Provably Correct Synthesis of Asynchronous Circuits

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

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

Abstract

Recently, powerful methods have been developed to aid in the construction of large asynchronous circuits. Asynchronous circuits are synthesized by starting with a high-level specification and incrementally transforming the specification to produce a circuit. As these methods are informal, or at best semi-formal, the circuits designed using these methods can only be guaranteed correct by applying *post hoc* verification techniques. It is the object of this paper to perform a rigorous justification of the correctness of the transformation rules. With rigorously justified transformations, specifications may be translated into circuits that provably meet their specification.

1. Introduction

Recently there has been increased interest in the design of asynchronous (self-timed) circuits. Such circuits have no clock to synchronize their events. Instead, completion signals are sent when a computation is finished, and circuits wait to receive completion signals on all input lines before proceeding. Until recently, the construction of large self-timed circuits has been avoided because of the complexity involved in their design and testing: there are no clocks or latches to control the flow of signals through the circuit. In the past few years, however, a collection of researchers have developed powerful methods to aid in the construction of large asynchronous circuits [17, 20]. These methods are a significant departure from the traditional design methodologies used in circuit development in that they are automatic or semi-automatic techniques for synthesizing asynchronous circuits from high-level specifications.

The asynchronous design method we focus on here is that of Martin *et al.* [6, 14, 15, 5, 16] Martin begins with a specification for a circuit written in a high-level language modeled after Hoare's CSP (Communicating Sequential Processes) [12], and incrementally applies a series of transformations that preserve meaning and eventually result in a circuit. The specification language describes circuits as a set of concurrently executing processes that can communicate via fixed channels. Each process is constructed from simple programming language-style constructs that include variables and assignments $x := a$, conditional branching, looping, and sequencing. Decomposition or compilation has analogues in the manipulation of mathematical equations. To simplify an equation, various known laws and operations are applied. Some of the laws are local, such as replacing $2+2$ in the midst of some large expression with 4; and other laws are global, such as showing an equation has a *min* or *max* by taking its derivative. For circuit synthesis, most of the transforma-

tions can be viewed as local rewriting operations, replacing one small chunk of the design specification with an equivalent chunk whose representation is incrementally closer to a silicon implementation. The initial CSP-style specification passes through several phases before arriving at an actual circuit. In the first phase, some language constructs are removed. For example, assignment statements are replaced with a single cell to which set and reset operations are communicated. In the second phase, communication commands are replaced with the handshaking protocol that implements the self-timed propagation of signals. Next, the description is expressed in a language called the *production rule description*, which is close to actual circuitry, and is then translated to a gate-and-wire description, *i.e.*, a circuit. The above translations have been implemented as a *circuit compiler* [5]. Surprisingly, the first microprocessor ever constructed using this general methodology has reasonable execution times [16].

In addition to its utility in the construction of asynchronous circuits, we believe this technique of asynchronous circuit design also has much to offer from a correctness standpoint. Most importantly, the absence of a global clock means that the behavior of a large block of circuitry can be understood in terms of the meaning of its parts. The major outstanding problem is that Martin's synthesis technique is not mathematically rigorous. In the conclusion to [16], Martin states,

> Since the method is based on semantics-preserving transformations, the object code generated by the procedure is correct by construction.

However, this statement can only be accepted if we take a less than rigorous interpretation of "semantics" and "correct." Since the purpose of hardware verification is to rigorously demonstrate the correctness of circuit designs, there must be no question that all of the transformation rules preserve meaning. The purpose of this paper is to give a methodology which allows the same statement to be made, only this time rigorously interpreting these words. The resulting method is an important technique for hardware verification because a specification in the high-level language will provably describe the behavior of the resulting circuit. For this initial paper we take an approach that is close to that of Martin and Burns. Since their method has been shown to be useful for actually building circuits, we feel there is no need to justify the usefulness of the general approach and refer readers to papers by Martin in the bibliography for such a justification.

This method may be contrasted with the more traditional methods of hardware verification such as [13, 9] that use theorem-proving systems such as the Boyer-Moore Theorem Prover and HOL [2, 8]. These traditional methods can give a more complete justification of correctness, because logics are inherently more expressive than programming languages and the CSP-style language is a programming language, not a logic. There may be some logical properties the high-level specification should have, for instance that an adder circuit in fact performs addition, but this property cannot be expressed in the language. The disadvantage of traditional methods compared to the approach taken here is that each time a circuit is designed it must be proven correct. Thus, someone involved in the design must be an expert in the use of theorem proving systems. In the transformational approach advocated here, once the transformation rules have been proven to preserve meaning, no more proofs have to be done. This means designers of verifiably correct circuits do not have to be experts in the use of theorem proving systems, a point of great

practical import.

What we present here can be taken as the lower half of a two-pronged method: we can get from a high-level CSP-style description to a circuit. The other half should be a logic in which we can prove properties of the CSP-style specifications. Numerous such logics have been constructed [12, 18, 10, 11, 1], so this is an eminently feasible task but is out of the scope of the present paper. We merely wish to point out that the technique of this paper together with such a logic and automated reasoning tools would provide a complete framework for hardware verification, so there is no inherent limitation of the method.

In order to make precise what has been left informal by Martin and Burns, we build on the theory of process algebra developed in Hoare's CSP, Milner's CCS, and Bergstra's ACP [12, 18, 1]. A process algebra consists of a simple language for expressing concurrent processes, an equivalence relation and associated model, and, usually, an equational logic for reasoning about processes. The basic process algebra language consists of communication actions plus constructs for sequencing, choice, parallel composition, recursion (repetition), renaming, some form of restriction or scoping, and doing nothing. Most notably, these languages seldom contain any sort of assignment or value passing. They are meant to model a very abstract form of concurrent execution in which all interaction occurs through synchronization.

Over the years, several different equivalences and associated models have been developed for studying process equivalence: bisimulation [18], traces [12], failures [3], testing [19], and acceptance trees [11], just to mention a few. Our circuit equivalence is built on *testing*. The *testing equivalence* distinguishes processes by their possibility of passing a test. It is a formalization of the idea of exhaustive testing, where both processes are tested with every possible test. The two processes are considered equivalent if their behavior is identical for all tests. Testing is particularly appropriate for circuit design, because it is an abstraction of the actual method whereby circuits are tested on a suite of data. An important feature of the various equivalence relations, including testing, is that they are used to obtain congruence relations. If an equality is a congruence it is possible to substitute equals for equals in any part of an equation, and thus justify local transformations on circuit specifications. Since most of the transformations we define are local, congruence is a critical component of the model we use.

The final component of the work in process algebras is the development of proof techniques for showing the equivalence between processes. Both modal logics and equational logics of processes have been developed. Equational logics may be used as a rewrite system, using the rules to transform processes into other syntactic forms which are semantically equivalent.

1.1. Proving Synthesis Correct

Before giving an overview of our methodology, we list the assumptions we make about the underlying hardware model. The assumptions are standard.

1. We assume the gate-and-wire-delay model of circuit behavior [4], a generalization of the gate-delay model of Muller.

2. To simplify the mathematics, only one value will be allowed per wire at a given instant in time.

3. A gate may ignore a spike on the input line.

4. All synchronization between gate level circuit components uses a 4-phase *handshaking protocol* [14].

We begin by defining the process algebra C-CSP (Circuit CSP) in Section 2. It shares many features with Martin's language. The main differences are that complete semantics are given to the language, dispelling any ambiguity of meaning; and that the high- and low-level descriptions of circuits, plus all the intermediate states involved in the translation process, may be specified in a single language. This means that we can do all the reasoning in a single language, simplifying the proofs. In Section 3, we give the language meaning via an operational semantics. Defining the semantics presents several challenges. The first problem is the addition of side effects to a process algebra (process algebras do not have assignment). Side effects are necessary because state-holding elements are a fundamental structure of digital circuits. Second, the semantics must formalize the mutual exclusion principles presented informally in Martin's work. The low-level handshaking synchronizations used in asynchronous circuits have no ability to arbitrate between two simultaneous requests for the same port. It is thus necessary to impose mutual exclusion conditions on synchronizations in the specification to guarantee no simultaneous requests will ever arise in the implementation it is translated to. Keeping track of necessary mutual exclusion throughout the translation process is difficult.

We define circuit equivalence in Section 4 based on testing equivalence to formalize the idea of *semantics-preserving transformations.*

In Section 5, we formalize decomposition as a 5-phase rewrite system for decomposing a high-level specification into an asynchronous implementation. Rewrite rules of the form $e \rhd_i e'$ indicate that for phase i, if an instance of e appears in a specification, it may be replaced by an instance of e' and the meaning will not change. We show that meaning will not change by proving $e = e'$. Each phase of the rewrite system performs a particular type of translation. A problem arises in the course of handshaking expansion (the implementation of high-level communication by a handshaking protocol): process equivalence is not preserved because different communication protocols are used before and after the expansion. This necessitates a different justification of correctness for handshaking expansion. The correctness of the translation is summarized in a theorem that shows to an external observer the high-level specification will behave identically to the digital circuit.

Section 6 applies our method to a simple example.

2. The Circuit Language — C-CSP

In this section we introduce C-CSP (Circuit-CSP), a variation of the CSP language [12] based on the version of CSP designed by Martin and Burns [5] for specifying asynchronous circuits. We remove some syntactic sugar and add constructs needed to guarantee correctness. The language is defined by the following grammar

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

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

with d **do** c **end**

d ::= **r** x d | **w** x d | $P!$ d | $P?$ d | ϵ

We will use e to denote boolean expressions, c to denote commands (also referred to as terms or processes), and d to denote declaration lists. $\mathcal{V}$ is the set of C-CSP variables; $x, y, z \ldots \in \mathcal{V}$ denote variables, and $!x, !y, !z, ?y, ?y, ?z \ldots \in \mathcal{V}$ denote *handshaking* variables, variables which take on special significance in the implementation of handshaking protocols. $\mathcal{P}$ is the set of all port names; $P?, C?, D? \ldots \in \mathcal{P}$ denote passive ports, and $P!, C!, D! \ldots \in \mathcal{P}$ are the corresponding active ports.

The boolean expressions e are the usual ones plus the probe $\bar{P}?$ [14], by which a passive port may unobtrusively check 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, $x := e$, immediately assigns the value of e to the boolean variable x. As a shorthand we represent $x :=$ **true** and $x :=$ **false** by $x \uparrow$ and $x \downarrow$. Delayed assignment, $x :=_d e$, immediately evaluates e and eventually assigns its value to x. It is used to model the actual delayed behavior of logic gates, and is not used in specifications. As a shorthand we represent $x :=_d$ **true** and $x :=_d$ **false** by $x \nearrow$ and $x \searrow$. Sequential composition is denoted by ";". Choice is written using guarded commands and repetition of c is designated by $*[c]$. Parallel composition is represented by $\|$. As in CSP, processes synchronize with one another through ports $P!$ and $P?$. Examples of C-CSP expressions may be found in section 6.

All variables are declared (scoped) twice: once for the scope in which they may be read (**r**), and once where they may be both read and written (**w**). These dual declarations are necessary for proving the transformation correct. Similarly, corresponding active and passive halves of a channel are declared separately. Outside of the scope of $P?$, for instance, $P?$ may not be used. To simplify the presentation, it is not possible to re-declare variables or ports. Scoping restrictions are made explicit by the following.

Definition 1 *A C-CSP term c is* syntactically well-formed *if and only if it can be generated by the above grammar and*

- *each send or receive port is declared at most once;*
- *each variable is declared at most once as read-write and at most once read-only; and*
- *within the read scope of a variable, no assignments to that variable may occur.*

Hereafter C-CSP terms are taken to be the syntactically well-formed terms. A C-CSP term is *closed* if and only if all ports used have both active and passive halves declared, and all variables assigned in the term are declared (variables only read need not be declared). The class of C-CSP terms that may be compiled are called *modules.* A *module m* is a (possibly non-closed) C-CSP term such that if $x := e$ or $x :=_d e$ occurs in m then **w** x is declared, and all ports $P!$ and $P?$ occurring in m are declared. Note it is possible (indeed very useful) to only declare only the active or passive half of a port in a module. This means the other half is external, and is how the module may interact with its environment. Modules here are useful in the same sense they are useful in programming languages: if a large specification is divided up into modules, each part may be compiled separately.

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

Some of the constructs of C-CSP are in the language only because they are used either during the translation process or in the final description of circuits. This arises because we wish to keep the entire translation process in one language; this is a very large gap for one language to span. We need to isolate the sublanguage of C-CSP that is the "pure" specification part of the language; terms in this language all may be compiled to circuits.

Definition 2 *S-CSP (Specification CSP) terms are C-CSP terms without instances of delaying assignment commands* $x :=_d e$ *or handshake variables* $!x$, $?x$.

The sublanguage H-CSP representings gate-level asynchronous circuits. Simple gates are represented by simple processes, and wires are represented by variables. The isochronic forks assumption of Martin and Burns is reflected in the fact that once a wire (*i.e.* a variable) is assigned a new value, all other gates see this new value instantaneously. This assumption is not always valid; for a discussion, see [5, Ch.4].

Definition 3 *H-CSP (Hardware CSP) modules m are C-CSP modules (ignoring declarations) of the form* $m = c_1 \| c_2 \ldots \| c_n$, *where the* c_i *are individual* gate processes *of one of the following forms.*

$*[z := x \wedge y]$ *for an and-gate,*

$*[z := x \vee y]$ *for an or-gate,*

$*[z := \neg x]$ *for a not-gate,*

$*[[x \wedge y \longrightarrow z \uparrow [] \neg x \wedge \neg y \longrightarrow z \downarrow]]$ *(abbreviated* $*[z := x \ \mathbf{C} \ y]$*) for a Muller C-element,*

and furthermore, all variables are assigned at most once.

H-CSP is an appropriate asynchronous hardware description language as can be seen in the representation of an and-gate. In a gate and wire delay model of circuits, a gate may take arbitrarily long to respond to a change in inputs. This is captured by the process $*[z := x \wedge y]$: there may be arbitrary delays between executions of the assignment as other processes execute.

3. Operational Semantics of C-CSP

An operational semantics describes the execution 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.

A *state* σ is a finite mapping from $\mathcal{V} \cup \mathcal{P}$ to $\mathbf{Bool} \times \mathbf{Bool}$. $\sigma(x)$ is thus a pair; the first value of the pair (abbreviated $\sigma_1(x)$) represents the current value of the port or variable. The second value $\sigma_2(x)$ represents a *delayed* value. In the high-level description

of circuits, both values will be identical. The use of delayed boolean values is necessary in order to correctly model the low level behavior of circuits. Augmenting or changing the state function σ is abbreviated $\sigma[x = (b, b')]$, where $b, b' \in$ Bool.

Definition 4 *The initial state of the computation and the configurations the computation passes through are formally defined as follows.*

- *initval* :C-CSP $\rightarrow$ States *maps terms c to initial states σ such that the domain of σ is all variables x and ports occurring in c, and $\sigma(x) = \{(\mathbf{false}, \mathbf{false})\})$ for all x in the domain of σ.*

- *A* configuration $\gamma = \langle c, \sigma \rangle$ *consists of a process c and a state σ that represents a point in the computation. There is additionally a special configuration,* **ERROR***, indicating that the computation has violated a mutual exclusion constraint.*

One important notational convenience is the notion of a term with a "hole" poked in it where another term may be placed; these objects are *contexts*. A certain class of contexts, *reduction contexts*, are very useful in the computation process.

- A *context* C is a term containing n holes "$-_i$", $1 \leq i \leq n$. $C[\![c_1]\!] \ldots [\![c_n]\!]$ is the result of syntactically replacing all occurrences of $-_i$ in C with c_i for each $1 \leq i \leq n$.

- A *closing context* for a process c is a context C where $C[\![c]\!]$ is closed.

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

  $$R \text{ is } -_i \text{ or } R;c \text{ or } R \| c \text{ or } c \| R \text{ or } \mathbf{with}\ d\ \mathbf{do}\ R\ \mathbf{end}.$$

  and each $-_i$ occurs at most once in R.

Most often single-hole contexts are used, in which case the subscript on the hole may be removed. We first define evaluation of boolean expressions, and then define the operational semantics for closed C-CSP terms.

3.1. Semantics of Expressions

All boolean expressions are evaluated with respect to a state σ by homomorphically extending the domain of σ_1 to all boolean expressions.

$$\begin{array}{rcll}
\sigma_1(\mathbf{true}) & = & \mathbf{true} & \\
\sigma_1(\mathbf{false}) & = & \mathbf{false} & \\
\sigma_1(x) & = & b & \text{where } \sigma = \sigma'[x = (b, b')] \\
\sigma_1(\bar{P}?) & = & \sigma_1(P!) & \\
\sigma_1(e_1 \wedge e_2) & = & \sigma_1(e_1) \wedge \sigma_1(e_2) & \\
\sigma_1(e_1 \vee e_2) & = & \sigma_1(e_1) \vee \sigma_1(e_2) & \\
\sigma_1(\neg e) & = & \neg \sigma_1(e) &
\end{array}$$

3.2. Semantics of Commands

The semantics of commands is defined by the 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 state σ has been augmented with the value of e assigned to x, $\langle R[\![\mathbf{skip}]\!], \sigma[x = \sigma(e)]\rangle$.

In order to enforce mutual exclusion for specifications, the configurations in which two concurrently executing processes simultaneously attempt to assign to the same variable, or synchronize on the same port leads to an **ERROR** configuration. In addition, the semantics of the conditional command force the guards to be mutually exclusive. If they are not, an **ERROR** configuration is reached.

Following Martin *et al.*, the asynchronous circuits we construct use a handshaking protocol to synchronize; see the references for a description of the protocol.

To ensure that the handshaking protocol is never violated, and thus to allow translations to be verified, there are configurations for handshaking variable assignments in which executing a delayed assignment leads to an **ERROR** configuration. These rules prevent spurious changes from being made to a handshaking variable during execution of the protocol. For example, if $!x$ goes high before $?x$ goes low, an **ERROR** state is reached.

Definition 5 *The one-step computation relation on configurations, $\rightarrow$, is*

Assignment

(1) $$\langle R[\![x := e]\!], \sigma\rangle \rightarrow \langle R[\![\mathbf{skip}]\!], \sigma[x = (\sigma_1(e), \sigma_1(e))]\rangle$$

Delayed Assignment

(1) $$\langle R[\![x :=_d e]\!], \sigma\rangle \rightarrow \langle R[\![\mathbf{skip}]\!], \sigma[\sigma_2(x) = \sigma_1(e)]\rangle$$

(2) $$\langle c, \sigma[x = (\mathbf{true}, \mathbf{false})]\rangle \rightarrow \langle c, \sigma[x = (\mathbf{false}, \mathbf{false})]\rangle$$

(3) $$\langle c, \sigma[x = (\mathbf{false}, \mathbf{true})]\rangle \rightarrow \langle c, \sigma[x = (\mathbf{true}, \mathbf{true})]\rangle$$

(4) $$\frac{(\$ =! \wedge \sigma_1(e) = \neg\sigma_1(!x) \wedge \sigma_1(e) = \sigma_1(?x)) \vee (\$ =? \wedge \sigma_1(e) = \neg\sigma_1(!x) \wedge \sigma_1(!x) = \sigma_1(?x))}{\langle R[\![\$x :=_d e]\!], \sigma\rangle \rightarrow \mathbf{ERROR}}$$

Sequencing

(1) $$\langle R[\![\mathbf{skip}; c]\!], \sigma\rangle \rightarrow \langle R[\![c]\!], \sigma\rangle$$

Selection

(1) $$\frac{\sigma_1(e_i) = \mathbf{true} \ \wedge \ \forall j \neq i . \sigma_1(e_j) = \mathbf{false}}{\langle R[\![[e_1 \longrightarrow c_1] \ldots [e_i \longrightarrow c_i] \ldots [e_n \longrightarrow c_n]]\!], \sigma\rangle \rightarrow \langle R[\![c_i]\!], \sigma\rangle}$$

(2) $$\frac{\sigma_1(e_i) = \mathbf{true} \ \wedge \ \sigma_1(e_j) = \mathbf{true} \ \wedge j \neq i}{\langle R[\![[e_1 \longrightarrow c_1] \ldots [e_i \longrightarrow c_i] \ldots [e_n \longrightarrow c_n]]\!], \sigma\rangle \rightarrow \mathbf{ERROR}}$$

Repetition

(1) $$\langle R[\![*[c]]\!], \sigma\rangle \rightarrow \langle R[\![c; *[c]]\!], \sigma\rangle$$

Parallelism

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

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

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

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

(5) $$\langle R[\![x := e_1]\!][\![x := e_2]\!], \sigma\rangle \rightarrow \mathbf{ERROR}$$

(6) $$\langle R[\![P\$]\!][\![P\$]\!], \sigma\rangle \rightarrow \mathbf{ERROR} \qquad \textit{where \$ is ! or ?}$$

4. Circuit Equivalence

The circuit equivalence we define is a variation on the *testing equivalence* of [19, 11]. It is a precise formalization of exhaustive testing, so if two processes are testing-equivalent, no difference will be ever be able to be ascertained between the two by a tester. We modify the classic notion of test so as to account for the imperative nature of the circuit language. We define *testing contexts*, the circuit equivalence counterparts to classic *tests*; fair computations; successful computations; equivalent observations; and testing equivalence.

We add a *success variable*, x_{success} to the existing C-CSP variables $\mathcal{V}$; call this extended language C-CSP*, the language of testers. The testing process sets x_{success} to **true** to indicate a test is successful. A *testing context* is a C-CSP* context.

Definition 6 *Let c_0 be a closed C-CSP* process. A computation of c_0 is of the form*

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

where $\sigma_0 = initval(c_0)$.
It is successful if for some i, $\sigma_i(x_{\text{success}}) = (\mathbf{true}, \mathbf{true})$.

Definition 7 *The* fair computations *of c are the computations of c in which no command is enabled infinitely often without being executed.*

Hereafter, "computations" are taken to implicitly mean fair computations.

To define observation equivalence for a nondeterministic language, observe testing computations do not always succeed or always fail; they may do both. In traditionaldefinitions of testing, predicates *c may* and *c must* mean c sometimes or always succeeds.

Definition 8 (Observation Equivalence) *Let c and c' be closed C-CSP* terms. Define $c =_{obs} c'$ iff*

- *there exists a successful computation of c iff there exists a successful computation of c'.*
- *all (fair) computations of c are successful iff all (fair) computations of c' are successful.*

Definition 9 (Testing Equivalence) *Let c, c' be C-CSP processes, and $\mathcal{T}$ a set of testing contexts. $c =^{\mathcal{T}} c'$ if for all closing testing contexts C in $\mathcal{T}$, $C[\![c]\!] =_{obs} C[\![c']\!]$.*

When $\mathcal{T}$ does not appear it may implicitly be taken to be the set of all testing contexts.

Theorem 1 (Congruence) *For any context C, $c{=}c'$ implies $C[\![c]\!]{=}C[\![c']\!]$.*

Proof: Suppose $c{=}c'$; show $C[\![c]\!]{=}C[\![c']\!]$, i.e. show for any closing context C' that $C'[\![C[\![c]\!]]\!]$ and $C'[\![C[\![c']\!]]\!]$ have the same may and must behavior. This is trivial by assumption, since $C'[\![C]\!]$ is a particular context our assumption holds for.

This congruence principle is critical for showing the correctness of the decomposition process: it allows us to substitute equals for equals, thus justifying the use of rewriting rules.

Corollary 1 (Substitution Principle) *If $c{=}c'$, then $C[\![c]\!]{=}C[\![c']\!]$.*

5. Compilation of C-CSP Specifications to Circuits

We now define a system for incrementally translating C-CSP process specifications to circuit implementations. Our translation follows that of [5], with changes necessary to make the translation provably correct.

We divide the translation process into five phases, and implement each phase using a different term rewriting system (see [7] for background and references on rewriting). The rewrite systems then may be used to compile a specification module as follows. Given a specification module m_0, the five rewrite systems are applied in turn, translating m_0 to m_1, to m_2, to m_3, to m_4, and finally to a circuit module m_5. The structure of these systems will first be outlined, and the individual rules will appear in the sections following.

Definition 10 *A rewrite system R over the C-CSP language consists of a finite set of rules of the form $e_0 \rhd_R e_1$, where the e_i are C-CSP metametavariables, i.e. they are terms which may themselves contain metavariables.*

For instance, if $c;\mathbf{skip} \rhd_R c$ is a rule in some rewrite system R, this rule allows any subexpression of the form $c;\mathbf{skip}$ to be replaced with c, for arbitrary terms c. We now define the application of rules to modules.

Definition 11 *Given a rewrite system R, the 2-place relations $\Rightarrow_R$, $\Rightarrow_R^*$, and $\Rightarrow_R^N$ on modules are defined as follows.*

1. *$m_0 \Rightarrow_R m_1$ (m_0 rewrites in one step to m_1) iff $m_0 = C'[\![c_0]\!]$, $m_1 = C'[\![c_1]\!]$, $e_0 \rhd_R e_1$, and the c_i are derived from the e_i by uniformly replacing metavariables by C-CSP terms.*

2. *$\Rightarrow_R^*$ is the transitive reflexive closure of $\Rightarrow_R$.*

3. *$m_0 \Rightarrow_R^N m_1$ (Normalizing rewrite) iff $m_0 \Rightarrow_R^* m_1$, and there is no m_2 such that $m_1 \Rightarrow_R m_2$.*

We proceed below to define five rewrite systems 1, 2, 3, 4, 5, by giving five sets of rules $\rhd_1 \ldots \rhd_5$. A specification m_0 is compiled to a circuit m_5 by the rewriting

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

For this we use the abbreviation $m_0 \Rightarrow_{1-5} m_5$ (m_0 compiles to m_5). There will be a need below to refer to modules that are the result of translating an initial specification through i levels, defined as follows.

Definition 12 *$\Rightarrow_i m$, for $i \in \{1, \ldots, 4\}$, is true iff $m_0 \Rightarrow_1^N m_1 \ldots \Rightarrow_i^N m$ and $m_0 \in$ S-CSP. $\Rightarrow_0 m$ holds if $m \in$ S-CSP.*

In each of the rewrite systems presented below, we have implicit rules that allow declarations to be freely moved about terms and allows the $\|$ and ; operators to associate freely, i.e. we rewrite over these algebraic laws.

Three important facts to establish about each rewrite system are strong normalization, so no rewriting may proceed infinitely; totality, so all specifications compile to realizable circuits; and semantics preservation, the rewritten term has the same behavior as the original. These properties will be shown to hold of our rewriting systems in Section 5.6.

Before defining the five systems, we briefly summarize the meaning of the five phases of translation.

- Phase 1 is a syntax-directed translation that simplifies uses of each of the constructs of the original language.

- Phase 2 further simplifies uses of the original constructs, though not in a syntax-directed manner.

- Phase 3 expands the high-level communication of C-CSP into an explicit handshaking protocol. In order to guarantee correctness all handshaking expansions must be performed simultaneously, so this translation must be isolated from the others in its own rewrite system.

- Phase 4 modularizes the result of handshaking expansion. This phase divides the evolving specification into small modules which may then be compiled independently. Again, these operations must be isolated from the others to guarantee correctness.

- Phase 5 translates each of the small modules that remain into actual circuitry.

5.1. Phase 1: Syntax-Directed Rewriting

Syntax-directed rewriting begins with a S-CSP term and produces a S-CSP module of the form $c_1 \| \ldots \| c_n$, where each c_i is in only of the following forms.

1. sequencing process: $*[[\bar{C}? \longrightarrow C_1!; C_2!; \ldots; C_n!; C?]]$

2. $*[[\bar{C}? \longrightarrow \mathbf{skip}; C?]]$

3. assignment: $*[[\bar{C}_1? \longrightarrow x\uparrow; C_1?[]\bar{C}_0? \longrightarrow x\downarrow; C_0?]]$

4. selection process: $*[[\bar{C}? \longrightarrow [e_0 \longrightarrow C_1![] \ldots []e_n \longrightarrow C_n!]]]$

5. communication process: $*[[\bar{C}? \longrightarrow A?; C?]]$

Initially, a "start port" is added to each global process, along with an active process that synchronizes with this port to initiate the process. Details of this are ignored for brevity. This phase creates "assignment cells" to help physically isolate storage locations and simplifies sequential composition to parallel composition. Assignment (immediate) is transformed by creating a separate assigning process for each variable and replacing all assignment statements by synchronizations with this new process. The assigning process consists of two subprocesses: one for assigning **true**($\uparrow$), and the other for assigning **false**($\downarrow$). There are three rewriting rules to accomplish this: the first creates the assignment process, the second synchronizes all assignments to the variable with the process, and the third moves the scoping around the relevant parts of the program. In the second rule, we assume that c is in the scope of the ports D_0, D_1 and that it contains a single occurrence of a term $x := e$. In the third rule, we assume that c_2 contains no occurrences

of the variable x.

(ASSIGN 1) $\quad$ **with w** x **do** c **end** $\triangleright_1$ $*[$**with** $D_1?, D_0?,$ **w** x **do**
$[\bar{D}_1? \longrightarrow x\uparrow; D_1?[]\bar{D}_0? \longrightarrow x\downarrow; D_0?]$
end$]\|$
with $D_1!, D_0!$ **do** c **end**

(ASSIGN 2) $\quad *[[\bar{D}_1? \longrightarrow x\uparrow; D_1?[]\bar{D}_0? \longrightarrow x\downarrow; D_0?]]\|c \triangleright_1$
$*[[\bar{D}_1? \longrightarrow x\uparrow; D_1?[]\bar{D}_0? \longrightarrow x\downarrow; D_0?]]\|$
$c[[e \longrightarrow D_1![]\neg e \longrightarrow D_0!]/x := e]$

(ASSIGN 3) $\quad$ **with w** x **do** $c_1\|c_2$ **end** $\triangleright_1$ **with w** x **do** c_1 **end**$\|c_2$

Rewriting the sequencing operator ";" introduces a separate process for each term in the sequencing construct. It is defined by the following two rules, where c is not an active communication $D!$ for any D.

(SEQ 1) $\quad c_1; c_2 \quad \triangleright_1$ **with** $D!, D?$
$(*[[\bar{D}? \longrightarrow c_1; D?]]\|D!; c_2)$
end

(SEQ 2) $\quad C!; c \quad \triangleright_1$ **with** $D!, D?$
$(*[[\bar{D}? \longrightarrow c; D?]]|C!; D!)$
end

We present here an outline of the correctness proof of the rule **(ASSIGN 1)** as an example of how individual rules are shown to preserve meaning.

Lemma 1 *Let* $cell = *[[\bar{D}_1? \longrightarrow x\uparrow; D_1?[]\bar{D}_0? \longrightarrow x\downarrow; D_0?]]$, *and* c *be a nonterminating process containing no declaration of* x. *Then*

with w x **do** c **end**=**with w** $x, D_0!, D_0?, D_1!, D_1?$ **do** $cell\|c$ **end**

Proof: To prove this lemma we must show that for all closing contexts C,

$$C[\![\textbf{with w } x \textbf{ do } c \textbf{ end}]\!] =_{obs} C[\![\textbf{with w } x, D_0!, D_0?, D_1!, D_1? \textbf{ do } cell\|c \textbf{ end}]\!].$$

For the sake of brevity we only show one is successful if the other is; the rest of the proof is similar. To ease notation, let

c_1 = **with w** x **do** c **end**
c_2 = **with w** $x, D_0!, D_0?, D_1!, D_1?$ **do** $cell\|c$ **end**

Choose an arbitrary testing context C such that $C[\![c_1]\!]$ has a successful computation. Thus there is some computation of this closed term that is successful in a finite number of steps. Since $D_0!, D_0?, D_1!, D_1?$ are new ports, c contains no occurrences of any of these ports. Therefore any step $C[\![c_1]\!]$ can do $C[\![c_2]\!]$ can also do, so $C[\![c_2]\!]$ contains a successful computation. The opposite direction is similar.

5.2. Phase 2: Non-Syntax-Directed Rewriting

Non-syntax-directed rewriting introduces further parallelism into the module. The first rule turns a process consisting of a sequence of active port commands into the parallel composition of these commands. The other rules simplify the structure of boolean guards and implement choice by parallel composition.

The first rule turns a sequence of active port commands into a collection of parallel compositions of processes of the form $*[[\bar{D}_j? \longrightarrow C_j!; D_{j+1}!; D_j?]]$. That is, each process receives a signal to to proceed on $D_j?$, signals the process guarded by $C_j!$, signals the next process in the sequential composition by $D_{j+1}!$, and then acknowledges the signal on $D_j?$.

$$(\mathbf{SEQ-PAR}) \quad *[[\bar{C}? \longrightarrow C_1!; C_2!; \ldots; C_n!; C?]] \triangleright_2 \quad \begin{array}{l} \textbf{with } D!, D? \textbf{ do} \\ \quad (*[[\bar{D}? \longrightarrow C_2!; \ldots; C_n!; D?]] \| \\ \quad *[[\bar{C}? \longrightarrow C_1!; D!; C?]]) \\ \textbf{end} \end{array}$$

Subprocesses consisting of repetition and selection are decomposed so that the evaluation of guards is separated from the evaluation of the command they are guarding. Since guards are guaranteed to be mutually exclusive, each one can be evaluated by a separate process.

$$(\mathbf{GUARD1}) \quad *[[\bar{D}? \longrightarrow \quad [e_1 \longrightarrow S_1!; D?[] \ldots []e_n \longrightarrow S_n!; D?]] \triangleright_2 \\ *[[e_1 \wedge \bar{D}? \longrightarrow S_1!; D?| \ldots |e_n \wedge \bar{D}? \longrightarrow S_n!; D?]]$$

This rewriting would not be legal with nonexclusive guards since it would allow two guards to simultaneously evaluate to true.

We apply de Morgan's law to complex guards of the form $g = \bar{P}? \wedge e$ obtaining g as the disjunctive normal form, $g = \bigvee_{j=0}^{m-1} f_j$ where each f_j is a simple conjunct containing $\bar{P}?$.

$$(\mathbf{GUARD2}) \quad *[[\bar{P}? \wedge e \longrightarrow Q!]] \triangleright_2 *[[f_0 \longrightarrow Q![] \ldots []f_{m-1} \longrightarrow Q!]]$$

Finally, we can combine all the simple conjuncts guarding the same command [5]:

$$(\mathbf{GUARD3}) \qquad *[[f_0 \longrightarrow Q![] \ldots []f_{m-1} \longrightarrow Q!]] \triangleright_2 \\ *[[\bar{P}? \wedge x_0 \wedge \ldots \wedge x_{k-1} \wedge \neg x_k \wedge \ldots \wedge \neg x_{n-1} \longrightarrow Q!]]$$

5.3. Phase 3: Handshaking Expansion

Handshaking expansion replaces the C-CSP synchronization constructs with boolean variables implementing the four-phase handshaking protocol. Since the active and passive ports need not be declared in the same scope, we must introduce three rules to carry out this rewriting. Each rule eliminates a channel scope construct by simultaneously substituting the variable assignments and tests that implement the handshaking protocol for all of the occurrences of ports. Assume in each rule that C contains no occurrences of

$P!$ or $P?$ or $\bar{P}?$.

(HS1) **(with** $P!, P?$ **do** $C[\![\bar{P}?]\!][\![P!]\!][\![P?]\!]$ **end)** $\triangleright_3$
with w $!p$, **w** $?p$ **do**
$C[\![!p]\!]$
$[\![!p\nearrow; [?p \longrightarrow \textbf{skip}]; !p\searrow; [\neg ?p \longrightarrow \textbf{skip}]]\!]$
$[\![[!p \longrightarrow \textbf{skip}]; ?p\nearrow; [\neg !p \longrightarrow ?p\searrow]]\!]$
end

(HS2) **(with** $P!$ **do** $C[\![P!]\!]$ **end)** $\triangleright_3$
with w $!p$, **r** $?p$ **do**
$C[\![!p\nearrow; [?p \longrightarrow \textbf{skip}]; !p\searrow; [\neg ?p \longrightarrow \textbf{skip}]]\!]$
end

(HS3) **(with** $P?$ **do** $C[\![\bar{P}?]\!][\![P?]\!]$ **end)** $\triangleright_3$
with r $!p$, **w** $?p$ **do**
$C[\![!p]\!]$
$[\![[!p \longrightarrow \textbf{skip}]; ?p\nearrow; [\neg !p \longrightarrow ?p\searrow]]\!]$
end

Simple testing equivalence $=$ cannot be used to prove that the handshaking expansion preserves equivalence, because it in fact does not preserve equivalence. See Section 5.6 for a description of this problem and its solution.

5.4. Phase 4: Modularization

Next the module being compiled is broken up into a collection of concurrently executing submodules, $m = m_1 \| m_2 \| \ldots \| m_n$. The different component processes that remain after handshaking expansion are all modules except communication processes, which may not be modular if there is more than one use of an active or passive port in the original. These components must be modularized at this point. Each $i-th$ instance of a handshaking variable $!p$ in a term is given a new variable $!p_i$, and the results are merged by disjunction. These rules are phrased to follow directly after handshaking, so for instance an application of **(HS1)** is always followed by an application of **(MOD1)**.

(MOD2) **with w** $!p$, **r** $?p$ **do**
$C[\![!p\nearrow; [?p \longrightarrow \textbf{skip}]; !p\searrow; [\neg ?p \longrightarrow \textbf{skip}]]\!]$
$\vdots$
$[\![!p\nearrow; [?p \longrightarrow \textbf{skip}]; !p\searrow; [\neg ?p \longrightarrow \textbf{skip}]]\!]$
end
$\triangleright_4$
with r $!p_1 \ldots !p_n$, **w** $!p$ **do** $*$ $[!p := !p_1 \vee \ldots \vee !p_n]$ **end** $\|$
with r $?p$ **do**
$C[\![$**with w** $!p_1$ **do** $!p_1\nearrow; [?p \longrightarrow \textbf{skip}]; !p_1\searrow; [\neg ?p \longrightarrow \textbf{skip}]$ **end**$]\!]$
$\vdots$
$[\![$**with w** $!p_n$ **do** $!p_n\nearrow; [?p \longrightarrow \textbf{skip}]; !p_n\searrow; [\neg ?p \longrightarrow \textbf{skip}]$ **end**$]\!]$
end

where $n > 1$ (if $n = 1$ processes are already modular), each hole $-_i$, $1 \leq i \leq n$ in C occurs at most once, and $!p$ is not assigned in C. **(MOD1)** and **(MOD3)** are similarly derived from the corresponding handshaking rules, but are not included for brevity. There is also are seperate rules not given here for the case $n = 1$, where the merge process is removed and $!p_1$ can be $!p$.

5.5. Phase 5: Final Compilation into circuits

The last part of the translation takes a module made up of submodule processes (the result of modularization), and transforms each submodule into an equivalent circuit representation. Rules are defined for making circuits for primitive assignment, sequencing, guards, and active and passive communication. before giving the rules we mention two valid principles used repeatedly below: actions may be placed in the middle of a handshaking protocol without altering meaning in certain cases, and new local state-holding cells may be added.

5.5.1. Assignment

Assignment cells after handshaking expansion look like*

$$
\begin{array}{l}
m_{assn} = \ \textbf{with w } x \textbf{ do} \\
\quad *[\textbf{with w } ?d_1, \textbf{r } !d_1 \textbf{ do } [!d_1 \longrightarrow \textbf{skip}]; x\uparrow; ?d_1\nearrow; [\neg !d_1 \longrightarrow \textbf{skip}]; ?d_1\searrow \textbf{ end}[] \\
\quad\ \ \textbf{with w } ?d_0, \textbf{r } !d_0 \textbf{ do } [!d_0 \longrightarrow \textbf{skip}]; x\downarrow; ?d_0\nearrow; [\neg !d_0 \longrightarrow \textbf{skip}]; ?d_0\searrow \textbf{ end}] \\
\quad \textbf{end}
\end{array}
$$

The first transformation we may make is

$$
\begin{array}{lll}
\textbf{(ASSN1)} & m_{assn} \rhd_5 & m_{assn1}, \text{ where} \\
 & m_{assn1} = & \textbf{with w } x \textbf{ do} \\
 & & \quad *[\textbf{with w } ?d_1, \textbf{r } !d_1 \textbf{ do} \\
 & & \quad\quad [!d_1 \longrightarrow \textbf{skip}]; x\nearrow; [x \longrightarrow \textbf{skip}]; ?d_1\nearrow; \\
 & & \quad\quad [\neg !d_1 \longrightarrow \textbf{skip}]; ?d_1\searrow \\
 & & \quad \textbf{end}[] \\
 & & \quad \textbf{with w } ?d_0, \textbf{r } !d_0 \textbf{ do} \\
 & & \quad\quad [!d_0 \longrightarrow \textbf{skip}]; x\searrow; [\neg x \longrightarrow \textbf{skip}]; ?d_0\nearrow; \\
 & & \quad\quad [\neg !d_0 \longrightarrow \textbf{skip}]; ?d_0\searrow \\
 & & \quad \textbf{end}] \\
 & & \textbf{end}
\end{array}
$$

This rewrite rule is valid because the two are provably equal as long as no other assignment to x intervenes, and this must be the case here because x is defined local to this process, and the guards $!d_1$ and $!d_0$ are, by well-formedness requirements, mutually exclusive.

The next translation is to divide each action into an individual process, and in a manner that may be implemented by a circuit. We then arrive at

*This assumes there were multiple assignments to the variable x in the module, and the **(MOD3)** rule caused the declarations to move in. If there was only one assignment to x the **(MOD3)** rule would not have applied, so there would be only one outermost declaration. Since the two are all but identical we assume them identical.

$$(\textbf{ASSN2}) \quad m_{assn1} \triangleright_5 \; m_{assn2}, \text{ where}$$
$$m_{assn2} = \textbf{with w } x, \textbf{ r } !d_1, \textbf{r } !d_0, \textbf{w } ?d_1, \textbf{w } ?d_0 \textbf{ do}$$
$$*[[!d_1 \longrightarrow x\nearrow]] \| *[[!d_1 \wedge x \longrightarrow ?d_1\nearrow]] \|$$
$$*[[\neg !d_1 \longrightarrow ?d_1\searrow]] \| *[[!d_0 \longrightarrow x\searrow]] \|$$
$$*[[!d_0 \wedge \neg x \longrightarrow ?d_0\nearrow]] \| *[[\neg !d_0 \longrightarrow ?d_0\searrow]]$$
$$\textbf{end}$$

m_{assn2} then rewrites to the circuit below.

$$(\textbf{ASSN3}) \quad m_{assn2} \triangleright_5 \; m_{assn3}, \text{ where}$$
$$m_{assn3} = \textbf{with w } x, \textbf{ r } !d_1, \textbf{r } !d_0, \textbf{w } ?d_1, \textbf{w } ?d_0 \textbf{ do}$$
$$*[x :=!d_1 \; \mathbf{C} \; \neg !d_0] \| *[?d_1 :=!d_1 \wedge x] \| *[?d_0 :=!d_0 \wedge \neg x]$$
$$\textbf{end}$$

For the remaining basic constructs we dispense with the intermediate phases for brevity.

5.5.2. Sequencing

Sequencing processes after handshaking expansion look like[†]

$$m_{seq} = \textbf{with } \textbf{r } !d, \textbf{r } ?t, \textbf{r } ?s, \textbf{w } ?d, \textbf{w } !t, \textbf{w } !s \textbf{ do}$$
$$*[[!d \longrightarrow \textbf{skip}]; !s\nearrow; [?s \longrightarrow \textbf{skip}]; !s\searrow; [\neg ?s \longrightarrow \textbf{skip}];$$
$$!t\nearrow; [?t \longrightarrow \textbf{skip}]; !t\searrow; [\neg !t \longrightarrow \textbf{skip}]; ?d\nearrow; [\neg !d \longrightarrow \textbf{skip}]; ?d\searrow]$$
$$\textbf{end}$$

Using the general reshuffling principle, $?d\nearrow; [\neg !d \longrightarrow \textbf{skip}]$ can be moved to right after $!s\searrow$. Next, a state-holding element x is inserted, necessary to implement the above sequence of actions as a circuit. The rule for producing a circuit from the above then is

$$(\textbf{SEQ}) \quad m_{seq} \triangleright_5 m_{seq2}, \text{ where}$$
$$m_{seq2} = \textbf{withw } x, \textbf{ r } !d, \textbf{r } ?t, \textbf{r } ?s, \textbf{w } ?d, \textbf{w } !t, \textbf{w } !s$$
$$*[!s :=!d] \| *[x :=?s \; \mathbf{C} \; \neg ?t] \|$$
$$*[?d := x \vee ?t] \| *[!t := \neg ?s \wedge x]$$
$$\textbf{end}$$

5.5.3. Guarded Command

The individual guard processes after handshaking expansion look like

$$m_{grd} = \textbf{with r } !d, \textbf{r } ?s, \textbf{w } ?d, \textbf{w } !s \textbf{ do}$$
$$*[[!d \wedge f \longrightarrow \textbf{skip}]; !s\nearrow; [?s \longrightarrow \textbf{skip}]; !s\searrow; [\neg ?s \longrightarrow \textbf{skip}];$$
$$?d\nearrow; [\neg !d \longrightarrow \textbf{skip}]; ?d\searrow]$$
$$\textbf{end}$$

One difficulty in compiling guards is the condition f may not stay fixed, i.e. $!d \wedge f$ may become false at any time; thus, a state-holding element is necessary to store this value.

[†]Since none of these communication channels occurs more than once, the **(MOD)** rules do not apply and so the declarations are as they were directly after handshaking.

Using the general reshuffling principle, $?d\nearrow$ can be moved to right before $!s\nearrow$, and $[\neg!d \longrightarrow \mathbf{skip}]$ can be inserted right before $[?s \longrightarrow \mathbf{skip}]$. This may be rewritten to a circuit by the following rule.

$$
\begin{array}{lll}
(\mathbf{GRD}) & m_{grd} \rhd_5 m_{grd2}, \text{ where} & \\
& m_{grd2} = & \mathbf{with\ w}\ x, \mathbf{r}\ !d, \mathbf{r}\ ?s, \mathbf{w}\ ?d, \mathbf{w}\ !s\ \mathbf{w}\ t_1 \mathbf{w}\ t_2 \mathbf{do} \\
& & \quad *[t_1 :=\!!d \wedge f] \| *[t_2 := \neg!d \wedge ?s] \| \\
& & \quad *[x := t_1\ \mathbf{C}\ t_2] \| *[?d := x \vee ?s] \| *[!s := x] \\
& & \mathbf{end}
\end{array}
$$

5.5.4. Active communication

The individual active communications after handshaking expansion look like

$$
\begin{array}{lll}
m_{act} & = & \mathbf{with\ r}\ !d, \mathbf{r}\ ?a, \mathbf{w}\ ?d, \mathbf{w}\ !a\ \mathbf{do} \\
& & \quad *[[!d \longrightarrow \mathbf{skip}]; !a\nearrow; [?a \longrightarrow \mathbf{skip}]; !a\searrow; [\neg?a \longrightarrow \mathbf{skip}]; \\
& & \quad\ ?d\nearrow; [\neg!d \longrightarrow \mathbf{skip}]; ?d\searrow] \\
& & \mathbf{end}
\end{array}
$$

This is similar to the guarded command except there is no guard f. Also, assignments $!s$ in the guarded command are known to be the unique by the scoping, but $!a$ could be assigned elsewhere in the circuit, originating from having two active channels $A!$ at different points in the original circuit. If $!a$ is only assigned here, it is possible to globally rename $!a$ and $?a$ to $!d$ and $?d$ to get an equivalent circuit, but if not, an implemenation similar to the guarded command implementation is used: reshuffle and add a state holding element. The reshuffling must be different, however. This rewrites to a circuit as follows.

$$
\begin{array}{lll}
(\mathbf{ACT}) & m_{act} \rhd_5 m_{act2}, \text{ where} & \\
& m_{act2} = & \mathbf{with\ w}\ x,\ \mathbf{r}\ !d, \mathbf{r}\ ?a, \mathbf{w}\ ?d, \mathbf{w}\ !a\ \mathbf{do} \\
& & \quad *[x :=?a\ \mathbf{C}\ \neg!d] \| *[?d := \neg x \wedge \neg?a] \| *[!a :=\!!d \wedge \neg x] \\
& & \mathbf{end}
\end{array}
$$

5.5.5. Passive communication

The individual passive communications after handshaking expansion look like

$$
\begin{array}{lll}
m_{pass} & = & \mathbf{with\ r}\ !d, \mathbf{r}\ !p, \mathbf{w}\ ?d, \mathbf{w}\ ?p\ \mathbf{do} \\
& & \quad *[[!d \longrightarrow \mathbf{skip}]; [!p \longrightarrow \mathbf{skip}]; c; ?p\nearrow; [\neg!p \longrightarrow \mathbf{skip}]; \\
& & \quad\ ?p\searrow; ?d\nearrow; [\neg!d \longrightarrow \mathbf{skip}]; ?d\searrow] \\
& & \mathbf{end}
\end{array}
$$

This rewrites to a circuit as follows.

$$
\begin{array}{lll}
(\mathbf{PASS}) & m_{pass} \rhd_5 m_{pass2}, \text{ where} & \\
& m_{pass2} = & \mathbf{with\ w}\ x, \mathbf{r}\ !d, \mathbf{r}\ !p, \mathbf{w}\ ?d, \mathbf{w}\ ?p\ \mathbf{do} \\
& & \quad *[x :=\!!d\ \mathbf{C}\ !p] \| *[?d := x] \| *[?p := x] \\
& & \mathbf{end}
\end{array}
$$

5.6. Summarizing the Translation Process

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

Lemma 2 *If $m \in$ S-CSP, then $m \Rightarrow_{1-5} m_5$ for some $m_5 \in$ H-CSP.*

We now summarize the results that show the overall translation to be meaning-preserving. Since the method of synchronization is changed by the handshaking expansion, the testers must also change. Hence, there is no absolute notion of equivalence between specification and circuit. This is an inherent problem to any translation process that changes the way interaction occurs with the environment, so all such definitions of equality must address this problem. For the other phases, however, strong results can be obtained.

Lemma 3 (Correctness of phases 1,2,5) *For $i = 1$, 2, or 5, if $m \Rightarrow_i^* m'$, then $m = m'$*

To address the problem of showing rewrite systems 3 and 4 preserve semantic equivalence, we show that if a module passes some test and rewrites to another module (the result of handshaking or modularization), then if we *also* rewrite the test, the rewritten module passes the rewritten test. This is a sensible view to take of circuit tests. The main result we obtain may be outlined as follows. Suppose specification m compiles to circuit m_5. If m_5 is to be correct it should (*informally*) pass all tests that the specification also passes. What we show is, the specification passes test m^t just when m^t compiles to m_5^t and the circuit passes test m_5^t, so meaning is preserved by the translation. In the above description, tests are also objects that are compiled, so tests are restricted to be modules.

Definition 13 *The set of testing modules $\mathcal{T}_i$ for level i is $\{(-\|m) \mid m \in$ C-CSP*, $\Rightarrow_i m.\}$.*

Lemma 4 (Handshaking Correctness) *For all m_2, if $\Rightarrow_2 m_2 \Rightarrow_3^N m_3$ then for any closing testing module m_2^t such that $\Rightarrow_2 m_2^t \Rightarrow_3^N m_3^t$, $(m_2 \| m_2^t) =_{obs} (m_3 \| m_3^t)$.*

Lemma 5 (Modularization Correctness) *For all m_3, if $\Rightarrow_3 m_3 \Rightarrow_4^N m_4$ then for any closing testing module m_3^t such that $\Rightarrow_3 m_3^t \Rightarrow_4^N m_4^t$, $(m_3 \| m_3^t) =_{obs} (m_4 \| m_4^t)$.*

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

Theorem 2 (Correctness) *For all $m \in$ S-CSP, if $m \Rightarrow_{1-5} m_5$ then for any closing testing module $m^t \in$ S-CSP* such that $m^t \Rightarrow_{1-5} m_5^t$, $(m \| m^t) =_{obs} (m_5 \| m_5^t)$.*

These results give a strong argument for the meaning-preserving nature of the translations. From them one additional useful result may be derived. It would be desirable to show the compilation process to be equality-preserving: if specifications $m = m'$, and

$m \Rightarrow_{1-5} m_5$ and $m' \Rightarrow_{1-5} m'_5$, it should be that $m_5 = m'_5$. However, this result *fails* because of handshaking expansion: consider

$$m = \textbf{with } A? \textbf{ do } x := \textbf{true}; (*[A?] \| *[x := \textbf{true}]) \textbf{ end},$$
$$m' = \textbf{with } A? \textbf{ do } *[x := \textbf{true}; [\bar{A}? \longrightarrow A? [] \neg \bar{A}? \longrightarrow \textbf{skip}]] \textbf{ end}.$$

$m = m'$, but after compiling through the handshaking expansion phase they may be distinguished because a testing context may initiate a handshake (set $!a \uparrow$), but not ever complete the protocol by setting $!a \downarrow$. If it then set $x := \textbf{false}$, m might **ERROR** because mutual exclusion could be violated, but m' would never **ERROR** because it is deadlocked by the partial handshake. Thus, the two are distinguished. However, It can be argued that the above negative result arises because we force equal terms to behave the same on too many tests. If testers were not allowed to abuse the handshaking protocol as in the above example, the result would still hold. The testing equality thus may be restricted by limiting the tests to be the results of translations themselves, producing a different $=_i$ for each translation phase i. Since only modules may be translated, this restricted equality is defined for modules only, not all terms.

Definition 14 *$m =_i^M m'$, $i \in \{0 \ldots 4\}$, iff for all closing $m^t \in \mathcal{T}_i$, $m^t \| m =_{obs} m^t \| m'$.*

Theorem 3 *For all $i \in \{1 \ldots 4\}$, if $\Rightarrow_{i-1} m_{i-1} \Rightarrow_i^N m_i$, $\Rightarrow_{i-1} m'_{i-1} \Rightarrow_i^N m'_i$ and $m_{i-1} =_{i-1}^M m'_{i-1}$, then $m_i =_i^M m'_i$.*

Proof: The proof follows directly from lemmas 3, 4, and 5 above.

This then allows a module in a multi-module specification to be replaced with an equivalent module and still get an equivalent circuit upon re-compilation. justified.

Corollary 2 *Given some specification $m \| m'$, if $m =_0^M m'$ and $m \Rightarrow_{1-5} m_5$, $m' \Rightarrow_{1-5} m'_5$, $m'' \Rightarrow_{1-5} m''_5$, then $m''_5 \| m_5 =_5^M m''_5 \| m'_5$.*

6. Example

We now translate a simple specification into a circuit. The specification of the circuit consists of two concurrent processes, both of which modify a shared variable x. They are prevented from violating mutual exclusion by synchronization on the ports $P!$ and $P?$.

with $Q?, P!, P?, \mathbf{w}\ x, \mathbf{r}\ y, \mathbf{r}\ z, \mathbf{r}\ v$ **do**
$\quad *[[\bar{Q}? \longrightarrow x := y \wedge z; P!; Q?]] \| *[[\bar{P}? \longrightarrow x := x \wedge v; P?]]$
end

We begin by applying the first rewrite system. The first step is to introduce an "assignment cell" for x and then to change all assignments to x to the appropriate synchronization with the cell. After one application of (ASSIGN 1), two of (ASSIGN 2), and a

simplification of a sequential composition (SEQ 1) we have,

$$
\begin{array}{l}
\textbf{with } Q?, P!, P?, \textbf{w } x, \textbf{r } y, \textbf{r } z, \textbf{r } v \textbf{ do} \\
\quad *[\textbf{with } D_1?, D_0? \textbf{ do} \\
\qquad [\bar{D}_1? \longrightarrow x\uparrow; D_1? [] \bar{D}_0? \longrightarrow x\downarrow; D_0?] \\
\quad \textbf{end}] || \\
\quad \textbf{with } D_1!, D_0! \textbf{ do} \\
\qquad *[\textbf{with } D!, D? \textbf{ do} \\
\qquad\quad *[[\bar{D}? \longrightarrow [[(y \wedge z) \longrightarrow D_1! [] \neg(y \wedge z) \longrightarrow D_0!]]; D?]] || \\
\qquad\quad *[[\bar{Q}? \longrightarrow D!; P!; Q?]] \\
\qquad \textbf{end}] || \\
\qquad *[[\bar{P}? \longrightarrow [x \wedge v \longrightarrow D_1! [] \neg(x \wedge v) \longrightarrow D_0!]; P?]] \\
\quad \textbf{end} \\
\textbf{end}
\end{array}
$$

We now apply the second rewrite system in which we simplify the guards. Two applications of (GUARD 1) and a simplification of four complex guards using (GUARD 2) leaves us with the following term.

$$
\begin{array}{l}
\textbf{with } P!, P?, \textbf{w } x, \textbf{r } y, \textbf{r } z, \textbf{r } v \textbf{ do} \\
\quad *[\textbf{with } D_1?, D_0? \textbf{ do} \\
\qquad [\bar{D}_1? \longrightarrow x\uparrow; D_1? [] \bar{D}_0? \longrightarrow x\downarrow; D_0?] \\
\quad \textbf{end}] || \\
\quad \textbf{with } D_1!, D_0! \textbf{ do} \\
\qquad *[\textbf{with } D!, D? \textbf{ do} \\
\qquad\quad *[[(\bar{D}? \wedge y \wedge z) \longrightarrow D_1!; D? [] (\bar{D}? \wedge \neg y \wedge \neg z) \longrightarrow D_0!; D?]] || \\
\qquad\quad *[[\bar{Q}? \longrightarrow D!; P!; Q?]] \\
\qquad \textbf{end}] || \\
\qquad *[[(\bar{P}? \wedge x \wedge v) \longrightarrow D_1!; P? [] (\bar{P}? \wedge \neg x \wedge \neg v) \longrightarrow D_0!; P?]] \\
\quad \textbf{end} \\
\textbf{end}
\end{array}
$$

To save space, we collapse the handshaking expansion and modularization into a single step. Handshaking replaces each port with corresponding handshaking variables. Multiple occurrences to writing to the same handshaking variables in different places (specifically, $!d_1, !d_0, !d, !p$) are replaced by separate communication variables

$(!d_{1_1}, !d_{1_2}, !d_{0_1}, !d_{0_2}, !d_{l_1}, !dl_2, !p_1, !p_2)$.

$$
\begin{array}{l}
\textbf{with r } !q\textbf{w } !p, \textbf{r } ?p, \textbf{w } x, \textbf{r } y, \textbf{r } z, \textbf{r } v \textbf{ do} \\
\quad \textbf{with w } ?p, \textbf{r } ?p_1, \textbf{r } ?p_2 \textbf{ do } *[?p :=?p_1 \vee ?p_2] \textbf{ end} \| \\
\quad *[\textbf{with r } !d_1, \textbf{r } !d_0, \textbf{w } ?d_1, \textbf{w } ?d_0 \textbf{ do} \\
\qquad [!d_1 \longrightarrow x\uparrow; [!d_1 \longrightarrow \textbf{skip}]; ?d_1\nearrow; [\neg !d_1 \longrightarrow ?d_1\searrow] [] \\
\qquad\ !d_0 \longrightarrow x\downarrow; [!d_0 \longrightarrow \textbf{skip}]; ?d_0\nearrow; [\neg !d_0 \longrightarrow ?d_0\searrow]] \\
\qquad \textbf{end}] \| \\
\quad \textbf{with r } ?d_1, \textbf{r } ?d_0 \textbf{ do} \\
\qquad \textbf{with r } !d_{1_1}, \textbf{r } !d_{1_2}, \textbf{w } !d_1 \textbf{ do } *[!d_1 :=!d_{1_1} \vee !d_{1_2}] \textbf{ end} \| \\
\qquad \textbf{with r } !d_{0_1}, \textbf{r } !d_{0_2}, \textbf{w } !d_0 \textbf{ do } *[!d_0 :=!d_{0_1} \vee !d_{0_2}] \textbf{ end} \| \\
\qquad *[\textbf{with w } !d, \textbf{r } ?d \textbf{ do} \\
\qquad\quad \textbf{with w } ?d, \textbf{r } ?d_{l_1}, \textbf{r } ?d_{l_2} \textbf{ do } *[?d :=?d_{l_1} \vee ?d_{l_2}] \textbf{ end} \| \\
\qquad\quad *[[(!d \wedge y \wedge z) \longrightarrow !d_{1_1}\nearrow; [?d_1 \longrightarrow !d_{1_1}\searrow; [\neg !d_1 \longrightarrow \textbf{skip}]]; \\
\qquad\qquad\qquad [!d \longrightarrow \textbf{skip}]; ?d_{l_1}\nearrow; [\neg !d \longrightarrow ?d_{l_1}\searrow] [] \\
\qquad\qquad (!d \wedge \neg y \wedge \neg z) \longrightarrow !d_{0_1}\nearrow; [?d_0 \longrightarrow !d_{0_1}\searrow; [\neg !d_0 \longrightarrow \textbf{skip}]]; \\
\qquad\qquad\qquad [!d \longrightarrow \textbf{skip}]; ?d_{l_2}\nearrow; [\neg !d \longrightarrow ?d_{l_2}\searrow]] \| \\
\qquad\quad *[[!q \longrightarrow !d\nearrow; [?d \longrightarrow !d\searrow; [\neg !d \longrightarrow \textbf{skip}]]; \\
\qquad\qquad\ !p\nearrow; [?p \longrightarrow !p\searrow; [\neg !p \longrightarrow \textbf{skip}]] \\
\qquad\qquad\ [!q \longrightarrow \textbf{skip}]; ?q\nearrow; [\neg !q \longrightarrow ?q\searrow]]] \\
\qquad\quad \textbf{end}] \| \\
\qquad\quad *[[(!p \wedge x \wedge v) \longrightarrow !d_{1_2}\nearrow; [?d_1 \longrightarrow !d_{1_2}\searrow; [\neg ?d_1 \longrightarrow \textbf{skip}]]; \\
\qquad\qquad [!p \longrightarrow \textbf{skip}]; ?p_1\nearrow; [\neg !p \longrightarrow ?p_1; \searrow] [] \\
\qquad\qquad (!p \wedge \neg x \wedge \neg v) \longrightarrow !d_{0_2}\nearrow; [?d_0 \longrightarrow !d_{0_2}\searrow; [\neg ?d_0 \longrightarrow \textbf{skip}]]; \\
\qquad\qquad [!p \longrightarrow \textbf{skip}]; ?p_2\nearrow; [\neg !p \longrightarrow ?p_2\searrow]]] \\
\quad \textbf{end} \\
\textbf{end}
\end{array}
$$

Although somewhat daunting, this representation now translates into its circuit representation quite easily. We use the rules for assignment cells, active and passive communication, and guarded commands. For ease of notation, let

$$
\begin{array}{rcl}
\mathrm{GRD}(d, d', s, s') & = & \textbf{with w } x, \textbf{r } d, \textbf{r } s, \textbf{w } d', \textbf{w } s', \textbf{w } t_1, \textbf{r } t_1, \textbf{w } t_2, \textbf{r } t_2 \textbf{ do} \\
 & & \quad *[t_1 := d \wedge f] \| *[t_2 := \neg d \wedge s] \| \\
 & & \quad *[x := t_1 \ \mathbf{C}\ t_2] \| *[d' := x \vee ?s] \| *[s' := x] \\
 & & \textbf{end}
\end{array}
$$

Note that this is a generalization of m_{grd2} which allows us to use the "modularized" variables. This does not affect the correctness of the handshaking protocol. The final

circuit is:

$$
\begin{array}{l}
\textbf{with } \mathbf{w}\ ?p, \mathbf{r}\ ?p_1, \mathbf{r}\ ?p_2 \textbf{ do } *[?p := ?p_1 \vee ?p_2] \textbf{ end} \| \\
\textbf{with } \mathbf{w}\ x, \mathbf{r}\ !d_1, \mathbf{r}\ !d_0, \mathbf{w}\ ?d_1, \mathbf{w}\ ?d_0 \textbf{ do} \\
\quad *[x := !d_1\ \mathbf{C}\ \neg !d_0] \| *[?d_1 := !d_1 \wedge x] \| *[?d_0 := !d_0 \wedge \neg x] \\
\textbf{end} \| \\
\textbf{with } \mathbf{r}\ !d_{1_1}, \mathbf{r}\ !d_{1_2}, \mathbf{w}\ !d_1 \textbf{ do } *[!d_1 := !d_{1_1} \vee !d_{1_2}] \textbf{ end} \| \\
\textbf{with } \mathbf{r}\ !d_{0_1}, \mathbf{r}\ !d_{0_2}, \mathbf{w}\ !d_0 \textbf{ do } *[!d_0 := !d_{0_1} \vee !d_{0_2}] \textbf{ end} \| \\
\textbf{with } \mathbf{w}\ ?d, \mathbf{r}\ ?d_{l_1}, \mathbf{r}\ ?d_{l_2} \textbf{ do } *[?d := ?d_{l_1} \vee ?d_{l_2}] \textbf{ end} \| \\
\textbf{with } \mathbf{r}\ !d, \mathbf{r}\ !p, \mathbf{r}\ ?d_1, \mathbf{r}\ ?d_0 \textbf{ do} \\
\quad \mathrm{GRD}(!d, ?d_{l_1}, ?d_1, !d_{1_1}) \| \mathrm{GRD}(!d, ?d_{l_2}, ?d_0, !d_{0_1}) \| \\
\quad \mathrm{GRD}(!p, ?p_1, ?d_1, !d_{1_2}) \| \mathrm{GRD}(!p, ?p_2, ?d_0, !d_{0_2}) \| \\
\textbf{end} \| \\
\textbf{with } \mathbf{w}\ x_1, \mathbf{r}\ !q, \mathbf{r}\ ?d, \mathbf{r}\ ?p, \mathbf{w}\ ?q, \mathbf{w}\ !d, \mathbf{w}\ !p \textbf{ do} \\
\quad *[!d := !q] \| *[x_1 := ?d\ \mathbf{C}\ \neg ?p] \| \\
\quad *[?q := x_1 \vee ?p] \| *[!p := \neg ?d \wedge x_1] \\
\textbf{end}
\end{array}
$$

7. Conclusions

We have shown that Martin *et al.*'s methodology can be formalized and proven correct. In order to do so, we have had to make certain ideas implicit in their work explicit. These include the notions of mutual exclusion, delaying assignment, partial declarations, decomposition and correctness. Mutual exclusion is enforced by our semantics, delaying assignment is built into our language, variables may be declared read-only, decomposition becomes rewriting systems, and correctness is enforced through proofs of equivalence. In addition, we have defined the closure properties necessary for compiling a circuit, the environment of a circuit, and notions of separate compilation. Finally, we agree with Martin's assessment of the utility of asynchronous circuit design, and believe correctness considerations give another justification for using asynchronous design methods.

7.1. Future Work

Our work can be extended in several directions. First, the language we use is very minimal. One useful extension is the incorporation of n-bit data paths. Another is the implementation of synchronization that does not require mutual exclusion. The transformation process we have presented is minimal in that it incorporates no optimizations. Burns presents some simple optimizations in his thesis 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 regard to correctness, meaning there is a strong feasibility of using this method to produce fast hand-optimized circuits that are nonetheless verifiably correct.

References

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

[2] Robert S. Boyer and J. Strother Moore. *A Computational Logic.* Academic Press, 1979.

[3] S.D. Brookes, C.A.R. Hoare, and A.W. Roscoe. A Theory of Communicating Sequential Processes. *Journal of the ACM*, 31(3):560–599, July 1984.

[4] J.A. Brzozowski and C-J.H. Seger. Advances in Asynchronous Circuit Theory. *Bulletin of EATCS*, pages 198–248, October 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] M. J. Gordon. HOL: a Machine Oriented Formulation of Higher-Order Logic. Technical Report 68, University of Cambridge, Computing Lab, 1985.

[9] Brian Graham and Graham Birtwistle. Formalising the design of an secd chip. In *Workshop on Hardware Specification, Verification and Synthesis: Mathematical Aspects*, Cornell University, 1989.

[10] M. Hennessy. Synchronous and Asynchronous Experiments on Processes. *Journal of Information and Control*, 59(1-3):36–83, 1983.

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

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

[13] Warren A. Hunt Jr. The Mechanical Verification of a Microprocessor Design. In *Proc. of the IFIP Intl. Working Conference: From HDL Descriptions to Guaranteed Correct Circuit Design*, September 1986.

[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*, 1:226–234, 1986.

[16] 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.

[17] T.H.-Y. Meng, R.W. Broderson, and D.G. Messerschmitt. Automatic Synthesis of Asynchronous Circuits from High-Level Specifications. *IEEE Trans. on Computer-Aided Design*, 8(11):1185–1205, November 1989.

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

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

[20] C.H. van Berkel, C. Niessen, M. Rem, and R.W.J.J. Saeijs. VLSI Programming and Silicon Compilation; A Novel Approach from Philips Research. In *Proceedings of the 1988 IEEE Int. Conf. on Computer Design: VLSI in Computers and Processors*, 1988.