Formal Aspects of Computing (1998) 3: 537–559
© 1998 BCS

Verified Compilation of Communicating Processes into Clocked Circuits

John O'Leary[1], Geoffrey Brown[1] and Wayne Luk[2]

[1]School of Electrical Engineering, Cornell University, Ithaca, New York, USA
[2]Department of Computing, Imperial College of Science, Technology and Medicine, London, UK

Keywords: Handshake protocols; Protocol conversion; Hardware verification

Abstract. We have previously developed a verified algorithm for compiling programs written in an occam-like language into delay-insensitive circuits. In this paper we show how to retarget our compiler for clocked circuits. Since verifying a hardware compiler is a huge effort, it is significant that we are able to retarget our compiler proof without recreating that effort.
The chief contribution of this paper is the methodology used for retargeting our compiler which is based upon a new model for systems with both synchronous and asynchronous behavior. The retargeting proof utilizes both theorems proved algebraically by hand and theorems proved automatically by state exploration. The technique of protocol conversion is used extensively in modularizing the proof of the clocked implementation.

1. Introduction

Systems of communicating processes are a natural and convenient way to specify the behavior of complex digital hardware. In such systems, processes execute "asynchronously", but communicate through synchronous exchanges of data. For example the instruction fetch unit and instruction decode unit of a processor may operate at independent rates, yet synchronize when they exchange data. While the specification for a complex system may be conceptually asynchronous, most modern digital hardware is clocked.[1] The main contribution of this paper is to show how an asynchronous specification consisting of a number of communicating processes can be formally related

[1] Throughout this paper, we refer to "clocked circuits" in place of the more traditional "synchronous circuits" to avoid confusion with "synchronous communication" which need not be clocked.

Correspondence and offprint requests to: John O'Leary, Strategic CAD Labs, Intel Corporation, 5200 NE Elam Young Parkway, Hillsboro, OR 97124, USA.

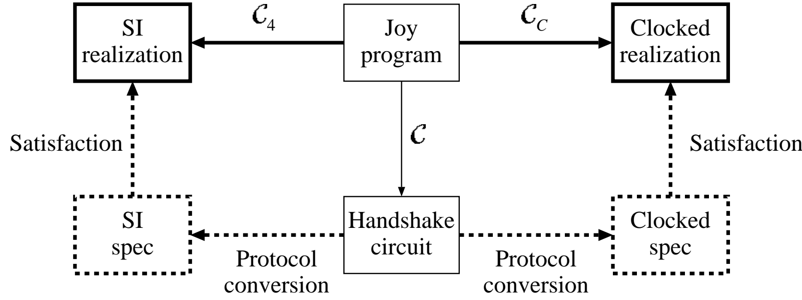


Fig. 1. Compiler Proof Structure

to a clocked hardware implementation. In particular we show how to relate the asynchronous intermediate form of a compiler for a communicating process language to the clocked hardware that the compiler back-end generates.

Perhaps the greatest contribution of communicating process languages such as occam is the development of semantic models that enjoy powerful sets of algebraic laws. These laws make it possible to show when one system is a satisfactory implementation of another, and therefore make communicating process languages attractive for the design of complex concurrent systems. The work described in this paper is part of a larger project to link concurrent system design with hardware design by developing verified hardware compilers for communicating process languages. It has previously been demonstrated that compiling occam-like languages into clocked hardware is feasible [PL91, WOB93], but it has not been shown that the intended behavior of the source language is preserved by the compiled hardware.

We have defined a core subset of occam called Joy, and developed verified compilation algorithms to translate source programs into both clocked and unclocked circuits. The basic compiler is structured into two components: a front end that translates arbitrary programs into a collection of primitive processes that communicate by handshaking and a back end that translates this intermediate form into circuits. Figure 1 shows the structure of the compiler and its proof. In [Bro91] we described a compiler $\mathcal{C}$ to intermediate form, and in [WBB92] and [H+96] we proved the correctness of this compiler. Joy is translated into either speed independent circuits or clocked circuits by the compilers $\mathcal{C}_4$ and $\mathcal{C}_C$, respectively. We do not discuss the compiler for speed independent circuits further in this paper.

An important technique that we adopt throughout this research is the use of protocol conversion in structuring our proofs. Protocol conversion is used in relating unclocked handshake modules to their clocked counterparts. In contrast to other proof methods such as those based on normal forms [HPB93], our approach allows us to exploit the environmental constraints imposed by the unclocked or the clocked handshake protocols, and formulating the correctness conditions is relatively straightforward.

The basic hardware building blocks for either clocked or unclocked circuits have the property (receptiveness) that they cannot refuse input events from their environment. Since unexpected inputs to a circuit may lead to aberrant behavior, it is the obligation of the circuit's environment to provide input only when the circuit is ready. The simplest way to guarantee satisfaction of this obligation is for the circuit and its environment to agree upon a basic communication protocol. We use a basic set of "handshake" protocols to guarantee that the circuits generated by our compiler do not diverge.

The remainder of this paper is organized as follows. We present our semantic model

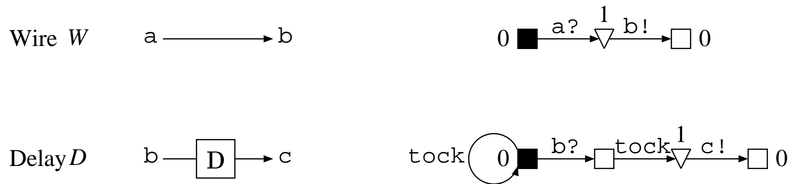


Fig. 2. Behavioral models of wire and delay

and, without proof, some of the algebraic properties it enjoys in Section 2. We briefly and informally discuss Joy and its intermediate form in Section 3. The translation from Joy to its intermediate form was inspired by the work of van Berkel *et al* [vB93] on compiling "handshake processes" into delay-insensitive circuits, and has been described in detail previously [WBB92, Bro91]. Section 4 introduces protocol conversion, the proof technique we use in verifying our clocked compiler, and discusses a form of refinement in our semantic model that allows the behaviors of our unclocked intermediate form and clocked realization to be related. In Section 5 we outline the correctness proof of our compiler, and examine the implementation of some clocked components in detail. The translation from Joy to clocked circuits was inspired by the work of Page and Luk [PL91], and the proof employs the protocol conversion technique introduced by Brown, Luk, and O'Leary [BLO96]. Section 6 contains some closing remarks.

2. Model

This section presents the highlights of our model of circuit behavior, discussed at greater length in [O'L95, OB97]. Our model is based on a variant of trace theory developed by Tom Verhoeff [Ver94]. Our model differs from Verhoeff's in that we use two types of events – directed asynchronous events and undirected synchronous events. No component can be prevented from producing asynchronous events at its outputs and no component can prevent asynchronous events from arriving at its inputs. In contrast, any component can prevent a synchronous event from occurring simply by refusing to participate. We use asynchronous events to model the communication that occurs between hardware components connected by wires and we use synchronous events to model the clocks that are are used in digital hardware. The existence of these clocks allows the digital designer to ignore most timing properties provided the period between clock events is sufficiently long. For example, in a system consisting of both combinational gates and clocked registers, the usual assumption is to treat the combinational logic as a function without delay. Rather than modeling delay directly and computing whether the clock period is "long enough", we model clocks by allowing the components that share a clock event to prevent the event from occurring until they are ready. Although adding a special clock event to model the passage of time seems an obvious approach, we found only one similar model reported in the literature [Bur89].

2.1. Overview

Figure 2 shows a wire W with input a and output b. The arrow from a to b is the symbol for a wire that we will use when we draw block diagrams of circuits. The inputs of the wire, denoted $\mathbf{i}W$, are the set $\{\mathtt{a}\}$, and the outputs of the wire, denoted $\mathbf{o}W$, are the

set $\{$b$\}$. Behaviors are described informally using state diagrams, in which we mark each state according to its "progressiveness". States which a system need not leave are marked $\Box$. States marked ∇ are ones which a system is bound to leave by performing an output. Thus, the state diagram of the wire in Figure 2 captures the requirement that each input on a causes an output on b, by marking state 1 as *transient* (∇). Input events are denoted in our diagrams by a trailing "?" and outputs by a trailing "!". We also observe the conventions that the darkened state in our diagrams denotes the initial state, and like-numbered states (for example, state 0 of the wire) are identical.

Figure 2 also shows a simple clocked circuit element – the *delay D* – and its behavior. Clocks are treated as a class of events distinct from inputs and outputs (the set of clock events of the delay, denoted **c***D*, is the set $\{$tock$\}$.) Clock periods are demarcated by occurrences of tock. From its initial state, the delay can either receive a b event or not; if a b event is received then a c event will be emitted in the following clock period. Our intent is that events in the behavioral model should correspond to unit-width voltage pulses in a physical circuit, with an event between tocks signifying a 1 bit and the lack of an event signifying a 0 bit. We chose the pulse interpretation of events to make it easier to relate abstract events in our compiler's asynchronous intermediate form to concrete events in the clocked implementation. The ordering of events between two successive tocks carries causality information that is essential to building this relation. The pulse interpretation of events requires a restriction on behavioral models: we consider it an error if two events arrive on a single input between two tocks, and we refrain from generating multiple events on any output between tocks. One function of the protocol converters we introduce in Section 4 will be to ensure that this restriction holds.

For a clock to tick, all components that share the clock must cooperate. States from which a tock departs are not generally deemed progressive (∇) unless some output event is also possible, since the occurrence of the tock event can be blocked by an unwilling component that shares the clock. The requirement for cooperation among all components allows us to capture the assumption, distributed among all components in a circuit, that the clock period is sufficiently long. Any physical realization of one of our specifications must satisfy this assumption. If clocks were treated as ordinary inputs, our models would need some additional mechanism to incorporate the assumption that the clock period is long enough.

Events that are neither inputs nor outputs nor clocks of a system cause no state change. Thus, in a system composed of the wire W followed by the delay D, W can allow any number of clock ticks to pass between receipt of an input event a and delivery of its output on b, while D enforces that exactly one clock tick pass between its input b and its output on c.

In addition to the indifferent and transient labels described above, states can also be labeled with Δ, $\perp$, and $\top$. States marked Δ, like ∇-states, are not indifferent to progress. Unlike ∇-states, the onus is upon the environment to force progress by supplying input. $\perp$ marks erroneous or "interfering" states – a system enters such a state if it receives an input it is not ready to accept, and its behavior is completely arbitrary afterward. $\top$ marks unreachable states. $\top$ and $\perp$ states normally do not appear in our state diagrams; we observe the convention that any unspecified input transition leads to a $\perp$ state, and any unspecified output or clock transition leads to a $\top$ state.

2.2. Trace Model of Systems

Following Verhoeff [Ver94], we assume a universal set Σ of events and call an element of Σ^* a *trace*. A system S is formally characterized in terms of its input events **i***S*, its output

events $\mathbf{o}S$, its clocks $\mathbf{c}S$, its hidden (internal) events $\mathbf{h}S$, and its *enhanced characteristic function* (ECF) $\mathbf{f}S$, a total function mapping traces to state labels ($\Sigma^* \to \{\top, \nabla, \Box, \Delta, \bot\}$).

$$S = (\mathbf{i}S, \mathbf{o}S, \mathbf{c}S, \mathbf{h}S, \mathbf{f}S)$$

We require that $\mathbf{i}S$, $\mathbf{o}S$, $\mathbf{c}S$, and $\mathbf{h}S$ be pairwise disjoint, and we define the alphabet of S $\mathbf{a}S = \mathbf{i}S \cup \mathbf{o}S \cup \mathbf{c}S \cup \mathbf{h}S$. The set of systems with inputs I, outputs O, and clocks C is denoted $\mathcal{SYS}(I,O,C)$.

As a simple example, the wire of Figure 2 corresponds to the system

$$W = (\{a\}, \{b\}, \{\}, \{\}, \mathbf{f}W)$$

We use a "." to denote function application, so the notation $\mathbf{f}W.\sigma$ should be read as "the ECF of W applied to trace σ". The ECF of the wire can be described using regular expressions as

$$\begin{aligned} \mathbf{f}W.\sigma &= \Box \quad \text{for } \sigma \uparrow \{a,b\} \in (ab)^* \\ &= \nabla \quad \text{for } \sigma \uparrow \{a,b\} \in (ab)^* a \\ &= \top \quad \text{for } \sigma \uparrow \{a,b\} \in (ab)^* b(a+b)^* \\ &= \bot \quad \text{for } \sigma \uparrow \{a,b\} \in (ab)^* aa(a+b)^* \end{aligned}$$

where $\sigma \uparrow \{a,b\}$ is the trace obtained by removing from σ all elements but a and b.

Systems in our model can be manipulated with three structural operators. First, the events of a system can be renamed according to a bijection $\Phi : \Sigma \to \Sigma$.

Definition 1. $\Phi.S = (\Phi.(\mathbf{i}S), \Phi.(\mathbf{o}S), \Phi.(\mathbf{c}S), \Phi.(\mathbf{h}S), \Phi.(\mathbf{f}S))$, where, for all $\sigma \in \Sigma^*$, $(\Phi.\mathbf{f}S).\sigma = \mathbf{f}S.(\Phi^{-1}.\sigma))$.

Second, certain output events of a system can be converted to internal events by means of the hiding operator "\".

Definition 2. $S \setminus H = (\mathbf{i}S, \mathbf{o}S \setminus H, \mathbf{c}S, \mathbf{h}S \cup H, \mathbf{f}S)$ provided $H \subseteq \mathbf{o}S$.

Finally, the parallel composition of systems S and T (written $S \parallel T$) is defined in terms of a composition operator $\Vert$ on trace labels:

$\Vert$	$\top$	∇	$\Box$	Δ	$\bot$
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$
∇	$\top$	∇	∇	∇	$\bot$
$\Box$	$\top$	∇	$\Box$	Δ	$\bot$
Δ	$\top$	∇	Δ	Δ	$\bot$
$\bot$	$\top$	$\bot$	$\bot$	$\bot$	$\bot$

$\Vert$ can be understood as follows. $\top$ states are intended to model states that would be unreachable in a physical implementation of the system. So, the composition of two systems is in an unreachable state if either of its constituents is. Suppose neither constituent is in an unreachable state. If either element is in a $\bot$ state, then their composition is in a $\bot$ state. Finally, suppose neither constituent is in a $\top$ nor a $\bot$ state. Then the system is transient if either of its constituents is transient, it is demanding if neither constituent is transient and one is demanding, and it is indifferent otherwise.

The composition of systems is defined only when S and T meet certain constraints: S and T must not have outputs in common, events that are clocks in S may not be inputs or outputs or T, and vice versa, and the internal events of each system must not interfere with the visible or internal events of the other.

Definition 3. If $\mathbf{o}S \cap \mathbf{o}T = \emptyset$, $\mathbf{c}S \cap (\mathbf{i}T \cup \mathbf{o}T) = (\mathbf{i}S \cup \mathbf{o}T) \cap \mathbf{c}T = \emptyset$, and $\mathbf{h}S \cap \mathbf{a}T = \mathbf{a}S \cap \mathbf{h}T = \emptyset$, then

$$S \parallel T = ((\mathbf{i}S \cup \mathbf{i}T) \setminus (\mathbf{o}S \cup \mathbf{o}T), \mathbf{o}S \cup \mathbf{o}T, \mathbf{c}S \cup \mathbf{c}T, \mathbf{h}S \cup \mathbf{h}T, \mathbf{f}(S \parallel T))$$

where $(\forall\sigma : \sigma \in \Sigma^* : \mathbf{f}(S \parallel T).\sigma = \mathbf{f}S.\sigma \| \mathbf{f}T.\sigma)$.

2.3. Satisfaction and equivalence

The central question in a model of system behavior is: when can two systems be used interchangeably? More generally, when can a system be safely used in place of another?

There is a natural ordering among state labels, $\top \sqsupset \nabla \sqsupset \Box \sqsupset \Delta \sqsupset \bot$. We say that a system S is correct (written *Correct*.S) if no reachable state is marked $\bot$ or Δ.

Definition 4. (Verhoeff) $Correct.S \iff (\forall\sigma : \sigma \in \Sigma^* : \mathbf{f}S.\sigma \sqsupseteq \Box)$

We then say that an implementation S satisfies its specification T (written $S\,\mathbf{sat}\,T$) if and only if S operates correctly in any context U in which T operates correctly.

Definition 5. For $S, T \in \mathcal{SYS}(I,O,C)$,

$$S\,\mathbf{sat}\,T \iff$$
$$(\forall U : U \in \mathcal{SYS}(O,I,C) : Correct.(S \parallel U) \Longleftarrow Correct.(T \parallel U))$$

$\Longrightarrow$ is reflexive and transitive, and the **sat** relation inherits these qualities. **sat** is therefore a preorder. Notice that in the formal definition of satisfaction, we restrict contexts to those whose inputs are exactly the outputs of S and T, and vice versa. The system and its context are therefore closed in the sense that their composition has no "dangling" inputs.

Definition 6. (Verhoeff) $S\,\mathbf{equ}\,T \iff S\,\mathbf{sat}\,T \wedge T\,\mathbf{sat}\,S$ □

equ is reflexive, symmetric, and transitive, and is thus an equivalence relation. Furthermore, relabeling, hiding, and composition are **sat**-monotonic and thus **equ** is a congruence.

An important result in the theory is that there is an algorithm to reduce any finite-state system T to a canonical form called the *composite of* T (denoted $[\![T]\!]$); furthermore, for finite-state systems S and T, $S\,\mathbf{sat}\,T$ can be proved or disproved by exploring the state space of $S \parallel\!\sim [\![T]\!]$[2]. This result is the basis of the model checking software we use in verifying our compiler.

We close this section by stating a number of algebraic laws our structural operators enjoy. The laws are those of a *circuit algebra* [Dil89], and codify some basic intuition about structural hardware description. For example, law 5 states that renaming the internal connections of a structure has no effect on its behavior. We will make extensive use of them in verifying the compiler.

Theorem 1. Let $\Phi : \Sigma \to \Sigma$, $\Psi : \Sigma \to \Sigma$ be bijections and $\mathbf{1}$ be the identity function on Σ. Then,

1. $\mathbf{1}.S\,\mathbf{equ}\,S$
2. $\Psi.(\Phi.S)\,\mathbf{equ}\,(\Phi \circ \Psi).S$
3. $S \setminus \emptyset\,\mathbf{equ}\,S$
4. $S \setminus C \setminus D\,\mathbf{equ}\,S \setminus (C \cup D)$

[2] $\sim$ is a "reflection" operator mapping $\bot$ states to $\top$ states and vice versa, Δ states to ∇ state and vice versa, and $\Box$ states to $\Box$ states.

5. $\Phi.(S \setminus C)\,\mathbf{equ}\,(\Phi.S) \setminus (\Phi.C)$
6. $S \parallel T\,\mathbf{equ}\,T \parallel S$
7. $(R \parallel S) \parallel T\,\mathbf{equ}\,R \parallel (S \parallel T)$
8. $\Phi.(S \parallel T)\,\mathbf{equ}\,(\Phi.S) \parallel (\Phi.T)$
9. $(S \setminus C) \parallel (T \setminus D)\,\mathbf{equ}\,(S \parallel T) \setminus (C \cup D)$ if $\mathbf{a}S \cap D = C \cap \mathbf{a}T = \emptyset$

□

3. Joy and its Intermediate Form

Our source language, Joy is a "core" subset of occam [Jon87]. In this section we introduce Joy and its compilation to an intermediate form. The picture we present here is necessarily incomplete – for a complete discussion, the reader is directed to [WBB92]. The discussion in this section is derived from that in [BLO96].

The simplest Joy process is **skip** which terminates without changing the state of variables or channels. More interesting is assignment, $v := B$, which assigns the value of expression B to variable v. In Joy, expressions and variables are restricted to Booleans, although the extension to allow other types is straightforward. Joy processes can be combined by sequential composition ($P_1;P_2$), or parallel composition ($P_1 \parallel_C P_2$). In parallel composition, the set of communication channels C over which the combined processes communicate is hidden.

To allow Joy processes to execute conditionally and iteratively, we introduce the notion of *guarded* processes. The primitive guarded process $B \to P$ is executable when the boolean guard expression B is true; its execution *succeeds* when P completes execution. If the guard expression B is false, $B \to P$ is said to *fail*. Two guarded processes may be composed to yield a third using $[\!]$, the "else" operator. In the guarded process $G_1 [\!] G_2$ we require that execution of G_2 be attempted only if execution of G_1 fails. The composite succeeds if either G_1 or G_2 succeeds, and fails if G_1 and G_2 both fail. The conditional **if** G **fi** attempts to execute the guarded process G until it succeeds. The iterative **do** G **od** repeatedly executes the guarded command G until G fails.

In Joy, communication between parallel processes is directed, though no values are passed. The process $c!$ awaits synchronization on channel c. We refer to $c!$ as a *send*; synchronization occurs when a receive – $c?$ – is evaluated as part of a guard expression in another process. The combination of guarded receives and unguarded sends allows Joy to emulate the value passing channels of occam using a one-hot code (that is, one channel per value).

The guarded process $B \& c? \to P$ is executable if B is true and another process executes the matching command $c!$. At most one receive is allowed per guard expression, and we only allow receives in guards contained in conditional (**if**) statements. There are other static restrictions on the use of channels and variables that are similar to the restrictions imposed by occam: channels must have exactly one sending and one receiving process, and variables must not be accessed by two or more concurrently active processes.

The Joy compiler translates each source program into an intermediate form consisting of a netlist of basic modules connected by wires. Each basic module corresponds directly to a Joy language construct. For example, Figure 3 shows the netlist of basic modules produced for the simple Joy program **skip ; skip**.

The basic modules in a netlist communicate through handshaking protocols. The *Skip* module, shown in Figure 4, provides a simple example. Execution of *Skip* is trig-

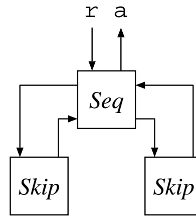


Fig. 3. Netlist for the program **skip ; skip**

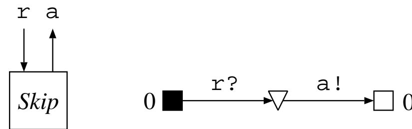


Fig. 4. Intermediate module for **skip**

gered by receipt of an event on r; *Skip* signals its termination by emitting an event on a.

Each basic module has one or more handshake interfaces; events on some wires trigger the execution of a module or its neighbors, while events on other wires signal the completion of execution. Figure 5 shows the intermediate module corresponding to the sequential composition of two processes. When triggered by receipt of an event on r, *Seq* starts the first process in the sequence by emitting an event on ar. When the first program signals termination via an event on aa, *Seq* starts the second program by emitting an event on br. *Seq* signals its own termination by emitting an event on a after it has received a termination signal from the second program on ba. Note that, to an observer who can see only events on r and a, the netlists of **skip** and **skip ; skip** are behaviorally indistinguishable.

Figure 6 illustrates the three types of handshaking interfaces we use for inter-module communication. The control interface, used by *Skip* and *Seq*, consists of request and acknowledgment signals, conventionally labeled r and q. The read interface is used in expression and guard evaluation. Evaluation of *P* is triggered by receipt of an event on r from *A*, and terminates when *P* returns an event on either a0 or a1, signifying that the value of *P* is 0 or 1. The write interface is used to assign values to variables: the Boolean value to be assigned is encoded on r0 and r1, and the variable returns an acknowledgment on a when the assignment is complete.

As an example of a module that has both read and write interfaces, consider the specification for a boolean variable illustrated in Figure 7. The labels 0, 1 in the state diagram denote the indifferent states in which the variable holds the values 0 and 1 respectively (initially the variable contains the value 0).

We have previously shown that, in the netlists our compiler generates, whenever a

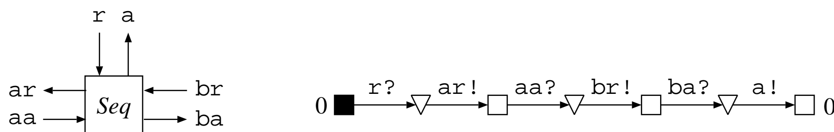
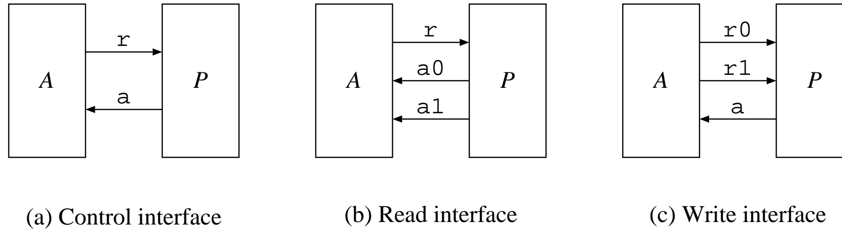


Fig. 5. Intermediate module for sequential composition



(a) Control interface (b) Read interface (c) Write interface

Fig. 6. Handshake Interfaces

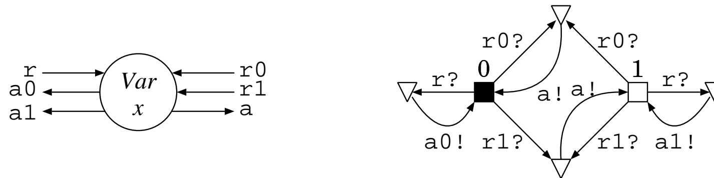


Fig. 7. Abstract variable specification

module wishes to transmit an event on a wire its partner is willing to receive it [WBB92]. Furthermore, the netlists we generate are delay-insensitive in the following sense: wires of arbitrary delay may be inserted in the handshake interfaces without affecting the correctness of the circuit's behavior. We will take advantage of the delay insensitivity of the intermediate netlists to introduce protocol converters that incorporate delay, as described in the next section.

4. Protocol Conversion and Timewise Satisfaction

The idea behind protocol conversion is extremely simple. Consider two elements P and Q, communicating via a protocol A. We wish to derive a pair of modified elements P' and Q' that communicate via protocol B. The first step to accomplish this is to insert a matched pair of converters – one converting protocol A to protocol B, and the other converting B back to A – between P and Q. This transformation is illustrated in Figure 8.

The second step is to obtain elements P' and Q' that do not require any converters. We start by computing the joint behavior of P and the AB converter (shown at the dashed

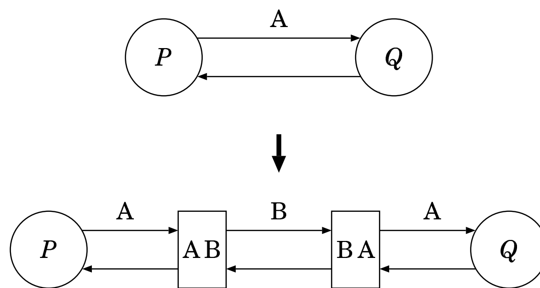


Fig. 8. Protocol conversion: transformation 1

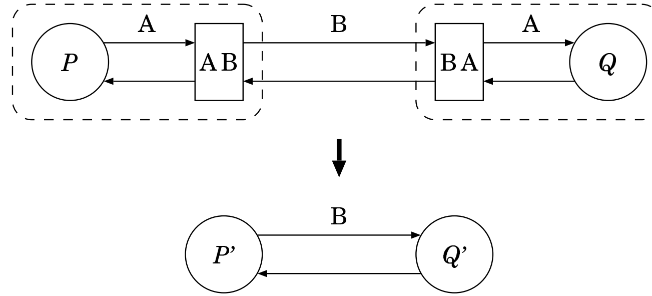


Fig. 9. Protocol conversion: transformation 2

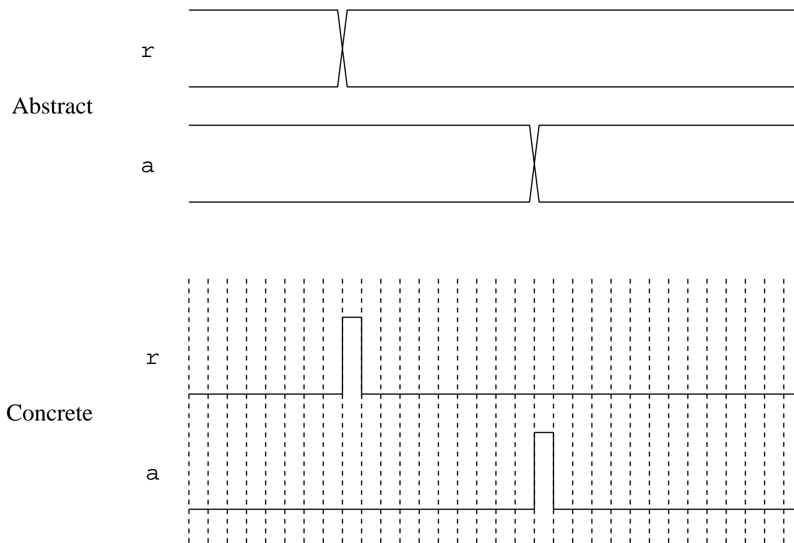


Fig. 10. Abstract (unclocked) and concrete (clocked) protocols

boundary). The result of this computation serves as the specification of P', and any P' meeting the specification is allowed. Q' is obtained in a similar way. This transformation is illustrated in Figure 9.

How do we know these transformations preserve correctness? The most difficult step to justify is the first one, and the property that we must prove is that of *transparency*. Informally, the transparency property states that P and Q cannot tell the difference between direct communication with one another and communication through the pair of protocol converters.

The second transformation simply requires proofs that

$$P' \ \mathbf{sat}\ (P \| AB) \quad \text{and} \quad Q' \ \mathbf{sat}\ (BA \| Q)$$

The monotonicity property of satisfaction in our theory ensures that this transformation preserves correct behavior in all contexts.

The principal function of the protocol converters in our proof is to add clocking information to the unclocked specifications. Figure 10 shows the abstract (unclocked) handshake above the concrete (clocked) handshake it corresponds to. The vertical lines

in the clocked handshake represent ticks of a global clock. These are modeled by `tock` events in the theory developed in Section 2.

Abstract events on `r` and `a` are represented concretely by unit-width voltage pulses. Intuitively, the correspondence between abstract and concrete levels is fairly straightforward because there are the same number of request and acknowledge events in the abstract and clocked protocols; however, the clocked protocol requires events to be scheduled to occur in specific time slots. Our mapping to concrete events allows at most one event per wire in each clock period.

To formally relate the behaviors of unclocked and clocked systems we introduce a variant of satisfaction called *timewise satisfaction* that accounts for the fact that progress in a clocked system can be driven by the passage of time. For example, the unclocked wire W of Figure 2 enters a transient state and is ready to emit an output immediately upon receipt of an input on `a`, but the clocked delay element D shown in the same figure needs one tock before it enters a transient state and is ready to emit an output. It is clearly not the case that D **sat** W; what is needed to relate D and W is a definition of satisfaction that recognizes that it is possible for D to "catch up" to W as time passes.

We begin by defining a new correctness predicate that allows a system (that is, a composition of modules) and its context to temporarily enter a "demanding" (Δ) state, if the system is sure to leave the Δ state within a finite number of clock events from a set C. In contrast, the correctness predicate of Definition 4 forbids Δ states entirely. In neither predicate do we tolerate $\bot$ states, because such states are symptomatic of a broken system.

Definition 7.

$$\begin{aligned} Correct_C.S \iff (\forall\sigma : \sigma \in \Sigma^* :\ & \mathbf{f}S.\sigma \sqsupseteq \Box \,\vee \\ & (\mathbf{f}S.\sigma = \Delta \wedge (\exists\rho : \rho \in C^* : \mathbf{f}S.\sigma\rho \sqsupseteq \Box))) \end{aligned}$$

□

A system S timewise satisfies T if, when T operates correctly in a context U, S eventually operates correctly in the same context.

Definition 8. For $S \in \mathcal{SYS}(I,O,C \cup C')$ and $T \in \mathcal{SYS}(I,O,C)$,

$$\begin{aligned} & S\ \mathbf{twsat}_{C'}\ T \iff \\ & \quad (\forall U : U \in \mathcal{SYS}(O,I,C) : Correct_{C'}.(S \parallel U) \Longleftarrow Correct.(T \parallel U)) \end{aligned}$$

□

Note that in the definition of **twsat**, S potentially has more clock labels than T, hence **twsat** can be used to relate clocked and unclocked system behaviors. Also, notice that $Correct.S \Longrightarrow Correct_C.S$, so it follows that systems related by **sat** are also related by **twsat**.

Theorem 2. $S\,\mathbf{sat}\,T \Longrightarrow S\ \mathbf{twsat}_C\,T$, where $C \subseteq \mathbf{c}S$ and $\mathbf{c}S = \mathbf{c}T$. □

Like **sat**, **twsat** is reflexive and transitive, and all our system-building operators are monotonic with respect to **twsat**.

To illustrate the use of the above results in the correctness proof of our clocked compiler, we will prove the transparency of protocol converters for the control interface. These converters, shown in Figure 11, capture an important decision in the design of the clocked Joy compiler: we require that all Joy programs take at least one clock cycle to

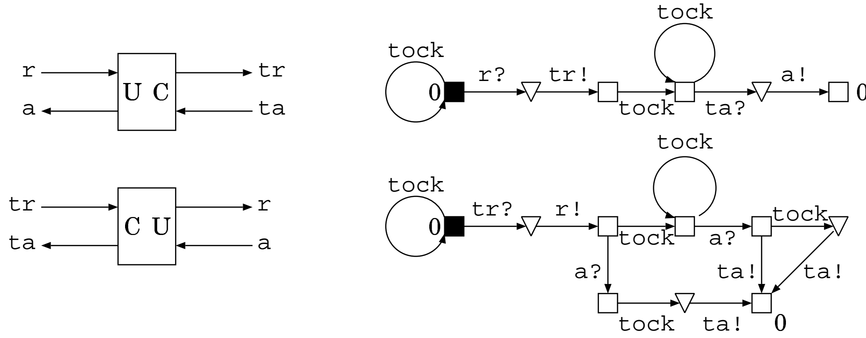


Fig. 11. Clocked protocol converters for the control interface

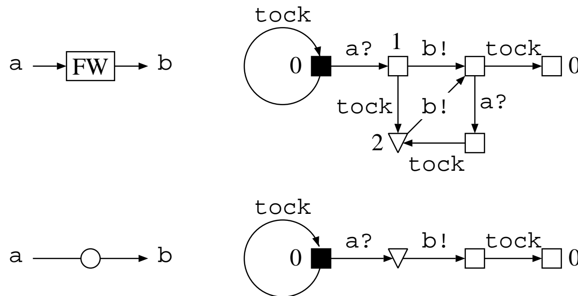


Fig. 12. Forgiving wire and clocked wire

execute[3]. The converter from the unclocked abstract protocol to the clocked protocol (top) assumes that the clocked acknowledgment ta must lag the clocked request tr by at least one clock cycle – it is an error for ta to arrive too early. The converter from the clocked to unclocked protocol (bottom) *enforces* that ta lags tr.

To help us prove our clocked converter pairs transparent, we introduce a component called a *forgiving wire*, shown as FW in Figure 12. For comparison, the behavior of a clocked wire is also shown in the same figure[4]. The forgiving wire has three interesting properties. First, it may choose to pass a received input to the output immediately (taking the trace $\langle \mathtt{a}?, \mathtt{b}!, \mathtt{tock} \rangle$) or it may choose to delay the output one clock cycle (taking the trace $\langle \mathtt{a}?, \mathtt{tock}, \mathtt{b}! \rangle$. State 1 of the forgiving wire is a $\square$ (indifferent) state because it is up to the wire to "decide" whether to immediately produce an output b! or wait. If the wire waits, it is then forced to emit b! without further delay, hence state 2 of the forgiving wire is ∇ (progressive). The variable delay of the forgiving wire will allow us to introduce protocol converters with variable delays, and ultimately will allow us some freedom of implementation. All our converters either do not delay, or delay only one clock period.

The second property earns the forgiving wire its nickname: it is able to absorb a second a? input after the trace $\langle \mathtt{a}?, \mathtt{b}! \rangle$. The need for forgiveness can be understood

[3] Requiring that all programs contain a delay is a simple way to ensure that no loops of combinational logic occur in the compiler's output.

[4] After receiving an input, the clocked wire delivers its output before allowing tock. In contrast, the delay of Figure 2 always performs tock before delivering its output.

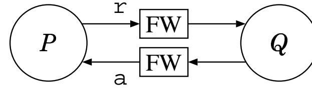


Fig. 13. Two systems connected with forgiving wires

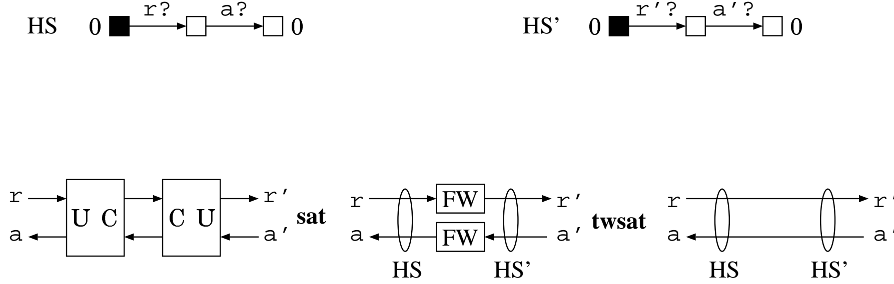


Fig. 14. Transparency of clocked protocol converters

by considering the small system shown in Figure 13, in which P and Q communicate through the abstract (unclocked) handshake protocol. Suppose P initiates a handshake by sending a request on `r`, the request is propagated to Q without delay, Q acknowledges without delay, and P receives the acknowledgment, again without delay. P is now entitled, according to the protocol, to issue a second request, yet there has been no `tock`. The connecting wires must be able to absorb the second request without error, but ordinary clocked wires are not able to do so.

The third and most important property of a forgiving wire is that it timewise satisfies an unclocked wire.

Lemma 1. FW $\mathbf{twsat}_{\{\texttt{tock}\}}$ W □

Lemma 1 and the monotonicity of || provide the first step of the transparency proof, namely that two forgiving wires timewise satisfy two unclocked wires. To complete the transparency proof, we show that a connected pair of converters satisfies (hence, timewise satisfies by Theorem 2) two forgiving wires. In this step we must supply some information about the handshake protocol followed by the Joy compiler's output; the model checking tool described in Section 2.3 is then able to establish the property automatically. Figure 14 shows the handshake constraints required, and the relationships that were proved among converter pairs, a constrained pair of forgiving wires and constrained pair of unclocked wires. The handshaking constraints are introduced by composing them in parallel with the wires. By Theorem 2 and transitivity of **twsat**, the clocked converter pair therefore timewise satisfies a pair of unclocked wires if the proper handshake protocols are obeyed.

Theorem 3. If handshake protocols are obeyed, then

$$(UC.(a,a'') \parallel CU.(a'',a')) \setminus \{a''\} \, \mathbf{twsat}_{\{\texttt{tock}\}} \, WIRES(a,a')$$

□

Because the handshaking modules in our intermediate netlist are delay-insensitive (that is, inserting extra wires between modules does not affect correctness), a netlist with clocked converter pairs inserted timewise satisfies a netlist with no converters.

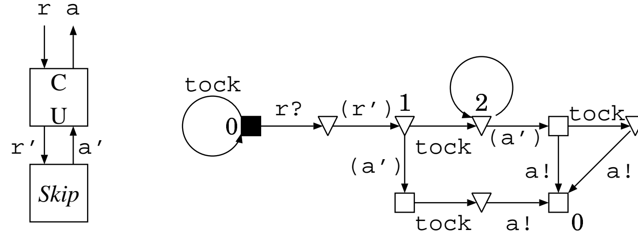


Fig. 15. Clocked **skip** specification

Transparency is proved in this way for each type of converter used in the compiler proof.

5. Deriving a verified compiler to clocked circuits

In this section we show how to refine the specifications of the abstract handshake modules our Joy compiler generates into specifications for equivalent modules that can be built with standard clocked components, using the protocol conversion technique described in Section 4. Correct clocked realizations of Joy programs are obtained in three steps:

1. We use a set of protocol converters along with our specifications for the modules of the intermediate form to generate specifications for clocked versions of the modules.
2. We then use an automatic tool to check that our proposed clocked implementations satisfy their clocked specifications.
3. Finally the laws presented in Theorem 1 are used to tie the individual component proofs together, showing that any valid Joy program has a correct clocked implementation.

The correctness criterion for our clocked Joy compiler is that the clocked realization of an arbitrary program, when viewed through an unclocked-to-clocked protocol converter at its control port, timewise satisfies its unclocked counterpart.

Theorem 4. $(\forall p : p \in \mathit{JoyProg} : (UC(a,a') \parallel \mathcal{C}_C.p.a) \setminus \{a'\}\ \mathbf{twsat}_{\{\mathtt{tock}\}}\ \mathcal{C}.p.a)$ $\square$

Here, $\mathcal{C}$ is the compilation function mapping Joy programs to netlists of intermediate modules, and $\mathcal{C}_C$ is the compiler from Joy to clocked realizations. The relationship between $\mathcal{C}$ and $\mathcal{C}_C$ was illustrated in Figure 1.

The proof of Theorem 4 is by induction on the structure of Joy programs. Space constraints prevent our presenting the entire proof in this paper. Instead, we will present several illustrative cases that demonstrate the use of our automatic verifier in deriving clocked realizations of individual modules, and we will outline the inductive proof of Theorem 4.

5.1. Clocked realization of handshake components

We now illustrate how our protocol converters are employed to derive clocked implementations of two simple components. Recall that the Joy program **skip** is implemented by the intermediate module *Skip*, whose behavior is shown in Figure 4. Figure 15 shows

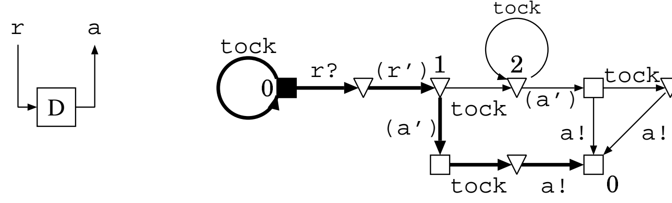


Fig. 16. Clocked **skip** realization

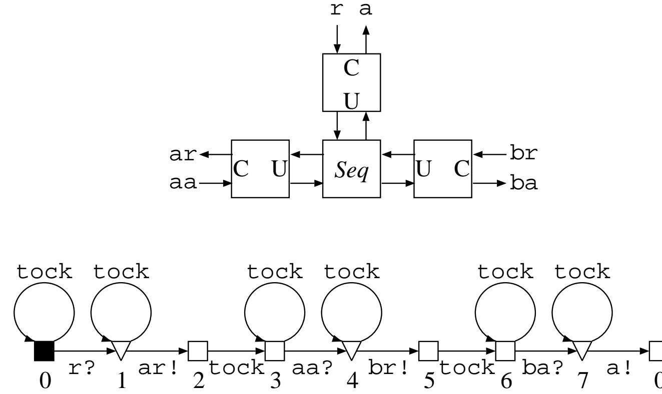


Fig. 17. Clocked specification of sequential composition

the behavior of *Skip* when combined with the clocked-to-unclocked protocol converter *CU*; we use the behavior of *Skip* plus converters as the specification for a legal clocked realization of *Skip*. The transitions `(r')` and `(a')` are parenthesized to indicate that they are internal to the specification; no event on `r'` or `a'` is visible externally.

Figure 16 shows our proposed realization of *Skip*: a clocked delay element. The behavior of the clocked delay element is shown in the figure in dark arrows, superimposed on the behavior of the specification. Informally, we can see that any system connected to the delay at `r`, `a` can observe one of the legal behaviors of the specification. We will refer to our realization as *SkipC*. Our automated model checker is able to prove that the following lemma holds.

Lemma 2. $\mathit{SkipC}.a$ **sat** $(\mathit{CU}.(a,a') \parallel \mathit{Skip}.a') \setminus \{a'\}$ □

The intermediate module for sequential composition provides a more complex example. Figure 17 shows the behavior of the intermediate module and its neighboring clocked protocol converters; the behavior of the intermediate module alone was shown in Figure 5. The tock self-loops at states 1, 4, and 7 of Figure 17 represent situations in which *Seq* is in a transient state and the protocol converters are in indifferent states.

Figure 18 shows the sequencing module, composed of three clocked wires, we use in our clocked Joy compiler. The figure also shows the behavior of the realization, drawn in dark arrows, superimposed on the state diagram of the specification. We do not show behaviors of the realization that step outside the specification's state diagram due to receipt of an unexpected input. The specification allows any behavior under these conditions. It is again evident from the diagram that the behaviors of *SeqC* are legal behaviors of its specification. Our model checker is able to prove the following result.

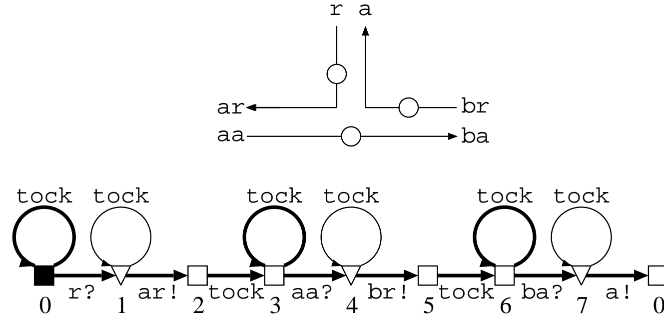


Fig. 18. Clocked wires realize clocked sequential composition

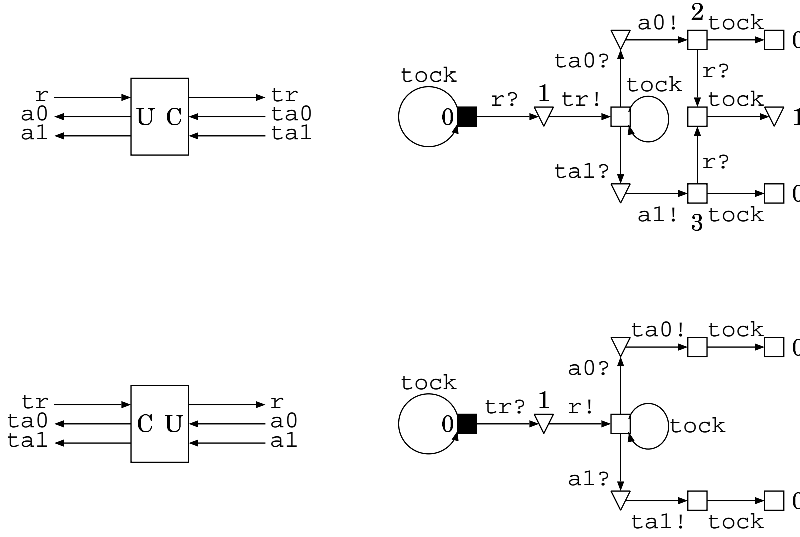


Fig. 19. Clocked protocol converters for the read interface

Lemma 3. $SeqC.(a, b, c)$
sat
$(CU.(a, a') \parallel Seq.(a', b', c') \parallel UC.(b', b) \parallel UC.(c', c)) \setminus \{a', b', c'\}$

□

The same method is used to derive clocked realizations of all the required components: the unclocked intermediate model is composed with protocol converters to yield a clocked specification, then we verify that clocked realizations meet their specifications. These verifications can all be handled by the automated verifier described in Section 2.

5.2. A Clocked Variable

To highlight some issues raised by a clocked implementation, we derive a clocked realization of the handshake component that implements a Boolean variable. The variable has two ports, one for reading and one for writing.

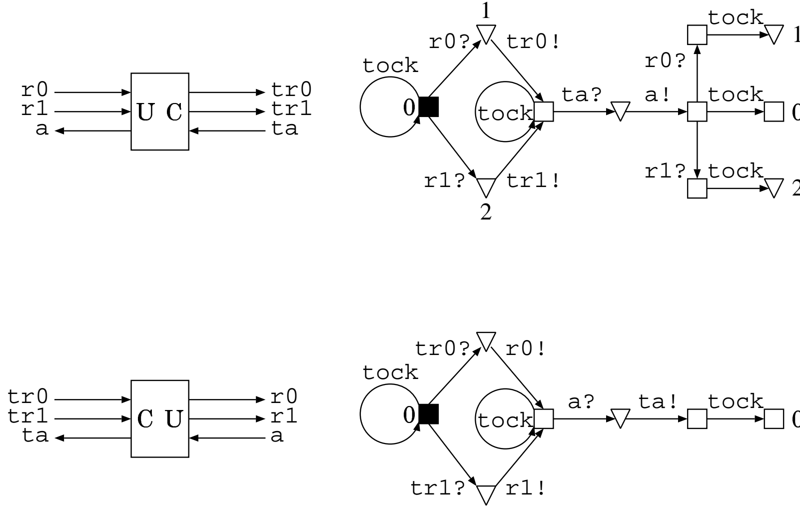


Fig. 20. Clocked protocol converters for the write interface

Protocol converters for the read handshake protocol are shown in Figure 19. Since events in the abstract protocol correspond closely with events in the clocked protocol, the two converters are very nearly symmetric. Upon receiving an input request or acknowledgment, the converters generate the appropriate output request or acknowledgment without delay. The difference between them is that the UC converter must be willing to forgive early requests on `r` received in states 2 and 3. The (`tr`, `ta0`, `ta1`) interface of the CU converter is clocked, and a clocked environment promises never to issue such events.

Figure 20 shows a pair of write converters, which are similar to the read converters we just discussed. The UC converter has the ability to absorb additional requests from the unclocked partner after completing a handshake but before `tock`. Both read and write converter pairs satisfy our transparency property when connected back-to-back.

We now have everything we need to derive the specification of a clocked variable. Recall the abstract specification of the variable in Figure 7. Figure 21 shows the abstract specification with the appropriate converters, and the concrete protocol behavior that is visible at the dashed boundary, subject to the following assumption:

If a read follows a write, there must be a `tock` between them.

The assumption is encoded much as the handshake constraints were encoded in proving transparency, and composed with the variable and converters to "filter out" behavior that violates the assumption.

We need to make the above assumption about read/write ordering because we wish eventually to implement the variable using a clocked storage element, whose state will not be updated until a clock period has elapsed. If a read were attempted after a write but before `tock`, an incorrect value could be returned. For example, from State 0 the clocked variable specification allows the sequence $\langle \mathtt{r}?, \mathtt{a0}!, \mathtt{r1}?, \mathtt{a}!, \mathtt{tock}\rangle$ but not the sequence $\langle \mathtt{r1}?, \mathtt{a}!, \mathtt{r}?, \mathtt{a0}!, \mathtt{tock}\rangle$. Of course, we are obliged to prove later that this assumption is justified. The justification is that programs composed in parallel cannot share variables due to restrictions in the source language. Although variables may be shared among programs composed in sequence or in different arms of an iterative or conditional construct,

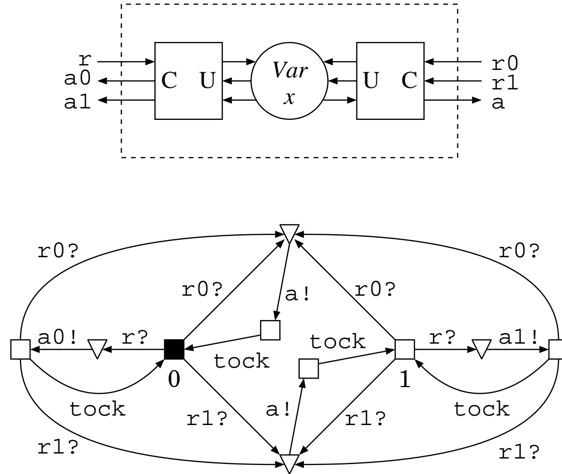


Fig. 21. Clocked variable specification

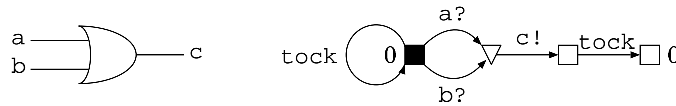


Fig. 22. Clocked OR gate

the nature of these constructs guarantees that no two subprograms are ever concurrently active. The only atomic program that can read and write to a single variable in a single clock cycle is assignment, and its behavior ensures that writes follow reads.

The variable's implementation uses an OR gate, AND gates, and a clocked SR flip-flop. The OR gate in Figure 22 responds to an event on one of its inputs with an event on its output `c`. Its specification suggests that OR is being used as a merge – at most one event can occur on either input per clock cycle. We use a restricted OR gate model because our implementation does not require more general behavior. In fact, the specification of this restricted model is satisfied by a more general model, hence, our implementations are correct when built with standard gates. The AND gate is shown in Figure 23.

The clocked SR flip-flop shown in Figure 24 is the most complicated component we will use. In its initial (darkened) state it stores a 0 and emits an event on `qb`. An event on its input `r` causes no change in the stored state, but an event on `s` causes an eventual transition into the lower half of the state graph, which is a mirror-image of the

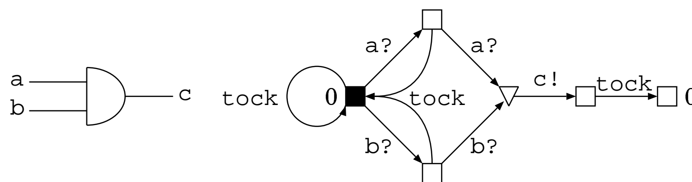


Fig. 23. Clocked AND gate

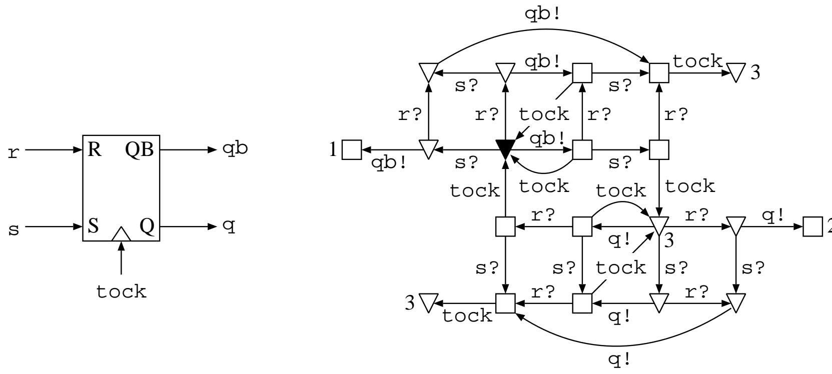


Fig. 24. Clocked SR flip-flop (set-dominant)

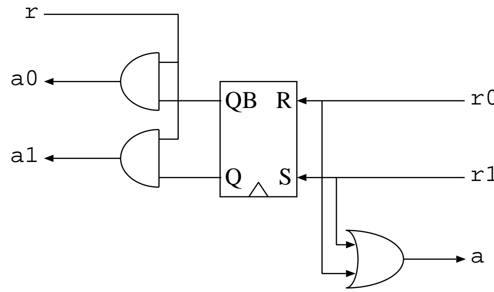


Fig. 25. Clocked variable implementation

upper. Our SR flip-flip is a synchronous component and is *set dominant* – if `r` and `s` are asserted simultaneously, `s` wins.

With these components in hand we can build the clocked variable shown in Figure 25. At the read side, two AND gates transfer the value stored in the flip-flop out to the reader when a request `r` is received. At the write side, `r0` and `r1` change the flip-flop's state [5]. The two requests are merged to form the acknowledgment. The correctness of this realization with respect to its specification,

$$VarC.(r,w) \ \mathbf{sat} \ (RdCU.(r,r') \parallel Var.(r',w') \parallel WrCU.(w,w')) \setminus \{r',w'\}$$

can then be checked by the model checker described in Section 2.3.

5.3. Correctness proof for clocked Joy compiler

Clocked hardware implementations for expressions and statements can be developed and verified using the technique of protocol conversion. In all, four sets of protocol converters are required to demonstrate the correctness of the clocked realizations of the abstract intermediate modules[6].

[5] `r0` and `r1` events are mutually exclusive due to our handshake protocol.
[6] We have already seen converters for the control, read, and write interfaces. The fourth converter type is a hybrid of the control and read converters, used to obtain clocked realizations of guarded processes.

To complete the verification of our clocked compilation scheme, we need to show that a clocked circuit produced by our compiler timewise satisfies the corresponding abstract netlist with abstract to clocked protocol converters at its external interfaces (as stated in Theorem 4). This result can be proved by induction on the structure of Joy programs, using the algebraic laws given in Theorem 1 and the result, outlined above, that clocked realizations timewise satisfy their unclocked specifications.

A typical base case is for the process **skip**, which compiles to the intermediate module *Skip* shown in Figure 4. Our abstract compilation function $\mathcal{C}$ defines $\mathcal{C}.\mathbf{skip}.a = \mathit{Skip}.a$, where a is the control interface for the module. The concrete compilation function $\mathcal{C}_C$ defines $\mathcal{C}_C.\mathbf{skip}.a = \mathit{SkipC}.a$; the clocked realization *SkipC* was shown in Figure 16. We have

$$
\begin{array}{ll}
 & (UC.(a,a') \parallel \mathcal{C}_C.\mathbf{skip}.a') \setminus \{a'\} \\
= & \{\text{Definition of } \mathcal{C}_C\} \\
 & (UC.(a,a') \parallel \mathit{SkipC}.a') \setminus \{a'\} \\
\mathbf{sat} & \{\text{Lemma 2 -- } \mathit{SkipC} \text{ realizes } \mathit{Skip}\} \\
 & (UC.(a,a') \parallel (CU.(a',a'') \parallel \mathit{Skip}.a'') \setminus \{a''\}) \setminus \{a'\} \\
\mathbf{equ} & \{\text{Theorem 1 -- algebraic reasoning}\} \\
 & ((UC.(a,a') \parallel CU.(a',a'')) \setminus \{a'\} \parallel \mathit{Skip}.a'') \setminus \{a''\} \\
\mathbf{twsat} & \{\text{Theorem 3 -- transparency of } UC \text{ and } CU\} \\
 & (\mathit{WIRES}.(a,a'') \parallel \mathit{Skip}.a'') \setminus \{a''\} \\
\mathbf{equ} & \{\text{Delay-insensitivity of } \mathit{Skip}\} \\
 & \mathit{Skip}.a \\
= & \{\text{Definition of } \mathcal{C}\} \\
 & \mathcal{C}.\mathbf{skip}.a
\end{array}
$$

A typical induction step involves the sequential composition operator, which compiles to the intermediate module *Seq* shown in Figure 5. Our abstract compilation function $\mathcal{C}$ defines $\mathcal{C}.(p_1;p_2).a = (\mathit{Seq}.(a,b,c) \parallel \mathcal{C}.p_1.b \parallel \mathcal{C}.p_2.c) \setminus \{b,c\}$, and the concrete compilation function $\mathcal{C}_C$ defines $\mathcal{C}_C.(p_1;p_2).a = (\mathit{SeqC}.(a,b,c) \parallel \mathcal{C}_C.p_1.b \parallel \mathcal{C}_C.p_2.c) \setminus \{b,c\}$. The clocked realization *SeqC* was shown in Figure 18. We have

$$
\begin{array}{ll}
 & (UC.(a,a') \parallel \mathcal{C}_C.(p_1;p_2).a') \setminus \{a'\} \\
= & \{\text{Definition of } \mathcal{C}_C\} \\
 & (UC.(a,a') \parallel \\
 & \quad (\mathit{SeqC}.(a',b,c) \parallel \mathcal{C}_C.p_1.b \parallel \mathcal{C}_C.p_2.c) \setminus \{b,c\}) \setminus \{a'\} \\
\mathbf{sat} & \{\text{Lemma 3 -- } \mathit{SeqC} \text{ realizes } \mathit{Seq}\} \\
 & (UC.(a,a') \parallel \\
 & \quad ((CU.(a',a'') \parallel \mathit{Seq}.(a'',b',c') \parallel \\
 & \quad\; UC.(b',b) \parallel UC.(c',c)) \setminus \{a'',b',c'\} \parallel \\
 & \quad \mathcal{C}_C.p_1.b \parallel \mathcal{C}_C.p_2.c) \setminus \{b,c\}) \setminus \{a'\}
\end{array}
$$

equu {Theorem 1 – algebraic reasoning}
$((UC.(a,a') \parallel CU.(a',a'')) \setminus \{a'\} \parallel$
$Seq.(a'',b',c') \parallel$
$(UC.(b',b) \parallel \mathcal{C}_C.p_1.b) \setminus \{b\}$
$(UC.(c',c) \parallel \mathcal{C}_C.p_2.c) \setminus \{c\}) \setminus \{a'',b',c'\}$

twsat {Theorem 3 – transparency of UC and CU}
$(WIRES.(a,a'') \parallel$
$Seq.(a'',b',c') \parallel$
$(UC.(b',b) \parallel \mathcal{C}_C.p_1.b) \setminus \{b\}$
$(UC.(c',c) \parallel \mathcal{C}_C.p_2.c) \setminus \{c\}) \setminus \{a'',b',c'\}$

sat {Delay-insensitivity of Seq}
$(Seq.(a'',b',c') \parallel$
$(UC.(b',b) \parallel \mathcal{C}_C.p_1.b) \setminus \{b\}$
$(UC.(c',c) \parallel \mathcal{C}_C.p_2.c) \setminus \{c\}) \setminus \{a'',b',c'\}$

twsat {Induction hypothesis}
$(Seq.(a'',b',c') \parallel \mathcal{C}.p_1.b' \parallel \mathcal{C}.p_2.c') \setminus \{a'',b',c'\}$

$=$ {Definition of $\mathcal{C}$}
$\mathcal{C}.(p_1;p_2).a$

The remaining cases in the proof for Joy processes are similar; similar proofs are required for the boolean expressions B and the guarded processes G.

6. Discussion

The key to verifying our compiler for clocked circuits is a circuit model capable of expressing both clocked and unclocked behavior. We introduced such a model, along with a notion of timewise satisfaction which provides the link between the asynchronous specification of our intermediate modules and their clocked implementation. Since the model is simple, it is amenable to a high degree of automation, hence eliminating the need to verify by hand the base cases of our proof. The close link between our theoretical development and the model checker illustrates that it is possible to produce a practical, demonstrably correct proof tool.

Protocol conversion has proven to be a useful technique in formulating correctness criteria and in modularizing their verification. The basic idea behind protocol conversion – outlined in Section 4 – is readily understandable, and different sets of protocol converters can be used in deriving different target implementations while retaining the structure of the proof [BLO96, O'L95]. The technique of protocol conversion is not merely applicable within our trace-based semantic model. For example, the simple compilation scheme that we verify in this paper can be improved by replacing the one-hot protocol we use for read and write handshake interfaces with a data-plus-valid protocol. Converters between the one-hot and data-plus-valid protocols can be constructed of combinational logic gates, and optimized implementations derived using Boolean algebra.

One potential problem with the approach we have taken is that it requires the use of two behavioral models – a high level model such as the failures-divergence model of CSP for compiling to intermediate form and the low level model of Section 2 for capturing circuit behavior. We wish to use CSP as a high level model because of the rich algebra developed for reasoning about CSP systems [Hoa85]. Given this choice, it is reasonable to ask why we do not perform the entire proof within the CSP model.

Every model for reasoning about concurrent systems makes compromises. These compromises are most evident in the systems that a model treats as identical and in

those that the model distinguishes. For example, we firmly believe that circuits (and their environments) should be considered as receptive. In our hardware model, a specification that allows two outputs to be produced in either order can be implemented by a circuit that produces the outputs in a fixed order. In CSP, the circuit is not a satisfactory implementation because it can be distinguished from the specification by an environment that will only accept outputs in the other order. The ability of the environment to "refuse" events allows it to distinguish the implementation from the specification.

Since we have chosen to use distinct high and low level models we are faced with the problem of showing that the descriptions of our intermediate modules in the two models are somehow equivalent or at least that our low level description is "better than" our high level description. With He Jifeng, we have recently shown a Galois connection between a class of CSP processes (which includes the CSP descriptions of our intermediate modules) and a class of circuits (which includes our low level model of the intermediate modules). The existence of such a Galois connection verifies the necessary correspondence between the two models [He89].

Some readers may recognize that currently our source language only has boolean variables and synchronization channels. We have begun work on extending our proof technique for a more realistic language, with facilities such as integer variables and value passing channels. Further work may include justified optimizations of the generated hardware, and the specification and verification of external interfaces to memories, buses and sensors.

Acknowledgments

We are grateful to He Jifeng for his comments on an earlier version of this work. John O'Leary was supported by NSF grants CCR-9058180 and CCR-9224575 under a joint ESPRIT-NSF program, and by a fellowship from Bell-Northern Research Limited. Geoffrey Brown was supported by NSF grant CCR-9058180 and matching funds from AT&T. Wayne Luk was supported by the ESPRIT OMI/HORN (7249) project.

References

[BLO96] Brown, G., Luk, W. and O'Leary, J.: Retargeting a Hardware Compiler Using Protocol Converters. *Formal Aspects of Computing*, **8**, 209–237 (1996).

[Bro91] Brown, G. M.: Towards Truly Delay-Insensitive Circuit Realizations of Process Algebras. *Designing Correct Circuits*, Jones, G. and Sheeran, M. (editors), pp. 120–131. Springer-Verlag, 1991.

[Bur89] Burch, J. R.: Modeling Timing Assumptions with Trace Theory. *International Conference on Computer Design*, pp. 208–211, 1989.

[Dil89] Dill, D. L.: *Trace Theory for Automatic Hierarchical Verification of Speed-Independent Circuits.* MIT Press, 1989.

[He89] He, J.: Process Simulation and Refinement. *Formal Aspects of Computing*, **1**, 229–241 (1989).

[H+96] He, J., Brown, G., Luk, W. and O'Leary, J.W.: Deriving Two-Phase Modules for a Multi-Target Hardware Compiler. Designing Correct Circuits, Springer Electronic Workshop in Computing series, 1996.

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

[HPB93] He, J., Page, I. and Bowen, J.: Towards a Provably Correct Hardware Implementation of occam. *Correct Hardware Design and Verification Methods*, volume 683 of *Lecture Notes in Computer Science*, pp. 214–225, Springer-Verlag, 1993.

[Jon87] Jones, G.. *Programming in occam.* Prentice-Hall International, 1987.

[OB97] O'Leary, J. and Brown, G.: A Model for Verifying Digital Circuits. *Acta Informatica*, in press, 1997.

[O'L95] O'Leary, J. W.: *A Model and Proof Technique for Verifying Hardware Compilers for Communicating Processes.* Ph.D. thesis, Cornell University, 1995.

[PL91] Page, I. and Luk, W.: Compiling occam into FPGAs. *FPGAs*, Moore, W. and Luk, W. (editors), pp. 271–283, Abingdon EE&CS Books, Abingdon, England, 1991.

[vB93] van Berkel, K.: *Handshake Circuits : An Asynchronous Architecture for VLSI Programming*. Cambridge University Press, 1993.

[Ver94] Verhoeff, T.: *A Theory of Delay-Insensitive Systems.* Ph.D. thesis, Eindhoven University of Technology, 1994.

[WBB92] Weber, S., Bloom, B. and Brown, G. M.: Compiling Joy into Silicon. *Advanced Research in VLSI and Parallel Systems: Proceedings of the 1992 Brown/MIT Conference*, pp. 79–98, 1992.

[WOB93] Wenban, A. S., O'Leary, J. W. and Brown, G. M.: Codesign of Communication Protocols. *Computer*, **26**(12), 46–52 (1993).