           $\rhd_4$
           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
(4 : PAR)  with  $\mathbf{r}\ !s, \mathbf{r}\ ?s_1, \ldots, ?s_n, \mathbf{w}\ ?s, \mathbf{w}\ !s_1 \ldots, !s_n$ do
                 $*[[!s \longrightarrow (\mathbf{AHS}(!s_1, ?s_1) || \ldots || \mathbf{AHS}(!s_n, ?s_n)); \mathbf{PHS}(!s, ?s)]]$
           end
           $\rhd_4$
           with  $\mathbf{r}\ !s, \mathbf{r}\ ?s', \mathbf{w}\ ?s, \mathbf{w}\ !s'$ do
                 $*[[!s \longrightarrow \mathbf{skip}]; !s' \uparrow; [?s' \longrightarrow \mathbf{skip}]; ?s \uparrow;$
                     $[\neg !s \longrightarrow \mathbf{skip}]; !s' \downarrow; [\neg ?s' \longrightarrow \mathbf{skip}]; ?s \downarrow]$
           end $||$
           with  $\mathbf{r}\ !s', \mathbf{w}\ !s_1, \ldots, !s_n$ do $*[!s_1 := !s'] || \ldots || *[!s_n := !s']$end $||$
           with  $\mathbf{r}\ ?s_1 \ldots, ?s_n, \mathbf{w}\ ?s'$ do $*[?s' := ?s_1 \wedge \ldots \wedge ?s_n]$end
(4 : ACT)  with  $\mathbf{r}\ !s, \mathbf{r}\ ?a, \mathbf{w}\ ?s, \mathbf{w}\ !a$ do
                 $*[[!s \longrightarrow \mathbf{AHS}(!a, ?a); \mathbf{PHS}(!s, ?s)]]$ end
           $\rhd_4$
           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
(4 : PASS) with  $\mathbf{r}\ !s, \mathbf{r}\ !p, \mathbf{w}\ ?s, \mathbf{w}\ ?p$ do
                 $*[[!s \longrightarrow \mathbf{PHS}(!p, ?p); \mathbf{PHS}(!s, ?s)]]$ end
           $\rhd_4$
           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
(4 : GD)   with  $\mathbf{w}\ !s_1, \ldots, \mathbf{w}\ !s_n, \mathbf{w}\ ?s'_1, \ldots \mathbf{w}\ ?s'_n, \mathbf{r}\ !s, \mathbf{r}\ ?s_1, \ldots, \mathbf{r}\ ?s_n$ do
                 $*[[!s \longrightarrow [e_1 \longrightarrow \mathbf{AHS}(!s_1, ?s_1); \mathbf{PHS}(!s, ?s'_1) [] \ldots []$
                            $e_n \longrightarrow \mathbf{AHS}(!s_n, ?s_n); \mathbf{PHS}(!s, ?s'_n)]]]$
           end
           $\rhd_4$
           with  $\mathbf{r}\ !s$ do
                 with $\mathbf{w}\ ?s'_1, \mathbf{w}\ !s_1, \mathbf{r}\ ?s_1$ do
                   $*[[!s \wedge e_1 \longrightarrow ?s'_1 \uparrow; [\neg !s \longrightarrow \mathbf{AHS}(!s_1, ?s_1); ?s'_1 \downarrow]]]$ end
                 $|| \ldots ||$
                 with $\mathbf{w}\ ?s'_n, \mathbf{w}\ !s_n, \mathbf{r}\ ?s_n$ do
                   $*[[!s \wedge e_n \longrightarrow ?s'_n \uparrow; [\neg !s \longrightarrow \mathbf{AHS}(!s_n, ?s_n); ?s'_n \downarrow]]]$ end
           end
```

Figure 5. Rewrite rules for Phase 4: Reshuffling

(**4 : SEQ**) reshuffles the passive handshake $\mathbf{PHS}(!s, ?s)$ into the active handshake $\mathbf{AHS}(!s_1, ?s_1)$, and eliminates the initial $!s$ since it is redundant. (**4 : ACT**) reshuffles the passive handshake $\mathbf{PHS}(!s, ?s)$ into the active handshake $\mathbf{AHS}(!a, ?a)$. (**4 : PASS**) causes the two passive handshakes to execute concurrently. (**4 : PAR**) gets further decomposed at this point: a single start signal $!s'$ is sent to all processes, and the received signals $?s_1, \ldots ?s_n$ are merged into a single $?s'$ by an and-gate, since the execution of the parallel statement is not complete unless all the parts are complere. The reshuffling of s' is then is identical to (**4 : ACT**) reshuffling. (**4 : GD**) reshuffles the guard process, and in addition replaces choice with parallelism. This is possible at this point because in phase 1 the guards were placed in disjoint disjunctive form, so at most one guard holds at any point. Note it is critical that $!s$ be low before the guard body is executed, otherwise the execution of the body could cause some other guarded command to become true and execute. Loop, skip, and assignment statements can be directly implemented without reshuffling.

After exhaustively applying the reshuffling rules, the term is in phase 4 normal form.

Definition 25 A C-CSP term m is in *phase 4 normal form* if *strip*(m) conforms to the following grammar.

$$\begin{array}{rcl} nf & ::= & \mathbf{AHS}(!s, ?s) \| nf' \| \ldots \| nf' \\ nf' & ::= & \mathit{strip}(c) \text{ where there is a phase 4 rule of the form } c_0 \rhd_4 c \\ & & \mid *[!p =!p_1 \vee \ldots \vee !p_n] \mid *[?p_i =!p_i \ \mathbf{C}\ ?p] \mid *[?p =?p_1 \vee \ldots \vee ?p_n] \\ & & \mid *[[!s \longrightarrow \mathbf{skip}; \mathbf{PHS}(!s, ?s)]] \mid *[[!s \longrightarrow *[\mathbf{AHS}(!s', ?s')]; \mathbf{PHS}(!s, ?s)]] \\ & & \mid *[[!s_0 \longrightarrow x\downarrow; \mathbf{PHS}(!s_0, ?s_0) [] !s_1 \longrightarrow x\uparrow; \mathbf{PHS}(!s_1, ?s_1)]] \end{array}$$

Lemma 6 If $\Rightarrow_3 m_3$ and $m_3 \Rightarrow_4^N m_4$, then $m_3 \overset{\Rightarrow}{\cong}_4 m_4$.

This Lemma is proven in Appendix D.4. Key to the proof is a general principle that allows $\mathbf{PHS}(!p, ?p)$ to be reshuffled when all active communications $\mathbf{AHS}(!p, ?p)$ on the same channel are still "pure", i.e. have not yet had anything reshuffled into them. Arbitrary reshuffling runs the risk of deadlock, and this principle captures one very useful form of reshuffling that is always sound. By a proper ordering of the rewriting process, this principle can be used to obtain correctness of the phase.

5.6. Phase 5: Final Compilation into circuits

The final part of the translation transforms each of the individual processes representing atomic assignment, sequencing, guarded commands, active and passive communication, looping, skip, and parallel execution into a circuit representation. The rules appear in Figure 6. To better understand why each of the circuits represents the "higher-level" construct, we describe the circuit realization of the assignment cell. Recall the assignment cell from after handshake expansion,

$$*[[!s_0 \longrightarrow x\downarrow; \mathbf{PHS}(!s_0, ?s_0) [\!] !s_1 \longrightarrow x\uparrow; \mathbf{PHS}(!s_1, ?s_1)]]$$

In response to $!s_0$ or $!s_1$ high (both cannot occur together as concurrent writes are disallowed), the circuit must (1) set x to the appropriate value (which may in fact involve no change if its value is already correct), and (2) correctly execute the passive synchronization in response. To get an idea how the right-hand side of (**5 : ASSN**) implements these two parts consider a synchronization with the cell in order to set x to $\mathbf{t}$ (we assume x is currently $\mathbf{f}$.). We may assume that the environment obeys the 4-phase handshaking protocol, and $!s_1, !s_0, ?s_1, ?s_0$ are all initially $\mathbf{f}$. The assignment is initiated by $!s_1$ going high. The only gate action that can now occur is that the C-element switches, setting x to $\mathbf{t}$. Note that the output of $!s_1 \wedge x$ remains $\mathbf{f}$ until x is set to $\mathbf{t}$. Once x has been set $\mathbf{t}$, $!s_1 \wedge x$ becomes $\mathbf{t}$, thus setting $?s_1$ $\mathbf{t}$. The active handshake will eventually respond by setting $!s_1$ to $\mathbf{f}$. The C-element will not change its value, but $!s_1 \wedge x$ now goes low, completing the passive handshake.

Note that assignment, passive handshake, and guard implementations all require C-elements be added, the remaining forms need no additional state-holding elements.

DEFINITION 26 A C-CSP term m is in *phase 5 normal form* if *strip*(m) conforms to the following grammar.

$$\begin{array}{rcl} nf & ::= & \mathbf{AHS}(!s, ?s) \| nf' \| \ldots \| nf' \\ nf' & ::= & strip(c) \text{ where there is a phase 5 rule of the form } c_0 \rhd_5 c \\ & & \mid *[!p = !p_1 \vee \ldots \vee !p_n] \mid *[?p_i = !p_i \ \mathbf{C}\ ?p] \mid *[?p = ?p_1 \vee \ldots \vee ?p_n] \\ & & \mid *[!s_i := !s'] \mid *[?s' := ?s_1 \wedge \ldots \wedge ?s_n] \end{array}$$

LEMMA 7 If $\Rightarrow_4 m_4$ and $m_4 \Rightarrow_5^N m_5$, then $m_4 \overset{\Rightarrow}{\cong}_5 m_5$.

This proof appears in Appendix D.5. The correctness of each rule hinges on the fact that the actual execution of the gates is constrained to “fire” in the same order as the “higher-level” construct they implement (recall the implementation of the assignment cell). The only real difficulty in proving the correctness of each rule is the error analysis, which requires a proof that no violations of the handshaking protocol can occur in semantically well-formed circuits. Once that has been established, the proofs proceed by a standard argument.

5.7. Example

In this section we show the translation of a simple example to give a flavor of the translation method. Our example specification is a module to let synchronizations

$$
\begin{array}{rll}
(\mathbf{5:\ ASSN}) & \mathbf{with} & \mathbf{w}\ x,\ \mathbf{w}\ ?s_0, \mathbf{w}\ ?s_1, \mathbf{r}\ !s_1, \mathbf{r}\ !s_0\ \mathbf{do} \\
 & & *[[!s_1 \longrightarrow x\uparrow;\ \mathbf{PHS}(!s_1, ?s_1) [\!] !s_0 \longrightarrow x\downarrow;\ \mathbf{PHS}(!s_0, ?s_0)]] \\
 & \mathbf{end} & \\
 & \rhd_5 & \\
 & \mathbf{with} & \mathbf{w}\ x,\ \mathbf{w}\ ?s_0, \mathbf{w}\ ?s_1, \mathbf{r}\ !s_1, \mathbf{r}\ !s_0\ \mathbf{do} \\
 & & *[x := !s_1\ \mathbf{C}\ \neg !s_0] \| *[?s_1 := !s_1 \wedge x] \| *[?s_0 := !s_0 \wedge \neg x] \\
 & \mathbf{end} & \\
(\mathbf{5:\ SEQ}) & \mathbf{with} & \mathbf{r}\ !s, \mathbf{r}\ ?s_1, \mathbf{r}\ ?s_2, \mathbf{w}\ ?s, \mathbf{w}\ !s_1, \mathbf{w}\ !s_2\ \mathbf{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]\ \mathbf{end} \\
 & \rhd_5 & \\
 & \mathbf{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\ \mathbf{do} \\
 & & *[!s_1 := !s] \| *[x := ?s_1\ \mathbf{C}\ \neg ?s_2] \| \\
 & & *[?s := x \vee ?s_2] \| *[!s_2 := x \wedge \neg ?s_1])\ \mathbf{end} \\
(\mathbf{5:\ GD}) & \mathbf{with} & \mathbf{r}\ !s, \mathbf{w}\ ?s, \mathbf{r}\ ?s_1, \mathbf{w}\ !s_1\ \mathbf{do} \\
 & & *[[!s \wedge e \longrightarrow ?s\uparrow; [\neg !s \longrightarrow \mathbf{AHS}(!s_1, ?s_1); ?s\downarrow]]]\ \mathbf{end} \\
 & \rhd_5 & \\
 & \mathbf{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\ \mathbf{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 !s]\ \mathbf{end} \\
(\mathbf{5:\ ACT/PAR}) & \mathbf{with} & \mathbf{r}\ !s, \mathbf{r}\ ?a, \mathbf{w}\ ?s, \mathbf{w}\ !a\ \mathbf{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]\ \mathbf{end} \\
 & \rhd_5 & \\
 & \mathbf{with} & \mathbf{r}\ !s\,\mathbf{r}\ ?a\,\mathbf{w}\ ?s\,\mathbf{w}\ !a\ \mathbf{do} \\
 & & *[!a := !s] \| *[?s := ?a]\ \mathbf{end} \\
(\mathbf{5:\ PASS}) & \mathbf{with} & \mathbf{r}\ !s, \mathbf{r}\ !p, \mathbf{w}\ ?s, \mathbf{w}\ ?p\ \mathbf{do} \\
 & & *[[!s \wedge !p \longrightarrow \mathbf{skip}]; (?p\uparrow \| ?s\uparrow); \\
 & & \ [\neg !p \wedge \neg !s \longrightarrow \mathbf{skip}]; (?p\downarrow \| ?s\downarrow)]\ \mathbf{end} \\
 & \rhd_5 & \\
 & \mathbf{with} & \mathbf{w}\ x, \mathbf{r}\ x, \mathbf{r}\ !s, \mathbf{r}\ !p, \mathbf{w}\ ?s, \mathbf{w}\ ?p\ \mathbf{do} \\
 & & *[x := !s\ \mathbf{C}\ !p] \| *[?s := x] \| *[?p := x]\ \mathbf{end} \\
(\mathbf{5:\ LOOP}) & \mathbf{with} & \mathbf{r}\ !s, \mathbf{r}\ ?a, \mathbf{w}\ ?s, \mathbf{w}\ !a\ \mathbf{do} \\
 & & *[[!s \longrightarrow \mathbf{skip}]; *[\mathbf{AHS}(!a, ?a)]; \mathbf{PHS}(!s, ?s)]\ \mathbf{end} \\
 & \rhd_5 & \\
 & \mathbf{with} & \mathbf{r}\ !s, \mathbf{r}\ ?a, \mathbf{w}\ ?s, \mathbf{w}\ !a, \mathbf{r}\ x, \mathbf{w}\ x\ \mathbf{do} \\
 & & *[!a := !s \wedge \neg ?a]\ \mathbf{end} \\
(\mathbf{5:\ SKIP}) & \mathbf{with} & \mathbf{r}\ !s, \mathbf{w}\ ?s\ \mathbf{do}\ *[[!s \longrightarrow \mathbf{skip}]; \mathbf{skip}; \mathbf{PHS}(!s, ?s)]\ \mathbf{end} \\
 & \rhd_5 & \\
 & \mathbf{with} & \mathbf{r}\ !s, \mathbf{w}\ ?s\ \mathbf{do}\ *[?s := !s]\ \mathbf{end}
\end{array}
$$

Figure 6. Rewrite rules for Phase 5: Circuit Generation

$$
\begin{array}{l}
*[[\bar{P}? \longrightarrow x := \neg x; \\
\qquad [x \vee y \longrightarrow Q! [\!] \neg x \wedge \neg y \longrightarrow R!]; P?]] \\
\qquad\quad \Downarrow_1 \text{ by } (\mathbf{1 : SEQ}) \\
*[[\bar{P}? \longrightarrow S_1!; S_2!; P?]] \| \\
*[[\bar{S}_1? \longrightarrow x := \neg x; S_1?]] \| \\
*[[\bar{S}_2? \longrightarrow [x \vee y \longrightarrow Q! [\!] \neg x \wedge \neg y \longrightarrow R!]; S_2?]] \\
\qquad\quad \Downarrow_1 \text{ by } (\mathbf{1 : ASSN1}) \\
\qquad\quad \Downarrow_1 \text{ by } (\mathbf{1 : ASSN2}) \\
*[[\bar{P}? \longrightarrow S_1!; S_2!; P?]] \| \\
*[[\bar{S}_1? \longrightarrow [\neg\neg x \longrightarrow S_{\mathbf{f}}!; S_1? [\!] \neg x \longrightarrow S_{\mathbf{t}}!]; S_1?]] \| \\
*[[\bar{S}_{\mathbf{f}}? \longrightarrow x \downarrow; S_{\mathbf{f}}? [\!] \bar{S}_{\mathbf{t}}? \longrightarrow x \uparrow; S_{\mathbf{t}}?]] \| \\
*[[\bar{S}_2? \longrightarrow [x \vee y \longrightarrow Q! [\!] \neg x \wedge \neg y \longrightarrow R!]; S_2?]] \\
\qquad\quad \Downarrow_1 \text{ by } (\mathbf{1 : GD}) \\
*[[\bar{P}? \longrightarrow S_1!; S_2!; P?]] \| \\
*[[\bar{S}_1? \longrightarrow [\neg\neg x \longrightarrow S_{\mathbf{f}}!; S_1? [\!] \neg x \longrightarrow S_{\mathbf{t}}!]; S_1?]] \| \\
*[[\bar{S}_{\mathbf{f}}? \longrightarrow x \downarrow; S_{\mathbf{f}}? [\!] \bar{S}_{\mathbf{t}}? \longrightarrow x \uparrow; S_{\mathbf{t}}?]] \| \\
*[[\bar{S}_2? \longrightarrow [x \longrightarrow T_1!; S_2? [\!] y \wedge \neg x \longrightarrow T_1!; S_2? [\!] \\
\qquad\qquad\qquad\qquad \neg x \wedge \neg y \longrightarrow T_2!; S_2?]]] \| \\
*[[\bar{T}_1? \longrightarrow Q!; T_1?]] \| \\
*[[\bar{T}_2? \longrightarrow R!; T_2?]]
\end{array}
$$

Figure 7. Example phase 1 translation

$$
\begin{aligned}
&*[[\bar{P}? \longrightarrow S_1!; S_2!; P?]] \|\\
&*[[\bar{S}_1? \longrightarrow [\neg\neg x \longrightarrow S_{\mathbf{f}}!; S_1? [\!] \neg x \longrightarrow S_{\mathbf{t}}!]; S_1?]] \|\\
&*[[\bar{S}_{\mathbf{f}}? \longrightarrow x\downarrow; S_{\mathbf{f}}? [\!] \bar{S}_{\mathbf{t}}? \longrightarrow x\uparrow; S_{\mathbf{t}}?]] \|\\
&*[[\bar{S}_2? \longrightarrow [x \longrightarrow T_1!; S_2? [\!] y \wedge \neg x \longrightarrow T_1!; S_2? [\!]\\
&\qquad\qquad\qquad \neg x \wedge \neg y \longrightarrow T_2!; S_2?]]] \|\\
&*[[\bar{T}_1? \longrightarrow Q!; T_1?]] \|\\
&*[[\bar{T}_2? \longrightarrow R!; T_2?]]\\
&\qquad\quad \Downarrow_2^{15} \text{ by } (\mathbf{2:HS\ ACT}) \text{ and } (\mathbf{2:HS\ PAS})\\
&*[[!p \longrightarrow \mathbf{AHS}(!s_1, ?s_1); \mathbf{AHS}(!s_2, ?s_2); \mathbf{PHS}(!p, ?p)]] \|\\
&*[[!s_1 \longrightarrow [\neg\neg x \longrightarrow \mathbf{AHS}(!s_{\mathbf{f}}, ?s_{\mathbf{f}}); \mathbf{PHS}(!s_2, ?s_2) [\!]\\
&\qquad\quad \neg x \longrightarrow \mathbf{AHS}(!s_{\mathbf{t}}, ?s_{\mathbf{t}})]; \mathbf{PHS}(!s_2, ?s_2)]] \|\\
&*[[!s_{\mathbf{f}} \longrightarrow x\downarrow; \mathbf{PHS}(!s_{\mathbf{f}}, ?s_{\mathbf{f}}) [\!] \bar{!}s_{\mathbf{t}} \longrightarrow x\uparrow; \mathbf{PHS}(!s_{\mathbf{t}}, ?s_{\mathbf{t}})]] \|\\
&*[[!s_2 \longrightarrow [x \longrightarrow \mathbf{AHS}(!t_1, ?t_1); \mathbf{PHS}(!s_2, ?s_2) [\!]\\
&\qquad\quad y \wedge \neg x \longrightarrow \mathbf{AHS}(!t_1, ?t_1); \mathbf{PHS}(!s_2, ?s_2) [\!]\\
&\qquad\quad \neg x \wedge \neg y \longrightarrow \mathbf{AHS}(!t_2, ?t_2); \mathbf{PHS}(!s_2, ?s_2)]]] \|\\
&*[[!t_1 \longrightarrow \mathbf{AHS}(!q; ?q); \mathbf{PHS}(!t_1, ?t_1)]] \|\\
&*[[!t_2 \longrightarrow \mathbf{AHS}(!r; ?r); \mathbf{PHS}(!t_2, ?t_2)]]
\end{aligned}
$$

Figure 8. Example phase 2 (handshaking) translation

on P produce an alteration of synchronizations on Q and R, respectively, with y an external override in favor of Q.

$$
\begin{aligned}
&\textbf{with } P?,\ Q!,\ R!,\ \mathbf{r}\ x, \mathbf{w}\ x,\ \mathbf{r}\ y\ \textbf{do}\\
&\quad *[[\bar{P}? \longrightarrow\ x := \neg x;\\
&\qquad\qquad\quad [x \vee y \longrightarrow Q! [\!] \neg x \wedge \neg y \longrightarrow R!]; P?]]\\
&\textbf{end}
\end{aligned}
$$

Since the above module is not a component (y is assigned externally) we cannot establish its semantic well-formedness independently. In particular, if the module was used in an environment where y was set *while* $P!$ was synchronizing, this would produce an error, and it is not possible to say precisely whose fault the error was. This module thus should be used in an environment where y is set only when it is known $P!$ is not synchronizing. If the override y were removed, the specification would be a component, and we could establish its well-formedness.

We now present the compilation through the five phases.

Syntax-directed translation We begin instead by introducing two new processes by the sequencing transformation. The assignment steps create a separate assignment cell for the variable x, and replaces assignments to x with synchronizations with this cell. Finally, the guard step places the guards in disjoint disjunctive form, so at most one guard is true at a time. The result is in Figure 7.

Handshaking expansion Each step of rewriting replaces one port by its handshaking expansion. There are 15 ports, so 15 steps are required to change all ports to handshakes. This appears in figure 8.

$$
\begin{array}{l}
*[[!s_2 \longrightarrow [x \longrightarrow \mathbf{AHS}(!t_1, ?t_1); \mathbf{PHS}(!s_2, ?s_2) [\!]\\
\qquad\qquad y \wedge \neg x \longrightarrow \mathbf{AHS}(!t_1, ?t_1); \mathbf{PHS}(!s_2, ?s_2) [\!]\\
\qquad\qquad \neg x \wedge \neg y \longrightarrow \mathbf{AHS}(!t_2, ?t_2); \mathbf{PHS}(!s_2, ?s_2)]]]||\\
\qquad \Downarrow_3 \text{ by } (\mathbf{3:\ MOD\ ACT})\\
\qquad \Downarrow_3 \text{ by } (\mathbf{3:\ MOD\ PASS})\\
*[[!s_2 \longrightarrow [x \longrightarrow \mathbf{AHS}(!t_{1a}, ?t_{1a}); \mathbf{PHS}(!s_2, ?s_{2a}) [\!]\\
\qquad\qquad y \wedge \neg x \longrightarrow \mathbf{AHS}(!t_{1b}, ?t_{1b}); \mathbf{PHS}(!s_2, ?s_{2b}) [\!]\\
\qquad\qquad \neg x \wedge \neg y \longrightarrow \mathbf{AHS}(!t_2, ?t_2); \mathbf{PHS}(!s_2, ?s_{2c})]]]||\\
*[!t_1 :=!t_{1a} \vee !t_{1b}]||\\
*[?t_{1a} :=!t_{1a}\ \mathbf{C}\ ?t_1]||\\
*[?t_{1b} :=!t_{1b}\ \mathbf{C}\ ?t_1]||\\
*[?s_2 :=?s_{2a} \vee ?s_{2b} \vee ?s_{2c}]
\end{array}
$$

Figure 9. Example phase 3 (modularization) translation

$$
\begin{array}{l}
*[[!p \longrightarrow \mathbf{AHS}(!s_1, ?s_1); \mathbf{AHS}(!s_2, ?s_2); \mathbf{PHS}(!p, ?p)]]\\
\qquad \Downarrow_4 \text{ by } (\mathbf{4:\ SEQ})\\
*[[!p \longrightarrow \mathbf{skip}]; !s_1 \uparrow; [?s_1 \longrightarrow \mathbf{skip}], ?p \uparrow; [\neg !p \longrightarrow \mathbf{skip}];\\
\qquad\qquad !s_1 \downarrow; [\neg ?s_1 \longrightarrow \mathbf{skip}]; \mathbf{AHS}(!s_2, ?s_2); ?p \downarrow]\\
\qquad \Downarrow_5 \text{ by } (\mathbf{5:\ SEQ})\\
*[?s_1 :=!p]|| *[z :=?s_1\ \mathbf{C}\ \neg ?s_2]||\\
*[?p := z \vee ?s_2]|| *[!s_2 := z \wedge \neg ?s_1]
\end{array}
$$

Figure 10. Example phase 4 and 5 translation

Modularization The occurences of multiple active or passive synchronizations are the ones introduced by the guard translation, $\mathbf{AHS}(!t_1, ?t_1)$ and $\mathbf{PHS}(!s_2, ?s_2)$. For conciseness we only show changes to the relevant portion; the remainder of the process is unchanged. This appears in figure 9.

Reshuffling and Circuit generation For the remainder of the translation it is merely the act of substituting lower-level modules for higher-level ones. We only illustrate how the sequencing module is rewritten to a circuit. This appears in figure 10.

6. Correctness of the Translation Process

We may now put together the correctness results for each phase to establish the correctness of the compiler.

THEOREM 3 (COMPILER CORRECTNESS) For S-CSP module m_0, if $m_0 \Rightarrow_{1-5} m_5$ then $m_0 \overset{\Rrightarrow}{\cong}_{0-5} m_5$.

Proof: The proof follows by transitivity from the correctness lemmas for each phase, Lemmas 2, 3, 5, 6, and 7. ■

Also, the compilation rewriting is normalizing, so the compilation process will always terminate in a circuit.

LEMMA 8 If $m \in$ S-CSP, then $m \Rightarrow_{1-5} m_5$ for some $m_5 \in$ H-CSP.

Proof: See Appendix C. ■

In addition to the correctness of the compilation process, we may also establish that all semantically well-formed S-CSP components compile to hazard-free circuits.

COROLLARY 3 If a closed S-CSP component k_0 is semantically well-formed and $k_0 \Rightarrow_{1-5} k_5$ then the circuit k_5 is semantically well-formed. Thus, k_5 will exhibit no hazards.

Proof: The first part is direct from the definition of transformational equivalence and the definition of semantic well-formedness for components. To prove there are no hazards, recall a hazard is defined to occur when an input value on a gate is changed when the gate is enabled to change its output value, and the input change causes the gate to become disenabled. Then, this produces an **ERROR** in execution by the **reading while writing** case of Definition 11. Thus, since k_5 is semantically well-formed, it cannot have a hazard. ■

7. Conclusions

The primary goal of this work has been to provide a rigorous proof of the correctness of Martin's methodology. In the process of accomplishing this task, we obtained a number of results that are interesting in their own right.

1. We incorporate notions of separately compilable unit (module), and of a standalone unit that can be fabricated on silicon (component) into an asynchronous circuit specification and implementation language.
2. The semantics provides a definition of fair behavior of asynchronous circuits and asynchronous circuit specifications.
3. The semantics rigorously defines what a hazard is in a circuit, and what a mutual exclusion violation is in a specfication, via the general concept of *semantic well-formedness.*
4. We proved that all hazard-free, arbiter-free asynchronous circuits are observably deterministic, an important mathematical characterization of these circuits.
5. We define a novel notion of equivalence to justify the correctness of the compilation process, *transformation equivalence.*

6. We use a formal rewriting system with equational rewriting and multiple phases to rigorously define the compilation process.

This work can be extended in several directions. First, the language used is simple, to allow for an easier proof of correctness. One important extension is the incorporation of n-bit data paths via dual-rail encodings. Another is the implementation of non-mutually-exclusive guarded commands.

The transformation process we have presented is also minimal in that it incorporates no optimizations. Burns presents some simple optimizations in [5] and we expect that these optimizations as well as others can be incorporated into our framework. Once optimizing transformations are proven correct, the designer can manually apply the transformations without worrying about correctness, meaning there is a feasibility of using this framework to produce fast hand-optimized circuits that are nonetheless verifiably correct.

As hardware verification what we present is not necessarily a "complete" verification. We show that given a high-level CSP-style description, an equivalent circuit may be produced. Since CSP is only a programming language, some high-level specifications cannot be expressed concisely in this language. For a more complete verification effort, a logic (including quantification) could be developed and used to specify and prove high-level properties of the CSP-style specifications. Numerous such logics have been constructed [13], [21], [12], [11], [1], so this is an eminently feasible task. The advantage of this approach as opposed to a *post-hoc* verification methodology is the high-level specification is relatively simple in comparison with an actual circuit, making it easier to reason about.

Another important problem to solve is the development of decision procedures to automatically test for semantic well-formedness of S-CSP components. Because specifications may be quite large, it may be useful to use BDD's [3] to increase efficiency of the decision procedure. This approach has been used successfully in [4].

References

1. J.A. Bergstra and J.W. Klop. Process Algebra for Synchronous Communication. *Information and Control*, 60:109–137, 1984.
2. 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.
3. R.E. Bryant. On the complexity of VLSI Implementations and Graph Representation of Boolean Functions with Application to Integer Multiplication. *IEEE Transactions on Computers*, 40(2), 1986.
4. J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, 1990.
5. Steven M. Burns. Automated compilation of concurrent programs into self-timed circuits. Technical Report Caltech-CS-TR-88-2, California Institute of Technology, 1988.
6. 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.

7. N. Dershowitz and J.-P. Jouannaud. Rewriting systems. In *Handbook of theoretical computer science*. MIT/Elsevier, 1990.
8. David L. Dill. Trace theory for automatic hierarchical verification of speed-independent circuits. In Jonathan Allen and F. Thomson Leighton, editors, *Advanced Research in VLSI: Proceedings of the Fifth MIT Conference*, pages 51–65. MIT Press, 1988.
9. Jo C. Ebergen. A formal approach to designing delay-insensitive circuits. Computing Science Notes 88/10, Dept. of Math. and C.S., Eindhoven Univ. of Technology, May 1988.
10. M. Felleisen, D. Friedman, and E. Kohlbecker. A syntactic theory of sequential control. *Theoretical Computer Science*, 52:205–237, 1987.
11. M. Hennessy. *Algebraic Theory of Processes.* MIT Press, 1988.
12. Matthew Hennessy. Synchronous and asynchronous experiments on processes. *Inform. and Control*, 59:36–83, 1983.
13. C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.
14. 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.
15. Alain J. Martin. Compiling communicating processes into delay-insensitive VLSI circuits. *Distributed Computing*, 1(4):226–234, 1986.
16. 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.
17. 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.
18. Alain J. Martin. Synthesis of asynchronous VLSI circuits. In J. Straunstrup, editor, *Formal Methods for VLSI Design*, pages 237–283. North-Holland, 1990.
19. 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.
20. 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.
21. Robin Milner. *Communication and Concurrency.* Prentice Hall, 1989.
22. R. De Nicola and M.C.B. Hennessy. Testing Equivalences for Processes. *Theoretical Computer Science*, 34:83–133, 1983.
23. G. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:223–255, 1977.
24. 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.
25. Scott F. Smith and Amy E. Zwarico. Correct compilation of specifications to deterministic asynchronous circuits. In George Milne, editor, *Correct Hardware Design and Verification Methods (CHARME)*, volume 683 of *Lecture Notes in Computer Science.* Springer Verlag, 1993.
26. C. H. van Berkel. Beware the isochronic fork. Nat. Lab. Unclassified Report UR 003/91, Philips Research Lab., Eindhoven, The Netherlands, 1991.
27. Kees van Berkel. *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*, volume 5 of *International Series on Parallel Computation.* Cambridge University Press, 1993.
28. 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.
29. 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.

Appendix A

Fairness

In this section we give a full, formal definition of when a computation is *fair*.

First it is useful to provide a more abstract characterization of the erroring computations. 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 27

1. $\mathit{changes}(e, \langle c, \sigma\rangle \rightarrow \langle c', \sigma'\rangle)$ iff $\sigma(e) \neq \sigma'(e)$.
2. $\mathit{depends}(e, \langle R[c_1][c_2], \sigma\rangle \rightarrow \langle R[c_1'][c_2'], \sigma'\rangle)$ iff either

   (A) $\sigma(e) = \mathbf{t}$ and for all σ'', σ''', if $\sigma''(e) = \mathbf{f}$, then

   $$\langle R[c_1][c_2], \sigma''\rangle \not\rightarrow \langle R[c_1'][c_2'], \sigma'''\rangle;$$

   or

   (B) c_1 is $x := e$ and $\bullet_2$ does not appear in R.

Lemma 9 (Error characterization) $\varepsilon(\langle c, \sigma\rangle)$ (the configuration $\langle c, \sigma\rangle$ is in error) iff either

1. $\mathit{changes}(e, \langle c, \sigma\rangle \rightarrow \langle c', \sigma'\rangle)$ and $\mathit{depends}(e, \langle c, \sigma\rangle \rightarrow \langle c'', \sigma''\rangle)$, and $c' \neq c''$; or
2. $\mathit{changes}(e, \langle c, \sigma\rangle \rightarrow \langle c', \sigma'\rangle)$ and $\mathit{changes}(e, \langle c, \sigma\rangle \rightarrow \langle c'', \sigma''\rangle)$, 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 $\sigma(e_i) = \sigma(e_j) = \mathbf{t}$ for $j \neq i$; or
4. $c = R[x := e_1][x := e_2]$.

Now, some notation for expressing a configuration with two distinct reductions possible is justified.

Lemma 10 (dual reduction factoring) If $\langle c_0, \sigma_0\rangle$ is semantically well-formed and

$$\begin{tikzcd} & \langle c_0, \sigma_0\rangle \arrow[dl] \arrow[dr] & \\ \langle c_1, \sigma_1\rangle & & \langle c_2, \sigma_2\rangle \end{tikzcd}$$

then $c_0 = R[c_a][c_b][c_c][c_d]$, $c_1 = R[c_a'][c_b'][c_c][c_d]$, $c_2 = R[c_a][c_b][c_c'][c_d']$ for some R, c_a, c_b, c_c, c_d.

Proof: This factoring implies the two redices do not overlap, namely the two reductions involve modification of distinct subexpressions. Consider all the cases for the two reductions. There are two cases where overlapping redices can in principle occur. (Parallelism)(1) can overlap with (Parallelism)(2), and (Parallelism)(2) can overlap with itself. All the other rules cannot overlap because the outermost form of the redices is distinct, and reductions can never occur inside redices by the definition of a reduction context. Two different guard selections also are not possible by the precondition on the (Selection) rule that at most one guard is true.

Consider first the case of (Parallelism)(1) overlapping with (Parallelism)(2), where the two redices share the subterm $P!$. This can immediately be ruled out by the fact that (Parallelism)(1) requires $\sigma(P!) = \mathbf{f}$, and (Parallelism)(2) requires $\sigma(P!) = \mathbf{t}$. For (Parallelism)(2), there is a possibility of the single subterm $P!$ synchronizing with two separate occurrences of $P?$ in c_0, or $P?$ synchronizing with two separate occurrences of $P!$ in c_0. We address only the first subcase since one follows from the other by symmetry. Observe both of these reductions will have the effect of setting $P!$ to $\mathbf{f}$, so $\mathit{changes}(P!, \langle c_0, \sigma_0\rangle \to \langle c_1, \sigma_1\rangle)$ and $\mathit{changes}(P!, \langle c_0, \sigma_0\rangle \to \langle c_2, \sigma_2\rangle)$, and thus by error condition 2 $\langle c_0, \sigma_0\rangle$ is in error and not semantically well-formed, contradicting our assumption. ■

Definition 28 (Fairness) Given a semantically well-formed configuration $\langle c_0, \sigma_0\rangle$,

- A finite computation path

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

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

- An infinite computation path

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

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

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

Appendix B

Observational Determinism

In this section we present a complete proof of Observational Determinism, Theorem 1. The proof of observational determinism depends on the Strong Diamond Property (Lemma 11). It implies that well-formed computations are locally highly deterministic: if two different single step reductions are possible from a particular configuration a, producing configurations b and c, there is a configuration d that b and c may reach in a single step. It is in the spirit of the diamond propery for the untyped lambda calculus.

LEMMA 11 (STRONG DIAMOND) Given an arbitrary semantically well-formed configuration $\langle R[c_a][c_b][c_c][c_d], \sigma\rangle$, if

$$\begin{tikzcd}
 & \langle R[c_a][c_b][c_c][c_d], \sigma\rangle \arrow[dl] \arrow[dr] & \\
\langle R[c_a'][c_b'][c_c][c_d], \sigma[v_{01} = b_{01}]\rangle & & \langle R[c_a][c_b][c_c'][c_d'], \sigma[v_{10} = b_{10}]\rangle
\end{tikzcd}$$

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

$$\begin{tikzcd}
\langle R[c_a'][c_b'][c_c][c_d], \sigma[v_{01} = b_{01}]\rangle \arrow[dr] & & \langle R[c_a][c_b][c_c'][c_d'], \sigma[v_{10} = b_{10}]\rangle \arrow[dl] \\
 & \langle R[c_a'][c_b'][c_c'][c_d'], \sigma[v_{01} = b_{01}, v_{10} = b_{10}]\rangle &
\end{tikzcd}$$

(Note that if one of these steps does not change memory, it is the case that $\sigma[v_{ij} = b_{ij}] = \sigma$, so the state is not changed by that operation.)

Proof: The proof proceeds by case analysis on the two initial reductions. There are 7 single-step reduction rules, and all combinations must be considered, giving 28 cases to consider (reduced by symmetry from 49 cases). Since the intial configuration is semantically well-formed, we know it will never produce an **ERROR**. For each of the 28 cases, we show any violation of the strong diamond property implies the configuration can reduce to **ERROR**.

Case either rule is (Sequencing) or (Repetition) or (Parallelism)(3) (18 cases in all): These rules do not depend on or change the state, so their effect is entirely local, so in all cases the diamond diagram above can be completed.

Case $c_a = x_1 := e_1$, $c_c = x_2 := e_2$ (two Assignments): First consider the case where $x_1 = x_2$. By error condition 4 of 9, this will produce an error. Now consider when $x_1 \neq x_2$. Executing $x_1 := e_1$ does not change the value of e_2, for otherwise error condition 1 would hold. In particular,

$$\begin{aligned}
\mathit{depends}(e_2,\ & \langle R[x_1 := e_1][c_b][x_2 := e_2][c_d], \sigma\rangle \rightarrow \\
& \langle R[x_1 := e_1][c_b][\mathbf{skip}][c_d], \sigma[x_2 = \sigma(e_2)]\rangle)
\end{aligned}$$

by clause (b) of the definition of *depends*, and

$$\begin{aligned}
\mathit{changes}(e_2,\ & \langle R[x_1 := e_1][c_b][x_2 := e_2][c_d], \sigma\rangle \rightarrow \\
& \langle R[\mathbf{skip}][c_b][x_2 := e_2][c_d], \sigma[x_1 = \sigma(e_1)]\rangle)
\end{aligned}$$

if $\sigma(e_1) \neq \sigma(x_1)$. We thus may assume $\sigma(e_1) = \sigma(x_1)$, so no change in the state occurs. We may apply this argument by symmetry to show executing $x_2 := e_2$ does not change the value of e_1. Thus, executing the two in either order will lead to the same configuration, namely

$$\langle R[\mathbf{skip}][c_b][\mathbf{skip}][c_d], \sigma[x_1 = \sigma(e_1), x_2 = \sigma(e_2)]\rangle.$$

Case $c_a = x := e$, $c_c = [e_1 \longrightarrow c_1 \| \ldots \| e_i \longrightarrow c_i \| \ldots\ldots \| e_n \longrightarrow c_n]$ (Assignment and Selection of guard i): The selection rule does not change the state, so the value assigned to x will not be affected by execution of the guard. The only problem is if $\sigma(e_i)$ becomes false by execution of the assignment statement. In this case, e_i *depends* on the guard execution step, so since e_i *changes* by the assignment, error condition 1 produces an error. In more detail,

$$\begin{aligned} \mathit{depends}(e_i,\ & \langle R[x_1 := e_1][c_b][[e_1 \longrightarrow c_1 \| \ldots \| e_i \longrightarrow c_i \| \ldots\ldots \| e_n \longrightarrow c_n]][c_d], \sigma\rangle \rightarrow \\ & \langle R[x_1 := e_1][c_b][c_i][c_d], \sigma\rangle) \end{aligned}$$

because this step will not happen for any σ'' if $\sigma''(e_i) = \mathbf{f}$.

Case (Assignment) with (Parallel)(1) or (Parallel)(2) (two cases): The only possible dependency is if $\bar{P}?$ occurs free in the assignment body e, which is not allowed syntactically. Therefore, there can be no dependency.

Case Two (Selection) rules: If two different guards in the same guarded command were firing, it would violate error condition 2. So, the two guarded commands must be different and neither reduction changes the state, so they may be executed in any order.

Case (Selection) with (Parallel)(1) or (Parallel)(2) (two cases): The only conflict possible is if the true guard e_i contains $\bar{P}?$, but if (Parallel)(1) or (Parallel)(2) execution *changes* the value of e_i to $\mathbf{f}$, error condition 1 of 9 results.

Case $c_a = P_1!$, $c_c = P_2!$, (Parallel)(1) with (Parallel)(1): If $P_1! = P_2!$, both rules *changes* $P_1!$ and error condition 2 results. Otherwise, the two are independent since by Lemma 10 redices cannot overlap.

Case $c_a = P_1!$, $c_c = P_2!$, $c_d = P_2?$, (Parallel)(1) with (Parallel)(2): If $P_1! \neq P_2!$, they are obviously independent; if $P_1! = P_2!$, one changes $P_1!$ to $\mathbf{t}$, the other changes it to $\mathbf{f}$, contradicting error condition 2.

Case $c_a = P_1!$, $c_b = P_1?$, $c_c = P_2!$, $c_d = P_2?$, (Parallel)(2) with (Parallel)(2): If $P_1! = P_2!$, both rules change $P_1!$ and error condition 2 results. Otherwise, the two are independent.

That completes the 28 possible cases. ■

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 12 (Progress) Given a semantically well-formed configuration $\langle c_0, \sigma_0\rangle$ 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

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

then there exists some n such that

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

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. Suppose not, i.e. the enabled redex is disabled first in step i. Then,

$$\langle R_i[c_a][c_b][c_{1i}][c_{2i}], \sigma_i\rangle \rightarrow \langle R_i[c_a][c_b][c'_{1i}][c'_{2i}], \sigma_{i+1}\rangle$$

and

$$\langle R_i[c_a][c_b][c_{1i}][c_{2i}], \sigma_i\rangle \rightarrow \langle R_i[c'_a][c'_b][c_{1i}][c_{2i}], \sigma'_i\rangle.$$

and

$$\langle R_i[c_a][c_b][c'_{1i}][c'_{2i}], \sigma_{i+1}\rangle \not\rightarrow \langle R_i[c'_a][c'_b][c'_{1i}][c'_{2i}], \sigma'\rangle$$

However, the strong diamond property directly implies that the third reduction above must follow from the first two, a contradiction. ■

The next Lemma uses the Strong Diamond Lemma (Lemma 11) 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 13 (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

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

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

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

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,

$$\begin{array}{c}
\langle R_0[c_a][c_b][c_{10}][c_{20}], \sigma_0\rangle \rightarrow \langle R_0[c'_a][c'_b][c_{10}][c_{20}], \sigma_0[v=b]\rangle \rightarrow \\
\langle R_1[c'_a][c'_b][c_{11}][c_{21}], \sigma_1[v=b]\rangle \xrightarrow{*} \langle R_n[c'_a][c'_b][c_{1n}][c_{2n}], \sigma_n[v=b]\rangle
\end{array}$$

Proof: Proceed by induction on n. For the case $n = 1$, the desired result is exactly the Strong Diamond Property (11). 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 (12),

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

By the induction hypothesis, we then have

$$(1)\ \ \langle R_0[c'_a][c'_b][c'_{10}][c'_{20}], \sigma_2[v=b]\rangle \xrightarrow{*} \langle R_n[c'_a][c'_b][c_{1n}][c_{2n}], \sigma_n[v=b]\rangle.$$

So, by the Strong Diamond Property (11), the first two steps

$$\begin{array}{l}
\langle R_0[c_a][c_b][c_{10}][c_{20}], \sigma_0\rangle \rightarrow \\
\langle R_0[c_a][c_b][c'_{10}][c'_{20}], \sigma_1\rangle \rightarrow \\
\langle R_0[c'_a][c'_b][c'_{10}][c'_{20}], \sigma_1[v=b]\rangle
\end{array}$$

may be swapped to give

$$\begin{array}{l}
\langle R_0[c_a][c_b][c_{10}][c_{20}], \sigma_0\rangle \rightarrow \\
\langle R_0[c'_a][c'_b][c_{10}][c_{20}], \sigma_0[v=b]\rangle \rightarrow \\
\langle R_0[c'_a][c'_b][c'_{10}][c'_{20}], \sigma_1[v=b]\rangle,
\end{array}$$

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

We now have all the apparatus in place to prove the observational determinism theorem, Theorem 1.

Proof: Suppose we have successful computation path

$$\langle c_0, \sigma_0\rangle \rightarrow \langle c_1, \sigma_1\rangle \rightarrow \ldots \rightarrow \langle c_{m-1}, \sigma_{m-1}\rangle \rightarrow \langle c_m, \sigma_{m-1}[x_{\text{success}} = \mathbf{t}]\rangle,$$

where $\sigma_i(x_{\text{success}}) = \mathbf{f}$ for all $i < m$. We show by induction on k for $k \leq m$ that any computation path that shares the first $m - k$ steps with the above successful path is successful.

The base case of $k = 0$ is immediate. Inductively assume this fact for $k - 1$, show for k provided $k \leq m$. Letting $l = m - k$, given any failing path that shares exactly l steps with the above,

$$\langle c_0, \sigma_0\rangle \rightarrow \langle c_1, \sigma_1\rangle \rightarrow \ldots \rightarrow \langle c_l, \sigma_l\rangle \rightarrow \langle c'_{l+1}, \sigma'_{l+1}\rangle \rightarrow \ldots \langle c'_n, \sigma'_n\rangle \rightarrow \ldots,$$

we show it in fact cannot be failing. Consider the $l+1$-st step of both the successful and failing computation: notating $c_l = R_l[c_a][c_b][c_{1l}][c_{2l}]$, we have

$$\begin{tikzcd}
 & \langle R_l[c_a][c_b][c_{1l}][c_{2l}], \sigma_l\rangle \arrow[dl, "successful"] \arrow[dr, "failing"] & \\
\langle R_l[c'_a][c'_b][c_{1l}][c_{2l}], \sigma_l[v = b]\rangle & & \langle R_l[c_a][c_b][c_{1(l+1)}][c_{2(l+1)}], \sigma'_{l+1}\rangle
\end{tikzcd}$$

where $(\sigma_l[v = b]) = \sigma_{l+1}$. By the Progress Lemma, 12, we know the c_a/c_b redex stays enabled in the failing path until it is eventually executed in some step $n + 1$:

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

where $\sigma'_l = \sigma_l$. This is the precondition to apply the Bubbling Lemma (13), and this gives us

$$\begin{array}{c}
\langle R_l[c_a][c_b][c_{1l}][c_{2l}], \sigma_l\rangle \rightarrow \langle R_l[c'_a][c'_b][c_{1l}][c_{2l}], \sigma_l[v = b]\rangle \rightarrow \\
\langle R_{l+1}[c'_a][c'_b][c_{1(l+1)}][c_{2(l+1)}], \sigma'_{l+1}[v = b]\rangle \xrightarrow{*} \langle R_n[c'_a][c'_b][c_{1n}][c_{2n}], \sigma'_n[v = b]\rangle \rightarrow \ldots
\end{array}$$

Note that since the original failing computation path did not set x_{success} to $\mathbf{t}$, the bubbled path also will not since $v = b$ cannot be $x_{\text{success}} = \mathbf{t}$ by the fact that the first such setting occurs at step m. Other than that the states of the two paths are identical.

Thus, we still have a failing computation, but note it has one more step in common with the successful computation (the l-th step in particular), so by our induction hypothesis it cannot be a failing path, contradiction.

Thus, any path that shares the first $l = m - k$ steps with the successful path is successful. In particular, for the case $k = m$, any path that shares no initial steps with the successful path is also successful, in other words all paths are successful.

Thus, it cannot be the case that some paths are successful and others are failing, so either all paths are successful or all paths are failing. ■

Appendix C
Normalization Results

In this Appendix we presents the proof of normalization for the five rewrite systems, justifying successful termination of the rewriting compiler.

LEMMA 14 (NORMALIZATION) If m_{i-1} is a module in phase $i-1$ normal form, then there exists a module m_i such that $m_{i-1} \Rightarrow_i^N m_i$ and *strip*(m_i) is in phase i normal form, for $1 \leq i \leq 5$.

(Note we declare the original S-CSP specification to vacuously be in phase 0 normal form.)

Proof: **Phase 1:** Before proving that the rules for Phase 1 produce normal forms, we define the depth of a term. **skip**, $x := e$, $P!$, $P?$ are all of depth 1. Let $m_1, \ldots, m_n, m'$ be of depth n, then $m_1 \| \ldots \| m_n$, $m_1; m_2$, $*[m']$ and $[e_1 \longrightarrow m_1 [\!] \ldots [\!] e_n \longrightarrow m_n]$ are all of depth $n+1$.

We assume that $(\mathbf{1 : INIT})$ has already been applied. We then prove that for all S-CSP terms m, $S! \| *[[\bar{S}? \longrightarrow m; S?]]$ can be converted to a phase 1 normal form. The proof of the existence of normal forms is by induction on the structure of m. In this proof, we drop all scoping to simplify the notation. For the base case we consider m consisting of a single instruction. There are four subcases.

1. If $m = \mathbf{skip}$ then $S! \| *[[\bar{S}? \longrightarrow \mathbf{skip}; S?]]$ is already in normal form.

2. If $m = x := e$ then by $(\mathbf{1 : ASSN1,2})$,

$$\begin{aligned} S! \| *[[\bar{S}? \longrightarrow x := e; S?]] \;\Rightarrow_1^2\; & S! \| *[[\bar{S}? \longrightarrow [\neg e \longrightarrow S_0!; S? [\!] e \longrightarrow S_1!; S?]]] \| \\ & *[[\bar{S}_0? \longrightarrow x \downarrow; S_0? [\!] \bar{S}_1? \longrightarrow x \uparrow; S_1?]] \end{aligned}$$

   which is in phase 1 normal form.

3,4. If $m = P\$$ where $\$ \in \{!, ?\}$, then $S! \| *[[\bar{S}? \longrightarrow P\$; S?]]$ is already in normal form.

Thus the base case holds. Furthermore, no additional rules from phase 1 can be applied to any of the final forms so the form is unique up to the commutativity of parallel composition and choice.

For the inductive hypothesis we assume that for all S-CSP terms m' of depth n there exists a term m_{nf} in phase 1 normal form such that $S! \| *[[\bar{S}? \longrightarrow m'; S?]] \Rightarrow_1^N m_{nf}$.

Suppose m is of depth $n+1$. Then we must consider the operators that increase the depth of a term: sequential composition, parallel composition, looping and choice. In each case we assume that $m_1, \ldots, m_m, m'$ are of depth at most n.

1. If $m = m_1; m_2$ and both m_1, m_2 are distinguished active synchronizations, then $S!|| * [[\bar{S}? \longrightarrow m_1; m_2; S?]]$ is in phase 1 normal form. Otherwise, neither m_1 nor m_2 are distinguished synchronizations and by application of (**1 : SEQ**) we obtain
$$\begin{aligned} S!|| * [[\bar{S} \longrightarrow m_1; m_2; S?]] \ \Rightarrow_1 \ & S!|| * [[\bar{S} \longrightarrow S_1!; S_2!; S?]]|| \\ & *[[\bar{S}_1? \longrightarrow m_1; S_1?]]|| * [[\bar{S}_2? \longrightarrow m_2; S_2?]] \end{aligned}$$
$*[[\bar{S} \longrightarrow S_1!; S_2!; S?]]$ conforms to one of the possible nf' terms from the definition of the phase 1 normal form. Since m_1, m_2 are of depth at most n, all $S_i!|| * [[\bar{S}_i? \longrightarrow m_i; S_i?]]$ $(i = 1, 2)$ have phase 1 normal forms, by the inductive hypothesis. Then, by dropping each singleton process $S_i!$, we obtain a phase 1 normal form for m.

2. If $m = m_1|| \dots ||m_n$ and all m_i are distinguished active synchronizations, then we are done. Otherwise, none of the m_i are distinguished active synchronizations and using (**1 : PAR**) we obtain
$$S!||*[[\bar{S} \longrightarrow (S_1!|| \dots ||S_n!); S?]]||*[[\bar{S}_1? \longrightarrow m_1; S_1?]]|| \dots ||*[[\bar{S}_n? \longrightarrow m_n; S_n?]]$$
$*[[\bar{S} \longrightarrow (S_1!|| \dots ||S_n!); S?]]$ is already one of the nf' forms for phase 1, and since all m_i are of depth at most n, for all $1 \leq i \leq n$ $S_i!|| * [[\bar{S}_i? \longrightarrow m_i; S_i?]]$ has a normal form, by the inductive hypothesis. Again, by dropping each process $S_i!$, we obtain a phase 1 normal form for m.

3. If $m = *[m']$ and m' is a distinguished active synchronization, then $S!||*[[\bar{S} \longrightarrow *[m']; S?]]$ is in normal form. If not, then by (**1 : LOOP**) we obtain $S!||*[[\bar{S} \longrightarrow *[S'!]; S?]]||*[[\bar{S'}? \longrightarrow m'; S'?]]$. Since m' is of depth at most n, by the inductive hypothesis $S'!||*[[\bar{S'}? \longrightarrow m'; S'?]]$ has a phase 1 normal form, and by dropping the singleton process $S'!$, we obtain a phase 1 normal form for m.

4. If $m = [e_1 \longrightarrow m_1 [\!] \dots [\!] e_n \longrightarrow m_n]$, all m_i are distinguished active synchronizations and none of the e_i can be further simplified by the disjoint guards algorithm, then $S!|| * [[\bar{S}? \longrightarrow m; S?]]$ is in normal form. Otherwise, we apply (**1 : GD**) to obtain
$$\begin{aligned} S!|| * [[\bar{S}? \longrightarrow [\ & e_{1_1} \longrightarrow S_1!; S? [\!] \dots [\!] e_{1_{k_1}} \longrightarrow S_1!; S? [\!] \dots [\!] \\ & e_{n_1} \longrightarrow S_n!; S? [\!] \dots [\!] e_{n_{k_n}} \longrightarrow S_n!; S?]]]|| \\ *[[\bar{S}_1? \longrightarrow m_1; S_1?]]|| \dots || * [[\bar{S}_n? \longrightarrow m_n; S_n?]] \end{aligned}$$
Since each m_i is of depth at most n, by the inductive hypothesis, each $S_i!|| * [[\bar{S}_i? \longrightarrow m_i; S_i?]]$ has a phase 1 normal form, and by dropping each process $S_i!$ we obtain a phase 1 normal form for m

Phase 2: To show that each process in phase 1 normal form can be converted to a process in phase 2 normal form, we observe that the phase 2 normal form only differs from the phase 1 normal form in that each occurrence of an active (passive) synchronization in replaced by a active (passive) handshake. The proof then proceeds by induction on the number of synchronizations on distinct ports in the term. For the base case, suppose m contains synchronizations on one port, without loss of generality, $P!$. Writing m as $C[P!]$, we then apply (**2 : HS ACT**) once to obtain $m_2 = C[\mathbf{AHS}(!p, ?p)]$. Since m was already in phase 1 normal form, m_2 must be in phase 2 normal form and no other handshaking expansions can be done. The case for a passive synchronization is identical except that we apply (**2 : HS PAS**) instead.

For the inductive hypothesis we must define an intermediate normal form by combining the phase 1 and phase 2 normal forms as follows.

$$\begin{array}{lcl} nf & ::= & nf_1 | nf_2 \\ nf_1 & ::= & S! || nf'_1 || \ldots || nf'_n | \mathbf{AHS}(!s, ?s) || nf'_1 || \ldots || nf'_n | \\ nf' & ::= & nf''_1 | nf''_2 \\ nf''_1 & ::= & nf' \text{ from the definition of the phase 1 normal form} \\ nf''_2 & ::= & nf' \text{ from the definition of the phase 2 normal form} \end{array}$$

For the inductive hypothesis we then assume that if m is in the above intermediate normal form and m contains synchronizations on n distinct ports, then there exists an m' in phase 2 normal form such that $m \Rightarrow_2^N m'$.

Suppose m contains synchronizations on $n+1$ distinct ports. There are two cases to consider, active synchronizations and passive synchronizations.

1. If there is an active synchronization on $P!$ we can write m as $C[P!]$ and by application of (**2 : HS ACT**), $C[P!] \Rightarrow_2 C[\mathbf{AHS}(!p, ?p)]$. Now $C[\mathbf{AHS}(!p, ?p)]$ contains synchronizations on n distinct ports and is in the intermediate normal form. Thus, by the inductive hypothesis, there exists an m' in phase 2 normal form such that $C[\mathbf{AHS}(!p, ?p)] \Rightarrow_2^N m'$. Combining these two results, we obtain $m \Rightarrow_2 C[\mathbf{AHS}(!p, ?p)] \Rightarrow_2^N m'$, where m' is in phase 2 normal form and no other phase 2 rules can be applied.

2. The case of a passive synchronization on $P?$ is identical except that we apply (**2 : HS PAS**).

Phase 3: The proof that phase 3 is normalizing is almost identical to that for phase 2. The only difference is that the intermediate normal form is that of phase 2 with the addition of the terms $*[!p :=!p_1 \vee \ldots \vee !p_n]$, $*[?p :=?p_1 \vee \ldots \vee ?p_n]$ and $*[?p_i :=!p_i \ \mathbf{C}\ ?p]$, and we apply (**3 : MOD ACT**) and (**3 : MOD PAS**). Thus we do not include the proof.

Phase 4: The proof that reshuffling is normalizing is by induction on the number of occurrences of subterms that can be reshuffled. For the base case we assume that m in phase 3 normal form contains exactly one subterm that may be reshuffled. There are five cases corresponding to the five different reshuffling rules. Each case requires one application of the appropriate rule to obtain a term in phase 4 normal form.

For the inductive hypothesis we again require an intermediate normal form, much like the one used in the proof for phase 2. This form is defined by the following BNF grammar.

$$\begin{array}{lll} nf & ::= & \mathbf{AHS}(!s, ?s)||nf'_1|| \dots ||nf'_m \\ nf' & ::= & nf''_3 | nf''_4 \\ nf''_3 & ::= & nf' \text{ from the definition of the phase 3 normal form} \\ nf''_4 & ::= & nf' \text{ from the definition of the phase 4 normal form} \end{array}$$

For the inductive hypothesis we assume that if m is a term in the above intermediate form containing at most n occurrences of subterms that can be reshuffled, then there exists a term m' in phase 4 normal form such that $m \Rightarrow_4^N m'$.

Let m be a term in phase 3 normal form containing $n+1$ occurrences of subterms that can be reshuffled. We write m as $nf'_1|| \dots ||nf'_{n+k}$, $k > 1$. As in the base case, we must consider five cases. Since parallel composition is associative and commutative, we always assume that the term to be reshuffled is in nf'_1.

1. If the term to be reshuffled is of the form $*[[!s \longrightarrow \mathbf{AHS}(!a, ?a); \mathbf{PHS}(!s, ?s)]]$, we apply $(\mathbf{4 : ACT})$ obtaining a term of the form m'' where

$$\begin{array}{ll} m'' = *[& [!s \longrightarrow \mathbf{skip}]; !a\uparrow; [?a \longrightarrow \mathbf{skip}]; ?s\uparrow; \\ & [\neg !s \longrightarrow \mathbf{skip}]; !a\downarrow; [\neg ?a \longrightarrow \mathbf{skip}]; ?s\downarrow] \; ||nf'_2|| \dots ||nf'_n. \end{array}$$

   m'' contains n occurrences of subterms that can be reshuffled, and by the inductive hypothesis, there exists a term m' in phase 4 normal form such that $m'' \Rightarrow_4^N m'$. Combining the two results, $m \Rightarrow_4 m'' \Rightarrow_4^N m'$, and m' cannot be further reduced.

2,3,4,5. These cases are essentially the same except that we apply $(\mathbf{4 : SEQ})$, $(\mathbf{4 : PAR})$, $(\mathbf{4 : PASS})$, $(\mathbf{4 : GD})$ respectively.

Phase 5: The proof for normalization of phase 5 is similar to that for phase 4, differing in the intermediate normal form (here a combination of the normal forms of phases 4 and 5), and the terms to be simplified. Thus we omit the proof.

■

Appendix D

Correctness Proofs

In this Appendix we present the proofs of correctness of the five translation phases. We present all the relevant styles of argument at least once in detail, and give somewhat less detail in subsequent uses. Correctness of the handshaking expansion phase, phase 2, is short enough that a fully detailed argument for that phase may be presented, and it appears first. The other phases are all proved using a similar basic technique, so in those phases a skeleton of a proof is given, and the rest can be reconstructed by inspection of the phase 2 proof.

There is one common proof technique that the proofs for each phase use. We define a series of bisimulation relations $\approx$ that correlate the steps of computation of the lhs of a rewrite rule with the steps of computation of the rhs of the rule. These relations are the core of the correctness proofs, they show how as a computation using the lhs proceeds, the rhs computation may proceed in similar fashion, with the $\approx$ relation defining precisely what "similar" is. In particular, we first prove two computational lemmas by induction on the $\approx$ relation: the first establishes that if we start with related lhs and rhs, for any single step of computation performed by the left-hand side, there is a sequence of computation steps that may be performed by the right-hand side that return the two to related states. The second lemma is a converse to this that from an rhs step constructs lhs steps, and additionaly must take **ERROR** states into account. This establishes that $\approx$ is a bisimulation.

Next we prove two Lemmas that verify we are in fact relating terms by $\approx$ that should be related. First, $\approx$-related terms must have identical values for x_{success} (Success Lemma) and an **ERROR** in the right-hand side must imply an error in the left-hand side (Error Lemma).

Combining these four lemmas, for each phase we then prove that the lhs and rhs have the same observations for all computations, and this leads to a proof of correctness of the phase.

D.1. Handshake Expansion Correctness

For the purpose of proofs, it is very useful to have a "mixed" notion of context M, containing some holes known to be at reduction points as in a reduction context R, and other holes occurring possibly multiply and at arbitrary points, as in normal contexts C. In order to distinguish the two classes of holes, substitution into reduction context holes will be indicate as before, and substitution into normal holes will be subscripted.

Definition 29 M denotes a *mixed context* with m holes, where holes $\bullet_1, \ldots \bullet_n$ are reduction context holes, and holes $\bullet_{n+1}, \ldots \bullet_m$ are arbitrary holes. We denote the substitution of each c_i into $\bullet_i$ for $0 \leq i \leq m$ by $M_{[c_{n+1}]\cdots[c_m]}[c_1]\ldots[c_n]$.

Theorem 4, the main theorem used to establish handshake correctness, shows one channel may be replaced by its handshake expansion in a closed expression. The proof is postponed until later in this section. In order to reduce notation, declarations will be taken to be implicit, and the development proceeds for fixed $P!, P?, !p, ?p$, so **AHS**$(!p, ?p)$ and **PHS**$(!p, ?p)$ will be abbreviated **AHS** and **PHS**.

Theorem 4 For closed term $C[P!][P?][\bar{P}?]$ for C containing no instances of $P!$ or $P?$ or $\bar{P}?$,

$$C[P!][P?][\bar{P}?] \overset{\rightrightarrows}{\cong}_{obserr} C[\mathbf{AHS}][\mathbf{PHS}][!p]$$

Relations $\approx_{hs}$ and $\approx_{hs+}$ are defined to relate intermediate computation states of channel-based synchronizations with their handshake expansions. $\approx_{hs}$ is used to get from computations involving channels to computations involving handshakes, and $\approx_{hs+}$ is used to go from handshakes to channels. We then show below, in Lemmas 15 and 16, that computing preserves these relations. These facts are then used to establish the main Theorem above.

DEFINITION 30 $\approx_{hs}$ is the least relation with the following properties, for arbitrary M, C containing no occcurrences of $P!, P?, \bar{P}?, !p, ?p$:

1. $\langle C[P!][P?][\bar{P}?], \sigma[P! = \mathbf{f}]\rangle \approx_{hs}$
   $\langle C[\mathbf{AHS}][\mathbf{PHS}][!p], \sigma[!p = \mathbf{f}, ?p = \mathbf{f}]\rangle$
2. $\langle M_{[P!][P?][\bar{P}?]}[P!], \sigma[P! = \mathbf{t}]\rangle \approx_{hs}$
   $\langle M_{[\mathbf{AHS}][\mathbf{PHS}][!p]}[\mathbf{AHS}_1], \sigma[!p = \mathbf{t}, ?p = \mathbf{f}]\rangle$

$\approx_{hs+}$ is the least relation extending $\approx_{hs}$ that satifies in addition the following:

3. $\langle M_{[P!][P?][\bar{P}?]}[P!][P?], \sigma[P! = \mathbf{t}]\rangle \approx_{hs}$
   $\langle M_{[\mathbf{AHS}][\mathbf{PHS}][!p]}[\mathbf{AHS}_1][\mathbf{PHS}_1], \sigma[!p = \mathbf{t}, ?p = \mathbf{f}]\rangle$
4. $\langle M_{[P!][P?][\bar{P}?]}[P!][P?], \sigma[P! = \mathbf{t}]\rangle \approx_{hs}$
   $\langle M_{[\mathbf{AHS}][\mathbf{PHS}][!p]}[\mathbf{AHS}_1][\mathbf{PHS}_2], \sigma[!p = \mathbf{t}, ?p = \mathbf{t}]\rangle$
5. $\langle M_{[P!][P?][\bar{P}?]}[P!][P?], \sigma[P! = \mathbf{t}]\rangle \approx_{hs}$
   $\langle M_{[\mathbf{AHS}][\mathbf{PHS}][!p]}[\mathbf{AHS}_2][\mathbf{PHS}_2], \sigma[!p = \mathbf{t}, ?p = \mathbf{t}]\rangle$
6. $\langle M_{[P!][P?][\bar{P}?]}[P!][P?], \sigma[P! = \mathbf{t}]\rangle \approx_{hs}$
   $\langle M_{[\mathbf{AHS}][\mathbf{PHS}][!p]}[\mathbf{AHS}_3][\mathbf{PHS}_2], \sigma[!p = \mathbf{f}, ?p = \mathbf{t}]\rangle$
7. $\langle M_{[P!][P?][\bar{P}?]}[P!][P?], \sigma[P! = \mathbf{t}]\rangle \approx_{hs}$
   $\langle M_{[\mathbf{AHS}][\mathbf{PHS}][!p]}[\mathbf{AHS}_3][\mathbf{PHS}_3], \sigma[!p = \mathbf{f}, ?p = \mathbf{t}]\rangle$
8. $\langle M_{[P!][P?][\bar{P}?]}[P!][P?], \sigma[P! = \mathbf{t}]\rangle \approx_{hs}$
   $\langle M_{[\mathbf{AHS}][\mathbf{PHS}][!p]}[\mathbf{AHS}_3][\mathbf{skip}], \sigma[!p = \mathbf{f}, ?p = \mathbf{f}]\rangle$
9. $\langle M_{[P!][P?][\bar{P}?]}[P!]\ldots[P!], \sigma[P! = \mathbf{t}]\rangle \approx_{hs}$
   $\langle M_{[\mathbf{AHS}][\mathbf{PHS}][!p]}[\mathbf{AHS}_1]\ldots[\mathbf{AHS}_1], \sigma[!p = \mathbf{t}, ?p = \mathbf{f}]\rangle$

where we abbreviate the intermediate stages in handshake protocols as follows

$$\begin{aligned}
\mathbf{AHS}_1 &= [?p \longrightarrow !p\downarrow]; [\neg ?p \longrightarrow \mathbf{skip}] \\
\mathbf{AHS}_2 &= !p\downarrow; [\neg ?p \longrightarrow \mathbf{skip}] \\
\mathbf{AHS}_3 &= [\neg ?p \longrightarrow \mathbf{skip}] \\
\mathbf{PHS}_1 &= ?p\uparrow; [\neg !p \longrightarrow ?p\downarrow] \\
\mathbf{PHS}_2 &= [\neg !p \longrightarrow ?p\downarrow] \\
\mathbf{PHS}_3 &= ?p\downarrow
\end{aligned}$$

We now prove two lemmas that show the $\approx_{hs}$ relation is preserved by computation.

LEMMA 15 If given semantically well-formed configurations $\langle c_0, \sigma_0\rangle$ and $\langle c'_0, \sigma'_0\rangle$ we have

$$\begin{tikzcd} \langle c_0,\sigma_0\rangle \arrow[r] \arrow[d, phantom, "\wr\wr_{hs}"] & \langle c_1,\sigma_1\rangle \\ \langle c'_0,\sigma'_0\rangle & \end{tikzcd}$$

then we may find $c'_i, \sigma'_i, 0 \leq i \leq m$, such that

$$\begin{tikzcd} \langle c_0,\sigma_0\rangle \arrow[r] \arrow[d, phantom, "\wr\wr_{hs}"] & \langle c_1,\sigma_1\rangle \arrow[d, phantom, "\wr\wr_{hs}"] \\ \langle c'_0,\sigma'_0\rangle \arrow[r, "*"] & \langle c'_m,\sigma'_m\rangle \end{tikzcd}$$

Proof: First, observe that any step uniform in the holes (namely the result is the same whatever is placed in the holes of C or M) trivially preserves $\approx_{hs}$ when both c_0 and c'_0 make this single step. Second, observe that any $\approx_{hs}$-related configurations always produce the same values for boolean subterms—all occurences of $\bar{P}?$ on the left are replaced by $!p$ on the right, and for each case of $\approx_{hs}$ $\bar{P}?$ and $!p$ can be seen by inspection to have the same boolean value. Thus, any step that depends on a boolean expression involving $\bar{P}$ and $!p$ proceeds uniformly and $\approx_{hs}$ is preserved. We may then focus on cases that depend on the non-boolean values in the holes. Proceed by cases on the assumption $\langle c_0, \sigma_0\rangle \approx_{hs} \langle c'_0, \sigma'_0\rangle$.
Case 1.: Consider the next step of $\langle c_0, \sigma_0\rangle = \langle C[P!][P?][\bar{P}?], \sigma[P! = \mathbf{f}]\rangle$. By inspection of the rules, the only rule involving $P!$ or $P?$ that could apply is (Parallelism)(1). In that case, for some $M_{[P!][P?][\bar{P}?]}[P!] = C[P!][P?][\bar{P}?]$ a single step places us in the configuration $\langle M_{[P!][P?][\bar{P}?]}[P!], \sigma[P! = \mathbf{t}]\rangle$, precisely the left-hand-side of case 2. of $\approx_{hs}$. We may complete the square by the computation

$$\begin{aligned} \langle c'_0, \sigma'_0\rangle = \; & \langle M_{[\mathbf{AHS}][\mathbf{PHS}][!p]}[\mathbf{AHS}], \sigma[!p = \mathbf{f}, ?p = \mathbf{f}]\rangle \rightarrow \\ & \langle M_{[\mathbf{AHS}][\mathbf{PHS}][!p]}[\mathbf{AHS}_1], \sigma[!p = \mathbf{t}, ?p = \mathbf{f}]\rangle \end{aligned}$$

Case 2.: From $\langle M_{[P!][P?][\bar{P}?]}[P!], \sigma[P! = \mathbf{t}]\rangle$, the only rule that involves $P!$ or $P?$ at this point is synchronization via (Parallelism)(2):

$$\langle M'_{[P!][P?][\bar{P}?]}[P!][P?], \sigma[P! = \mathbf{t}]\rangle \rightarrow \langle M'_{[P!][P?][\bar{P}?]}[\mathbf{skip}][\mathbf{skip}], \sigma[P! = \mathbf{f}]\rangle$$

Now, we may complete the square by completing the handshake protocol via the computation

$$\begin{aligned} & \langle M'_{[\mathbf{AHS}][\mathbf{PHS}][!p]}[\mathbf{AHS}_1][\mathbf{PHS}], \sigma[!p = \mathbf{t}, ?p = \mathbf{f}]\rangle \xrightarrow{*} \\ & \langle M'_{[\mathbf{AHS}][\mathbf{PHS}][!p]}[\mathbf{skip}][\mathbf{skip}], \sigma[!p = \mathbf{f}, ?p = \mathbf{f}]\rangle, \end{aligned}$$

and the two resulting states are clearly related by condition (1.) of $\approx_{hs}$. ■

LEMMA 16 If given configurations $\langle c_0, \sigma_0\rangle$ and $\langle c'_0, \sigma'_0\rangle$ we have

$$\begin{tikzcd}\langle c_0, \sigma_0\rangle \arrow[d, phantom, "\wr\wr_{hs+}"] & \\ \langle c'_0, \sigma'_0\rangle \arrow[r] & \langle c'_1, \sigma'_1\rangle\end{tikzcd}$$

then either

1. $\langle c_0, \sigma_0\rangle \xrightarrow{*} \mathbf{ERROR}$, or
2. we may find $c_i, \sigma_i, 0 \leq i \leq n$, such that

$$\begin{tikzcd}\langle c_0, \sigma_0\rangle \arrow[r, "*"] \arrow[d, phantom, "\wr\wr_{hs+}"] & \langle c_n, \sigma_n\rangle \arrow[d, phantom, "\wr\wr_{hs+}"] \\ \langle c'_0, \sigma'_0\rangle \arrow[r] & \langle c'_1, \sigma'_1\rangle\end{tikzcd}$$

Proof: This proof parallels the proof of the previous Lemma. As in that Lemma, there are two cases that may first be factored out of the proof. If the hole in C or M is not touched, the relation is clearly preserved. If a boolean expression is evaluated, inspection of all the cases of $\approx_{hs+}$ shows $\sigma(\bar{P}?) = \sigma(!p)$ except in cases 6-8, so the only potential differences are in those cases. For the case of a guard, for some guard expression e containing $!p$ to be true requires $!p$ to occur negatively in the guard since $\sigma(!p) = \mathbf{f}$, but by syntactic restriction 1 of Definition 2, probes cannot occur negatively in guard expressions, so $!p$ will also not occur negatively, so a true guard of this form in fact could never arise. Thus, evaluation of a boolean expression will either lead to **ERROR** or will proceed uniformly in the two cases.

Then, proceed by cases on the assumption $\langle c_0, \sigma_0\rangle \approx_{hs+} \langle c'_0, \sigma'_0\rangle$.

Case 1.: Consider the next step of $\langle c'_0, \sigma'_0\rangle = \langle C[\mathbf{AHS}][\mathbf{PHS}][!p], \sigma[!p = \mathbf{f}, ?p = \mathbf{f}]\rangle$. The only step involving **AHS** or **PHS** that could apply is the first step of **AHS**: **PHS** cannot execute even if it is enabled because the initial condition requires $\sigma(!p) = \mathbf{t}$.

Thus, for $M_{[\mathbf{AHS}][\mathbf{PHS}][!p]}[\mathbf{AHS}] = C[\mathbf{AHS}][\mathbf{PHS}][!p]$ a single step places us in the configuration $\langle M_{[\mathbf{AHS}][\mathbf{PHS}][!p]}[\mathbf{AHS}_1], \sigma[!p = \mathbf{t}, ?p = \mathbf{f}]\rangle$, precisely the right-hand-side of case 2. of $\approx_{hs+}$. We may complete the square by the computation

$$\langle c_0, \sigma_0\rangle = \langle M_{[P!][P?][\bar{P}?]}[P!], \sigma[P! = \mathbf{f}]\rangle \rightarrow \langle M_{[P!][P?][\bar{P}?]}[P!], \sigma[P! = \mathbf{t}]\rangle$$

Case 2.: From the state $\langle M_{[\mathbf{AHS}][\mathbf{PHS}][!p]}[\mathbf{AHS}_1], \sigma[!p = \mathbf{t}, ?p = \mathbf{f}]\rangle$, the only steps that involve $\mathbf{AHS}_1$ or **AHS** or **PHS** at this point are (1) some passive protocol **PHS** could be enabled and the guard $!p$ true, causing it to step to $\mathbf{PHS}_1$ or (2) another **AHS** could be enabled and it could step to $\mathbf{AHS}_1$. For case (1), the configuration $M'_{[\mathbf{AHS}][\mathbf{PHS}][!p]}[\mathbf{AHS}_1][\mathbf{PHS}] = M_{[\mathbf{AHS}][\mathbf{PHS}][!p]}[\mathbf{AHS}_1]$ steps to $\langle M'_{[\mathbf{AHS}][\mathbf{PHS}][!p]}[\mathbf{AHS}_1][\mathbf{PHS}_1], \sigma[!p = \mathbf{t}, ?p = \mathbf{f}]\rangle$, and this is related to the same left-hand side by case 3. For case (2), this execution places us in state 9.

Case 3.: From the state $\langle M_{[\mathbf{AHS}][\mathbf{PHS}][!p]}[\mathbf{AHS}_1][\mathbf{PHS}_1], \sigma[!p = \mathbf{t}, ?p = \mathbf{f}]\rangle$, the only steps involving $\mathbf{AHS}_1$ or $\mathbf{PHS}_1$ or **AHS** or **PHS** are (1) the passive protocol $\mathbf{PHS}_1$ stepping to $\mathbf{PHS}_2$ and setting $?p = \mathbf{t}$ or (2) another **AHS** setting $!p = \mathbf{t}$ or (3) another **PHS** stepping to $\mathbf{PHS}_1$. Cases (2) and (3) may be shown to lead to an error in the high-level synchronization, satifying case (1.) of what we are trying to prove. We consider (2) only, (3) is similar. For this case, the term must initially be of the form $M_{[\mathbf{AHS}][\mathbf{PHS}][!p]}[\mathbf{AHS}_1][\mathbf{AHS}][\mathbf{PHS}_1]$, thus it is $\approx_{hs+}$-related to term $M_{[P!][P?][\bar{P}?]}[P!][P!][P?]$, and that configuration is in error since there are two different synchronizations possible, each changing the value of $P!$ (see Definition 11). Thus, only case (1) remains, and this step clearly gets us to state 4. of $\approx_{hs+}$.
Case 4.–7.: These states closely parallel state 3.: from state 4. we move to 5., 5. to 6., and 6. to 7., all the while performing no steps on the left-hand-side terms.
Case 8.: From the state $\langle M_{[\mathbf{AHS}][\mathbf{PHS}][!p]}[\mathbf{AHS}_3][\mathbf{skip}], \sigma[!p = \mathbf{f}, ?p = \mathbf{f}]\rangle$ we clearly step to $\langle M_{[\mathbf{AHS}][\mathbf{PHS}][!p]}[\mathbf{skip}][\mathbf{skip}], \sigma[!p = \mathbf{f}, ?p = \mathbf{f}]\rangle$, and we may complete the square by

$$\langle M_{[P!][P?][\bar{P}?]}[P!][P?], \sigma[P! = \mathbf{t}]\rangle \to \langle M_{[P!][P?][\bar{P}?]}[\mathbf{skip}][\mathbf{skip}], \sigma[P! = \mathbf{f}]\rangle$$

Case 9.: There are two possibilities here: if a **PHS** is enabled and executes to $\mathbf{PHS}_1$, it means, as in the case 3. argument above, the high-level synchronization would **ERROR**. If another **AHS** is enabled and executes to $\mathbf{AHS}_1$, we stay in state 9. Note there are no possibilities of a synchronization completing successfully from this state, the best we can do is to hang forever. ∎

Lemma 17 If $\langle c, \sigma\rangle \approx_{hs+} \langle c', \sigma'\rangle$, then $\sigma(x_{\text{success}}) = \sigma'(x_{\text{success}})$.

Proof: By inspection of each of the cases of the definition of $\approx_{hs+}$, related states are identical on all variables excepting $P!, !p, ?p$. ∎

Lemma 18 If $\langle c, \sigma\rangle \approx_{hs+} \langle c', \sigma'\rangle$, then $\varepsilon(\langle c', \sigma'\rangle)$ implies $\varepsilon(\langle c, \sigma\rangle)$.

Proof: Assume $\varepsilon(\langle c', \sigma'\rangle)$. If the cause of error is independent of $!p/?p$, the above property is uniformly true. Thus, the error must involve $!p$ and/or $?p$ in some way. The possible errors may be seen to be as follows: (1) one of $!p$ or $?p$ is concurrently written, or (2) one of $!p$ or $?p$ is concurrently read and changed (the "read" action is either in a guard or an assignment). For case (1), since all writes of $!p/?p$ occur in **AHS** and **PHS**, it must be that two active or passive synchronizations are simultaneously executing, and this will also introduce an error in the high-level synchronization $\langle c', \sigma'\rangle$. Case (2) can occur in two possible manners. One is as in case (1), where the outcome is the same. The other is when a probe, here of the form $!p$, is read in a guard at the same time $!p$'s value is changed. Two subcases arise depending on whether $!p$ was set high or low. If it is set high, we must be in state 1. of $\approx_{hs+}$, and $P!$ is also set high so the same error will result. If $!p$ is set low we are in state 5 of $\approx_{hs+}$, and similarly, $P!$ may be set low by completing the synchronization at this point, so the same error will again result at the high level. ■

We may now prove the main result which allows one channel to be replaced by its handshake expansion, Theorem 4.

Proof: From the definition of $\overset{\Rightarrow}{\cong}_{obserr}$, it suffices to prove three things.

1. Assuming $C[P!][P?][\bar{P}?]$ and $C[\mathbf{AHS}][\mathbf{PHS}][!p]$ are semantically well-formed, if
$$\langle C[P!][P?][\bar{P}?], \iota\rangle \xrightarrow{*} \langle c, \sigma[x_{\text{success}} = \mathbf{t}]\rangle$$
then
$$\langle C[\mathbf{AHS}][\mathbf{PHS}][!p], \iota\rangle \xrightarrow{*} \langle c', \sigma'[x_{\text{success}} = \mathbf{t}]\rangle.$$

2. Assuming $C[P!][P?][\bar{P}?]$ and $C[\mathbf{AHS}][\mathbf{PHS}][!p]$ are semantically well-formed, if
$$\langle C[\mathbf{AHS}][\mathbf{PHS}][!p], \iota\rangle \xrightarrow{*} \langle c, \sigma[x_{\text{success}} = \mathbf{t}]\rangle$$
then
$$\langle C[P!][P?][\bar{P}?], \iota\rangle \xrightarrow{*} \langle c', \sigma'[x_{\text{success}} = \mathbf{t}]\rangle.$$

3. If $\langle C[\mathbf{AHS}][\mathbf{PHS}][!p], \iota\rangle \xrightarrow{*} \mathbf{ERROR}$, then $\langle C[P!][P?][\bar{P}?], \iota\rangle \xrightarrow{*} \mathbf{ERROR}$.

We establish these in turn.

For (1.), by Lemma 17 it suffices to show $\langle c, \sigma[x_{\text{success}} = \mathbf{t}]\rangle \approx_{hs} \langle c', \sigma'[x_{\text{success}} = \mathbf{t}]\rangle$ for some c', σ' such that
$$\langle C[\mathbf{AHS}][\mathbf{PHS}][!p], \iota\rangle \xrightarrow{*} \langle c', \sigma'[x_{\text{success}} = \mathbf{t}]\rangle$$
This in turn follows by immediate induction from Lemma 15.

For (2.), the proof parallels the previous case, only lemma 16 is used, and note under the assumption that $\langle C[P!][P?][\bar{P}?], \iota\rangle$ is semantically well-formed, so case (1.) of that Lemma will never hold.

For (3.), proceed by induction on the length of the computation
$$\langle C[\mathbf{AHS}][\mathbf{PHS}][!p], \iota\rangle \xrightarrow{*} \mathbf{ERROR}$$
By Lemma 16, either case (1.) of that Lemma holds, in which case the conclusion is vacuous, or (2.) holds, in which case the conclusion follows directly from Lemma 18. ■

This then suggests a general scheme for establishing $\overset{\Rightarrow}{\cong}_{obserr}$: if we can define a relation $\approx$ and establish properties as expressed in Lemmas 15 through 18, $\overset{\Rightarrow}{\cong}_{obserr}$ follows.

We now may esablish the correctness of the handshake expansion phase, Lemma 3. Before giving the main proof we present a few auxiliary Lemmas.

Lemma 19 $\Rightarrow_2$ has the strong diamond property, namely if $m \Rightarrow_2 m'$ and $m \Rightarrow_2 m''$, then $m' \Rightarrow_2 m'''$ and $m'' \Rightarrow_2 m'''$ for some m'''.

Proof: By inspection of the rules, the rewritings to m' and m'' must either involve different variables or one must involve an active port and the other the passive port. These are then obviously interchangable since there can be no syntactic overlap of the two. ■

Corollary 4 (Uniqueness of phase 2 normal forms) If $m \Rightarrow_2^N m'$ and $m \Rightarrow_2^N m''$ then $m' = m''$.

Proof: By direct induction from Lemma 19. ■

Now for the proof of Lemma 3.

Proof: Given $\Rightarrow_1 m_1$ and $m_1 \Rightarrow_2^N m_2$, we wish to show $m_1 \overset{\Rightarrow}{\cong}_2 m_2$. By the definition of $\overset{\Rightarrow}{\cong}_2$, this means we are given m_1^t such that $m_1||m_1^t$ is closed, $\Rightarrow_1 m_1^t$, and $m_1^t \Rightarrow_2^N m_2^t$, we wish to show $m_1||m_1^t \overset{\Rightarrow}{\cong}_{obserr} m_2||m_2^t$.

First, observe that $m_1||m_1^t \Rightarrow_2^N m_2||m_2^t$: $m_1||m_1^t \Rightarrow_2^* m_2||m_1^t \Rightarrow_2^* m_2||m_2^t$ since the same rewrite sequence as before may be applied in an expanded context.

Next, by uniqueness of phase 2 normal forms, Corollary 4, the reduction steps of $m_1||m_1^t$ can be ordered as we please and the same normal form will always be reached. We thus choose the following particular reduction sequence to normal form:

$$m_1||m_1^t = m_{10}||m_{10}^t \Rightarrow_2^2 m_{11}||m_{11}^t \Rightarrow_2^2 \ldots \Rightarrow_2^2 m_{1n}||m_{1n}^t = m_2||m_2^t$$

where $m_{1i}||m_{1i}^t$ differs from $m_{1i+1}||m_{1i+1}^t$ for $i < n$ in that exactly one channel P has been replaced by its handshake expansion. This requires one application of each of the phase 2 rules, so each step from $m_{1i}||m_{1i}^t$ to $m_{1i+1}||m_{1i+1}^t$ requires two steps.

Now, for each $i < n$, $m_{1i}||m_{1i}^t \overset{\Rightarrow}{\cong}_{obserr} m_{1i+1}||m_{1i+1}^t$ by Theorem 4 since exactly one channel is replaced with its handshake expansion. Thus, by transitivity over the above rewrite sequence we have $m_1||m_1^t \overset{\Rightarrow}{\cong}_{obserr} m_2||m_2^t$. ■

D.2. Correctness of Phase 1

We present outlines of the proofs of correctness of the phase 1 transformation rules. The proof of (**1 : ASSN1**) is immediate since the scoping rules prevent any context from using $S_0!$, $S_1!$ and $S_0!, S_0?, S_1!, S_1?$ do not occur in c.

Theorem 5 ((**1 : INIT**) Correct)

$$c \overset{\Rightarrow}{\cong} \textbf{with } S! \textbf{ do } S! \textbf{ end } || \textbf{with } S? \textbf{ do } *[[\bar{S} \longrightarrow c; S?]] \textbf{ end}$$

Proof: $S!, S?$ are new ports used nowhere in c nor in the enclosing context because of the scoping restrictions, so the right-hand side always computes to c. ■

THEOREM 6 ((**1 : ASSN1**) CORRECT) $p \overset{\Rightarrow}{\cong} p'$ where

$$\begin{array}{rcl} p & = & \textbf{with w } x \textbf{ do } *[c] \textbf{ end} \\ p' & = & \textbf{with w } x \textbf{ do} \\ & & \quad \textbf{with } S_0?, S_1? \textbf{ do} \\ & & \quad\quad *[[\bar{S}_0? \longrightarrow x \downarrow; S_0? [] \bar{S}_1? \longrightarrow x \uparrow; S_1?]] \\ & & \quad \textbf{end} \parallel \\ & & \quad \textbf{with } S_0!, S_1! \textbf{ do } *[c] \textbf{ end} \\ & & \textbf{end} \end{array}$$

Proof: Immediate. ■

We now turn our attention to establishing the correctness of the remaining phase 1 rules:
(**1 : ASSN2**), (**1 : SEQ**), (**1 : GD**), (**1 : LOOP**) and (**1 : PAR**). The proofs of each of these rules are similar in form, and are thus condensed. The general technique for proving the rules correct is given in detail in the correctness proof of handshaking expansion, section D.1. We begin by defining the necessary relations for each rule: $\approx_{as2}$, $\approx_{seq}$, $\approx_{guard}$, $\approx_{loop}$, $\approx_{par}$. Except for $\approx_{as2}$ each of these relations is defined with respect to two other relations $\approx_{ss}$ and $\approx_{es}$ that abstract the computation involved in starting and completing the initial synchronization between $S!, S?$. Note that left- and right-hand sides of the transformations have been embedded into the same closing testing context.

DEFINITION 31 $\approx_{as2}$ is the least relation with the following properties for arbitrary C, C'.

1. $\langle R_1[S!], \sigma[S! = \mathbf{f}, S_0! = \mathbf{f}, S_1! = \mathbf{f}]\rangle \approx_{as2}$
   $\langle R_1'[S!], \sigma[S! = \mathbf{f}, S_0! = \mathbf{f}, S_1! = \mathbf{f}]\rangle$

2. $\langle R_2[p_2], \sigma[S! = \mathbf{t}, S_0! = \mathbf{f}, S_1! = \mathbf{f}]\rangle \approx_{as2}$
   $\langle R_2'[p_2'], \sigma[S! = \mathbf{t}, S_0! = \mathbf{f}, S_1! = \mathbf{f}]\rangle$

3. $\langle R_3[x := e], \sigma[S! = \mathbf{t}, S_0! = \mathbf{f}, S_1! = \mathbf{f}]\rangle \approx_{as2}$
   $\langle R_3'[[\neg e \longrightarrow S_0!; S? [] e \longrightarrow S_1!; S?]], \sigma[S! = \mathbf{t}, S_0! = \mathbf{f}, S_1! = \mathbf{f}]\rangle$

4. $\langle R_3[x := e], \sigma[S! = \mathbf{t}, S_0! = \mathbf{f}, S_1! = \mathbf{f}]\rangle \approx_{as2}$
   $\langle R_4'[S_i!], \sigma[S! = \mathbf{t}, S_i! = \mathbf{t}, S_{i-1}! = \mathbf{f}]\rangle$

5. $\langle R_3[x := e], \sigma[S! = \mathbf{t}, S_0! = \mathbf{f}, S_1! = \mathbf{f}]\rangle \approx_{as2}$
   $\langle R_5'[S_i!][[S_0! \longrightarrow x \downarrow [] S_1! \longrightarrow x \uparrow]], \sigma[S! = \mathbf{t}, S_i! = \mathbf{t}, S_{i-1}! = \mathbf{f}]\rangle$

6. $\langle R_3[x := e], \sigma[S! = \mathbf{t}, S_0! = \mathbf{f}, S_1! = \mathbf{f}]\rangle \approx_{as2}$
   $\langle R_6'[x := i], \sigma[S! = \mathbf{t}, S_i! = \mathbf{t}, S_{i-1}! = \mathbf{f}]\rangle$

7. $\langle R_4[S?], \sigma[x = \sigma(e), S! = \mathbf{t}, S_0! = \mathbf{f}, S_1! = \mathbf{f}]\rangle \approx_{as2}$
$\langle R'_7[S_i!][S_i?], \sigma[x = i, S! = \mathbf{t}, S_i! = \mathbf{t}, S_{i-1}! = \mathbf{f}]\rangle$

8. $\langle R_4[S?], \sigma[x = \sigma(e), S! = \mathbf{t}, S_0! = \mathbf{f}, S_1! = \mathbf{f}]\rangle \approx_{as2}$
$\langle R'_8[S?], \sigma[x = i, S! = \mathbf{t}, S_i! = \mathbf{f}, S_{i-1}! = \mathbf{f}]\rangle$

where

$$\begin{array}{lcl}
R_1[S!] & = & C'[cell_1 || C[p_1]] \\
R_2[p_2] & = & C'[cell_1 || C[p_2; p_1]] \\
R_3[x := e] & = & C'[cell_1 || C[x := e; S?; p_1]] \\
R_4[S?] & = & C'[cell_1 || C[S?; p_1]] \\
 & & \\
R'_1[S!] & = & C'[cell_1 || C[p'_1]] \\
R'_2[p_2] & = & C'[cell_1 || C[p'_2; p'_1]] \\
\multicolumn{3}{l}{R'_3[[\neg e \longrightarrow S_0!; S? [\!] e \longrightarrow S_1!; S?]]} \\
 & = & C'[cell_1 || C[[\neg e \longrightarrow S_0!; S? [\!] e \longrightarrow S_1!; S?]; p'_1]] \\
R'_4[S_i!] & = & C'[cell_1 || C[S_i!; S?; p'_1]] \\
\multicolumn{3}{l}{R'_5[S_i!][[\bar{S}_0? \longrightarrow x \downarrow; S_0? [\!] \bar{S}_1 \longrightarrow x \uparrow; S_1?]]} \\
 & = & C'[cell_2 || C[S_i!; S?; p'_1]] \\
R'_6[x := i] & = & C'[cell_3 || C[S_i!; S?; p'_1]] \\
R'_7[S_i!][S_i?] & = & C'[cell_4 || C[S_i!; S?; p'_1]] \\
R'_8[S?] & = & C'[cell_1 || C[S?; p'_1]]
\end{array}$$

and

$$\begin{array}{lcl}
cell_1 & = & *[[\bar{S}_0? \longrightarrow x \downarrow; S_0? [\!] \bar{S}_1? \longrightarrow x \uparrow; S_1?]] \\
cell_2 & = & [\bar{S}_0? \longrightarrow x \downarrow; S_0? [\!] \bar{S}_1? \longrightarrow x \uparrow; S_1?]; cell_1 \\
cell_3 & = & x := i; S_i?; cell_1 \\
cell_4 & = & S_i?; cell_1 \\
p_1 & = & *[[\bar{S}? \longrightarrow x := e; S?]] \\
p_2 & = & [\bar{S}? \longrightarrow x := e; S?] \\
p'_1 & = & *[[\bar{S}? \longrightarrow [\neg e \longrightarrow S_0!; S? [\!] e \longrightarrow S_1!; S?]]] \\
p'_2 & = & [\bar{S}? \longrightarrow [\neg e \longrightarrow S_0!; S? [\!] e \longrightarrow S_1!; S?]]
\end{array}$$

The relations for the other rules depend on two additional relations, $\approx_{ss}$ and $\approx_{es}$, that relate beginning synchronization on $S!, S?$ and completing synchronization on $S!, S?$ respectively.

Definition 32 (Start Synchronization) $\approx_{ss}$ is the least relation with the following properties for arbitrary C.

1. $\langle R_1[S!], \sigma[S! = \mathbf{f}]\rangle \approx_{ss}$
$\langle R'_1[S!], \sigma'[S! = \mathbf{f}]\rangle$

2. $\langle R_2[[\bar{S}? \longrightarrow c; S?]], \sigma[S! = \mathbf{t}]\rangle \approx_{ss}$
$\langle R'_2[[\bar{S}? \longrightarrow c'; S?]], \sigma'[S! = \mathbf{t}]\rangle$

3. $\langle R_3[c], \sigma[S! = \mathbf{t}]\rangle \approx_{ss}$
$\langle R_3'[c'], \sigma'[S! = \mathbf{t}]\rangle$

where

$$\begin{array}{lclllcl} R_1[S!] & = & C[p_1] & R_1[S!] & & = & C[p_1'] \\ R_2[[\bar{S}? \longrightarrow c; S?]] & = & C[p_2] & R_2'[[\bar{S}? \longrightarrow c'; S?]] & & = & C[p_2'] \\ R_3[c] & = & C[p_3] & R_3'[c'] & & = & C[p_3'] \end{array}$$

and

$$\begin{array}{rcl} p_1 & = & *[[\bar{S}? \longrightarrow c; S?]] \\ p_2 & = & [\bar{S}? \longrightarrow c; S?]; *[[\bar{S}? \longrightarrow c; S?]] \\ p_3 & = & c; S?; *[[\bar{S}? \longrightarrow c; S?]] \\ p_1' & = & *[[\bar{S}? \longrightarrow op(S_1!, \ldots, S_n!); S?]] || * [[\bar{S}_1? \longrightarrow c_1; S_1?]] || \ldots * [[\bar{S}_n? \longrightarrow c_n; S_n?]] \\ p_2' & = & [\bar{S}? \longrightarrow op(S_1!, \ldots, S_n!); S?]; *[[\bar{S}? \longrightarrow op(S_1!, \ldots, S_n!); S?]] || \\ & & *[[\bar{S}_1? \longrightarrow c_1; S_1?]] || \ldots * [[\bar{S}_n? \longrightarrow c_n; S_n?]] \\ p_3' & = & op(S_1!, \ldots, S_n!); S?; *[[\bar{S}? \longrightarrow op(S_1!, \ldots, S_n!); S?]] || \\ & & *[[\bar{S}_1? \longrightarrow c_1; S_1?]] || \ldots * [[\bar{S}_n? \longrightarrow c_n; S_n?]] \end{array}$$

where $op(S_1!, \ldots, S_n!)$ is shorthand for a composition of n active synchronizations using one of sequencing, choice, looping or parallel composition.

DEFINITION 33 (COMPLETE SYNCHRONIZATION) $\approx_{es}$ is the least relation with the property

$$\langle R[S!][S?], \sigma[S! = \mathbf{t}]\rangle \approx_{es} \langle R'[S!][S?], \sigma'[S! = \mathbf{t}]\rangle$$

for arbitrary C, where

$$\begin{array}{rcl} R[S!][S?] & = & C[S?; *[[\bar{S}? \longrightarrow c; S?]]] \\ R'[S!][S?] & = & C[S?; *[[\bar{S}? \longrightarrow op(S_1!, \ldots, S_n!); S?]] || \\ & & \quad *[[\bar{S}_1? \longrightarrow c_1; S_1?]] || \ldots * [[\bar{S}_n? \longrightarrow c_n; S_n?]]] \end{array}$$

We now define the four relations $\approx_{seq}$, $\approx_{guard}$, $\approx_{loop}$ and $\approx_{par}$ that correlate the execution of the original body c to the transformed body in which $\bar{S}?$ guards a set of synchronizations. The four relations correspond to sequential composition, (guarded) choice, looping and parallelism.

DEFINITION 34 $\approx_{seq}$ is the least relation with properties $\approx_{ss}, \approx_{es}$ in addition to the following properties for arbitrary C.

1. $\langle R_1[c_1], \sigma[S! = \mathbf{t}]\rangle \approx_{seq}$
$\langle R_1'[S_1!], \sigma[S! = \mathbf{t}, S_1! = \mathbf{f}, S_2! = \mathbf{f}]\rangle$

2. $\langle R_1[c_1], \sigma[S! = \mathbf{t}]\rangle \approx_{seq}$
$\langle R_2'[[\bar{S}_1? \longrightarrow c_1; S_1?]], \sigma[S! = \mathbf{t}, S_1! = \mathbf{t}, S_2! = \mathbf{f}]\rangle$

3. $\langle R_1[c_1], \sigma[S! = \mathbf{t}]\rangle \approx_{seq}$
$\langle R_3'[c_1], \sigma[S! = \mathbf{t}, S_1! = \mathbf{t}, S_2! = \mathbf{f}]\rangle$

4. $\langle R_2[c_2], \sigma[S! = \mathbf{t}]\rangle \approx_{seq}$
$\langle R'_4[S_1!][S_1?], \sigma[S! = \mathbf{t}, S_1! = \mathbf{t}, S_2! = \mathbf{f}]\rangle$

5. $\langle R_2[c_2], \sigma[S! = \mathbf{t}]\rangle \approx_{seq}$
$\langle R'_5[S_2!], \sigma[S! = \mathbf{t}, S_1! = \mathbf{f}, S_2! = \mathbf{f}]\rangle$

6. $\langle R_2[c_2], \sigma[S! = \mathbf{t}]\rangle \approx_{seq}$
$\langle R'_6[[\bar{S}_2? \longrightarrow c_2; S_2?]], \sigma[S! = \mathbf{t}, S_1! = \mathbf{f}, S_2! = \mathbf{t}]\rangle$

7. $\langle R_2[c_2], \sigma[S! = \mathbf{t}]\rangle \approx_{seq}$
$\langle R'_7[c_2], \sigma[S! = \mathbf{t}, S_1! = \mathbf{f}, S_2! = \mathbf{t}]\rangle$

8. $\langle R_3[\mathbf{skip}], \sigma[S! = \mathbf{t}]\rangle \approx_{seq}$
$\langle R'_8[S_2!][S_2?], \sigma[S! = \mathbf{t}, S_1! = \mathbf{f}, S_2! = \mathbf{t}]\rangle$

where

$$\begin{aligned}
R_1[c_1] &= C[S!][c_1; c_2; S?; *[[\bar{S}? \longrightarrow c_1; c_2; S?]]] \\
R_2[c_2] &= C[S!][c_2; S?; *[[\bar{S}? \longrightarrow c_1; c_2; S?]]] \\
R_3[\mathbf{skip}] &= C[S!][\mathbf{skip}; S?; *[[\bar{S}? \longrightarrow c_1; c_2; S?]]] \\
R'_1[S_1!] &= C[S!][S_1!; S_2!; S?; *[[\bar{S}? \longrightarrow S_1!; S_2!; S?]] \| \\
&\qquad *[[\bar{S}_1? \longrightarrow c_1; S_1?]] \| *[[\bar{S}_2? \longrightarrow c_2; S_2?]]] \\
R'_2[[\bar{S}_1? \longrightarrow c_1; S_1?]] & \\
&= C[S!][S_1!; S_2!; S?; *[[\bar{S}? \longrightarrow S_1!; S_2!; S?]] \| \\
&\qquad [\bar{S}_1? \longrightarrow c_1; S_1?]; *[[\bar{S}_1? \longrightarrow c_1; S_1?]] \| *[[\bar{S}_2? \longrightarrow c_2; S_2?]]] \\
R'_3[c_1] &= C[S!][S_1!; S_2!; S?; *[[\bar{S}? \longrightarrow S_1!; S_2!; S?]] \| \\
&\qquad c_1; S_1?; *[[\bar{S}_1? \longrightarrow c_1; S_1?]] \| *[[\bar{S}_2? \longrightarrow c_2; S_2?]]] \\
R'_4[S_1!][S_1?] &= C[S!][S_1!; S_2!; S?; *[[\bar{S}? \longrightarrow S_1!; S_2!; S?]] \| \\
&\qquad S_1?; *[[\bar{S}_1? \longrightarrow c_1; S_1?]] \| *[[\bar{S}_2? \longrightarrow c_2; S_2?]]] \\
R'_5[S_2!] &= C[S!][S_2!; S?; *[[\bar{S}? \longrightarrow S_1!; S_2!; S?]] \| *[[\bar{S}_1? \longrightarrow c_1; S_1?]] \| \\
&\qquad *[[\bar{S}_2? \longrightarrow c_2; S_2?]]] \\
R'_6[[\bar{S}_2? \longrightarrow c_2; S_2?]] & \\
&= C[S!][S_2!; S?; *[[\bar{S}? \longrightarrow S_1!; S_2!; S?]] \| \\
&\qquad *[[\bar{S}_1? \longrightarrow c_1; S_1?]] \| [\bar{S}_2? \longrightarrow c_2; S_2?]; *[[\bar{S}_2? \longrightarrow c_2; S_2?]]] \\
R'_7[c_2] &= C[S!][S_2!; S?; *[[\bar{S}? \longrightarrow S_1!; S_2!; S?]] \| *[[\bar{S}_1? \longrightarrow c_1; S_1?]] \| \\
&\qquad c_2; S_2?; *[[\bar{S}_2? \longrightarrow c_2; S_2?]]] \\
R'_8[S_2!][S_2?] &= C[S!][S_2!; S?; *[[\bar{S}? \longrightarrow S_1!; S_2!; S?]] \| *[[\bar{S}_1? \longrightarrow c_1; S_1?]] \| \\
&\qquad S_2?; *[[\bar{S}_2? \longrightarrow c_2; S_2?]]]
\end{aligned}$$

Definition 35 $\approx_{guard}$ is the least relation with properties $\approx_{ss}, \approx_{es}$ in addition to the following properties for arbitrary C.

1. $\langle R_1[choice], \sigma[S! = \mathbf{t}]\rangle \approx_{guard}$
$\langle R'_1[choice'], \sigma[S! = \mathbf{t}, S_1! = \ldots = S_n! = \mathbf{f}]\rangle$

2. $\langle R_2[c_i], \sigma[S! = \mathbf{t}]\rangle \approx_{guard}$
$\langle R'_2[S_i!], \sigma[S! = \mathbf{t}, S_i! = \mathbf{f}]\rangle$

3. $\langle R_2[c_i], \sigma[S! = \mathbf{t}]\rangle \approx_{guard}$
   $\langle R'_3[[S_i? \longrightarrow c_i; S_i?]], \sigma[S! = \mathbf{t}, S_i! = \mathbf{t}]\rangle$

4. $\langle R_2[c_i], \sigma[S! = \mathbf{t}]\rangle \approx_{guard}$
   $\langle R'_4[c_i], \sigma[S! = \mathbf{t}, S_i! = \mathbf{t}]\rangle$

5. $\langle R_2[\mathbf{skip}], \sigma[S! = \mathbf{t}]\rangle \approx_{guard}$
   $\langle R'_5[S_i!][S_i?], \sigma[S! = \mathbf{t}, S_i! = \mathbf{t}]\rangle$

where

$$\begin{array}{lcl}
R_1[\mathit{choice}] & = & C[S!][\mathit{choice}; S?; *[[\bar{S}? \longrightarrow \mathit{choice}; S?]]] \\
R_2[c_i] & = & C[S!][c_i; S?; *[[\bar{S}? \longrightarrow \mathit{choice}; S?]]] \\
R_3[\mathbf{skip}] & = & C[S!][\mathbf{skip}; S?; *[[\bar{S}? \longrightarrow \mathit{choice}; S?]]] \\
R'_1[\mathit{choice}'] & = & C[S!][\mathit{choice}'; *[[\bar{S}? \longrightarrow \mathit{choice}']]|| \\
 & & *[[S_1? \longrightarrow c_1; S_1?]]|| \ldots || * [[S_n? \longrightarrow c_n; S_n?]]] \\
R'_2[S_i!] & = & C[S!][S_i!; S?; *[[\bar{S}? \longrightarrow \mathit{choice}']]|| * [[\bar{S}_i? \longrightarrow c_i; S_i?]]||\mathit{rest}] \\
\multicolumn{3}{l}{R'_3[[\bar{S}_i? \longrightarrow c_i; S_i?]]} \\
 & = & C[S!][S_i!; S?; *[[\bar{S}? \longrightarrow \mathit{choice}']]|| \\
 & & [\bar{S}_i? \longrightarrow c_i; S_i?]; *[[\bar{S}_i? \longrightarrow c_i; S_i?]]||\mathit{rest}] \\
R'_4[c_i] & = & C[S!][S_i!; S?; *[[\bar{S}? \longrightarrow \mathit{choice}']]||c_i; S_i?; *[[\bar{S}_i? \longrightarrow c_i; S_i?]]||\mathit{rest}] \\
R'_5[S_i!][S_i?] & = & C[S!][S_i!S?; *[[\bar{S}? \longrightarrow \mathit{choice}']]||S_i?; *[[\bar{S}_i? \longrightarrow c_i; S_i?]]||\mathit{rest}] \\
\mathit{choice} & = & [e_1 \longrightarrow c_1 [\!] \ldots [\!] e_n \longrightarrow c_n] \\
\mathit{choice}' & = & [e_{1_1} \longrightarrow S_1!; S? [\!] \ldots [\!] e_{1_{m_1}} \longrightarrow S_1!; S? [\!] \ldots \\
 & & [\!] e_{n_1} \longrightarrow S_n!; S? [\!] \ldots [\!] e_{n_{m_n}} \longrightarrow S_n!; S?] \\
\mathit{rest} & = & *[[\bar{S}_1? \longrightarrow c_1; S_1?]]|| \ldots || * [[\bar{S}_{i-1}? \longrightarrow c_{i-1}; S_{i-1}?]]|| \\
 & & *[[\bar{S}_{i+1}? \longrightarrow c_{i+1}; S_{i+1}?]]|| \ldots || * [[\bar{S}_n? \longrightarrow c_n; S_n?]]
\end{array}$$

Note that in the state for the right-hand side of the relation we only include the port $S_i!$ that will be changed. This improves the readibility.

DEFINITION 36 $\approx_{loop}$ is the least relation with properties $\approx_{ss}, \approx_{es}$ in addition to the following properties for arbitrary C.

1. $\langle R_1[\,*\,[c]], \sigma[S! = \mathbf{t}]\rangle \approx_{loop}$
   $\langle R'_1[\,*\,[S!']], \sigma[S! = \mathbf{t}, S!' = \mathbf{f}]\rangle$

2. $\langle R_2[c], \sigma[S! = \mathbf{t}]\rangle \approx_{loop}$
   $\langle R'_2[S'!], \sigma[S! = \mathbf{t}, S'! = \mathbf{f}]\rangle$

3. $\langle R_2[c], \sigma[S! = \mathbf{t}]\rangle \approx_{loop}$
   $\langle R'_3[[\bar{S}'? \longrightarrow c; S'?]], \sigma[S! = \mathbf{t}, S'! = \mathbf{t}]\rangle$

4. $\langle R_2[c], \sigma[S! = \mathbf{t}]\rangle \approx_{loop}$
   $\langle R'_4[c], \sigma[S! = \mathbf{t}, S'! = \mathbf{t}]\rangle$

5. $\langle R_3[\mathbf{skip}], \sigma[S! = \mathbf{t}]\rangle \approx_{loop}$
   $\langle R'_5[S'!][S'?], \sigma[S! = \mathbf{t}, S'! = \mathbf{t}]\rangle$

where

$$\begin{array}{lcl}
R_1[*[c]] & = & C[S!][*[c]; S?; *[[\bar{S}? \longrightarrow *[c]; S?]]] \\
R_2[c] & = & C[S!][c; *[c]; S?; *[[\bar{S}? \longrightarrow *[c]; S?]]] \\
R_3[\mathbf{skip}] & = & C[S!][\mathbf{skip}; *[c]; S?; *[[\bar{S}? \longrightarrow *[c]; S?]]] \\
R'_1[*[S'!]] & = & C[S!][*[S'!]; S?; *[[\bar{S}? \longrightarrow *[S'!]; S?]]||*[[\bar{S}'? \longrightarrow c; S'?]]] \\
R'_2[S'!] & = & C[S!][S'!; *[S'!]; S?; *[[\bar{S}? \longrightarrow *[S'!]; S?]]||*[[\bar{S}'? \longrightarrow c; S'?]]] \\
\multicolumn{3}{l}{R'_3[[\bar{S}'? \longrightarrow c; S'?]]} \\
 & = & C[S!][S'!; *[S'!]; S?; *[[\bar{S}? \longrightarrow *[S'!]; S?]]|| \\
 & & [\bar{S}'? \longrightarrow c; S'?]; *[[\bar{S}'? \longrightarrow c; S'?]]] \\
R'_4[c] & = & C[S!][S'!; *[S'!]; S?; *[[\bar{S}? \longrightarrow *[S'!]; S?]]||c; S'?; *[[\bar{S}'? \longrightarrow c; S'?]]] \\
R'_5[S'!][S'?] & = & C[S!][S'!; *[S'!]; S?; *[[\bar{S}? \longrightarrow *[S'!]; S?]]||S'?; *[[\bar{S}'? \longrightarrow c; S'?]]]
\end{array}$$

Note that we do not ever need to consider $\approx_{es}$ since the loop never terminates.

Definition 37 $\approx_{par}$ is the least relation with properties $\approx_{ss}, \approx_{es}$ in addition to the following properties for arbitrary C.

1. $\langle R_1[c_i], \sigma[S! = \mathbf{t}]\rangle \approx_{par}$
   $\langle R'_1[S_i!], \sigma[S! = \mathbf{t}, S'_1! = \ldots = S'_n! = \mathbf{f}]\rangle$

2. $\langle R_1[c_i], \sigma[S! = \mathbf{t}]\rangle \approx_{par}$
   $\langle R'_2[[\bar{S}_i? \longrightarrow c_i; S_i?]],$
   $\sigma[S! = \mathbf{t}, S_i! = \mathbf{t}, S_1! = \ldots = S_{i-1}! = S_{i+1}! = \ldots = S_n! = \mathbf{f}]\rangle$

3. $\langle R_1[c_i], \sigma[S! = \mathbf{t}]\rangle \approx_{par}$
   $\langle R'_3[c_i], \sigma[S! = \mathbf{t}, S_i! = \mathbf{t}, S_1! = \ldots = S_{i-1}! = S_{i+1}! = \ldots = S_n! = \mathbf{f}]\rangle$

4. $\langle R_2[\mathbf{skip}], \sigma[S! = \mathbf{t}]\rangle \approx_{par}$
   $\langle R'_4[S_i!][S_i?], \sigma[S! = \mathbf{t}, S_i! = \mathbf{t}, S_1! = \ldots = S_{i-1}! = S_{i+1}! = \ldots = S_n! = \mathbf{f}]\rangle$

where

$$\begin{array}{lcl}
R_1[c_i] & = & C[S!][(c'_1||\ldots||c'_{i-1}||c'_{i+1}||\ldots||c_n); S?; *[[\bar{S}? \longrightarrow (c_1||\ldots||c_n); S?]]] \\
R_2[\mathbf{skip}] & = & C[S!][S?; *[[\bar{S}? \longrightarrow (c_1||\ldots||c_n); S?]]] \\
R'_1[S_i!] & = & C[S!][(S_1!||\ldots||S_n!); S?; *[[\bar{S}? \longrightarrow (S_1!||\ldots||S_n!); S?]]|| \\
 & & *[[\bar{S}_i? \longrightarrow c_i; S_i?]]||\mathit{rest}] \\
\multicolumn{3}{l}{R'_2[[\bar{S}_i? \longrightarrow c_i; S_i?]]} \\
 & = & C[S!][(S_1!||\ldots||S_n!); S?; *[[\bar{S}? \longrightarrow (S_1!||\ldots||S_n!); S?]]|| \\
 & & [\bar{S}_i? \longrightarrow c_i; S_i?]; *[[\bar{S}_i? \longrightarrow c_i; S_i?]]||\mathit{rest}] \\
R'_3[c_i] & = & C[S!][(S_1!||\ldots||S_n!); S?; *[[\bar{S}? \longrightarrow (S_1!||\ldots||S_n!); S?]]|| \\
 & & c_i; S_i?; *[[\bar{S}_i? \longrightarrow c_i; S_i?]]||\mathit{rest}] \\
R'_4[S_i!][S_1?] & = & C[S!][(S_1!||\ldots||S_n!); S?; *[[\bar{S}? \longrightarrow (S_1!||\ldots||S_n!); S?]]|| \\
 & & S_i?; *[[\bar{S}_i? \longrightarrow c_i; S_i?]]||\mathit{rest}] \\
\mathit{rest} & = & c'_1; *[[\bar{S}_1? \longrightarrow c_1; S_1?]]||\ldots||c'_{i-1}; *[[\bar{S}_{i-1}? \longrightarrow c_{i-1}; S_{i-1}?]]|| \\
 & & c'_{i+1}; *[[\bar{S}_{i+1}? \longrightarrow c_{i+1}; S_{i+1}?]]||\ldots||c'_n; *[[\bar{S}_n? \longrightarrow c_n; S_n?]]
\end{array}$$

With these relations defined, we can proceed with the proofs of the five rules.

LEMMA 20 ($\Rightarrow_r$) If given semantically well-formed configurations $\langle c_0, \sigma_0 \rangle$, $\langle c'_0, \sigma'_0 \rangle$ we have $\langle c_0, \sigma_0 \rangle \longrightarrow \langle c_1, \sigma_1 \rangle$ and $\langle c_0, \sigma_0 \rangle \approx_r \langle c'_0, \sigma'_0 \rangle$ then there exist c'_i, σ'_i $(0 \leq i \leq m)$ such that $\langle c'_0, \sigma'_0 \rangle \xrightarrow{*} \langle c'_m, \sigma'_m \rangle$ and $\langle c'_0, \sigma'_0 \rangle \approx_{as2} \langle c'_m, \sigma'_m \rangle$, where $r \in \{as2, seq, guard, loop, par\}$.

LEMMA 21 ($\Leftarrow_r$) If given configurations $\langle c_0, \sigma_0 \rangle$, $\langle c'_0, \sigma'_0 \rangle$ we have $\langle c_0, \sigma_0 \rangle \approx_r \langle c'_0, \sigma'_0 \rangle$ and $\langle c'_0, \sigma'_0 \rangle \longrightarrow \langle c'_1, \sigma'_1 \rangle$ then either

1. $\langle c_0, \sigma_0 \rangle \xrightarrow{*}$ **ERROR**; or
2. we can find c'_i, σ'_i $(0 \leq i \leq n)$ such that $\langle c_0, \sigma_0 \rangle \xrightarrow{*} \langle c_n, \sigma_n \rangle$ and $\langle c_n, \sigma_n \rangle \approx_r \langle c'_1, \sigma'_1 \rangle$.

for $r \in \{as2, seq, guard, loop, par\}$.

LEMMA 22 (SUCCESS LEMMA) If $\langle c, \sigma \rangle \approx_r \langle c', \sigma' \rangle$ then $\sigma(x_{\text{success}}) = \sigma'(x_{\text{success}})$, for $r \in \{as2, seq, guard, loop, par\}$.

LEMMA 23 (ERROR LEMMA) If $\langle c, \sigma \rangle \approx_r \langle c', \sigma' \rangle$ then $\epsilon(\langle c, \sigma \rangle)$ implies $\epsilon(\langle c', \sigma' \rangle)$, for $r \in \{as2, seq, guard, loop, par\}$.

THEOREM 7 (**1 : ASSN2**), (**1 : SEQ**), (**1 : GD**), (**1 : LOOP**), (**1 : PAR**) are correct. That is, $p_l \overset{\Rightarrow}{\cong} p_r$ for p_l and p_r a left- and right-hand side of one of these rules.

Proof: In order to prove that the left- and right-hand sides of each of these rules are testing equivalent, $p_l \overset{\Rightarrow}{\cong} p_r$, we must show that for every closing testing context C, $C[p_l] \overset{\Rightarrow}{\cong}_{obserr} C[p_r]$. Using Corollary 2, we can simplify the proof into showing three properties.

1. If $C[p_l]$ has a successful computation then so does $C[p_r]$. Suppose $C[p_l]$ has a successful computation, written
$$\langle C[p_l], \iota \rangle \xrightarrow{*} \langle c, \sigma[x_{\text{success}} = \mathbf{t}] \rangle.$$
Then by induction using Lemma $\Rightarrow_r$, we can find a computation
$$\langle C[p_r], \iota \rangle \xrightarrow{*} \langle c', \sigma'[x_{\text{success}} = \mathbf{t}] \rangle$$
such that
$$\langle c, \sigma[x_{\text{success}} = \mathbf{t}] \rangle \approx_r \langle c', \sigma'[x_{\text{success}} = \mathbf{t}] \rangle.$$
The result then follows from the Success Lemma.

2. If $C[p_r]$ has a successful computation then so does $C[p_l]$. Suppose $C[p_r]$ has a successful computation, written
$$\langle C[p_r], \iota \rangle \xrightarrow{*} \langle c, \sigma[x_{\text{success}} = \mathbf{t}] \rangle.$$

Then by induction using Lemma $\Leftarrow_r$, we can find a computation

$$\langle C[p_l], \iota\rangle \xrightarrow{*} \langle c', \sigma'[x_{\text{success}} = \mathbf{t}]\rangle$$

such that

$$\langle c, \sigma[x_{\text{success}} = \mathbf{t}]\rangle \approx_r \langle c', \sigma'[x_{\text{success}} = \mathbf{t}]\rangle.$$

The result then follows from the Success Lemma.

3. If $C[p_r]$ has a failing computation then so does $C[p_l]$. Suppose $C[p_r]$ has a successful computation, written

$$\langle C[p_r], \iota\rangle \xrightarrow{*} \mathbf{ERROR}$$

Then by induction using Lemma $\Leftarrow_r$, case 1, we can find a computation

$$\langle C[p_l], \iota\rangle \xrightarrow{*} \langle c', \sigma'\rangle$$

and either Case 1 of Lemma $\Leftarrow_r$ holds, in which case the conclusion is immediate, or Case 2 holds in which case the conclusion follows directly from the Error Lemma.

■

We now may prove the correctness of this phase, Lemma 2.

Proof: We have $m_0 \Rightarrow_1^N m_1$ and $m_0^t \Rightarrow_1^N m_1^t$, show $m_0 || m_0^t \overset{\Rightarrow}{\cong}_{obserr} m_1 || m_1^t$. By transitivity on Theorems phase1-init, 6 and 7 we have $m_0 \overset{\Rightarrow}{\cong} m_1$ and $m_0^t \overset{\Rightarrow}{\cong}_{obserr} m_1^t$. Thus, by congruence of $\overset{\Rightarrow}{\cong}$, $m_0 || m_0^t \overset{\Rightarrow}{\cong} m_1 || m_0^t \overset{\Rightarrow}{\cong} m_1 || m_1^t$, and then picking the context in the definition of $\overset{\Rightarrow}{\cong}$ to be $\bullet$, we have $m_0 || m_0^t \overset{\Rightarrow}{\cong}_{obserr} m_1 || m_1^t$, completing the proof. ■

D.3. Modularization Correctness

In this section we present the correctness proofs for modularization. The proof technique used is similar to that used for the Phase 1 transformations and handshaking expansion, see those proofs for more details. We only present here the proof for (**3 : MOD ACT**), the passive modularization rule (**3 : MOD PAS**) is similar (but simpler), so for space considerations will be skipped.

Definition 38 $\approx_{ma}$ is the least relation with the following properties for arbitrary C, C_0, C'.

1. $\langle R_1[!p \uparrow], \sigma[!p = \mathbf{f}, ?p = \mathbf{f}]\rangle \approx_{ma}$
   $\langle R_1'[!p_i \uparrow], \sigma[!p = \mathbf{f}, ?p = \mathbf{f}, !p_i = \mathbf{f}, ?p_i = \mathbf{f}]\rangle$

2. $\langle R_1[!p \uparrow], \sigma[!p = \mathbf{f}, ?p = \mathbf{f}]\rangle \approx_{ma}$
$\langle R_2'[!p :=!p_i \vee \ldots \vee !p_n], \sigma[!p = \mathbf{f}, ?p = \mathbf{f}, !p_i = \mathbf{t}, ?p_i = \mathbf{f}]\rangle$

3. $\langle R_2[[!p \longrightarrow ?p \uparrow]], \sigma[!p = \mathbf{t}, ?p = \mathbf{f}]\rangle \approx_{ma}$
$\langle R_3'[[!p \longrightarrow ?p \uparrow]], \sigma[!p = \mathbf{t}, ?p = \mathbf{f}, !p_i = \mathbf{t}, ?p_i = \mathbf{f}]\rangle$

4. $\langle R_3[?p \uparrow], \sigma[!p = \mathbf{t}, ?p = \mathbf{f}]\rangle \approx_{ma}$
$\langle R_4'[?p \uparrow], \sigma[!p = \mathbf{t}, ?p = \mathbf{f}, !p_i = \mathbf{t}, ?p_i = \mathbf{f}]\rangle$

5. $\langle R_4[[?p \longrightarrow !p \downarrow]], \sigma[!p = \mathbf{t}, ?p = \mathbf{t}]\rangle \approx_{ma}$
$\langle R_5'[?p_i :=!p_i \ \mathbf{C}\ ?p], \sigma[!p = \mathbf{t}, ?p = \mathbf{t}, !p_i = \mathbf{t}, ?p_i = \mathbf{f}]\rangle$

6. $\langle R_4[[?p \longrightarrow !p \downarrow]], \sigma[!p = \mathbf{t}, ?p = \mathbf{t}]\rangle \approx_{ma}$
$\langle R_6'[[?p_i \longrightarrow !p_i \downarrow]], \sigma[!p = \mathbf{t}, ?p = \mathbf{t}, !p_i = \mathbf{t}, ?p_i = \mathbf{t}]\rangle$

7. $\langle R_5[!p \downarrow], \sigma[!p = \mathbf{t}, ?p = \mathbf{t}]\rangle \approx_{ma}$
$\langle R_7'[!p_i \downarrow], \sigma[!p = \mathbf{t}, ?p = \mathbf{t}, !p_i = \mathbf{t}, ?p_i = \mathbf{t}]\rangle$

8. $\langle R_5[!p \downarrow], \sigma[!p = \mathbf{t}, ?p = \mathbf{t}]\rangle \approx_{ma}$
$\langle R_8'[!p :=!p_1 \vee \ldots \vee !p_n], \sigma[!p = \mathbf{t}, ?p = \mathbf{t}, !p_i = \mathbf{f}, ?p_i = \mathbf{t}]\rangle$

9. $\langle R_6[[\neg !p \longrightarrow ?p \downarrow]], \sigma[!p = \mathbf{f}, ?p = \mathbf{t}]\rangle \approx_{ma}$
$\langle R_9'[[\neg !p \longrightarrow ?p \downarrow]], \sigma[!p = \mathbf{f}, ?p = \mathbf{t}, !p_i = \mathbf{f}, ?p_i = \mathbf{t}]\rangle$

10. $\langle R_7[?p \downarrow], \sigma[!p = \mathbf{f}, ?p = \mathbf{t}]\rangle \approx_{ma}$
$\langle R_{10}'[?p \downarrow], \sigma[!p = \mathbf{f}, ?p = \mathbf{t}, !p_i = \mathbf{f}, ?p_i = \mathbf{t}]\rangle$

11. $\langle R_8[[\neg ?p \longrightarrow \mathbf{skip}]], \sigma[!p = \mathbf{f}, ?p = \mathbf{f}]\rangle \approx_{ma}$
$\langle R_{11}'[?p_i :=!p_i \ \mathbf{C}\ ?p], \sigma[!p = \mathbf{f}, ?p = \mathbf{f}, !p_i = \mathbf{f}, ?p_i = \mathbf{t}]\rangle$

12. $\langle R_8[[\neg ?p \longrightarrow \mathbf{skip}]], \sigma[!p = \mathbf{f}, ?p = \mathbf{f}]\rangle \approx_{ma}$
$\langle R_{12}'[[\neg ?p_i \longrightarrow \mathbf{skip}]], \sigma[!p = \mathbf{f}, ?p = \mathbf{f}, !p_i = \mathbf{f}, ?p_i = \mathbf{f}]\rangle$

where

$$\begin{aligned}
&R_1[!p\uparrow] &&= C'[C_0[\mathbf{PHS}][p_1]]\\
&R_2[[!p\longrightarrow ?p\uparrow]] &&= C'[C_0[[!p\longrightarrow ?p\uparrow];[\neg !p\longrightarrow ?p\downarrow][p_2]]\\
&R_3[?p\uparrow] &&= C'[C_0[?p\uparrow;[\neg !p\longrightarrow ?p\downarrow]][p_2]]\\
&R_4[[?p\longrightarrow !p\downarrow]] &&= C'[C_0[[\neg !p\longrightarrow ?p\downarrow]][p_2]]\\
&R_5[!p\downarrow] &&= C'[C_0[[\neg !p\longrightarrow ?p\downarrow]][p_3]]\\
&R_6[[\neg !p\longrightarrow ?p\downarrow]] &&= C'[C_0[[\neg !p\longrightarrow ?p\downarrow]][p_5]]\\
&R_7[?p\downarrow] &&= C'[C_0[?p\downarrow][p_5]]\\
&R_8[[\neg ?p\longrightarrow \mathbf{skip}]] &&= C'[C_0[\mathbf{skip}][p_5]]\\
&R'_1[!p_i\uparrow] &&= C'[C_0[\mathbf{PHS}][p'_1]]\\
&R'_2[!p:=!p_i\vee\ldots\vee !p_n] &&= C'[C_0[\mathbf{PHS}][p'_2]]\\
&R'_3[[!p\longrightarrow ?p\uparrow]] &&= C'[C_0[[!p\longrightarrow ?p\uparrow];[\neg !p\longrightarrow ?p\downarrow]][p'_3]]\\
&R'_4[?p\uparrow] &&= C'[C_0[?p\uparrow;[\neg !p\longrightarrow ?p\downarrow]][p'_3]]\\
&R'_5[?p_i:=!p_i\ \mathbf{C}\ ?p] &&= C'[C_0[[\neg !p\longrightarrow ?p\downarrow]][p'_4]]\\
&R'_6[[?p_i\longrightarrow !p_i\downarrow]] &&= C'[C_0[[\neg !p\longrightarrow ?p\downarrow]][p'_3]]\\
&R'_7[!p_i\downarrow] &&= C'[C_0[[\neg !p\longrightarrow ?p\downarrow]][p'_5]]\\
&R'_8[!p:=!p_1\vee\ldots\vee !p_n] &&= C'[C_0[[\neg !p\longrightarrow ?p\downarrow]][p'_6]]\\
&R'_9[[\neg !p\longrightarrow ?p\downarrow]] &&= C'[C_0[[\neg !p\longrightarrow ?p\downarrow]][p'_7]]\\
&R'_{10}[?p\downarrow] &&= C'[C_0[?p\downarrow][p'_7]]\\
&R'_{11}[?p_i:=!p_i\ \mathbf{C}\ ?p] &&= C'[C_0[\mathbf{skip}][p'_8]]\\
&R'_{12}[[\neg ?p_i\longrightarrow \mathbf{skip}]] &&= C'[C_0[\mathbf{skip}][p'_9]]
\end{aligned}$$

$$\begin{aligned}
p_1 &= C[\mathbf{AHS}(!p,?p)]\ldots[\mathbf{AHS}(!p,?p)]\\
p_2 &= C[\mathbf{AHS}(!p,?p)]\ldots[[?p\longrightarrow !p\downarrow];[\neg ?p\longrightarrow \mathbf{skip}]]\ldots[\mathbf{AHS}(!p,?p)]\\
p_3 &= C[\mathbf{AHS}(!p,?p)]\ldots[!p\downarrow;[\neg ?p\longrightarrow \mathbf{skip}]]\ldots[\mathbf{AHS}(!p,?p)]\\
p_4 &= C[\mathbf{AHS}(!p,?p)]\ldots[[\neg ?p\longrightarrow \mathbf{skip}]]\ldots[\mathbf{AHS}(!p,?p)]\\
p'_1 &= *[!p:=!p_1\vee\ldots\vee !p_n]\|*[?p_1:=!p_1\ \mathbf{C}\ ?p]\|\ldots\|*[?p_n:=!p_n\ \mathbf{C}\ ?p]\|\\
&\quad C[\mathbf{AHS}(!p_1,?p_1)]\ldots[\mathbf{AHS}(!p_n,?p_n)]\\
p'_2 &= !p:=!p_1\vee\ldots\vee !p_n;*[!p:=!p_1\vee\ldots\vee !p_n]\|\\
&\quad *[?p_1:=!p_1\ \mathbf{C}\ ?p]\|\ldots\|*[?p_n:=!p_n\ \mathbf{C}\ ?p]\|\\
&\quad C[\mathbf{AHS}(!p_1,?p_1)]\ldots[[?p_i\longrightarrow !p_i];[\neg !p_i\longrightarrow ?p_i\downarrow]]\ldots[\mathbf{AHS}(!p_n,?p_n)]\\
p'_3 &= *[!p:=!p_1\vee\ldots\vee !p_n]\|*[?p_1:=!p_1\ \mathbf{C}\ ?p]\|\ldots\|*[?p_n:=!p_n\ \mathbf{C}\ ?p]\|\\
&\quad C[\mathbf{AHS}(!p_1,?p_1)]\ldots[[?p_i\longrightarrow !p_i];[\neg !p_i\longrightarrow ?p_i\downarrow]]\ldots[\mathbf{AHS}(!p_n,?p_n)]\\
p'_4 &= *[!p:=!p_1\vee\ldots\vee !p_n]\|*[?p_1:=!p_1\ \mathbf{C}\ ?p]\|\ldots\|\\
&\quad ?p_i:=!p_i\ \mathbf{C}\ ?p;*[?p_i:=!p_i\ \mathbf{C}\ ?p]\|\ldots\|*[?p_n:=!p_n\ \mathbf{C}\ ?p]\|\\
&\quad C[\mathbf{AHS}(!p_1,?p_1)]\ldots[[?p_i\longrightarrow !p_i];[\neg !p_i\longrightarrow ?p_i\downarrow]]\ldots[\mathbf{AHS}(!p_n,?p_n)]\\
p'_5 &= *[!p:=!p_1\vee\ldots\vee !p_n]\|*[?p_1:=!p_1\ \mathbf{C}\ ?p]\|\ldots\|*[?p_n:=!p_n\ \mathbf{C}\ ?p]\|\\
&\quad C[\mathbf{AHS}(!p_1,?p_1)]\ldots[!p_i\downarrow;[\neg ?p_i\longrightarrow \mathbf{skip}]]\ldots[\mathbf{AHS}(!p_n,?p_n)]\\
p'_6 &= !p:=!p_1\vee\ldots\vee !p_n];*[!p:=!p_1\vee\ldots\vee !p_n]\|\\
&\quad *[?p_1:=!p_1\ \mathbf{C}\ ?p]\|\ldots\|*[?p_n:=!p_n\ \mathbf{C}\ ?p]\|\\
&\quad C[\mathbf{AHS}(!p_1,?p_1)]\ldots[[\neg ?p_i\longrightarrow \mathbf{skip}]]\ldots[\mathbf{AHS}(!p_n,?p_n)]\\
p'_7 &= *[!p:=!p_1\vee\ldots\vee !p_n]\|*[?p_1:=!p_1\ \mathbf{C}\ ?p]\|\ldots\|*[?p_n:=!p_n\ \mathbf{C}\ ?p]\|\\
&\quad C[\mathbf{AHS}(!p_1,?p_1)]\ldots[[\neg ?p_i\longrightarrow \mathbf{skip}]]\ldots[\mathbf{AHS}(!p_n,?p_n)]\\
p'_8 &= *[!p:=!p_1\vee\ldots\vee !p_n]\|*[?p_1:=!p_1\ \mathbf{C}\ ?p]\|\ldots\|\\
&\quad ?p_i:=!p_i\ \mathbf{C}\ ?p;*[?p_i:=!p_i\ \mathbf{C}\ ?p]\|\ldots\|*[?p_n:=!p_n\ \mathbf{C}\ ?p]\|\\
&\quad C[\mathbf{AHS}(!p_1,?p_1)]\ldots[[\neg ?p_i\longrightarrow \mathbf{skip}]]\ldots[\mathbf{AHS}(!p_n,?p_n)]\\
p'_9 &= *[!p:=!p_1\vee\ldots\vee !p_n]\|*[?p_1:=!p_1\ \mathbf{C}\ ?p]\|\ldots\|*[?p_n:=!p_n\ \mathbf{C}\ ?p]\|\\
&\quad C[\mathbf{AHS}(!p_1,?p_1)]\ldots[[\neg ?p_i\longrightarrow \mathbf{skip}]]\ldots[\mathbf{AHS}(!p_n,?p_n)]
\end{aligned}$$

LEMMA 24 ($\Rightarrow_m$) If given semantically well-formed configurations $\langle c_0, \sigma_0\rangle$, $\langle c'_0, \sigma'_0\rangle$ we have $\langle c_0, \sigma_0\rangle \longrightarrow \langle c_1, \sigma_1\rangle$ and $\langle c_0, \sigma_0\rangle \approx_m \langle c'_0, \sigma'_0\rangle$ then there exist c'_i, σ'_i $(0 \leq i \leq m)$ such that $\langle c'_0, \sigma'_0\rangle \xrightarrow{*} \langle c'_m, \sigma'_m\rangle$ and $\langle c'_0, \sigma'_0\rangle \approx_{as2} \langle c'_m, \sigma'_m\rangle$, where $m \in \{ma, mp\}$.

LEMMA 25 ($\Leftarrow_m$) If given configurations $\langle c_0, \sigma_0\rangle$, $\langle c'_0, \sigma'_0\rangle$ we have $\langle c_0, \sigma_0\rangle \approx_m \langle c'_0, \sigma'_0\rangle$ and $\langle c'_0, \sigma'_0\rangle \longrightarrow \langle c'_1, \sigma'_1\rangle$ then either

1. $\langle c_0, \sigma_0\rangle \xrightarrow{*}$ **ERROR**; or
2. we can find c'_i, σ'_i $(0 \leq i \leq n)$ such that $\langle c_0, \sigma_0\rangle \xrightarrow{*} \langle c_n, \sigma_n\rangle$ and $\langle c_n, \sigma_n\rangle \approx_m \langle c'_1, \sigma'_1\rangle$.

for $m \in \{ma, mp\}$.

LEMMA 26 (SUCCESS LEMMA) If $\langle c, \sigma\rangle \approx_m \langle c', \sigma'\rangle$ then $\sigma(x_{\text{success}}) = \sigma'(x_{\text{success}})$, for $m \in \{ma, mp\}$.

LEMMA 27 (ERROR LEMMA) If $\langle c, \sigma\rangle \approx_m \langle c', \sigma'\rangle$ then $\epsilon(\langle c, \sigma\rangle)$ implies $\epsilon(\langle c', \sigma'\rangle)$, for $m \in \{ma, mp\}$.

THEOREM 8 (**3 : MOD ACT**) and (**1 : MOD PAS**) preserve $\overset{\Rightarrow}{\cong}$.

With this theorem, we may prove the correctness of the entire phase of compilation, Lemma 5. See the end of section D.2 for the general structure of this proof.

D.4. Reshuffling Correctness

The Reshuffling Lemma is key in the correctness proofs for this phase. It states that a passive handshake may be reshuffled, provided its corresponding active handshake has not been reshuffled (we call this a *pure* handshake). This Lemma suffices to prove correctness, because by performing reshuffling steps in a certain order it is always possible to reshuffle passive commications in the presence of a non-reshuffled active communication.

Before stating the Lemma we introduce the abbreviations **AHS** and **PHS** for **AHS**$(!p, ?p)$ and **PHS**$(!p, ?p)$. Also, **RPHS**$(!p, ?p)$, abbreviated **RPHS**, is the reshuffled passive handshake

$$[!p \longrightarrow \textbf{skip}]; c_0; ?p\uparrow; c_1; [\neg !p \longrightarrow \textbf{skip}]; c_2; ?p\downarrow$$

LEMMA 28 (RESHUFFLING PRINCIPLE) .

$$\begin{array}{l} C' \ [\textbf{with r } ?p \textbf{ w } !p \textbf{ do } C[\textbf{AHS}]\textbf{end}] \\ \quad\ [\textbf{with w } ?p \textbf{ r } !p \textbf{ do } [!p \longrightarrow \textbf{skip}]; c_0; c_1; c_2; \textbf{PHS end}] \\ \overset{\Rightarrow}{\cong}_{obserr} \\ C' \ [\textbf{with r } ?p \textbf{ w } !p \textbf{ do } C[\textbf{AHS}]\textbf{end}] \\ \quad\ [\textbf{with w } ?p \textbf{ r } !p \textbf{ do } \textbf{RPHS end}] \end{array}$$

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

Relation $\approx_{rshfl}$ is defined to relate intermediate computation states of non-reshuffled terms with their reshufled counterparts. Many cases are required because the active handshake runs in parallel with the c_i computations, and all interleavings must be considered. The proofs in this section leave out some details, see section D.1 for a complete description of the technique for the case of handshake expansion.

Definition 39 $\approx_{rshfl}$ is the least relation with the following properties, for arbitrary M, C, c_0, c_1, c_2 containing no occcurrences of $!p, ?p$:

1. $\langle C[\mathbf{AHS}][[!p \longrightarrow \mathbf{skip}]; c_0; c_1; c_2; \mathbf{PHS}], \sigma[!p = \mathbf{f}, ?p = \mathbf{f}]\rangle \approx_{rshfl}$ $\langle C[\mathbf{AHS}][\mathbf{RPHS}], \sigma[!p = \mathbf{f}, ?p = \mathbf{f}]\rangle$
2. $\langle M_{[\mathbf{AHS}][[!p \longrightarrow \mathbf{skip}]; c_0; c_1; c_2; \mathbf{PHS}]}[\mathbf{AHS}_1], \sigma[!p = \mathbf{t}, ?p = \mathbf{f}]\rangle \approx_{rshfl}$ $\langle M_{[\mathbf{AHS}][\mathbf{RPHS}]}[\mathbf{AHS}_1], \sigma[!p = \mathbf{t}, ?p = \mathbf{f}]\rangle$
3. $\langle M_{[\mathbf{AHS}][[!p \longrightarrow \mathbf{skip}]; c_0; c_1; c_2; \mathbf{PHS}]}[\mathbf{AHS}_1][c_0; c_1; c_2; \mathbf{PHS}], \sigma[!p = \mathbf{t}, ?p = \mathbf{f}]\rangle \approx_{rshfl}$ $\langle M_{[\mathbf{AHS}][\mathbf{RPHS}]}[\mathbf{AHS}_1][c_0; ?p \uparrow; c_1; [\neg !p \longrightarrow \mathbf{skip}]; c_2; ?p \downarrow], \sigma[!p = \mathbf{t}, ?p = \mathbf{f}]\rangle$
4. $\langle M_{[\mathbf{AHS}][[!p \longrightarrow \mathbf{skip}]; c_0; c_1; c_2; \mathbf{PHS}]}[\mathbf{AHS}_1][c_1; c_2; \mathbf{PHS}], \sigma[!p = \mathbf{t}, ?p = \mathbf{f}]\rangle \approx_{rshfl}$ $\langle M_{[\mathbf{AHS}][\mathbf{RPHS}]}[\mathbf{AHS}_1][?p \uparrow; c_1; [\neg !p \longrightarrow \mathbf{skip}]; c_2; ?p \downarrow], \sigma[!p = \mathbf{t}, ?p = \mathbf{f}]\rangle$
5. $\langle M_{[\mathbf{AHS}][[!p \longrightarrow \mathbf{skip}]; c_0; c_1; c_2; \mathbf{PHS}]}[\mathbf{AHS}_1][c_1; c_2; \mathbf{PHS}], \sigma[!p = \mathbf{t}, ?p = \mathbf{f}]\rangle \approx_{rshfl}$ $\langle M_{[\mathbf{AHS}][\mathbf{RPHS}]}[\mathbf{AHS}_1][c_1; [\neg !p \longrightarrow \mathbf{skip}]; c_2; ?p \downarrow], \sigma[!p = \mathbf{t}, ?p = \mathbf{t}]\rangle$
6. $\langle M_{[\mathbf{AHS}][[!p \longrightarrow \mathbf{skip}]; c_0; c_1; c_2; \mathbf{PHS}]}[\mathbf{AHS}_1][c_2; \mathbf{PHS}], \sigma[!p = \mathbf{t}, ?p = \mathbf{f}]\rangle \approx_{rshfl}$ $\langle M_{[\mathbf{AHS}][\mathbf{RPHS}]}[\mathbf{AHS}_1][[\neg !p \longrightarrow \mathbf{skip}]; c_2; ?p \downarrow], \sigma[!p = \mathbf{t}, ?p = \mathbf{t}]\rangle$
7. $\langle M_{[\mathbf{AHS}][[!p \longrightarrow \mathbf{skip}]; c_0; c_1; c_2; \mathbf{PHS}]}[\mathbf{AHS}_1][c_1; c_2; \mathbf{PHS}], \sigma[!p = \mathbf{t}, ?p = \mathbf{f}]\rangle \approx_{rshfl}$ $\langle M_{[\mathbf{AHS}][\mathbf{RPHS}]}[\mathbf{AHS}_2][c_1; [\neg !p \longrightarrow \mathbf{skip}]; c_2; ?p \downarrow], \sigma[!p = \mathbf{t}, ?p = \mathbf{t}]\rangle$
8. $\langle M_{[\mathbf{AHS}][[!p \longrightarrow \mathbf{skip}]; c_0; c_1; c_2; \mathbf{PHS}]}[\mathbf{AHS}_1][c_2; \mathbf{PHS}], \sigma[!p = \mathbf{t}, ?p = \mathbf{f}]\rangle \approx_{rshfl}$ $\langle M_{[\mathbf{AHS}][\mathbf{RPHS}]}[\mathbf{AHS}_2][[\neg !p \longrightarrow \mathbf{skip}]; c_2; ?p \downarrow], \sigma[!p = \mathbf{t}, ?p = \mathbf{t}]\rangle$
9. $\langle M_{[\mathbf{AHS}][[!p \longrightarrow \mathbf{skip}]; c_0; c_1; c_2; \mathbf{PHS}]}[\mathbf{AHS}_1][c_1; c_2; \mathbf{PHS}], \sigma[!p = \mathbf{f}, ?p = \mathbf{t}]\rangle \approx_{rshfl}$ $\langle M_{[\mathbf{AHS}][\mathbf{RPHS}]}[\mathbf{AHS}_3][c_1; [\neg !p \longrightarrow \mathbf{skip}]; c_2; ?p \downarrow], \sigma[!p = \mathbf{f}, ?p = \mathbf{t}]\rangle$
10. $\langle M_{[\mathbf{AHS}][[!p \longrightarrow \mathbf{skip}]; c_0; c_1; c_2; \mathbf{PHS}]}[\mathbf{AHS}_1][c_2; \mathbf{PHS}], \sigma[!p = \mathbf{t}, ?p = \mathbf{f}]\rangle \approx_{rshfl}$ $\langle M_{[\mathbf{AHS}][\mathbf{RPHS}]}[\mathbf{AHS}_3][[\neg !p \longrightarrow \mathbf{skip}]; c_2; ?p \downarrow], \sigma[!p = \mathbf{f}, ?p = \mathbf{t}]\rangle$
11. $\langle M_{[\mathbf{AHS}][[!p \longrightarrow \mathbf{skip}]; c_0; c_1; c_2; \mathbf{PHS}]}[\mathbf{AHS}_1][c_2; \mathbf{PHS}], \sigma[!p = \mathbf{t}, ?p = \mathbf{f}]\rangle \approx_{rshfl}$ $\langle M_{[\mathbf{AHS}][\mathbf{RPHS}]}[\mathbf{AHS}_3][c_2; ?p \downarrow], \sigma[!p = \mathbf{f}, ?p = \mathbf{t}]\rangle$

12. $\langle M_{[\mathbf{AHS}][[!p \longrightarrow \mathbf{skip}];c_0;c_1;c_2;\mathbf{PHS}]}[\mathbf{AHS}_1][\mathbf{PHS}], \sigma[!p = \mathbf{t}, ?p = \mathbf{f}]\rangle \approx_{rshfl}$
$\langle M_{[\mathbf{AHS}][\mathbf{RPHS}]}[\mathbf{AHS}_3][?p\downarrow], \sigma[!p = \mathbf{f}, ?p = \mathbf{t}]\rangle$

13. $\langle M_{[\mathbf{AHS}][[!p \longrightarrow \mathbf{skip}];c_0;c_1;c_2;\mathbf{PHS}]}[\mathbf{AHS}_1][\mathbf{PHS}_1], \sigma[!p = \mathbf{t}, ?p = \mathbf{f}]\rangle \approx_{rshfl}$
$\langle M_{[\mathbf{AHS}][\mathbf{RPHS}]}[\mathbf{AHS}_3][?p\downarrow], \sigma[!p = \mathbf{f}, ?p = \mathbf{t}]\rangle$

14. $\langle M_{[\mathbf{AHS}][[!p \longrightarrow \mathbf{skip}];c_0;c_1;c_2;\mathbf{PHS}]}[\mathbf{AHS}_1][\mathbf{PHS}_2], \sigma[!p = \mathbf{t}, ?p = \mathbf{t}]\rangle \approx_{rshfl}$
$\langle M_{[\mathbf{AHS}][\mathbf{RPHS}]}[\mathbf{AHS}_3][?p\downarrow], \sigma[!p = \mathbf{f}, ?p = \mathbf{t}]\rangle$

15. $\langle M_{[\mathbf{AHS}][[!p \longrightarrow \mathbf{skip}];c_0;c_1;c_2;\mathbf{PHS}]}[\mathbf{AHS}_2][\mathbf{PHS}_2], \sigma[!p = \mathbf{t}, ?p = \mathbf{t}]\rangle \approx_{rshfl}$
$\langle M_{[\mathbf{AHS}][\mathbf{RPHS}]}[\mathbf{AHS}_3][?p\downarrow], \sigma[!p = \mathbf{f}, ?p = \mathbf{t}]\rangle$

16. $\langle M_{[\mathbf{AHS}][[!p \longrightarrow \mathbf{skip}];c_0;c_1;c_2;\mathbf{PHS}]}[\mathbf{AHS}_3][\mathbf{PHS}_2], \sigma[!p = \mathbf{f}, ?p = \mathbf{t}]\rangle \approx_{rshfl}$
$\langle M_{[\mathbf{AHS}][\mathbf{RPHS}]}[\mathbf{AHS}_3][?p\downarrow], \sigma[!p = \mathbf{f}, ?p = \mathbf{t}]\rangle$

17. $\langle M_{[\mathbf{AHS}][[!p \longrightarrow \mathbf{skip}];c_0;c_1;c_2;\mathbf{PHS}]}[\mathbf{AHS}_3][\mathbf{PHS}_3], \sigma[!p = \mathbf{f}, ?p = \mathbf{t}]\rangle \approx_{rshfl}$
$\langle M_{[\mathbf{AHS}][\mathbf{RPHS}]}[\mathbf{AHS}_3][?p\downarrow], \sigma[!p = \mathbf{f}, ?p = \mathbf{t}]\rangle$

18. $\langle M_{[\mathbf{AHS}][[!p \longrightarrow \mathbf{skip}];c_0;c_1;c_2;\mathbf{PHS}]}[\mathbf{AHS}_3][\mathbf{skip}], \sigma[!p = \mathbf{f}, ?p = \mathbf{t}]\rangle \approx_{rshfl}$
$\langle M_{[\mathbf{AHS}][\mathbf{RPHS}]}[\mathbf{AHS}_3][\mathbf{skip}], \sigma[!p = \mathbf{f}, ?p = \mathbf{t}]\rangle$

where $\mathbf{AHS}_i$, $\mathbf{PHS}_i$ are as defined in section D.1, and $\mathbf{RPHS}_1 \ldots \mathbf{RPHS}_6$ are

$$\begin{array}{lcl lcl}
\mathbf{RPHS}_1 & = & c_0; ?p\uparrow; c_1; [\neg !p \longrightarrow \mathbf{skip}]; c_2; ?p\downarrow & \mathbf{RPHS}_4 & = & [\neg !p \longrightarrow \mathbf{skip}]; c_2; ?p\downarrow \\
\mathbf{RPHS}_2 & = & ?p\uparrow; c_1; [\neg !p \longrightarrow \mathbf{skip}]; c_2; ?p\downarrow & \mathbf{RPHS}_5 & = & c_2; ?p\downarrow \\
\mathbf{RPHS}_3 & = & c_1; [\neg !p \longrightarrow \mathbf{skip}]; c_2; ?p\downarrow & \mathbf{RPHS}_6 & = & ?p\downarrow
\end{array}$$

Cases 5.–10. describe all the possible interleavings of the active handshake transitioning to $\mathbf{AHS}_2$ and $\mathbf{AHS}_3$ with execution of c_1 in the reshuffled form, since the two may occur in parallel. Cases 13.–17. describe how the non-reshuffled synchronization is completed.

We now prove two lemmas that show the $\approx_{rshfl}$ relation is preserved by computation.

LEMMA 29 If given semantically well-formed configurations $\langle c_0, \sigma_0\rangle$ and $\langle c_0', \sigma_0'\rangle$ we have

$$\begin{tikzcd}
\langle c_0, \sigma_0\rangle \arrow[r] \arrow[d, phantom, "\wr\wr_{rshfl}"] & \langle c_1, \sigma_1\rangle \\
\langle c_0', \sigma_0'\rangle &
\end{tikzcd}$$

then we may find $c_i', \sigma_i', 0 \leq i \leq m$, such that

$$\begin{tikzcd}
\langle c_0, \sigma_0\rangle \arrow[r] \arrow[d, phantom, "\wr\wr_{rshfl}"] & \langle c_1, \sigma_1\rangle \arrow[d, phantom, "\wr\wr_{rshfl}"] \\
\langle c_0', \sigma_0'\rangle \arrow[r, "*"] & \langle c_m', \sigma_m'\rangle
\end{tikzcd}$$

Lemma 30 If given configurations $\langle c_0, \sigma_0 \rangle$ and $\langle c'_0, \sigma'_0 \rangle$ we have

$$\begin{tikzcd}
\langle c_0, \sigma_0 \rangle \arrow[d, phantom, "\wr\wr_{rshfl}"] & \\
\langle c'_0, \sigma'_0 \rangle \arrow[r] & \langle c'_1, \sigma'_1 \rangle
\end{tikzcd}$$

then either

1. $\langle c_0, \sigma_0 \rangle \xrightarrow{*} \mathbf{ERROR}$, or
2. we may find $c_i, \sigma_i, 0 \leq i \leq n$, such that

$$\begin{tikzcd}
\langle c_0, \sigma_0 \rangle \arrow[r, "*"] \arrow[d, phantom, "\wr\wr_{rshfl}"] & \langle c_n, \sigma_n \rangle \arrow[d, phantom, "\wr\wr_{rshfl}"] \\
\langle c'_0, \sigma'_0 \rangle \arrow[r] & \langle c'_1, \sigma'_1 \rangle
\end{tikzcd}$$

Lemma 31 If $\langle c, \sigma \rangle \approx_{rshfl} \langle c', \sigma' \rangle$, then $\sigma(x_{\text{success}}) = \sigma'(x_{\text{success}})$.

Lemma 32 If $\langle c, \sigma \rangle \approx_{rshfl} \langle c', \sigma' \rangle$, then $\varepsilon(\langle c', \sigma' \rangle)$ implies $\varepsilon(\langle c, \sigma \rangle)$.

We may now prove the result which allows one channel to be replaced by its handshake expansion, Lemma 28. The proof is similar to the proof of Lemma 4 at the end of Section D.1.

Two related results are needed to allow reshufflings of handshake protocols that have already been modularized. This is an annoying problem that must be faced at some point. The other possible solution is to reshuffle *before* modularizing, but then the modularization phase must take into account all the reshuffling. Proofs of these principles are similar to the previous reshuffling principle and will not be given here. For readibility, declarations have been removed from the Lemma.

Lemma 33 (Modularized Reshuffling Principles)

1.
$$\begin{array}{c}
*[!p :=!p_1 \vee \ldots \vee !p_n] \,\|\, *\,[?p_1 :=!p_1 \;\mathbf{C}\; ?p] \,\|\, \ldots \,\|\, *\,[?p_n :=!p_n \;\mathbf{C}\; ?p] \,\| \\
C[\mathbf{AHS}(!p_1, ?p_1)] \ldots [\mathbf{AHS}(!p_n, ?p_n)][[!p \longrightarrow \mathbf{skip}]; c_0; c_1; c_2; \mathbf{PHS}] \\
\overset{\Rightarrow}{\cong}_{obserr} \\
*[!p :=!p_1 \vee \ldots \vee !p_n] \,\|\, *\,[?p_1 :=!p_1 \;\mathbf{C}\; ?p] \,\|\, \ldots \,\|\, *\,[?p_n :=!p_n \;\mathbf{C}\; ?p] \,\| \\
C[\mathbf{AHS}(!p_1, ?p_1)] \ldots [\mathbf{AHS}(!p_n, ?p_n)][\mathbf{RPHS}]
\end{array}$$

2.
$$\begin{array}{c}
*[?p :=?p_1 \vee \ldots \vee ?p_n] \,\| C[\mathbf{AHS}(!p, ?p)][[!p \longrightarrow \mathbf{skip}]; c_0; c_1; c_2; \mathbf{PHS}(!p, ?p_i)] \\
\overset{\Rightarrow}{\cong}_{obserr} \\
*[?p :=?p_1 \vee \ldots \vee ?p_n] \,\| C[\mathbf{AHS}(!p, ?p)][\mathbf{RPHS}(!p, ?p_i)]
\end{array}$$

provided there are no occurrences of $?p$ or $!p$ or $?p_i$ or $!p_i$, for any i, in C, c_0, c_1, or c_2.

There is in fact a third such principle, where both active and passive handshakes have been modularized, but it is not needed in our particular compilation method.

There is one additional matter that must be addressed. The guard reshuffling rule also parallelizes the guards, so we must prove a Lemma that states the reshuffled guard is equivalent to the parallelized reshuffled guard, and that together with the reshuffling principle will allow this rule to be proved correct.

LEMMA 34

$$\begin{array}{c} *[[!s \longrightarrow [e_1 \longrightarrow [!s \longrightarrow ?s_1' \uparrow]; [\neg !s \longrightarrow \mathbf{AHS}(!s_1, ?s_1); ?s_1' \downarrow] [\!] \ldots [\!] \\ e_n \longrightarrow [!s \longrightarrow ?s_n' \uparrow]; [\neg !s \longrightarrow \mathbf{AHS}(!s_n, ?s_n); ?s_n' \downarrow]]] \\ \overset{\Rightarrow}{\cong} \\ *[[!s \wedge e_1 \longrightarrow ?s_1' \uparrow; [\neg !s \longrightarrow \mathbf{AHS}(!s_1, ?s_1); ?s_1' \downarrow]]] || \ldots || \\ *[[!s \wedge e_n \longrightarrow ?s_n' \uparrow; [\neg !s \longrightarrow \mathbf{AHS}(!s_n, ?s_n); ?s_n' \downarrow]]] \end{array}$$

This principle may be proved by the usual method of defining a relation $\approx$. It is informally justified because all guards must be mutually exclusive, and thus evaluating guards in parallel should not change meaning. One subtle issue is when one guard e_i becomes true, the execution of its body must not inadvertantly make some other guard e_j true and cause that guard to execute. For this reason, the reshuffling is critical, because the guard body will not execute until $!s$ is already low, so the precondition $!s \wedge e_j$ will be false for all j.

We now can prove correctness of the reshuffling phase, Lemma 6.

Proof: Assume $\Rightarrow_3 m_3$ and $m_3 \Rightarrow_4^N m_4$, show $m_3 \overset{\Rightarrow}{\cong}_4 m_4$. By the definition of $\overset{\Rightarrow}{\cong}_4$, this means we are given m_3^t such that $m_3 || m_3^t$ is closed, $\Rightarrow_3 m_3^t$, and $m_3^t \Rightarrow_4^N m_4^t$, and we wish to show $m_3 || m_3^t \overset{\Rightarrow}{\cong}_{obserr} m_4 || m_4^t$. By locality of rewriting we have $m_3 || m_3^t \Rightarrow_4^N m_4 || m_4^t$, so we may assume each rewrite step is a closed expression. Furthermore, since $\Rightarrow_4^N$ has the strong diamond property, *any* rewrite ordering suffices since all will produce the same normal form. We thus pick a particular ordering below to suit our purposes.

By inspection of the reshuffling rules in table 5, all of the transformations of those rules may be expressed in the form of a transformation from

$$[!p \longrightarrow \mathbf{skip}]; c_0; c_1; c_2; \mathbf{PHS}$$

to **RPHS** for some c_0, c_1, c_2, *except* for two rules. (**4 : PASS**) is of a slightly different form, placing some of the operations in parallel. This style of reshuffling is just as easily justified as the form presented in Lemma 28, by a nearly identical argument, so details will not be presented here. (**4 : GD**) is reshuffled and then parallelized, so we will consider this case separately. For all other rules, Each step in a derivation $m_3 || m_3^t \Rightarrow_4^N m_4 || m_4^t$ amounts to one replacement of $[!p \longrightarrow \mathbf{skip}]; c_0; c_1; c_2; \mathbf{PHS}$ by **RPHS**, so each rewriting step in this derivation preserves $\overset{\Rightarrow}{\cong}_{obserr}$ by Lemma 28, *provided* the reshuffling takes place in the presence of a pure active synchronization **AHS**, namely the active synchronization has yet to be reshuffled. For the guard case, rewriting preserves $\overset{\Rightarrow}{\cong}_{obserr}$ by the combination of Lemma 28 and Lemma 34, again provided the active handshake is pure.

We must thus establish that there is *some* derivation $m_3||m_3^t {\Rightarrow}_4^N m_4||m_4^t$ such that each reshuffling of **PHS** takes place in the context of a non-reshuffled ("pure") **AHS**, and the proof will be complete. Since ${\Rightarrow}_3 m_3$ and ${\Rightarrow}_3 m_3^t$, $m_3||m_3^t$ is of a very particular form: there must be a m_0, m_0^t such that $m_0 {\Rightarrow}_1 {\Rightarrow}_2 {\Rightarrow}_3 m_3$ and $m_0^t {\Rightarrow}_1 {\Rightarrow}_2 {\Rightarrow}_3 m_3^t$. Thus, we may use this fact to observe properties about the ports in $m_3||m_3^t$.

Define a directed graph G based on the syntax of m_3 and m_3^t. There is a node in this graph for every process in m_3 and m_3^t. There is an edge from each process containing an active handshake to each process containing its corresponding passive handshake (or modularization thereof). Inspection of the rules of phases 1-3 should convince the reader that these graphs have a structure based on the syntax tree of the original specifications m_0 and m_0^t: there are edges from each construct to its component arguments. Further, each active synchronization in the original specification has edges to all the corresponding passive synchronizations from the original specification. Passive synchronizations in the original specification are leaf processes with two in-edges, one for the corrresponding active process, and one to enable the passive process. This graph is thus a dag, a tree with cross-edges. We thus may apply reshuffling rules in a bottom-up order on the dag. First, leaves are reshuffled by (**4 : PASS**). Then, we iteratively reshuffle any node for which all its immediate children needing reshuffling have been reshuffled already.

With this ordering, a parent is always reshuffled after all of its children, thus active handshake protocols of the parent will not be altered at the time the child process is reshuffled.

Thus, this ordering defines a rewriting of $m_3||m_3^t {\Rightarrow}_4^N m_4||m_4^t$ such that, by Lemmas 28 and 33, each step preserves $\overset{\Rightarrow}{\cong}_{observ}$. Thus, $m_3||m_3^t \overset{\Rightarrow}{\cong}_{observ} m_4||m_4^t$ and the correctness of this phase is established. ■

D.5. Circuit Correctness

The proof of correctness of the last phase is similar to the phase 1 proof in that each rule may be shown to preserve correctness through use of a simulation relation $\approx$. However, the **ERROR** case is more complicated. In phases 2-4 it was obvious that the handshaking protocols were performed correctly since the protocols are represented by code **AHS** or **PHS**. In the circuits this is not clear, so we must prove that no violations of the protocol occur in any of the semantically well-formed circuit modules.

In the proofs below it is useful to have a precise characterization of all the possible well-formedness violations that may occur at this point.

LEMMA 35 For closed m such that ${\Rightarrow}_4 m_4 {\Rightarrow}_5^* m$, $m \xrightarrow{*}$ **ERROR** iff if $\langle m, \iota\rangle \xrightarrow{*} \langle m', \sigma'\rangle \rightarrow \langle m'', \sigma''\rangle$ and one of the following conditions hold for σ' and σ'':

1. There is an active modularization process (see (**3 : MOD ACT**)) with $?p = \mathbf{t}$, i.e. $\sigma'(?p) = \sigma''(?p) = \mathbf{t}$ and $!p_i \uparrow$ in this step, i.e. $\sigma'(!p_i) = \mathbf{f}$ and $\sigma''(!p_i) = \mathbf{t}$.

2. There is an active modularization process with $!p_i = !p_j = \mathbf{t}$, $i \neq j$, and $?p\uparrow$.
3. There is a passive modularization process with $?p_i\uparrow$, and $?p_j = \mathbf{t}$, for some $i \neq j$.
4. in an **ASSN** cell, $!s_1 = \mathbf{t}$ and $x = \mathbf{f}$ and $!s_0\uparrow$.
5. in an **ASSN** cell, $!s_0 = \mathbf{t}$ and $x = \mathbf{t}$ and $!s_1\uparrow$.
6. in a **GD** cell, $!s = \mathbf{t}$ while $e\downarrow$.

DEFINITION 40 For closed m s.t. $\Rightarrow_4 m_4 \Rightarrow_5^* m$, *$m$ strongly violates handshake protocols* iff there is a computation $\langle m, \iota\rangle \xrightarrow{*} \langle m', \sigma'\rangle \rightarrow \langle m'', \sigma''\rangle$ and one of the following conditions holds for the step from σ' to σ'', for some $!p, ?p$:

1. $?p = \mathbf{t}$ while $!p\uparrow$.
2. $?p = \mathbf{f}$ while $!p\downarrow$.
3. $!p = \mathbf{f}$ while $?p\uparrow$.
4. $!p = \mathbf{t}$ while $?p\downarrow$.

DEFINITION 41 For closed m s.t. $\Rightarrow_4 m_4 \Rightarrow_5^* m$, *$m$ violates handshake protocols* iff *m strongly violates handshake protocols*, and not by certain innocuous means: in a passive process process, $!p$'s value may change arbitrarily provided $!s$ is low, and in a guard process, $!s$ may change arbitrarily while e has a $\mathbf{f}$ value.

LEMMA 36 If for closed m such that $\Rightarrow_4 m_4 \Rightarrow_5^* m$, if m violates handshaking protocols then m is semantically ill-formed.

Proof: We prove the converse: suppose m is well-formed but violates handshake protocols, establish a contradiction. Suppose the first such instance was from variables $!p/?p$ in step k of the computation. Then, the violation must have been by one of the four cases and there was an assignment to $!p$ or $?p$ responsible for the problem. Since all assignments have been localized at this point in the compilation process, there is a single process where $!p/?p$ is assigned. Proceed by cases on which process this is. In each case, it may be shown that such a violation leads to contradiction, we leave out the lengthy case analysis. ■

We now define a family of relations $\approx_{\text{assnckt}}$, $\approx_{\text{seqckt}}$, $\approx_{\text{guardckt}}$, $\approx_{\text{actckt}}$, $\approx_{\text{passckt}}$, $\approx_{\text{loopckt}}$, and $\approx_{\text{skipckt}}$ (notated collectively as relations $\approx_{\text{ckt}}$) relating intermediate points in computation for each phase 5 rule. The proof of correctness of this phase is thus similar to the phase 1 proof. The definition of these relations is straightforward: the circuit process does not change because there are no events to be sequenced in a circuit. The left-hand side processes execute the appropriate actions step-by-step. We give the assignment rule as illustration and leave the others for the reader to fill in.

Definition 42 We define $\approx_{\text{assnckt}}$ as the least relation with the following properties, for arbitrary m and σ_0:

1. $\langle m\|\mathbf{ASSN}_{\text{lhs}}, \sigma\rangle \approx_{\text{assnckt}} \langle m\|\mathbf{ASSN}_{\text{rhs}}, \sigma\rangle$
   where $\sigma = \sigma_0[!s_1 = \mathbf{f}, ?s_1 = \mathbf{f}, !s_0 = \mathbf{f}, ?s_0 = \mathbf{f}]$
2. $\langle m\|(x\uparrow; \mathbf{PHS}(!s_1, ?s_1); \mathbf{ASSN}_{\text{lhs}}), \sigma\rangle \approx_{\text{assnckt}} \langle m\|\mathbf{ASSN}_{\text{rhs}}, \sigma\rangle$
   where $\sigma = \sigma_0[!s_1 = \mathbf{t}, ?s_1 = \mathbf{f}, !s_0 = \mathbf{f}, ?s_0 = \mathbf{f}]$
3. $\langle m\|(\mathbf{PHS}(!s_1, ?s_1); \mathbf{ASSN}_{\text{lhs}}), \sigma\rangle \approx_{\text{assnckt}} \langle m\|\mathbf{ASSN}_{\text{rhs}}, \sigma\rangle$
   where $\sigma = \sigma_0[!s_1 = \mathbf{t}, ?s_1 = \mathbf{f}, !s_0 = \mathbf{f}, ?s_0 = \mathbf{f}, x = \mathbf{t}]$
4. $\langle m\|([\neg !s_1 \longrightarrow ?s_1\downarrow]; \mathbf{ASSN}_{\text{lhs}}), \sigma\rangle \approx_{\text{assnckt}} \langle m\|\mathbf{ASSN}_{\text{rhs}}, \sigma\rangle$
   where $\sigma = \sigma_0[!s_1 = \mathbf{t}, ?s_1 = \mathbf{t}, !s_0 = \mathbf{f}, ?s_0 = \mathbf{f}, x = \mathbf{t}]$
5. $\langle m\|(?s_1\downarrow; \mathbf{ASSN}_{\text{lhs}}), \sigma\rangle \approx_{\text{assnckt}} \langle m\|\mathbf{ASSN}_{\text{rhs}}, \sigma\rangle$
   where $\sigma = \sigma_0[!s_1 = \mathbf{f}, ?s_1 = \mathbf{t}, !s_0 = \mathbf{f}, ?s_0 = \mathbf{f}, x = \mathbf{t}]$

Cases 6.–9. are the symmetrical ones to the above when $!s_0\uparrow$. Note $\mathbf{ASSN}_{\text{lhs}}$ and $\mathbf{ASSN}_{\text{rhs}}$ are the left- and right-hand sides of rule (**6 : ASSN**).

We then have two main computational correspondence Lemmas for each of the $\approx_{\text{ckt}}$ relations.

Lemma 37 If given semantically well-formed configurations $\langle c_0, \sigma_0\rangle$ and $\langle c_0', \sigma_0'\rangle$, for each $\approx_{\text{ckt}}$ relation, if

$$\begin{tikzcd}\langle c_0, \sigma_0\rangle \arrow[r] \arrow[d, phantom, "\wr\wr_{\text{ckt}}"] & \langle c_1, \sigma_1\rangle \\ \langle c_0', \sigma_0'\rangle & \end{tikzcd}$$

then we may find $c_i', \sigma_i', 0 \leq i \leq m$, such that

$$\begin{tikzcd}\langle c_0, \sigma_0\rangle \arrow[r] \arrow[d, phantom, "\wr\wr_{\text{ckt}}"] & \langle c_1, \sigma_1\rangle \arrow[d, phantom, "\wr\wr_{\text{ckt}}"] \\ \langle c_0', \sigma_0'\rangle \arrow[r, "*"] & \langle c_m', \sigma_m'\rangle \end{tikzcd}$$

Proof: The proof proceeds by case analysis on how the initial configurations are related by $\approx_{\text{ckt}}$, as in previous proofs. The one important difference in this proof is there is *prima facie* no reason to expect the environment (in this case m) to obey the handshaking protocols. It must then be argued that for each possible violation of handshake protocols by the environment (*e.g.* from state 2. of $\approx_{\text{assnckt}}$, $!s_1$ is set low), Lemma 36 then shows the computation of m to be ill-formed, violating the assumption of well-formedness. By this line of reasoning, in every case we may assume there are no errors in the handshaking protocols performed by the environment.

The remainder of the proof consists in showing steps of left-hand side modules such as $\mathbf{ASSN}_{\text{lhs}}$ may be mimicked by their corresponding right-hand sides, a task we leave to the reader. ■

LEMMA 38 If given configurations $\langle c_0, \sigma_0\rangle$ and $\langle c'_0, \sigma'_0\rangle$ for each $\approx_{\mathrm{ckt}}$ relation, if

$$\begin{tikzcd}
\langle c_0, \sigma_0\rangle \arrow[d, phantom, "\wr_{\mathrm{ckt}}"] & \\
\langle c'_0, \sigma'_0\rangle \arrow[r] & \langle c'_1, \sigma'_1\rangle
\end{tikzcd}$$

then either

1. $\langle c_0, \sigma_0\rangle \xrightarrow{*}$ **ERROR**, or
2. we may find $c_i, \sigma_i, 0 \leq i \leq n$, such that

$$\begin{tikzcd}
\langle c_0, \sigma_0\rangle \arrow[r, "*"] \arrow[d, phantom, "\wr_{\mathrm{ckt}}"] & \langle c_n, \sigma_n\rangle \arrow[d, phantom, "\wr_{\mathrm{ckt}}"] \\
\langle c'_0, \sigma'_0\rangle \arrow[r] & \langle c'_1, \sigma'_1\rangle
\end{tikzcd}$$

Proof: As in the previous Lemma, it may be assumed that the environment causes no handshake errors, because if so, by Lemma 36 the computation of m is ill-formed, so case 1. may be established here. The rest of the argumentation involves showing all possible executions of the circuit correspond to some execution of the left-hand side specifications. A simple but lengthy case analysis allows these conditions to be established. ■

LEMMA 39 If $\langle c, \sigma\rangle \approx_{\mathrm{ckt}} \langle c', \sigma'\rangle$, then $\sigma(x_{\mathrm{success}}) = \sigma'(x_{\mathrm{success}})$.

LEMMA 40 If $\langle c, \sigma\rangle \approx_{\mathrm{ckt}} \langle c', \sigma'\rangle$, then $\varepsilon(\langle c', \sigma'\rangle)$ implies $\varepsilon(\langle c, \sigma\rangle)$.

Proof: This is proved using Lemma 35 above: if the rhs errors, one of the cases of Lemma 35 must hold, in which case the lhs has the same ill-formed case since all nonlocal variables have same values in related states. ■

We may thus establish that every single-step rewriting preserves meaning.

LEMMA 41 If for closed m, $\Rightarrow_4 m_4 \Rightarrow_5^* m \Rightarrow_5 m'$, then $m \overset{\Rightarrow}{\cong}_{obserr} m'$.

Proof: Direct from the previous four Lemmas, by case analysis on which rule was applied. ■

The correctness of this phase, Lemma 7, may now be established.
Proof: By Lemma 41 and transitivity. ■