ELECTRONIC **WORKSHOPS IN COMPUTING**

Series edited by Professor C.J. van Rijsbergen

Mary Sheeran, Chalmers Technical University, Sweden, and Satnam Singh, University of Glasgow, UK (Eds)

Designing Correct Circuits

Proceedings of the 3rd Workshop on Designing Correct Circuits (DCC96), Båstad, Sweden, 2-4 September 1996

ISBN: 3-540-76102-0

Paper:

Deriving Two-Phase Modules for a Multi-Target Hardware Compiler

J. He, G. Brown, W. Luk and J. O'Leary

Published in collaboration with the
British Computer Society

BCS

Springer

Deriving Two-Phase Modules for a Multi-Target Hardware Compiler

Jifeng He
Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford OX1 3QD, UK

Geoffrey Brown
School of Electrical Engineering, Cornell University
Ithaca, NY 14853, USA

Wayne Luk
Department of Computing, Imperial College
180 Queen's Gate, London SW7 2BZ, UK

John O'Leary
Microprocessor Products Group, Intel Corporation
2111 NE 25th Avenue, Mail Stop JF1-81, Hillsboro, Oregon 97124-5961, USA

Abstract

This paper adopts the CSP framework for deriving a compilation scheme from a simple imperative language to two-phase modules. Two-phase modules are processes that communicate with one another using two-phase handshake protocols. The two-phase modules generated by our compilation scheme can be implemented as asynchronous or clocked circuits. The derivation techniques have been applied to a concurrent language which is a superset of the language discussed.

1 Introduction

A verified compiler is essential to a structured approach for verifying designs: if its users are confident of the correctness of the translation from the source language to the target description, they can focus on getting the source program right. The verification of the compiler should itself be structured to make it simple, modular and flexible.

We have been investigating a method for verifying a compilation scheme for occam-like languages that targets both asynchronous [2] and clocked [8] hardware. The method involves two steps; the first is to translate the constructs of the source language systematically into an intermediate form known as two-phase modules, which interact with one another using simple two-phase protocols. Figure 1 shows the hardware modules implementing the following program:

var x:
do
  true $\rightarrow$ $x := \neg x$
od

The second step of our method is to use appropriate protocol converters to derive the implementation of the two-phase modules in a particular target technology. The second step has been described elsewhere [3]; the purpose of this paper is to describe the first step.

The verification of asynchronous and clocked realisations of occam-like languages has been undertaken by a number of researchers, including He *et. al.* [5], Smith and Zwarico [11], van Berkel [12], and Weber *et. al.* [14]. There are three principal differences between our work and theirs. First, our approach provides a means of capturing and reasoning about asynchronous and clocked systems within a unified framework; it can also deal with systems containing both asynchronous and clocked elements. Second, our verification strategy is structured into a number of stages to improve modularity and reusability of proofs. Finally, many of our proofs can be automated; we shall identify some of them later.

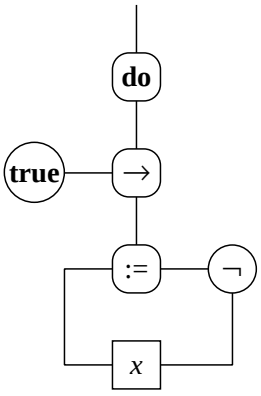


Figure 1: Two-phase modules for a simple program. Note that each connection actually contains several wires: the assignment module for instance, labelled ':=' above, has a top 2-wire control interface, a right 3-wire read interface and a left 3-wire write interface. The definition of these interfaces can be found in Figure 2.

Our derivation of the two-phase modules includes several novelties. The use of the CSP framework [6, 10] makes our descriptions concise and simplifies our presentation. Using the CSP algebra, we can systematically refine source programs into a collection of interacting two-phase modules; program variables are modelled by data modules, while control constructs result in control modules. We also exploit the regularity of the handshake protocols which govern the interaction between these modules – this leads to a notion of refinement for such modules which greatly simplifies our derivation.

The rest of the paper is organised as follows. Section 2 provides an overview of our derivation strategy. Section 3 introduces the source language for our compiler and the two-phase handshake protocols adopted by the intermediate form. Section 4 reviews the algebra of CSP and shows how it can be used to capture handshake protocols. Sections 5, 6 and 7 are devoted to translating variables, expressions and statements from the source program into two-phase modules. Section 8 describes the specification of two-phase modules, and Section 9 outlines the derivation of implementations. Concluding remarks are presented in Section 10.

2 Overview of strategy

This section provides an overview of our derivation strategy for the sequential subset of our source language; the CSP notation used here will be summarised in Section 4. The network of two-phase modules implementing the source program P is specified as $\Psi^r_a(P)$, where the request signal r activates P and the acknowledgement signal a indicates that P has terminated. $\Psi^r_a(P)$ satisfies

$$\Psi^r_a(P) \quad = \quad r \rightarrow P \,;\, a \rightarrow \Psi^r_a(P)\,.$$

Thus $\Psi^r_a(P)$ can be activated multiple times, while P cannot. We define a compilation function to be the parallel composition of an activatable master control process $\Psi^r_a(\mathcal{M}(P))$ and a data process $\mathcal{D}(P)$:

$$C^r_a(P) \quad \stackrel{def}{=} \quad \Psi^r_a(\mathcal{M}(P)) \;\parallel\; \mathcal{D}(P)\,. \tag{1}$$

The master control process $\mathcal{M}(P)$ involves only synchronised communications with $\mathcal{D}(P)$, which maintains the program state. To avoid deadlock on the internal links between $\mathcal{M}(P)$ and $\mathcal{D}(P)$, we shall ensure that the communications on the added channels satisfy the related two-phase handshake protocols.

Our task is to verify that the compiled design is at least as good as its specification derived from the source program:

$$\Psi^r_a(P) \quad \sqsubseteq \quad C^r_a(P)\,.$$

The verification task can be broken down into two stages. In the first stage, we demonstrate the correct refinement of a program P into the master process $\mathcal{M}(P)$ and the data process $\mathcal{D}(P)$ executing in parallel:

$$P \quad \sqsubseteq \quad \mathcal{M}(P) \;\parallel\; \mathcal{D}(P)\,.$$

This result will be used in the second stage, the main challenge of which is to show that Ψ^r_a is a homomorphism. Taking sequential composition as an example, calling Ψ^r_a a homomorphism means that the activatable control of the composite can be implemented by those of its components together with the control module *SEQ*:

$$\Psi^r_a(\mathcal{M}(P;Q)) \quad \sqsubseteq \quad \Psi^{rp}_{ap}(\mathcal{M}(P)) \parallel \mathit{SEQ} \parallel \Psi^{rq}_{aq}(\mathcal{M}(Q))$$

(see Equation 15). We can then establish that

$$\Psi^r_a(P;Q) \quad \sqsubseteq \quad \Psi^{rp}_{ap}(\mathcal{M}(P)) \parallel \mathit{SEQ} \parallel \Psi^{rq}_{aq}(\mathcal{M}(Q)) \parallel \mathcal{D}(P;Q).$$

An outline of these two verification stages can be found in the appendix. Once we check that the specification of modules like *SEQ* satisfies the above formula, various asynchronous and clocked implementations can be developed using the technique of protocol conversion [3]. An example is included in Section 9.

3 Joy and two-phase modules

To simplify the presentation we shall focus on the sequential subset of Joy ([3], [9], [14]), our source language. Its syntax is given by the following BNF rules, where v stands for program variable of Boolean type, B for Boolean expression and P for process.

$$\begin{array}{rcl} B & ::= & \mathbf{false} \mid \mathbf{true} \mid v \mid B \vee B \mid B \wedge B \mid \neg B \\ P & ::= & \mathbf{skip} \mid v := B \mid \mathbf{if}\, BG\, \mathbf{fi} \mid \mathbf{do}\, BG\, \mathbf{od} \mid P;P \\ BG & ::= & B \rightarrow P \mid BG \,[\!]\, BG \end{array}$$

Note that this subset of Joy deals only with Boolean variables and expressions. A Joy program may include **skip**, which does nothing except terminating successfully, and assignment, conditional, iteration and sequential composition statements. The Boolean guarded process $B \rightarrow P$ is executed when the Boolean guard B is true; its execution completes when P completes execution. Two guarded processes may be composed using $[\!]$, the choice operator. The statement $\mathbf{if}\, BG\, \mathbf{fi}$ executes its boolean guarded command set until one succeeds; $\mathbf{do}$ repeatedly evaluates its boolean guarded command set until execution fails.

The translation of Joy into its intermediate form is performed in a purely syntax-directed manner. The networks that the Joy compiler generates are delay-insensitive in the following sense: wires with arbitrary (bounded) delay can be introduced between any two primitive components without affecting the functional behaviour of the system.

The idea of using modules communicating with handshake protocols to represent the main components of the source language has been explored by van Berkel [12]. Our work takes an 'indirect approach' in which our language is given a denotation in an existing notation (CSP) and a mapping from CSP to two-phase modules. In contrast, van Berkel takes a direct approach in which each command of the source language is defined by a corresponding handshake process. Furthermore, van Berkel focused on a trace-based model, while CSP provides a more sophisticated failures/divergences model and a rich algebraic system.

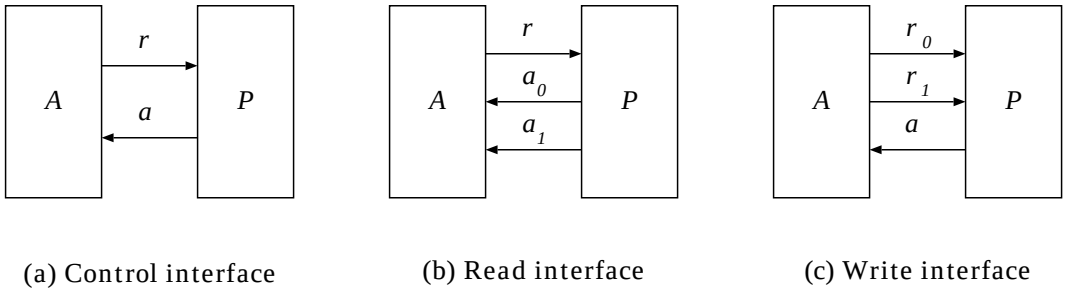
The signaling interfaces used by the two-phase components are shown in Figure 2. Each handshake interface requires an active partner (marked *A* in the figure) that begins the handshake by sending a request, and a passive partner (marked *P*) that responds to the handshake by sending an acknowledgement.

The simplest interface, shown in Figure 2(a), is the two-wire control interface, which consists of one request and one acknowledgement signal. A handshake begins when *A* sends an event to *P* along the wire marked r (for request). *A* then awaits a response on the wire marked a (for acknowledgement). When *A* has received an acknowledgement from *P*, the handshake is complete.

Figure 3 shows a module that composes two programs in sequence and has both passive and active interfaces. When triggered through the top input `req` port, the *SEQ* module starts the first program in the sequence by dispatching the event through the bottom left output *P0.req*. When the first program signals termination via the event *P0.ack*, the second program is started by a *P1.req* output. *SEQ* returns *ack* after it receives *P1.ack*.

Some interfaces pass data as well as control information. Figure 2(b) shows two interfaces for passing Boolean data encoded on two wires, sometimes known as dual-rail encoding. The read interface is used in expression and guard evaluation. The active partner in the handshake requests a value by sending an event on r; the passive partner sends an acknowledgement on a_0 or a_1, according to the value it wishes to return.

The three-wire interface in Figure 2(c) is called the write interface, and is used to assign values to variables. Writing a value begins with a request on either r_0 or r_1, depending upon the value to be written; completion is signalled by an acknowledgement event on a.



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

Figure 2: Handshake interfaces.

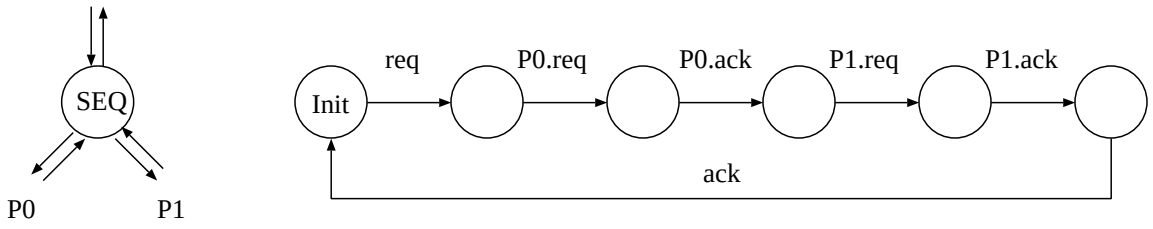


Figure 3: Two-phase module for sequential composition.

4 CSP

We shall regard the Joy language as a subset of CSP. To describe the operation of the target architecture, we need to include a few more operators. $P \sqcap Q$ represents the nondeterministic choice between P and Q in which the environment plays no part, while $P \,[\!]\, Q$ stands for the external choice between P and Q where the environment decides which branch is selected to execute. The pattern **if** $B \rightarrow P \,[\!]\, \neg B \rightarrow Q$ **fi** will be abbreviated as $P \lhd B \rhd Q$. The alphabet of P, αP, corresponds to the set of events that the process P can engage in. **var** $w : P$ declares w to be a local variable of P. $\perp$ denotes the chaotic process.

Recursion is handled by the μ operator. For instance the handshake protocol for the two-wire interface shown in Figure 2(a), which always performs the event r before the event a and never engages in two consecutive events r before the occurrence of a, is given by:

$$HP(r,\ a) \quad \stackrel{def}{=} \quad \mu X :: ((r \rightarrow a \rightarrow X) \,[\!]\, \mathbf{skip}).$$

The presence of **skip** as a possible choice allows $HP(r, a)$ to terminate when its partner in a parallel composition terminates.[1] The modelling of handshake protocols forms an important part of our derivation techniques. What follows elaborates on this example.

Using the CSP algebra, one can show that a sequence of $HP(r,\ a)$ is still a handshake protocol:

$$HP(r,\ a); HP(r,\ a) \quad = \quad HP(r,\ a).$$

Parallel composition of P and Q, represented by $P \ \|_A \ Q$, synchronises on the set of events A common to both P and Q; the A will usually be dropped since it can be deduced from context. Sometimes we abuse the notation by using $P \parallel Q$ to represent a parallel program where the events in the set $\alpha P \cap \alpha Q$ are *not* hidden. The synchronised communication events between components of a parallel composition are sometimes called channels: $c?$ is an input channel while $c!$ is an output channel. Channels may pass values: $c!e$ sends the value of the expression e to the output channel c, and $c?x$ reads a value from the input channel c and assigns it to the variable x. $\mathbf{InputChan}(P)$ and

[1] To reassure those who remember Tony Hoare's remark ([6], page 178), including an unguarded **skip** in an external choice is acceptable here.

$\mathbf{OutputChan}(P)$ represent the sets of input channel names and output channel names respectively, and $\mathbf{Chan}(P) \stackrel{def}{=} \mathbf{OutputChan}(P) \cup \mathbf{InputChan}(P)$.

A process Q with a and b in its alphabet satisfies the two-phase handshake protocol on $(a,\ b)$ if $(Q \parallel HP(a,\ b)) = Q$. This condition will be signified by $\vdash_{(a,\ b)}\ Q$. Given this condition, one can show that for any process R, $HP(a,b)$ distributes into $(Q;R)$ to give

$$(Q\ ;R) \parallel HP(a,\ b) \ =\ (Q \parallel HP(a,\ b))\ ;(R \parallel HP(a,\ b)) \ =\ Q\ ;(R \parallel HP(a,\ b)). \tag{2}$$

A similar law will be used later in showing that $P;Q \ \sqsubseteq\ \mathcal{M}(P;Q) \ \parallel\ \mathcal{D}(P;Q)$.

We need to generalise $HP(a,b)$ in order to pass data in the handshake protocol, as shown in Figure 2(b) and Figure 2(c). One way to achieve this is to let I be a finite set, B be an I-indexed family of finite set of events, and $A = \{(a(i),\ B(i)) \mid i \in I\}$. The two-phase handshake protocol on A can then be defined as follows:

$$HP(A) \stackrel{def}{=} \mu X :: [\!]_{i\in I}\,(a(i) \rightarrow ([\!]_{b\in B(i)}\, b \rightarrow X)\ [\!]\ \mathbf{skip}).$$

A process Q with

$$(\{a(i) \mid i \in I\} \cup \bigcup_{i\in I} B(i)) \subseteq \alpha(Q)$$

is said to obey the two-phase handshake protocol on the set A if $Q \parallel HP(A) = Q$. This condition will be referred to as $\vdash_A\ Q$.

We can now define the two-phase handshake refinement operator $\sqsubseteq_A$ as follows:

$$R\ \sqsubseteq_A\ S \quad \stackrel{def}{=} \quad (R \parallel HP(A)) \ \sqsubseteq\ (S \parallel HP(A)),$$

where $\sqsubseteq$ adopts the refinement ordering in the failures-divergences model of CSP. In other words, $R\ \sqsubseteq_A\ S$ means that S behaves better than R in any environment which obeys the handshake protocol $HP(A)$. One can show that, if $R\ \sqsubseteq_A\ S$ and $\vdash_A\ Q$, then

$$(Q \parallel R) \quad \sqsubseteq \quad (Q \parallel S). \tag{3}$$

Later we shall illustrate how this property can be used in replacing $\mathcal{D}(P)$ by a sequential version, $\mathcal{SD}(P)$, which has useful algebraic laws to simplify our derivation.

5 Variables

This section and the next two show how to construct communicating processes to model program variables and expressions of the source program P, and to transform P into a network of communicating processes such that $P \ \sqsubseteq\ \mathcal{M}(P) \parallel \mathcal{D}(P)$. The master process $\mathcal{M}(P)$ should retain the control structure of the source program P, while the data process $\mathcal{D}(P)$ can be expressed as $\mathtt{Var}(P) \parallel \mathtt{Exp}(P)$, where $\mathtt{Var}(P)$ and $\mathtt{Exp}(P)$ implement respectively the variables and expressions of P.

Given that $\mathbf{VAR}(P)$ denotes the set of all program variable names for P, we define $\mathtt{Var}(P)$ as the parallel composition of all the processes $\mathtt{Var}(x)$ representing program variables in the source program P:

$$\mathtt{Var}(P) \quad \stackrel{def}{=} \quad \parallel \{x\ :\ x \in \mathbf{VAR}(P)\ :\ \mathtt{Var}(x)\}.$$

$\mathtt{Var}(x)$ models a program variable x by providing a pair of channels $(x.req,\ x.val)$ for read access: after $x.req$ is activated, the value of x should be available from the channel $x.val$. More precisely, given that $\mathbf{Chan}(\mathtt{Var}(x)) \subseteq \mathbf{Chan}(Q)$, $\mathtt{Var}(x)$ should satisfy

$$(x.req! \rightarrow\ x.val?v \rightarrow\ Q) \parallel \mathtt{Var}(x) \quad = \quad (v\ :=\ x)\ ;(Q \parallel \mathtt{Var}(x)). \tag{4}$$

Similarly, assigning a new value to x can be achieved by communicating with the channels $(x.write, x.ack)$. Formally, given that $\mathbf{Chan}(\mathtt{Var}(x)) \subseteq \mathbf{Chan}(Q)$, one should be able to show that

$$(x.write!v \rightarrow\ x.ack? \rightarrow\ Q) \parallel \mathtt{Var}(x) \quad = \quad (x\ :=\ v)\ ;(Q \parallel \mathtt{Var}(x)). \tag{5}$$

The application of these lemmas in proofs will be demonstrated in the appendix.

We use three components to implement $\texttt{Var}(x)$. $\texttt{Cell}(x)$ is the basic storage cell, and $\texttt{RMux}(x)$ and $\texttt{Wmux}(x)$ are multiplexors allowing multiple users to connect to $\texttt{Cell}(x)$:

$$\texttt{Var}(x) \stackrel{def}{=} \texttt{Cell}(x) \parallel \texttt{RMux} \parallel \texttt{WMux}. \tag{6}$$

Let us consider each of these components in turn. As explained in [6], a Boolean variable x can be modelled by a communicating process $\texttt{Cell}(x)$ which models a storage cell:

$$\texttt{Cell}(x) \stackrel{def}{=} \mathbf{skip} \,[\!]\, \texttt{Read}(x) \,[\!]\, \texttt{Write}(x).$$

The process $\texttt{Read}(x)$ describes how the user can read the value of the variable x using communications:

$$\texttt{Read}(x) \stackrel{def}{=} x.req? \rightarrow x.val!x \rightarrow \texttt{Cell}(x).$$

The local variable x in the process `Cell` is used to hold the current value of the program variable x. It is required that the user of the process `Read` must obey the handshake protocol on $(x.req,\, x.val)$ in order to avoid deadlock on these two channels.

The process $\texttt{Write}(x)$ describes how the value of x is updated by its user process,

$$\texttt{Write}(x) \stackrel{def}{=} x.write?x \rightarrow x.ack! \rightarrow \texttt{Cell}(x).$$

Users of $\texttt{Write}(x)$ are also required to obey the handshake protocol on the channels $(x.write,\, x.ack)$.

As a state holder, the process $\texttt{Cell}(x)$ should be able to communicate with a number of users. To serve multiple-user requests, we treat the reading and writing actions as atomic. For this purpose we introduce multiplexors `RMux` and `WMux`:

$$\begin{aligned}
\texttt{RMux} &\stackrel{def}{=} \mathbf{skip} \,[\!]\, ([\!]_{i\in I}\,(x.req_i? \rightarrow x.req! \rightarrow x.val?w \rightarrow x.val_i!w \rightarrow \texttt{RMux})),\\
\texttt{WMux} &\stackrel{def}{=} \mathbf{skip} \,[\!]\, ([\!]_{j\in J}\,(x.write_j?w \rightarrow x.write!w \rightarrow x.ack? \rightarrow x.ack_j! \rightarrow \texttt{WMux})),
\end{aligned}$$

where the sets I and J are both finite. It is clear that the process `RMux` is a legal user of the process $\texttt{Read}(x)$ since it obeys the handshake protocol on the channels $(x.req,\, x.val)$. Furthermore, the process `RMux` complies with the handshake protocol on $(x.req_i,\, x.val_i)$ for every $i \in I$.

Given that $\texttt{Var}(x)$ is associated with the alphabet

$$\begin{aligned}
\mathbf{InputChan} &= \{x.req_i \mid i \in I\} \cup \{x.write_j \mid j \in J\},\\
\mathbf{OutPutChan} &= \{x.val_i \mid i \in I\} \cup \{x.ack_j \mid j \in J\}.
\end{aligned}$$

Each user process of $\texttt{Read}(x)$ is allocated a pair $(x.req_i,\, x.val_i)$ of channels for accessing the variable x via the multiplexor `RMux`, and in turn each user process is required to satisfy the handshake protocol over the corresponding channels. The users of $\texttt{Write}(x)$ can be treated in a similar way.

6 Expressions

Let $\texttt{Exp}(P)$ be the parallel composition of the expression processes required by the program P. The characterisation of an expression process is similar to that for a variable process: for any Boolean expression b in P, and for any process Q with $\mathbf{Chan}(Q) \supseteq \mathbf{Chan}(\texttt{Exp}(P) \parallel \texttt{Var}(P))$,

$$(b.req_i! \rightarrow b.val_i?v \rightarrow Q) \parallel \mathcal{D}(P) \;=\; (v := b)\,;\,(Q \parallel \mathcal{D}(P)).$$

A Boolean expression can be modelled by a communicating two-phase module in the same way as a program variable. For example, the evaluation of the expression $x \vee y$ can be described by the process

$$\begin{aligned}
\texttt{OR}(x,\,y) \stackrel{def}{=}\; & \mathbf{skip} \,[\!]\\
& \mathbf{var}\ w\ :\ (req? \rightarrow x.req_i! \rightarrow x.val_i?w \rightarrow ((val!w \rightarrow \texttt{OR}(x,\,y))\\
& \qquad \lhd w \rhd (y.req_j! \rightarrow y.val_j?w \rightarrow val!w \rightarrow \texttt{OR}(x,\,y)))).
\end{aligned}$$

The module operates as follows. It receives an event from the req channel, and it activates $x.req_i$, the req channel of the two-phase module evaluating the x operand. The result of the evaluation will be received from the $x.val_i$ channel, and its value will be assigned to the local variable w. If w has the value *true*, then this value will be passed to the val channel; otherwise the two-phase module evaluating the y operand will be activated, and the value returned from it will be passed to the val channel.

Taking the multiple-user issue into account, we adopt the following definition for $(x \vee y)$:

$$\mathtt{Exp}(or(x,\ y)) \stackrel{def}{=} \mathtt{OR}(x,\ y)) \parallel \mathtt{RMux}(req,\ val)$$

where $\mathtt{RMux}(req,\ val)$ becomes the process $\mathtt{RMux}(x.req,\ x.val)$ after proper channel renaming:

$$\mathtt{RMux}(req,\ val) \stackrel{def}{=} \mathbf{skip} \,[\!]\, ([\!]_{i\in I}\,(req_i? \rightarrow req! \rightarrow val?w \rightarrow val_i!w \rightarrow \mathtt{RMux}(req,\ val))).$$

In general, a composite expression $b = b_1 \vee b_2$ can be defined in the same way as the expression $(x \vee y)$ except that the former will communicate with the processes $\mathtt{Exp}(b_1)$ and $\mathtt{Exp}(b_2)$ rather than $\mathtt{Var}(x)$ and $\mathtt{Var}(y)$. To avoid channel name clash among the expression processes, we will rename the channels req and val used in the process $\mathtt{Exp}(b)$ to $b.req$ and $b.val$ respectively.

To be able to execute the expression processes in parallel with the master control process, we must make sure that the processes representing expressions have disjoint sets of channels. In particular, since most expression processes may need to access the variable processes, the allocation of the channels $x.req_i,\ x.val_i$ turns out to be an important issue of the hardware compiler.

For simplicity we assume that there are index functions $\mathbf{RIdx}$ and $\mathbf{WIdx}$. For each variable process $\mathtt{Var}(x)$, in addition to the channels used by the expression processes, the following set of channels

$$\{x.req_i,\ x.val_i \mid i \in \mathbf{RIdx}(x)\} \cup \{x.write_j,\ x.ack_j \mid j \in \mathbf{WIdx}(x)\}$$

is available at the disposal of the master process. For the expression process we adopt the similar convention that the set $\{b.req_i,\ b.val_i \mid i \in \mathbf{RIdx}(b)\}$ of channels can be used by the master process.

To conclude this section, note that the set of two-phase handshake channels consists of data-read channels, data-write channels and expression evaluation channels:

$$\begin{aligned} V \stackrel{def}{=}\ & \{(x.req_i,\ x.val_i) \mid i \in \mathbf{RIdx}(x)\ \&\ x \in \mathbf{VAR}(P)\} \cup \\ & \{(x.write_j,\ x.ack_j) \mid j \in \mathbf{WIdx}(x)\ \&\ x \in \mathbf{VAR}(P)\} \cup \\ & \{(b.req_k,\ b.val_k) \mid k \in \mathbf{RIdx}(b)\ \&\ b\ is\ an\ expression\ in\ P\}. \end{aligned} \tag{7}$$

7 Master control

This section will complete the first stage of our derivation of two-phase modules for Joy programs: to show that $P \sqsubseteq \mathcal{M}(P) \parallel \mathcal{D}(P)$.

The task can be simplified if we exploit the regularity of handshake protocols to construct $\mathcal{SD}(P)$, a sequential version of $\mathcal{D}(P)$, which does not contain parallel composition:

$$\begin{aligned} \mathcal{SD}(P) \stackrel{def}{=}\ & [\!]_{i\in\mathbf{RIdx}(x)\ \&\ x\in\mathbf{VAR}(P)}\,(x.req_i? \rightarrow x.val_i!x \rightarrow \mathcal{SD}(P))\,[\!] \\ & [\!]_{j\in\mathbf{WIdx}(x)\ \&\ x\in\mathbf{VAR}(P)}\,(x.write_j?x \rightarrow x.ack_j! \rightarrow \mathcal{SD}(P))\,[\!] \\ & [\!]_{k\in\mathbf{RIdx}(b)\ \&\ b\in\mathbf{EXP}(P)}\,(b.req_i? \rightarrow b.val_i!b \rightarrow \mathcal{SD}(P))\,[\!] \\ & \mathbf{skip}. \end{aligned}$$

Since all users of $\mathcal{D}(P)$ must follow the handshake protocol $HP(V)$ where V is defined in equation 7, from lemma 3 we can replace $\mathcal{D}(P)$ by $\mathcal{SD}(P)$ within a handshake environment:

$$\mathcal{D}(P) =_V \mathcal{SD}(P). \tag{8}$$

$\mathcal{SD}(P)$ has two properties which are used extensively in our derivation:

$$\mathcal{SD}(P);\mathcal{SD}(P) = \mathcal{SD}(P), \tag{9}$$

$$(\mathcal{M}(P);Q) \parallel \mathcal{SD}(P) = (\mathcal{M}(P) \parallel \mathcal{SD}(P));(Q \parallel \mathcal{SD}(P)). \tag{10}$$

The second equation shows the synchronised termination of $\mathcal{M}(P)$ and $\mathcal{SD}(P)$, a simpler version of which is given in lemma 2.

The construction of the master process is based on a translator $\mathcal{M}$ whose task is to replace each direct evaluation of the expression b by communicating with the process $\mathtt{Exp}(b)$ using the multiplexor $\mathtt{RMux}(b.req,\ b.val)$, and to replace every assignment to the variable x by communicating with the variable process using the multiplexor $\mathtt{WMux}(x.write,\ x.ack)$.

The master processes of the primitive commands have straightforward definitions:

$$\mathcal{M}(\mathbf{skip}) \stackrel{def}{=} \mathbf{skip}, \tag{11}$$

$$\mathcal{M}(x := e) \stackrel{def}{=} \sqcap_{i \in \mathbf{RIdx}(e),\, j \in \mathbf{WIdx}(x)}\ \mathbf{var}\ v :\ (e.req_i! \rightarrow e.val_i?v \rightarrow x.write_j!v \rightarrow x.ack_j? \rightarrow \mathbf{skip}). \tag{12}$$

The definition of $\mathcal{M}(x := e)$ suggests that the choice of channels used to communicate with $\mathtt{Var}(x)$ and $\mathtt{Exp}(e)$ are rather irrelevant. This nondeterminism allows us later to allocate a specific pair $(i,\ j)$ of channel indices for implementing $\mathcal{M}(x := e)$.

The master process of a composite program is formed by those of its individual components:

$$\mathcal{M}(P\,;Q) \stackrel{def}{=} \mathcal{M}(P)\,;\mathcal{M}(Q), \tag{13}$$

$$\mathcal{M}(G_1 \,[\!]\, G_2) \stackrel{def}{=} \mathcal{M}(G_1) \,[\!]\, \mathcal{M}(G_2). \tag{14}$$

The master process for a conditional statement evaluates its Boolean guard by interacting with the related expression process:

$$\mathcal{M}(\mathbf{if}\ b \rightarrow P\ \mathbf{fi}) \stackrel{def}{=} \sqcap_{i \in \mathbf{RIdx}(b)}\ \mathbf{var}\ w :\ ((b.req_i! \rightarrow b.val_i?w \rightarrow\ (\mathcal{M}(P) \lhd w \rhd \bot)))$$

where the local variable w is not used in $\mathcal{M}(P)$. As for conditional statements with multiple branches,

$$\mathcal{M}(\mathbf{if}\ b \rightarrow P \,[\!]\, BG\ \mathbf{fi}) \stackrel{def}{=} \sqcap_{i \in \mathbf{RIdx}(b)}\ \mathbf{var}\ w :\ (b.req_i! \rightarrow b.val_i?w \rightarrow\ (\mathcal{M}(P) \lhd w \rhd \mathcal{M}(\mathbf{if}\ BG\ \mathbf{fi})))$$

where the local variable w is not used in either $\mathcal{M}(P)$ or $\mathcal{M}(\mathbf{if}\ BG\ \mathbf{fi})$.

The master process for an iteration statement is constructed in a similar way:

$$\mathcal{M}(\mathbf{do}\ BG\ \mathbf{od}) \stackrel{def}{=} \mu X :: \mathbf{var}\, w : \mathcal{M}(BG)$$

where

$$\mathcal{M}(b \rightarrow P) \stackrel{def}{=} \sqcap_{i \in \mathbf{RIdx}(b)} (b.req_i! \rightarrow b.val_i?w \rightarrow ((\mathcal{M}(P); X) \lhd w \rhd \mathbf{skip})),$$

$$\mathcal{M}(B \rightarrow P \,[\!]\, BG) \stackrel{def}{=} \sqcap_{i \in \mathbf{RIdx}(b)} (b.req_i! \rightarrow b.val_i?w \rightarrow\ ((\mathcal{M}(P); X) \lhd w \rhd \mathcal{M}(BG))),$$

and the variable w does not appear in either $\mathcal{M}(P)$ or $\mathcal{M}(BG)$.

The correctness of the translator $\mathcal{M}$ can now be shown by structural induction:

$$P \ \sqsubseteq\ \mathcal{M}(P) \parallel \mathcal{D}(P).$$

An outline of the proof will be given in the appendix.

8 Specification of two-phase modules

For a source program P, we define the specification of its target circuit with request channel r and acknowledgement channel a as

$$\Psi^r_a(P) = \mu X :: r? \rightarrow P\,;\, a! \rightarrow X.$$

We include in the appendix a derivation of the main theorem

$$\Psi^r_a(P) \quad \sqsubseteq \quad \Psi^r_a(\mathcal{M}(P)) \parallel \mathcal{D}(P).$$

The right-hand side of this formula is defined to be $C^r_a(P)$, the compilation function mentioned in definition 1. Note that $C^r_a(P)$ involves $\mathcal{M}(P)$, a purely communication-based process without program variables or assignments.

Next, we shall demonstrate how to implement the process $\Psi^r_a(\mathcal{M}(P))$ by a network of two-phase modules within an environment obeying both $HP(r, a)$ and $HP(V)$, where V is the set of channels for variable-read, variable-write and expression evaluation (equation 7). We define

$$(R \sqsubseteq_{Env} S) \quad \textbf{iff} \quad (R \parallel HP(r, a) \parallel HP(V)) \sqsubseteq (S \parallel HP(r, a) \parallel HP(V)).$$

The main objectives of our design are to preserve the modular structure of the source program, and to use a small number of two-phase modules.

We have already introduced a number of two-phase modules, such as `Cell` for implementing variables and `OR` for expression evaluation. The following describes three further examples; the implementation of two of these will be outlined in the next section.

First, the **skip** statement can be implemented by the *SKIP* module:

$$\Psi^r_a(\mathcal{M}(\mathbf{skip})) \quad =_{(r,a)} \quad \mathit{SKIP},$$

where *SKIP* has the property that

$$\mathit{SKIP} \quad = \quad r \rightarrow a \rightarrow \mathit{SKIP}.$$

Second, consider the assignment statement. Let $i \in \mathbf{RIdx}(e)$ so that the channels $e.req_i$ and $e.val_i$ can be used for communicating with the expression evaluation module for e, and let $j \in \mathbf{WIdx}(x)$ so that the channels $x.write_j$ and $x.ack_j$ can be used for communicating with the variable module for x. One can then show that:

$$\Psi^r_a(\mathcal{M}(x := e)) \quad \sqsubseteq_{Env} \quad \mathit{ASGN},$$

where *ASGN* should satisfy

$$\mathit{ASGN} \quad = \quad r? \rightarrow e.req_i! \rightarrow e.val_i?w \rightarrow x.write_j!w \rightarrow x.ack_j? \rightarrow a! \rightarrow \mathit{ASGN}.$$

The third example is sequential composition. If $\mathbf{VAR}(P0) \cap \mathbf{VAR}(P1) = \emptyset$, then

$$\Psi^r_a(P0; P1) \quad \sqsubseteq_{Env} \quad \Psi^{r0}_{a0}(P0) \parallel \mathit{SEQ} \parallel \Psi^{r1}_{a1}(P1) \tag{15}$$

where *SEQ* should satisfy

$$\mathit{SEQ} \quad = \quad r? \rightarrow r0! \rightarrow a0? \rightarrow r1! \rightarrow a1? \rightarrow a! \rightarrow \mathit{SEQ}.$$

This CSP description of the *SEQ* module matches exactly the state diagram shown in Figure 3. Other two-phase modules, such as those for the conditional and iteration statements (Figure 1), can be developed in a similar way.

9 Implementation

The CSP description of two-phase modules serves two purposes. First, we use it as a behavioural specification for further refinement into circuits. Second, it can be regarded as a normal form which provides a basis for hardware/software partitioning in a codesign environment. Most of our work has been focused on hardware synthesis. To illustrate this approach, we outline an implementation within the CSP model. In practice, we use a separate hardware model [9] which is formally linked to CSP.

Let us first introduce a CSP description for a wire:

$$Wire(a, b) \quad \stackrel{def}{=} \quad \mu X : a? \rightarrow ((a? \rightarrow \bot) \,[\!]\, (b! \rightarrow X)).$$

This description captures the behaviour of a wire, which becomes chaotic if it receives a second input before the first signal has propagated to the output. As expected, two wires can be connected into a single one: $Wire(a, b) \parallel_b Wire(b, c) = Wire(a, c)$. Since the two-phase modules communicate with two-phase two-phase protocols, the

two-phase implementations of these modules are relatively straightforward [2]. For instance, it can be shown that the two-phase module *SKIP* can be implemented by $Wire(r, a)$.

The two-phase implementation of *SEQ* involves three wires:

$$\mathit{SEQ} \quad \stackrel{def}{=} \quad Wire(r,\, r0) \parallel Wire(a0,\, r1) \parallel Wire(a1,\, a),$$

and one can show that this definition of *SEQ* satisfies formula 15. This proof is an example which can be checked using an automatic tool such as FDR [4].

A four-phase implementation of *SEQ* can be obtained from the two-phase specification as follows. Given that `4-2` and `2-4` are respectively converters that transform a four-phase protocol to a two-phase protocol and vice versa, we first generate a specification for the four-phase sequential composition operator by connecting converters to the two-phase version as shown in Figure 4. Automatic tools are then used to check that the four-phase implementation shown in Figure 5 satisfies this specification. Further details of this method can be found in [3].

A clocked implementation can be obtained using protocol converters that transform between two-phase handshaking and clocked control. This approach requires a clocked circuit model [9] based on the theory proposed by Verhoeff [13], and we have established a formal link between CSP and this clocked model using a Galois connection. The details are beyond the scope of this paper.

10 Concluding remarks

We have presented in this paper a systematic approach for deriving two-phase modules, an intermediate form for a compilation scheme that targets a simple imperative language for asynchronous and clocked implementations. The use of CSP enables us to structure our derivation into several stages, making the proofs modular and reusable. Our derivation is simplified by algebraic laws governing the operation and refinement of two-phase modules.

There are two significant extensions to our framework which have been developed. The first involves adapting our derivation to accommodate the extension of the source language to cover parallelism and communication [3]. This can be accomplished by including processes that implement channels in the source language, and by using a further translator – in addition to $\mathcal{M}$ – that introduces communication between the master control processes and the channel processes. The second extension involves optimising the hardware produced by our method. For instance, we have developed protocol converters for clocked implementations which can generate conventional clocked circuits [8] from designs with dual-rail encoded data.

Acknowledgements

Thanks to Mike Dean and the anonymous referees for their comments and suggestions. Geoffrey Brown was supported by NSF grant CCR-9058180 and matching funds from AT&T, and by an EPSRC Visiting Fellowship at Oxford University Computing Laboratory. 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 programme, and by a fellowship from Bell-Northern Research Limited. The support of ESPRIT PROCOS-US (ECUS027) project is gratefully acknowledged.

Appendix: some proofs

What follows is a sketch of the proof of

$$\Psi^r_a(P) \quad \sqsubseteq \quad \Psi^r_a(\mathcal{M}(P)) \parallel \mathcal{D}(P)$$

where $\Psi^r_a(P) = \mu X :: (r? \rightarrow (P; (a! \rightarrow X)))$. We shall first prove that $P \sqsubseteq \mathcal{M}(P) \parallel \mathcal{D}(P)$ by structural induction, when P is in the form of an assignment statement (base case) and sequential composition (induction case).

(i) Given $P \stackrel{def}{=} (x := e)$,

$$\begin{array}{ll} & \mathcal{M}(x := e) \parallel \mathcal{D}(P) \\ = & \quad \{\text{definition of } \mathcal{M}(x := e) \text{ (equation 12) and lemma 4}\} \end{array}$$

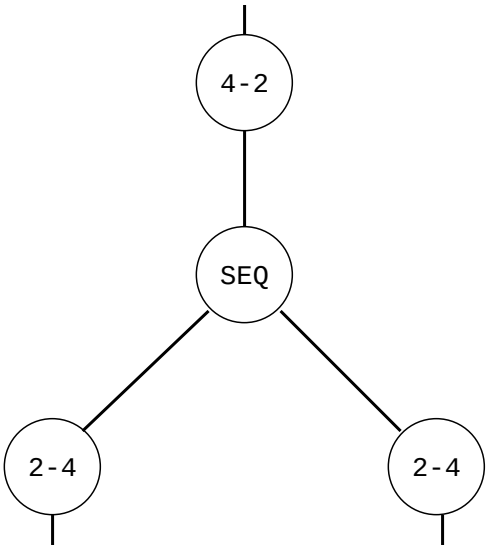
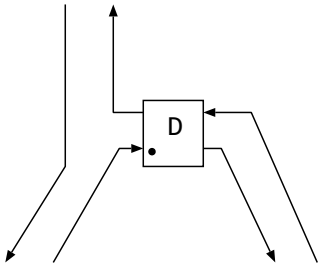


Figure 4: Generating a four-phase specification for a sequential composition circuit.



four-phase
SEQ

Figure 5: A four-phase implementation for sequential composition. The element labelled ‘D’ is described in [7].

$\sqcap_{i \in \mathbf{RIdx}(e),\, j \in \mathbf{WIdx}(x)}\ \mathbf{var}\ v\ :$
$v := e; ((x.write_j!v \rightarrow x.ack_j \rightarrow \mathbf{skip}) \parallel \mathcal{D}(P))$
$=$ {lemma 5}
$\sqcap_{i \in \mathbf{RIdx}(e),\, j \in \mathbf{WIdx}(x)}\ \mathbf{var}\ v\ :$
$(v := e; x := v); (\mathbf{skip} \parallel \mathcal{D}(P))$
$=$ {merging assignments}
$x := e$

(ii) Given $P \stackrel{def}{=} Q; R$,

$\mathcal{M}(Q\,;R) \parallel \mathcal{D}(P)$
$=$ {definition of $\mathcal{M}(Q;R)$ (equation 13)}
$(\mathcal{M}(Q)\,;\mathcal{M}(R)) \parallel \mathcal{D}(P)$
$=$ {lemma 8}
$(\mathcal{M}(Q)\,;\mathcal{M}(R)) \parallel \mathcal{SD}(P)$
$=$ {lemma 10}
$(\mathcal{M}(Q) \parallel \mathcal{SD}(P)); (\mathcal{M}(R) \parallel \mathcal{SD}(P))$
$\sqsupseteq$ {lemma 8}
$(\mathcal{M}(Q) \parallel \mathcal{D}(P)); (\mathcal{M}(R) \parallel \mathcal{D}(P))$
$=$ {induction hypothesis}
$Q\,;R$

Now proceed to the main proof: $\Psi^r_a(P) \sqsubseteq \Psi^r_a(\mathcal{M}(P)) \parallel \mathcal{D}(P)$.

RHS
$\sqsupseteq$ {lemma 8}
$\Psi^r_a(\mathcal{M}(P)) \parallel \mathcal{SD}(P)$
$=$ {definition of $\Psi^r_a(\mathcal{M}(P))$}
$r? \rightarrow ((\mathcal{M}(P); (a! \rightarrow \Psi^r_a(\mathcal{M}(P)))) \parallel (\mathcal{SD}(P))$
$=$ {lemma 10}
$r? \rightarrow ((\mathcal{M}(P) \parallel \mathcal{SD}(P)); ((a! \rightarrow \Psi^r_a(\mathcal{M}(P))) \parallel \mathcal{SD}(P)))$
$=$ {lemma 8}
$r? \rightarrow ((\mathcal{M}(P) \parallel \mathcal{D}(P)); ((a! \rightarrow \Psi^r_a(\mathcal{M}(P))) \parallel \mathcal{D}(P)))$
$\sqsupseteq$ {theorem above: $P \sqsubseteq \mathcal{M}(P) \parallel \mathcal{D}(P)$}
$r? \rightarrow (P; (a! \rightarrow RHS))$
$=$ {unique fixed point of guarded recursion}
LHS

References

[1] Brookes SD, Hoare CAR, Roscoe AW. A theory of communicating sequential processes. JACM 1984; 31:560-599

[2] Brown GM. Towards truly delay-insensitive circuit realizations of process algebras. In: Jones J, Sheeran M (eds) Designing correct circuits. Springer-Verlag, 1991, pp 120-131 (Workshops in Computing)

[3] Brown G, Luk W, O'Leary JW. Retargeting a hardware compiler proof using protocol converters. In: Proc. international symposium on advanced research in asynchronous circuits and systems. IEEE Computer Society Press, 1994

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

[5] He J, Page I, Bowen JP. Towards a provably correct hardware implementation of occam. In: Milne M, Pierre L (eds) Correct hardware design and verification methods. Springer-Verlag, Heidelberg, 1993, pp 214-225 (Lecture Notes in Computer Science No. 683)

[6] Hoare CAR. Communicating sequential processes. Prentice Hall International Series in Computer Science, 1985

[7] Martin AJ. Programming in VLSI: from communicating processes into delay-insensitive circuits. In: Hoare CAR (ed) Developments in concurrency and communication. Addison-Wesley, 1990.

[8] Page I., Luk W. Compiling occam into FPGAs. In: Moore W and Luk W (eds) FPGAs. Abingdon EE&CS Books, 1991, pp 271-283

[9] O'Leary JW. A model and proof technique for verifying hardware compilers for communicating processes. PhD thesis, Cornell University, 1995

[10] Roscoe AW, Hoare CAR. Laws of occam programming. Theoretical Computer Science 1988; 60:177-229

[11] Smith SF, Zwarico AE. Correct compilation of specifications to deterministic asynchronous circuits. Formal Methods in System Design 1995; 7:155–226

[12] van Berkel K. Handshake circuits: an intermediary between communicating processes and VLSI. PhD thesis, Eindhoven University of Technology, 1992

[13] Verhoeff T. A theory of delay-insensitive systems. PhD thesis, Eindhoven University of Technology, 1994

[14] Weber S, Bloom B, Brown GM. Compiling Joy into silicon. In: Advanced research in VLSI and parallel systems, Proc. 1992 Brown/MIT conference. MIT Press, 1992, pp 79-98