Retargeting a Hardware Compiler Proof using Protocol Converters

Geoffrey Brown* Wayne Luk† John O'Leary‡

Abstract

We show how to retarget the correctness proof of a hardware compiler generating two-phase delay-insensitive circuits to a compiler generating four-phase speed-independent circuits. We use protocol converters to convert the specifications of our compiler's two-phase circuit elements into equivalent specifications for four-phase elements. The processes of converting the specifications and verifying their implementations are automated.

1 Introduction

Large asynchronous circuits are difficult to design by hand, so it is desirable to provide a compilation scheme which assures the correct production of such circuits. We have previously developed and verified a compiler from Joy, an occam-like language, to delay-insensitive circuits [WBB92, Bro91]. Most of the correctness proof involved showing that the output of the compiler – a netlist of components communicating via two-phase handshake protocols – is behaviorally equivalent to the original program. Showing that each of the handshake components was correctly implemented in terms of primitive circuit elements was done using automatic tools.

In this paper, we show how to use our specifications of the two-phase handshake components targeted by our compiler to generate a set of equivalent specifications for components which use four-phase handshake protocols. We compose our two-phase specifications with *protocol converters* to obtain specifications for the equivalent four-phase components. We then check that our four-phase implementations satisfy their specifications. In practice there are many choices in the design of the protocol converters which lead to significantly different four-phase implementations of our Joy compiler. Rather than consider all the options for converting our compiler, we take as our challenge the task of developing a set of protocol converters which yield a four-phase compiler that is roughly equivalent to one developed by Burns and Martin [BM88]. The principal differences are due to differences in the source languages.

Two automated tools were used to obtain the four-phase specifications and validate their implementations. The first, developed by Dill [Dil89], is purpose-built for verifying asynchronous circuits. The second tool, known as FDR, was developed by Formal Systems (Europe) Limited [For93] and checks refinement in the failures-divergence model of CSP. FDR is able to detect certain types of misbehavior (deadlock and livelock) which are not detected by Dill's tool, and facilitates the link with other CSP-based proofs for our compiler.

Proving a hardware compilation scheme correct is a substantial undertaking. Our approach is attractive because it allows an existing correctness proof (for compilation to two-phase circuits) to be reused for the four-phase compiler. Most of the additional proofs required in retargeting the compiler can be carried out by the automatic tools. Since the protocol converters that we use are implementable, our work also facilitates the verification of systems containing both two-phase and four-phase circuits which interact with one another through protocol converters.

Previously, Smith and Zwarico have verified the compiler of Burns and Martin for a subset of the input language [SZ92]. Their proof was performed by verifying a set of refinement rules from the input language to target circuits. Our technique is somewhat different – we verify the compilation algorithm to an abstract intermediate form and then hardware implementations of that intermediate form. van Berkel has developed and verified a compiler from a similar language to a similar intermediate form [vB92]. The principal difference between the source languages is that his does

*School of Electrical Engineering, Cornell University, Ithaca, NY, USA 14853; gbrown@ee.cornell.edu

†Programming Research Group, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, England OX1 3QD; Wayne.Luk@prg.oxford.ac.uk

‡School of Electrical Engineering, Cornell University, Ithaca, NY, USA 14853; jwo@ee.cornell.edu

not support guarded choice.

The remainder of this paper is organized as follows. Section 2 summarizes the source language for our compiler. Section 3 considers some issues which arise in the development of a suitable set of protocol converters. We illustrate our technique by generating a four-phase specification and implementation of a sequential composition element. Because space is limited, we do not describe the conversion of every element; a complete account is given in [BLO94]. Section 4 is devoted to an extended example: the specification and implementation of a module realizing channel communication. Channels proved to be the most difficult part of the retargeting effort, chiefly because our two-phase specification is delay-insensitive, while Martin gives a highly optimized four-phase implementation which is speed-independent but not delay-insensitive. We present some concluding remarks and suggestions for further work in Section 5.

2 Joy and Its Compiler

We consider, as a source language, a variant of occam which we refer to as Joy. We assume a set *VID* of variable identifiers and a set *CID* of channel identifiers, and let $v \in \mathit{VID}$, $c \in \mathit{CID}$, $C \subseteq \mathit{CID}$. Joy then has the following syntax:

(boolean expression)
$B \ ::= \ \mathbf{false} \mid \mathbf{true} \mid v \mid \neg B \mid B \wedge B \mid B \vee B$
(process)
$P \ ::= \ \mathbf{skip} \mid \mathbf{stop} \mid v := B \mid c! \mid P;P \mid P \parallel_C P \mid \mathbf{if}\, G\, \mathbf{fi} \mid \mathbf{do}\, BG\, \mathbf{od}$
(guarded process)
$G \ ::= \ B \rightarrow P \mid B \,\&\, c? \rightarrow P \mid G \,[\!]\, G$
(Boolean guarded process)
$BG \ ::= \ B \rightarrow P \mid BG \,[\!]\, BG$

Boolean expressions are composed of the constants **false** and **true**, variable identifiers, and the negation, conjunction, and disjunction operators. A brief explanation of the process terms is as follows.

skip	terminates, leaving the values of variables unchanged.
stop	deadlocks.
$v := B$	assigns the value of expression B to variable v. We stipulate that B may not contain a reference to v.
$c!$	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.
$P_1;P_2$	the sequential composition of P_1 and P_2.
$P_1 \parallel_C P_2$	the parallel composition of P_1 and P_2, with communications between them along the channels in C; these communications are concealed.
if G **fi**	attempts to execute the guarded process G until it succeeds;
do BG **od**	repeatedly executes the boolean guarded process BG until it fails.

There are two composite guarded command statements – **if** and **do**. **if** corresponds to occam's **PRI-ALT** – the command repeatedly attempts to execute its guarded command set until such an attempt succeeds. The primitive guarded process $B \rightarrow P$ is executable when the boolean guard expression B is true; its execution succeeds when P completes execution. Execution of $B \rightarrow P$ (read as "B then P") fails when B is false. The behavior of $B \,\&\, c? \rightarrow P$ is similar; $c?$ succeeds if another process executes the matching communication command $c!$. We require that $c?$ be evaluated only if B's value is true. 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 G_2 be executed only if execution of G_1 fails. The composite process succeeds if either G_1 or G_2 succeeds, and fails if G_1, G_2 both fail. **do** repeatedly evaluates its boolean guarded command set until execution fails.

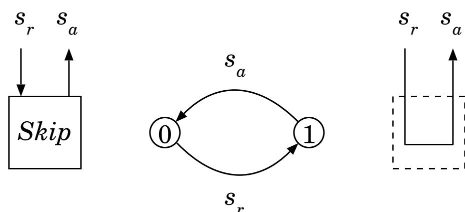
Since evaluation of $c?$ as part of a guard expression potentially has a side effect – completion of synchronization – we allow, at most, one receive per guard expression. Other syntactic restrictions include the condition that input and output channels used by concurrent processes must be disjoint, and that parallel processes may not share any variables. For simplicity only boolean operators are given in the above syntax; it is possible to extend Joy to include, for instance, integer operations.

Our Joy compiler was inspired by the work of van Berkel and others at Philips Research [NvBRS88], by Martin [Mar86], and by Brunvand and Sproull [BS89]. Each syntactic unit of the source language is mapped to a corresponding hardware component; these components communicate using two-phase handshake protocols. Each basic module has one or more *ports*, or bundles of signals. Some signals in each bundle trigger the execution of the module or its neighbors; other signals indicate the completion of execution.

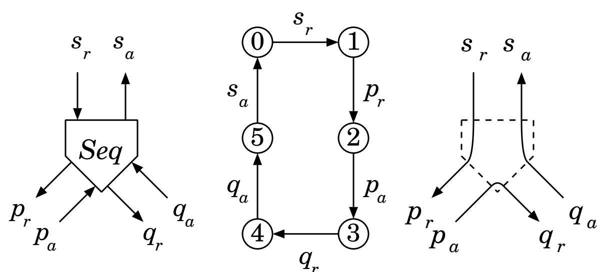
The simplest handshake interface is the *control* interface, consisting of request and acknowledgement

signals. On a port p we denote the request and acknowledgement signals p_r and p_a, by convention. Execution of a process P with port p is triggered by receipt of an event on p_r; P produces an output event on p_a to notify the requester that execution has completed. The physical interpretation of events is as voltage transitions (either low to high or high to low).

For example, a module implementing the **skip** process has a single control interface. It does nothing but issue an acknowledgement s_a after receiving a request s_r, and is specified by the following state machine. If two-phase signalling were used, *Skip* could be realized by a single wire.



As another example, consider the module implementing sequential composition – P; Q – which has three control interfaces. Once triggered by an event on s_r, *Seq* starts the first process by issuing a request on p_r. When the first process signals completion by an event on p_a, the second process is started by an event on q_r. Completion of the second process (q_a) leads to termination of the sequence (s_a). A two-phase version of *Seq* can be implemented using three wires.

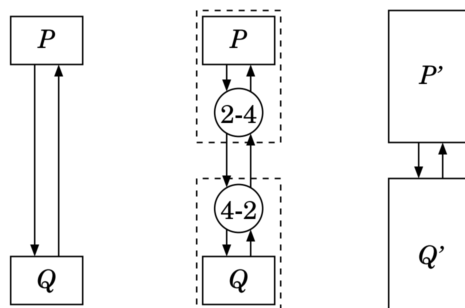


In addition to this simple control interface, more complex protocols are needed for passing data among handshake components. Boolean data are encoded on two wires, conventionally denoted b_0 and b_1 for a port b. The read interface is used in expression and guard evaluation. For an expression B with port b, evaluation is triggered by receipt of an event on b_r from A, and terminates with an event on either b_0 or b_1 depending upon the value of the expression. There is also a write interface used to assign values to variables: assignment to variable V begins with receipt of an event on either v_0 or v_1 from U; completion is signaled by an event on v_a.

3 From Two-phase to Four-phase Specifications

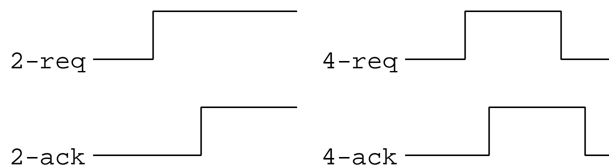
In this section we show how to use our specifications for two-phase modules to automatically generate specifications for "equivalent" four-phase modules in a manner which guarantees that our correctness proof for the two-phase compiler is valid for the four-phase compiler. Our technique is to develop a set of converters which translate between two-phase and four-phase handshake protocols. The specification for a four-phase component is generated by adding protocol converters to all the handshake interfaces of the corresponding two-phase component. The protocol converters have the property that a matched pair of them ($2 \rightarrow 4$, $4 \rightarrow 2$) behave like a bundle of wires, provided that the components connected by the converters obey the handshake protocol. By definition, the behavior of a delay-insensitive circuit is not changed by introducing additional wires (delays) into the circuit. Thus, a circuit created by combining four-phase components is equivalent to one created from two-phase components with converters added to each handshake port, and thus is equivalent to a circuit composed of just the two-phase components.

The figure below illustrates our method. Suppose P and Q specify delay-insensitive elements communicating over a pair of wires using a two-phase handshake protocol. We introduce the protocol converters $2-4$ and $4-2$ — these converters communicate with each other using the four-phase protocol, but the pair of them behave like a bundle of wires and are therefore transparent to P and Q. The combined behavior of P and $2-4$ is P', the four-phase counterpart of P; similarly, Q and $4-2$ combine to yield Q'.



The design of suitable converters is complicated by the additional freedom allowed by the four-phase protocol which is not present in the two-phase protocol. Consider the waveforms associated with two-

phase and four-phase handshaking over a simple two-wire interface.



We say that the two-phase protocol is *idle* before the `req` transition and after the subsequent `ack` transition. A handshake is *initiated* when the `req` transition occurs and is *completed* when the subsequent `ack` transition occurs. For the four-phase protocol, the terms *initiated* and *idle* are similar, but when is the handshake *completed*?

One answer would be after the second `ack` transition. This conservative interpretation has the advantage that it makes it easy to detect when a circuit consisting of many communicating components is idle. This is important when accesses over multiple interfaces to a component (e.g. a channel) must be mutually exclusive. We have chosen this interpretation for control interfaces as it is consistent with Burns' and Martin's compiler.

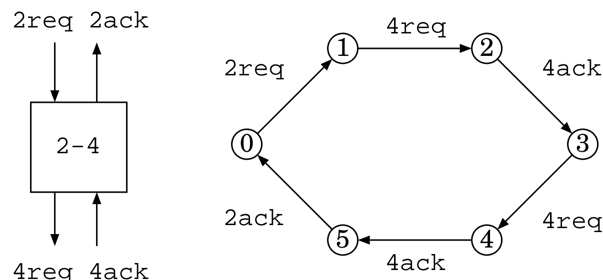
Another answer would be to say the communication is *completed* after the first `ack` (it is clear that after the first `ack` the two partners have agreed to synchronize) but that there is some additional work which must be done to return the interface to an *idle* state. This interpretation is useful for value-passing interfaces – it is desirable to use the value returned as soon as it is available. We refer to interfaces of this type as *quick return* interfaces, and we choose this interpretation for most read interfaces and all write interfaces, again for consistency with Burns' and Martin's compiler. As we shall see in section 4, the quick return interface is not appropriate in evaluating guarded commands: an extra type of read interface is required which adopts the conservative interpretation for "true" acknowledgements and the quick return interpretation for "false" acknowledgements.

In discussing the various converters we use the following naming convention. Protocol converters have two ports: one port obeys a two-phase protocol and one obeys a four-phase protocol. Additionally, one of the ports has its request line as an input (we call this a *passive* port) and one port has its request line as an output (we call this an *active* port). In naming protocol converters, the type of the passive port will be listed first and the type of the active port second. For example, `2-4` has a (two-wire) passive two-phase port and an active four-phase port. For value passing converters we will indicate the source of the value by adding the symbol `val` to either the passive or active port as appropriate. For example, `2val-4` is a converter in which the value to be passed is supplied at the passive two-phase port. The six basic types of converters are thus `2-4, 4-2, 2val-4, 4val-2, 2-4val,` and `4-2val`. Later, we will add `2-4val-qr0`, a converter with quick return in the case of "false" acknowledgement only.

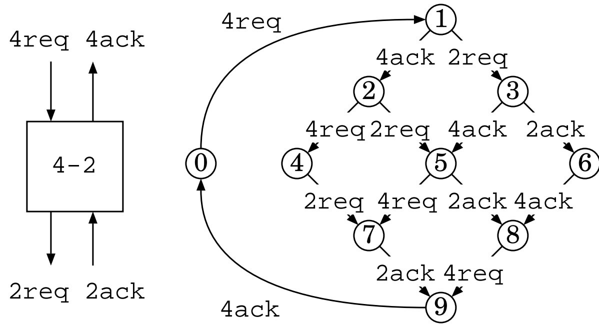
A guiding principle has been to restrict the specifications of protocol converters as little as possible. Since we are using converters to generate specifications for four-phase components, using less restrictive protocol converters leads to less restrictive four-phase specifications, and therefore allows more freedom in choosing a four-phase implementation.

3.1 Two-wire Converters

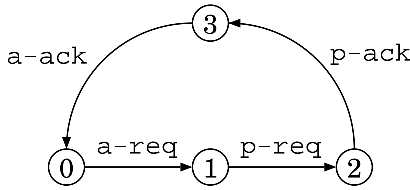
We begin by discussing the two-wire converters `2-4` and `4-2`. For the two-wire four-phase protocol, recall that the handshake is *completed* only when the second `ack` has occurred. The converter `2-4` accepts a transition on input `2req`, completes a four-phase handshake on output `4req` and input `4ack`, and produces a transition on output `2ack`.



The `2-4` converter is the simplest which we shall present. The design of the `4-2` converter is somewhat more subtle. A handshake is initiated by a transition on `4req`. Since a full four-phase handshake requires two transitions on `4req` and `4ack`, the converter may respond with `4ack` before it has even initiated a transition on `2req`! This is an instance of our principle that converters should not be unduly restrictive: given our definition for "completed", we need only require that the initiating (completing) transition of the four-phase handshake precede (follow) that of the two-phase handshake. Note also that in the state diagram for the `4-2` converter, the transitions on the leading diagonals are identical (either `2req` or `2ack`), and similarly for the transitions on the trailing diagonals (either `4req` or `4ack`). This symmetry helps in understanding how the converter works.



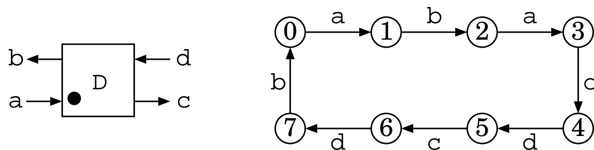
In order to ensure that the use of converters does not affect the behavior of the system, we must verify that the behavior of two converters connected back to back with wires between them is not detectable to the communicating two-phase components. In particular, we must check that the connection `2-4, 4-2` behaves correctly. The strongest possible requirement is for this pair of components to behave like a pair of wires. Fortunately, a weaker requirement suffices. Since we are restricting connected components to obeying the handshake protocol, we need only require that a pair of converters connecting an active port `a` to a passive port `p` have the following behavior:



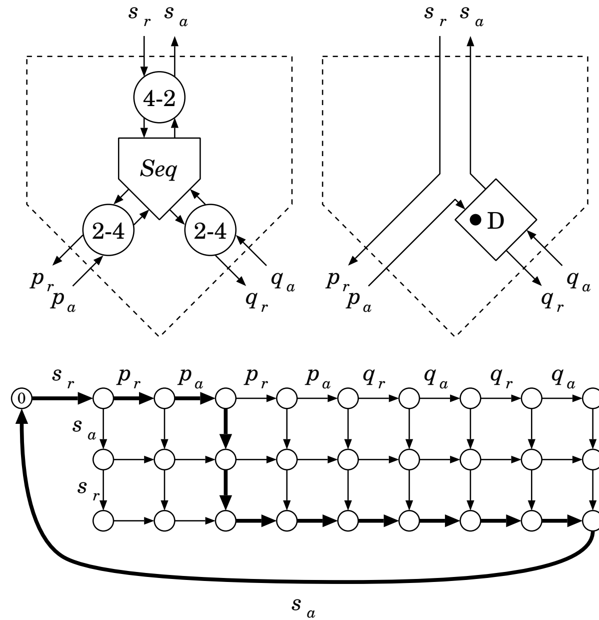
We were able to show this using the FDR tool and Dill's tool.

3.2 Sequential Composition

We now derive a four-phase specification and implementation for one of our simpler components, which implements sequential composition. In developing our four-phase implementation, we make use of the D-element to enforce sequencing [Mar86]. The D-element has the following state table:



The four-phase specification is obtained by composing the two-phase specification (*Seq* below) with converters for four-phase handshake protocols. The state diagram of the composite is also shown.



As used above, the D-element enforces that the sequential composition element performs the following sequence of actions:

$$s_r, p_r, p_a, s_a, s_r, p_r, p_a, q_r, q_a, q_r, q_a, s_a, \ldots$$

These transitions appear as darker arrows in the state diagram of the four-phase sequential composition element. This component is identical to that used in Martin's compiler [Mar86]. In contrast, van Berkel [vB92] implements sequential composition with an element performing the sequence

$$s_r, p_r, p_a, p_r, p_a, q_r, q_a, s_a, s_r, q_r, q_a, s_a, \ldots$$

Both groups have recognized and exploited the fact that the only requirements of the sequential composition component are that the first transition of a cycle be s_r, the last transition be s_a, and that the handshake with P be completed before the handshake with Q is begun. As the diagram shows, the four-phase specification we generated is sufficiently general that either Martin's or van Berkel's implementation would be permitted.

4 Channels

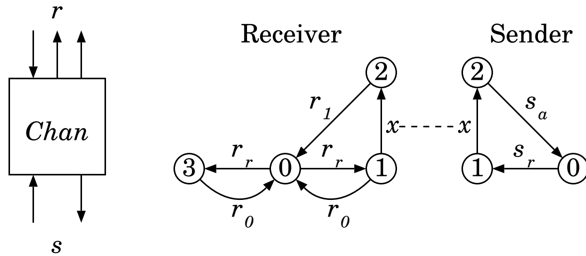
Processes executing in parallel can interact by synchronized communication over named channels, which our compiler implements by handshaking. Our two-phase compiler generates circuits which are "truly" delay-insensitive. By this we mean that we build our circuits using a small set of primitive delay insensitive components, and make no assumptions about wire

delays. While it is possible to implement four-phase channels in a truly delay-insensitive fashion, here we present a *speed-independent* solution which is similar to that used by Burns and Martin.

We shall begin by explaining the channels and call-modules used by our two-phase compiler. Next, we will introduce the protocol converters needed to obtain the four-phase specification. These converters incorporate assumptions about the environment in which four-phase implementations may be placed, and allow us to perform aggressive optimization. Finally, we shall show how to derive a four-phase, multi-receiver, multi-sender channel which is similar to that generated by Burns and Martin's compiler.

4.1 Two-Phase Channels

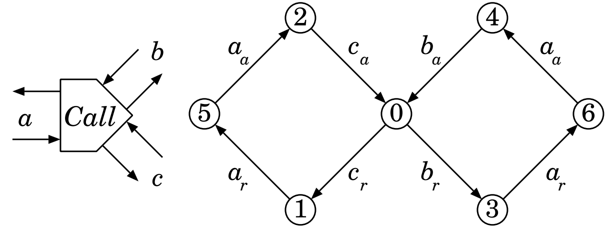
Communication channels have a control interface at the sending end and a read interface at the receiving end. A sender, in executing $c!$, requests synchronization by generating an event on s_r and awaits an event on s_a signaling that synchronization has taken place. A receiver, executing $c?$, tests a sender's readiness to communicate by generating an event on r_r. An acknowledgement on r_1 means that synchronization *has occurred*; otherwise an acknowledgement on r_0 is returned. The sending and receiving ends of the channel synchronize on the internal event x.



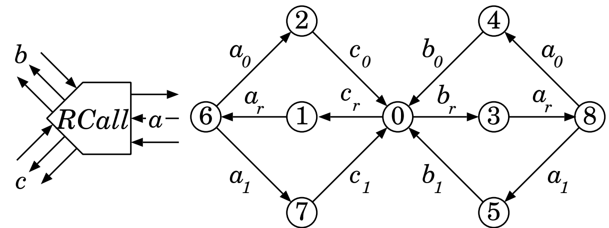
A noteworthy feature of this specification is that the channel may ignore the sender's request for synchronization – after r_r, the receiver may be in state 1 and be willing to cooperate with the sender on x, or in state 3 and refuse x. This is necessary for the specification to be delay-insensitive: a delay-insensitive circuit which has its interface buffered (delaying the arrival of input and output events) must be indistinguishable from the unbuffered circuit. As a further consequence of delay insensitivity, new synchronization requests on r_r may be received by the channel before completion of a previous handshake with the sender, and vice versa. Of course a satisfactory implementation of the channel should be fair, in the sense that if the sender is continually ready to perform x and the receiver is infinitely often in states 1 or 3, then x will eventually be taken. We will not consider fairness in this paper.

Our model of a channel is slightly different from that of Burns and Martin since we do not separate the "probe"[1] from synchronization. This is equivalent to performing the probe and immediately following the probe by a synchronization request. Our compiler does not support value passing channels; however, this can be simulated by using a separate synchronization channel for value. The circuits used by Burns and Martin are (roughly) implemented in this way; however, they allow a single synchronization event to pass values in both directions simultaneously.

It may be the case that synchronization on a channel is required at several different points in the program. Consider the simple program "$c!$; $c!$": since the send interface to channels is delay-insensitive, the acknowledgements must be properly routed back to the requester. The send interface of a channel is shared among multiple senders by means of the *Call* module:



Similarly, the receive interface of a channel may be shared among multiple receivers using the *RCall* module:



Further senders and receivers can be served by having a number of *Call* or *RCall* modules connected together in the form of a tree.

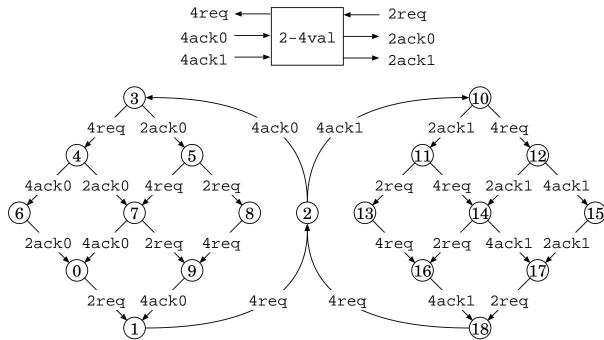
4.2 Protocol Converters

In this section we present a set of three-wire converters for read interfaces. The two-wire converters discussed in section 3.1 were quite conservative – the four-phase handshake was *completed* only after the second `ack` transition. However, with value passing

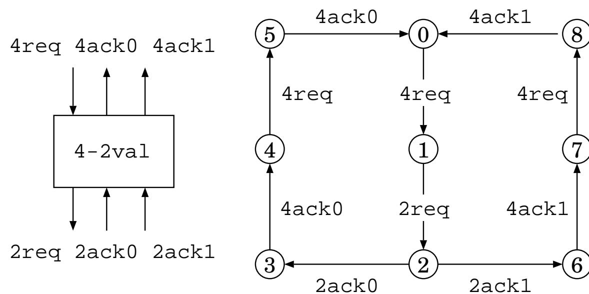
[1] The probe is an operation developed by Martin which allows a process to test whether a matching process is ready to communicate

interfaces it is convenient to be able to use a value as soon as it is available. Thus, for the read interface, one of the converters is of the quick return type: the two-phase handshake is *completed* immediately after the value is passed via the first acknowledgement, even if the handshake interface has not returned to the idle state.

The **2-4val** converter receives a request from the two-phase component, forwards it to the four-phase component, waits for **4-ack0** or **4-ack1**, returns **2-ack0** or **2-ack1** as appropriate, and then completes the four-phase handshake. Notice that any subsequent request on the two-phase port is held until the four-phase handshake is complete.



The **4-2val** component is unlike the **4-2** component in that it cannot return an acknowledgement to the four-phase interface until the handshake with the two-phase interface is complete. This is because it must wait to receive either **2-ack0** or **2-ack1** in order to choose between **4-ack0** and **4-ack1**.

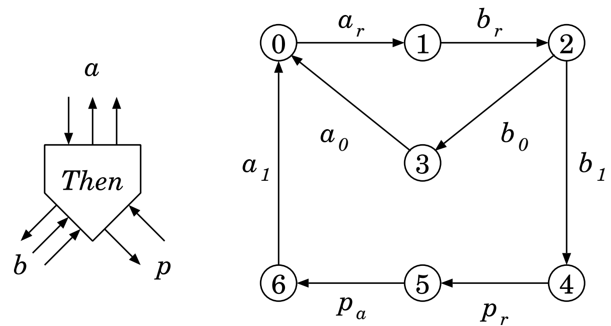


The pair of converters **2-4val** and **4-2val** connected back-to-back are transparent, provided they are inserted between components obeying the two-phase handshake protocol.

A Special Case

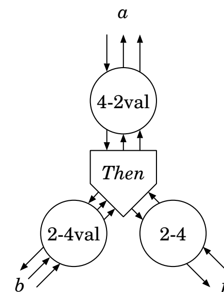
Our two-phase compiler generates shared resources, such as variables and channels, by building trees of "call-modules" which are essentially delay-insensitive multiplexors and de-multiplexors. In contrast, Burns and Martin implement variables and channels which may have an arbitrary number of ports. To obtain efficient implementations, however, they make compromises to delay-insensitivity by using large *isochronic* regions – regions in which wire delays are assumed to be negligible – and assuming that certain mutual exclusion properties hold. If a write port of a variable is active, all other ports must be idle; for channels, at most one send and one receive port may be active at any time.

The protocol converters presented in the previous sections are sufficient to derive most of the four-phase building blocks necessary to transform our two-phase compiler. For the components which perform guard evaluation, unfortunately, we need an additional converter to ensure that the mutual exclusion properties are not violated. To understand where the difficulty arises, consider the component which corresponds to $\rightarrow$ (as in $B \rightarrow P$).



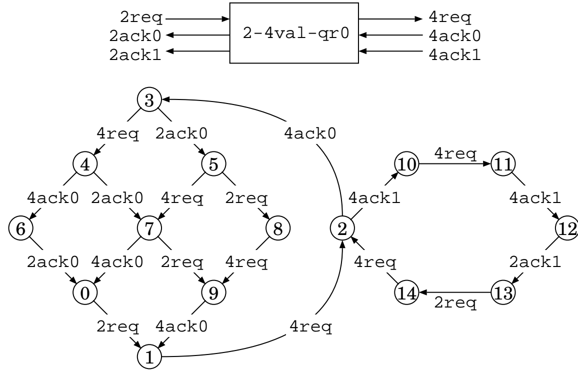
Then operates as follows. When a request a_r is received, *Then* requests the evaluation of B by issuing b_r. The result of the evaluation determines whether the process P will be executed. If b_0 is returned, the guard evaluation has "failed" and *Then* returns a_0 to its caller. If b_1 is returned, the guard evaluation has "succeeded" and *Then* initiates execution of the process P by issuing p_r. *Then* returns a_1 to its caller after P has signalled completion by p_a.

In an attempt to obtain a four-phase specification of *Then*, we might attach protocol converters:

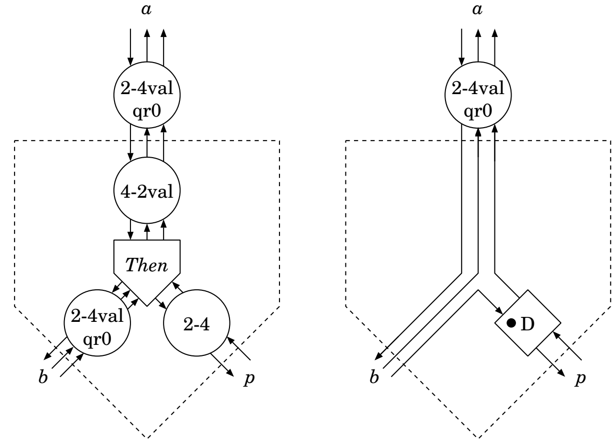


Suppose the handshake proceeds as follows. A request is made at the port a for the guarded command $B \rightarrow P$ to be executed. A request is issued at b for evaluation of the guard expression B, and a "true" acknowledgement is returned. Because **`2-4val`** is a quick return converter, the two-phase *Then* receives a "true" acknowledgement *before the four-phase handshake at b is completed*. *Then* is then able to initiate execution of P, leading to a problem: P might assign to a variable present in the guard expression B, but the read port of that variable may not be idle. This violates the mutual exclusion property that Burns' and Martin's compiler requires.

The solution to this problem is to introduce an alternative converter for read interfaces – **`2-4val-qr0`** – which uses a mixed interpretation of completion. A "false" response means that the evaluation of the guard is complete and hence evaluation of the guarded process is complete; a "true" response only means that the evaluation of the guard is complete, not that the corresponding process has executed. The simple converters which we have presented do not admit this distinction. The new converter **`2-4val-qr0`** is similar to the **`2-4val`** component presented in the previous section, except that it allows quick return only in the case in which a "false" is returned. The two converters **`4-2val`** and **`2-4val-qr0`** are transparent when connected back to back.



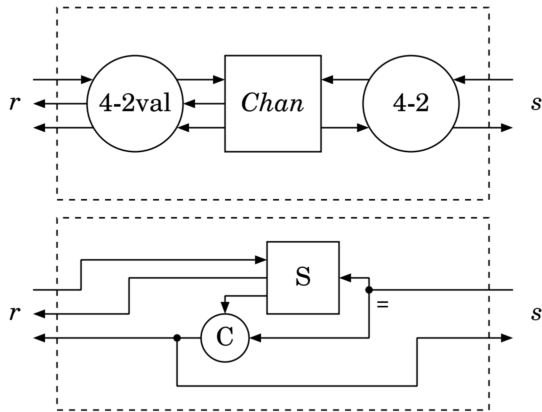
When we derive the four-phase implementation of guarded commands we also use this new converter to introduce contextual information to our proofs. By considering how a component's environment interprets events at its ports (quick-return "false", conservative "true" in the case of the four-phase *Then*) we are able to obtain an optimized implementation. The specification and implementation of our four-phase *Then* are shown below. The D-element is used to complete the handshake with the guard when the guard is true (compare with the implementation of *Seq*). Note that the **`2-4val-qr0`** converter is only required for "context"; it does not appear in the implementation.



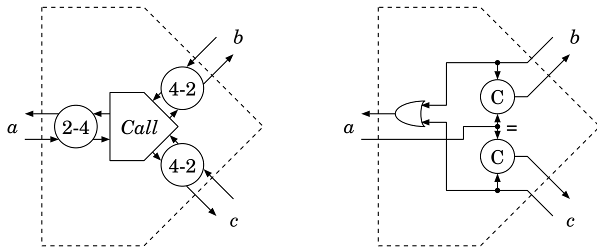
4.3 Four-Phase Channels

We begin by presenting the basic channel module for a single sender and a single receiver. We then derive a speed-independent multi-sender, multi-receiver channel using the *Call* and *RCall* modules specified for the two-phase compiler.

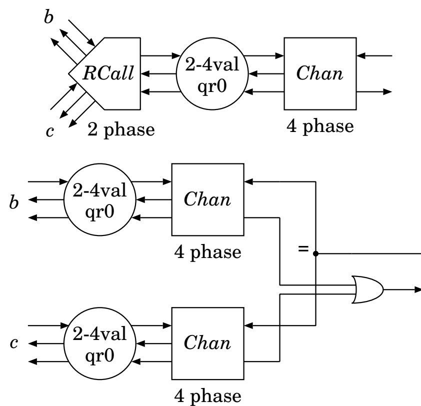
Recall that the *Chan* module has a read interface for the receiver and a control interface for the sender. To derive the specification of the four-phase channel, we place one of our three-wire converters, **`4-2val`**, at the receiver's interface of the two-phase specification, and a **`4-2`** converter at the sender's interface (for brevity, we do not show the state diagram of the four-phase specification). The four-phase implementation uses a Muller C-element and a *synchronizer* [Mar86]. The synchronizer has a one-wire input (to be synchronized) and a read interface. When it receives a request on its read interface the synchronizer examines its input; some time later it issues either a "true" or "false" acknowledgement depending upon the value it saw there. The fork marked "=" signifies an *isochronic* region, in which events are assumed to occur at all points simultaneously. The presence of the isochronic region distinguishes this implementation as speed-independent, rather than delay-insensitive.



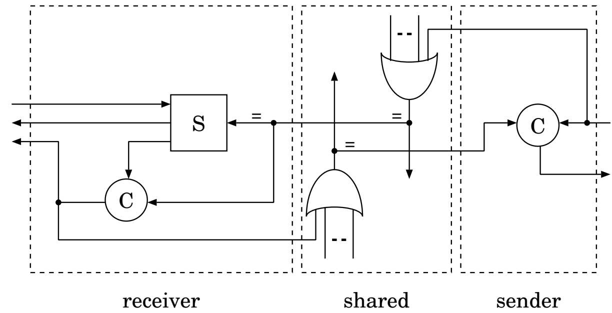
We can derive a two-sender channel by considering the development of a speed-independent *Call* module. We obtain the four-phase specification by using the appropriate protocol converters. Note that the four-phase implementation contains an isochronic region.



Similarly, we can derive a two-receiver channel by considering the *RCall* module. In this case, however, it turns out to be efficient to implement the combination of a *RCall* and a *Chan* by duplicating the *Chan* module. As with the guarded commands we then need to consider some information about the context of the channel. The combination of a four-phase channel and a *RCall* module can be implemented as follows:



We obtain a multi-sender, multi-receiver channel by combining and generalizing the two-sender and two-receiver channels. The channel, shown below, consists of three parts. Each receiver accessing the channel duplicates the hardware of a single-sender, single-receiver channel. Each sender's hardware consists of a `c-element`; the sender blocks until synchronization has occurred. Finally, the shared portion of the channel is implemented using distributed or-gates.



The channel is speed-independent in the sense that all inputs from the channel to the receivers and senders are assumed to form an isochronic region. An additional proof is needed to justify compressing the senders' call tree into a large isochronic region and one `c-element` per sender. This represents a design tradeoff between safety (relatively small isochronic regions) and efficiency.

5 Discussion

We have described and illustrated a simple method for developing and verifying alternative implementations for a hardware compiler. This approach facilitates the re-use of existing proofs, and many of the new proofs can be automated. Our method consists of three steps. First, we develop protocol converters for transforming between the protocols used in the existing and in the new compilation schemes. In the case of our Joy compiler, seven protocol converters were required. The second step in our method is to generate the specifications of the basic modules for the new compilation scheme from the old ones using protocol converters; finally, we show that the implementations of the new compilation scheme satisfy their specifications. The final two steps are performed by automated tools.

In developing our four-phase modules, we assumed certain mutual exclusion properties on access to variables and channels. The final "loose end" in our proof is to prove our implementation has these properties. We can define predicates on the four-phase modules which capture the state of their handshake interfaces (e.g. *idle*) and formulate the required properties as invariants; these can be proved easily by induction on

the structure of programs. The proof was done by hand in [BLO94], but could be partially automated using model-checking techniques.

There are several possibilities for further work. It would be desirable to extend our derivation to include other asynchronous design styles, such as the bundled data approach [BS89], or to include clocked implementations [PL91]. We believe that our approach would simplify the verification of hardware compilers for occam-like languages, and would provide a mathematical basis for developing implementations consisting of a number of synchronization methods, such as those proposed by Seitz [Sei80]. If this venture is successful, then the next question could be: how to characterize efficient designs adopting mixed synchronization protocols?

Acknowledgements

Thanks to Mark Josephs, He Jifeng, and Tony Hoare for their comments and suggestions. 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. 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. This work was completed while the first author was a SERC Visiting Fellow at Oxford University Computing Laboratory.

References

[BLO94] Geoffrey Brown, Wayne Luk, and John O'Leary. Retargeting a hardware compiler using protocol converters. Technical Report EE-CEG-94-1, School of Electrical Engineering, Cornell University, February 1994. Submitted to *Formal Aspects of Computing*.

[BM88] Steven M. Burns and Alain J. Martin. Synthesis of self-timed circuits by program transformation. In *Advanced Research in VLSI: Proceedings of the 5th MIT Conference*, pages 35–50, 1988.

[Bro91] Geoffrey M. Brown. Towards truly delay-insensitive circuit realizations of process algebras. In Geraint Jones and Mary Sheeran, editors, *Designing Correct Circuits*, pages 120–131. Springer-Verlag, 1991.

[BS89] Eric Brunvand and Robert F. Sproull. Translating concurrent communicating programs into delay-insensitive circuits. Technical Report CMU-CS-89-126, Carnegie-Mellon University, 1989.

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

[For93] Formal Systems (Europe) Limited. *FDR user manual and tutorial*, 1993.

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

[NvBRS88] Cees Niessen, C. H. van Berkel, Martin Rem, and Ronald W. J. J. Saeijs. VLSI programming and silicon compilation; a novel approach from Philips Research. In *International Conference on Computer Design*, pages 150–151. IEEE, October 1988.

[PL91] Ian Page and Wayne Luk. Compiling occam into FPGAs. In W. Moore and W. Luk, editors, *FPGAs*, pages 271–283, Abingdon, England, 1991. Abingdon EE&CS Books.

[Sei80] Charles L. Seitz. System timing. In Carver Mead and Lynn Conway, *Introduction to VLSI Systems*, pages 218–262. Addison-Wesley, 1980.

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

[vB92] Kees van Berkel. *Handshake Circuits: an Intermediary Between Communicating Processes and VLSI*. PhD thesis, Eindhoven University of Technology, 1992.

[WBB92] Sam Weber, Bard Bloom, and Geoffrey M. Brown. Compiling Joy into silicon. In *Advanced Research in VLSI and Parallel Systems: Proceedings of the 1992 Brown/MIT Conference*, pages 79–98. MIT Press, 1992.