A “Hardware Compiler” Semantics for Handel-C

Andrew Butterfield ¹,²
Department of Computer Science
University of Dublin
Dublin, Ireland

Jim Woodcock³
Department of Computer Science
University of York, UK

Abstract
We present a denotational semantics for the hardware compilation language Handel-C that maps language constructs to a set of equations, which describe the structure of the resulting hardware. This semantics is then shown to be useful for validating various algebraic laws which should hold for Handel-C programs, as well as exposing a key principle which governs how such hardware should be operated.

Key words: Handel-C, Hardware Compilation, Denotational Semantics, CSP

1 Introduction

This paper describes a semantics for Handel-C which gives a program a meaning as a collection of equations describing a possible (very naive) hardware implementation — hence the term “Hardware Compiler” Semantics, in the title. This semantics, which sounds very operational in nature, turns out in fact to have a strong denotational character, albeit in an unconventional sense.

Handel-C⁴ [3] is a language originally developed by the Hardware Compilation Group at Oxford University Computing Laboratory, and now marketed by Celoxica

¹ Done while on sabbatical at the University of Kent
² Email: Andrew.Butterfield@cs.tcd.ie
³ Email: Jim.Woodcock@cs.york.ac.uk
⁴ Handel-C is the registered trademark of Celoxica Ltd (www.celoxica.com)

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs
Butterfield and Woodcock Ltd. It is a hybrid of CSP \cite{5} and C, designed to target hardware implementations, specifically field-programmable gate arrays (FPGAs) \cite{7}. The language has sequential and parallel constructs and global variable assignment and channel communication. The language targets synchronous hardware with multiple clock domains. All assignments and channel communication events take one clock cycle. All expression and conditional evaluations, as well as priority resolutions are deemed to be instantaneous, effectively being completed before the current clock-cycle ends.

As the Handel-C language targets hardware, it is ideal for implementing embedded systems, often in situations where high levels of assurance would be desirable \cite{4}. There is a clear need for both a formal semantics of Handel-C (or a reasonable subset) as well as an appropriate methodology and tool support. The research described here is part of program to provide just such an industrial-strength formal framework.

2 Syntax

We introduce here the “mathematical” syntax of a stripped-down version of Handel-C, which albeit simpler, has all the essential features of the synchronous core of the full language.

We have identifiers for channels ($c \in \text{Ch}$) and variables ($x \in \text{Var}$), and we assume the existence of an expression syntax ($e \in \text{Exp}$) whose details need not concern us here. We consider all the above as having either boolean or integer type. We also have the notion of guards ($g \in \text{Grd}$), which denote the offering and accepting of communication actions. Guards either denote expression output along a channel ($c!e$), variable input via a channel ($c?x$), or a skip guard which always succeeds ($!?$).

A syntax of a process is as follows:

\[ P, Q ::= \text{Skip} \mid \text{Delay} \mid x := e \mid P; \ Q \mid P \parallel Q \mid P \triangleleft e \triangleright Q \mid e \ast P \mid \langle g_i \rightarrow P_i \rangle \]

We use notation like $\langle g_i : p_i \rangle$ as shorthand for $\langle g_1 : p_1, \ldots, g_n : p_n \rangle$ where $i$ is assumed to index over $1 \ldots n$ for appropriate $n$. In the last construct, if the $!?$ guard appears it must appear only once, as the last guard.

We can briefly summarise the behaviour of a Handel-C process as follows: \text{Skip} does nothing, in zero time; \text{Delay} does nothing, but takes one clock cycle to do it; $x := e$ assigns the value of $e$ into $x$, taking one clock cycle; $P; Q$ first executes $P$, and once it has terminated immediately starts $Q$; $P \parallel Q$ runs both $P$ and $Q$ in lock-step parallel, terminating when they have both finished; $P \triangleleft e \triangleright Q$ evaluates $e : \mathbb{B}$ and executes $P$ immediately if $e$ is True, otherwise it runs $Q$; and $e \ast P$ tests $e : \mathbb{B}$ and if True it runs $P$ and then repeats, otherwise it terminates.

The $\langle g_i \rightarrow P_i \rangle$ construct (“prialt”) is an ordered sequence of guard-process pairs. Each guard is checked against the process environment to see if it is able to execute.
If no guards are so enabled, then the prialt blocks until the next clock cycle when it tries again. If one or more guards are enabled, then the first such in the list is executed, and the corresponding process is executed subsequently. An input guard \((c?x)\) is enabled if there is a corresponding output guard \((c!e)\) in some other prialt executing at the same time, and \textit{v.v.} The skip guard \((!?\)\) is always enabled. The input \((c?x)\) and output \((c!e)\) guards perform their actions taking one clock-cycle, while the skip guard \((!?\)\) acts like \textit{Skip} so the subsequent process starts execution immediately. It is this “instant” execution of \(!?\) guards that so complicates the formal semantics of Handel-C \cite{2}.

To see the problem, consider the following process:

\[(1) \quad \langle c!66 \rightarrow \text{Skip} \rangle \parallel \langle d!99 \rightarrow \text{Skip}, \!? \rightarrow (b \ast P; \langle c?x \rightarrow \text{Skip} \rangle) \rangle\]

In order to establish the outcome here (using the operational semantics of \cite{2} for example) we proceed as shown in Figure 1, where \(|_n|\) indicates a transition-sequence annotated with the number of clock cycles elapsing; \([\text{cond}]\) denotes a side-condition; and \([\text{effect}]\) denotes some change to internal state.

\[
\begin{align*}
\langle c!66 \rightarrow \text{Skip} \rangle & \parallel \langle d!99 \rightarrow \text{Skip}, \!? \rightarrow (b \ast P; \langle c?x \rightarrow \text{Skip} \rangle) \rangle \\
|_0 & \quad \text{[requests lodged]} \\
\langle c!66 \rightarrow \text{Skip} \rangle & \parallel \langle d!99 \rightarrow \text{Skip}, \!? \rightarrow (b \ast P; \langle c?x \rightarrow \text{Skip} \rangle) \rangle \\
|_0 & \quad \text{[requests resolved]} \\
\langle c!66 \rightarrow \text{Skip} \rangle & \parallel \langle b \ast P; \langle c?x \rightarrow \text{Skip} \rangle \rangle \\
|_0 & \quad [b = \text{False}] \\
\langle c!66 \rightarrow \text{Skip} \rangle & \parallel \langle c?x \rightarrow \text{Skip} \rangle \\
|_1 & \quad [x \mapsto 66] \\
\text{Skip} & \parallel \text{Skip}
\end{align*}
\]

Fig. 1. Program Execution

Details of how requests are “lodged” and “resolved” can be found in \cite{1}.

Prialtls nested inside default clauses of other prialtls may become active in the same clock cycle as those enclosing prialtls, which requires us to iterate this request–resolve loop several times, in any given clock cycle. Managing this micro-cycle activity severely complicates the operational semantics. However, the underlying
Butterfield and Woodcock

hardware doesn’t iterate, as it computes what is to be active in any given clock cycle using combinatorial logic. The “Hardware Compilation” semantics described here was initially developed to see if such a semantics would give some insight into a simpler, less “micro”-iterative operational semantics. In other words, can we find a way to compute the outcome in one (functional) step?

3 Hardware Compilation

The key concept behind the hardware semantics is to recognise that the resulting hardware simply consists of a fixed bank of registers, connected by fixed combinatorial logic — in effect a large (finite-)state machine. On each clock cycle, new values for the register state are computed as a function of the current values. Program execution simply repeats this fixed calculation on every clock cycle. The hardware semantics of a Handel-C program is therefore simply a fixed function $f: \text{State} \to \text{State}$, where $\text{State}$ denotes the contents of all the registers. The contribution of this work is to describe how $f$ is determined from the Handel-C language constructs in a compositional manner. We use equations that model the behaviour of the hardware to describe how $f$ is computed.

The main features of the hardware that need to be modelled are:

- Registers, loaded on a clock edge used to store variable values and control tokens to manage control flow.
- Multiplexers used to route expression results to registers and channels (wires) and channel data to registers.
- Program statement hardware has two key signals: $\text{start} : \mathbb{B}$, an input, starts execution of the statement; while $\text{done} : \mathbb{B}$, an output, indicates its termination.
- A control token is a register whose input is a $\text{done}$ signal, and whose output is fed to one or more $\text{starts}$.

We shall express all these components using a set of equations which distinguish between combinatorial (pure functions) and sequential (stateful) hardware. An equation simply equates a variable on its lefthand-side with either a combinatorial or sequential expression on its righthand-side:

$$\text{Eqn} \equiv \text{Var} \times \text{Rhs}$$

$$\text{Rhs} \equiv \text{CombExpr} \mid \text{SeqExpr}$$

We differentiate between combinatorial and sequential expressions by using parentheses for the former ($z = f(x, y)$) and square brackets for the latter ($w = g[x, y]$). The overall system is described as a list of such equations,

$$\text{Sys} \doteq \{ \text{Eqn} \}$$
which we expect to have no “combinatorial cycles”: Any circular chain of dependencies must include a sequential equation. Generally we either list the equations one to a line as follows:

\[
\begin{align*}
x &= f(u, v) \\
y &= g(w, x) \\
z &= h(u, y)
\end{align*}
\]

or we list several on one line, separated by semi-colons:

\[x = f(u, v); \ y = g(w, x); \ z = h(u, y)\]

The combinatorial building blocks provided are the usual functions over \(\mathbb{B}\) and \(\mathbb{Z}\) such as \(\land, \lor, \neg, +, -, /, \) etc., plus multiplexers (with non-standard controls\(^5\)):

\[
\text{mux}_n : \mathbb{B}^n \rightarrow \mathbb{Z}^n \rightarrow \mathbb{Z}
\]

\[
\text{mux}_n(c_1, \ldots, c_n)(\text{data}_1, \ldots, \text{data}_n) \triangleq \text{data}_i, \text{ if } c_i
\]

If no \(c_i\), or more than one, is true, then the output is undefined. The latter case corresponds to more than one process trying to update a variable in a given clock cycle.

These multiplexers are required because a single process variable \(x\) may participate in many assignment statements only one of which should be active during any clock cycle (e.g.):

\[
\ldots; \ x := 0; \ \ldots; \ x := x + 1; \ \ldots; \ x := y - 2 * z; \ \ldots; \ x := -1; \ \ldots
\]

A multiplexer connects the four pieces of hardware implementing the expressions \(0, \ x + 1, \ y - 2 * z, \ -1\) to the input of register \(x\). The \textit{start} signals for each assignment statement above control the multiplexer to determine which expression is routed through to the output. The logical-or of these \textit{start} signals enables the loading of the register. All register updates occur on the appropriate edge of the global clock, which is implicit in this semantic model\(^6\).

We use three sequential building blocks:

- Registers: \textit{register[load : \mathbb{B}, in : \mathbb{Z}] : \mathbb{Z}}
  
  When \textit{load} is true, \textit{in} is stored at the clock edge.

- Wait Block (Control Token): \textit{wait[fini : \mathbb{B}] : \mathbb{B}}
  
  The value of \textit{fini} is stored and appears on output after clock edge.

- Synchronisation Block: \textit{sync_n[done_1 : \mathbb{B}, \ldots, done_n : \mathbb{B}] : \mathbb{B}}
  
  \textit{sync_n}’s output is initially false. It waits, over many clock cycles if necessary, for

\(^5\) The standard \(n\)-way multiplexer encodes the controls in \(\lceil \log_2 n \rceil\) bits.

\(^6\) Dealing with multi-clock Handel-C would require the use of explicit clock variables in the equations, but this is relatively easy to add in if required.
all $n$ done, to go true. Then its output goes true immediately, and reverts to false at the next clock edge.

In order to generate hardware equations we need to generate hardware equation variables (not to be confused with process variables), which is achieved by giving every process statement (atomic and compound) a unique label. So, for example, the conditional statement $p < e > q$ might be labelled as $e:(m:p < c > n:q)$, where $\ell$, $m$ and $n$ are unique labels, with $\ell$ labelling the entire conditional construct, while $m$ and $n$ label the true and false branches respectively.

The trick now is come up with a way of generating the hardware equations for a process in a compositional manner. Initially this seems impossible, simply because some of the hardware generated seems to require global knowledge about the whole process for which hardware is being produced. For example, the multiplexers that feed results into variables need to have one data and one control input for every use of the variable in the entire (top-level) process! This seems to mitigate against a compositional semantics in this case.

However, there is a technical trick we can employ to make our semantics compositional: we generate partial hardware descriptions, and use an equation join operator $(\uplus : \mathbb{P} \text{Eqn} \times \mathbb{P} \text{Eqn} \to \mathbb{P} \text{Eqn})$ to collect equations together with appropriate merging of partial hardware elements into more complete ones. This technique for merging partial hardware descriptions is required in three cases:

- **Register multiplexers:**
  We generate a “singleton” multiplexer: $\text{in}.x = \text{mux}_1(\text{start}_{m2})(x+1)$, or an empty one: $c = \text{mux}_0()$ (for input channels).
  We merge them using:

  \[
  \text{in}.x = \text{mux}_m(c_1, \ldots, c_m)(d_1, \ldots, d_m) \uplus \text{in}.x = \text{mux}_n(c_{m+1}, \ldots, c_{m+n})(d_{m+1}, \ldots, d_{m+n})
  \]

  \[
  \equiv
  \]

  \[
  \text{in}.x = \text{mux}_{n+m}(c_1, \ldots, c_m, c_{m+1}, \ldots, c_{m+n})(d_1, \ldots, d_m, d_{m+1}, \ldots, d_{m+n})
  \]

- **Distributed-Or** (used for register-load/channel-data controls)
  We generate either a singleton distributed-or: $v = \bigvee \{ c \}$, or an empty one (for some channel cases): $v = \bigvee \{ \}$

---

7 Merge (\uplus) as defined here is non-commutative, but as the behaviour of $\text{mux}$ is invariant of any consistent re-ordering of controls and data, this has no real effect on the overall semantics, where we would expect merge to be commutative.
Butterfield and Woodcock

\[ [\text{Skip}] \equiv \text{done}_e = \text{start}_e \]
\[ [\text{Delay}] \equiv \text{done}_e = \text{wait}[\text{start}_e] \]
\[ [\text{x} := e] \equiv \text{in}.x = \text{max}_1(\text{start}_e([e])); \text{load}.x = \bigvee \{\text{start}_e\} \]
\[ x = \text{register}[\text{load}.x, \text{in}.x]; \text{done}_e = \text{wait}[\text{start}_e] \]

Fig. 2. Compiling Atomic Statements

We merge these using

\[ v = \bigvee \{c_1, \ldots, c_m\} \uplus v = \bigvee \{c_{m+1}, \ldots, c_{m+n}\} \]
\[ \equiv \]
\[ v = \bigvee \{c_1, \ldots, c_m, c_{m+1}, \ldots, c_{m+n}\} \]

- Register instances:
  
  We generate a register for every use: \( x = \text{register}[\text{load}.x, \text{in}.x] \)
  
  Merged using

  \[ x = \text{register}[\text{load}.x, \text{in}.x] \uplus x = \text{register}[\text{load}.x, \text{in}.x] \equiv x = \text{register}[\text{load}.x, \text{in}.x] \]

  i.e., they all refer to the same register

The merging of all other sets of equations simply involves lumping them together as a larger set of separate equations\(^8\).

3.1 Hardware Compilation Semantics

We are now in a position to give the hardware semantics for all the process types in our language. We introduce a semantic function which maps processes into sets of hardware equations:

\[ [-] : P \rightarrow \mathcal{P} \text{Eqn} \]

The semantics of the atomic statements is given in Figure 2.

\text{Skip} asserts \text{done} the instant is \text{started}.

\text{Delay} waits for the clock cycle in which is was \text{started} to end before asserting that it is \text{done}. Hence it always takes one clock cycle to execute.

The semantics of assignment \( x := e \) is simply to route the current value of \( e \) through a multiplexer to the input (\text{in}.x) of register \( x \). The assignment statement’s \text{start} control is used to route the multiplexer and load the register (via \text{load}.x).

\(^8\) There is another problem with compositionality and the requirement for unique labels, but this can be resolved by having \( \psi \) use renaming to avoid label clashes.
We rely on the merge operator as previously described to link up the multiplexers, distributed-ors, and to merge identical register invocations to get the global hardware required.

The semantics of the standard compound statements are given in Figure 3.

The conditional \(<c>\) uses \(c\) to determine which branch to start. It is done when either branch is.

The loop \((\ast)\) looks at its condition. If false it terminates immediately, otherwise it starts its body. It itself starts on an external request, or if its body has just terminated.

Sequential composition \(\;\) starts its first sub-statement immediately, its second the instant the first is done, and it terminates when the second does.

Parallel composition \(||\) starts all its sub-statements immediately once it is itself started. It is done when all its sub-statements are done, as signalled by \(\text{sync}_2\).

The compilation of the prialt statement is shown in Figure 4.

The condition \(<g_0 =!?>\) is a “compile-time” conditional, which does not translate into hardware. Once started, a prialt gets its first guard to “offer” to communicate. The guard will report if it is active. If not, then each next guard in sequence is made to “offer”. Once a guard gets an active response, it executes in this cycle, followed by its continuation process in the next (except for default guards,
\[
\ell::(m_1::g_1 \rightarrow n_1::p_1, \ldots, m_k::g_k \rightarrow n_k::p_k)\\ \\
\triangleq \bigcup_i [m_i::g_i] \uplus \bigcup_i [n_i::p_i] \uplus\\ \\
offer_{m_1} = \text{start}_\ell \lor \text{retry}_\ell; \quad offer_{m_{i+1}} = offer_{m_i} \land \neg \text{active}_{m_i}\\ \\
\text{start}_{n_i} = offer_{m_i} \land g_i = !? \Rightarrow \text{wait}[\text{active}_{m_i}]\\ \\
\text{inactive}_\ell = \neg \bigvee \{\text{active}_{m_1}, \ldots, \text{active}_{m_k}\}\\ \\
\text{retry}_\ell = \text{wait}[(\text{start}_\ell) \lor \text{retry}_\ell) \land \text{inactive}_\ell]\\ \\
\text{done}_\ell = \bigvee \{\text{done}_{n_1}, \ldots, \text{done}_{n_k}\}
\]

Fig. 4. Compiling “prialt”

\[
[e!!?] \triangleq \text{active}_\ell = \text{offer}_\ell\\ \\
[e; c!e] \triangleq out.c = \bigvee \{\text{offer}_\ell\}; \quad in.c = \bigvee \{}\\ \\
c = \text{mux}_1(\text{active}_\ell)([e]); \quad \text{active}_\ell = \text{offer}_\ell \land in.c\\ \\
[e; c?]x \triangleq in.c = \bigvee \{\text{offer}_\ell\}; \quad out.c = \bigvee \{}; \quad \text{active}_\ell = \text{offer}_\ell \land out.c\\ \\
in.x = \text{mux}_1(\text{active}_\ell)(c); \quad load.x = \bigvee \{\text{active}_\ell\}\\ \\
x = \text{register}[\text{load}.x, in.x]; \quad c = \text{mux}_0()\\
\]

Fig. 5. Compiling Guards

whose continuation process starts immediately). The prialt terminates when the continuation process is done. If all guards are inactive, it retries next clock cycle.

The prialt semantics also makes use of a compilation scheme for guards:

\[[-]: \text{Grd} \rightarrow \mathbb{P} \text{Eqns}\]

The compilation semantics for guards is described in Figure 5.

The guards do not have done and start control tokens, but instead use signals offer and active respectively to offer to perform their corresponding action, and to be told that their action is to be active.

The skip guard !? is implemented with a piece of wire, since it is always active if it offers (Who cares about how it complicates the semantics !!). An output guard c!e makes a global output offer on out.c, and becomes active if it offers. An output guard e!x makes a global output offer on in.c, and becomes active if it offers. An output guard c!x makes a global output offer on in.c, and becomes active if it offers.
sees a global output offer on \textit{out.c}. If \textit{active}, it behaves like an assignment \( x := c \) where \( c \) is the channel data. It needs the value of \( c \) but cannot provide it, so an empty multiplexer is used to complete the semantics and avoid a dangling reference.

### 3.2 Where has the fixed point gone?

A standard feature of denotational semantics is the use of fixed points to reason about recursion and iteration. However a look at the semantics of \( c \ast p \) shows no sign of a fixed point. Fixed points are used to ensure that the semantics so given is compositional — the semantics of a compound language construct is built up from the semantics of its components.

First we point out that the semantics given here is compositional — for example \([c \ast p]\\) is given in terms of \([c]\) and \([p]\). The merging of the semantics of program fragments is achieved by \( \sqcup \), which is defined at the semantic level. Secondly, note that we are defining the behaviour of the program, or indeed any well-formed fragment, by giving its computational behaviour for a single clock-cycle. The running program is characterised by a sequence of states generated on successive clock-cycles by the repeated use of \( f \) on some starting state \( s_0 : \text{State} \)

\[
s_0, f(s_0), f^2(s_0), f^3(s_0), \ldots
\]

This is where the fixed point has gone — the iteration and its fixed-point semantics is effectively lifted up to a top-level, were it effectively covers the whole program’s execution trace.

This is why we refer to this semantics as “denotational”, but admittedly in an unconventional manner.

### 4 Laws of Handel-C

We would like to be able to validate a variety of algebraic laws for Handel-C process, such as:

\[
\begin{align*}
\text{Skip}; P & \equiv P \equiv P; \text{Skip} \\
\textit{P } || \textit{Q} & \equiv \textit{Q } || \textit{P} \\
\textit{P ; (Q } || \textit{R}) & \equiv (\textit{P } || \textit{Q}); \textit{R} \\
\textit{b } \ast \textit{P} & \equiv \textit{P}; \textit{b } \ast \textit{P} < \textit{b } \triangleright \text{Skip}
\end{align*}
\]

We consider two programs as equivalent \((\equiv)\) if they both make the same variable assignments on each clock cycle.

The hardware semantics makes it surprisingly easy to prove some of these laws, in particular the structural ones.

In order to perform the proofs we need to introduce the notion of a process variable denoting an arbitrary process \((\ell : P, \text{say})\), and referring to its hardware
semantics expansion as
\[ done_\ell = P[\text{start}_\ell] \]

Here \( P[\ldots] \) represents all the hardware equations that correspond to the semantics of \( P \). The equation above simply serves to name the start and done signals for that hardware.

4.1 Proving Skip a unit for ;

We now consider the following three-way equation on processes:

\[ \text{Skip;} \; P \equiv P \equiv P; \; \text{Skip} \]

We introduce labels:
\[ \ell::(s::\text{Skip} ; p::P) = \ell::(p::P) = \ell::(p::P ; t::\text{Skip}) \]

The extra label on the middle process simply serves to make it easier to compare the results. Expanding out the lefthand-side:
\[ [\ell::(s::\text{Skip} ; p::P)] \]
\[ \cong done_p = P[\text{start}_p]; \; done_s = \text{start}_s \]
\[ \text{start}_s = \text{start}_\ell; \; \text{start}_p = \text{done}_s; \; \text{done}_\ell = \text{done}_p \]

Expanding out the middle:
\[ [\ell::(p::P)] \]
\[ \cong \text{start}_p = \text{start}_\ell; \; \text{done}_\ell = \text{done}_p; \; \text{done}_p = P[\text{start}_p] \]

Expanding out the righthand-side:
\[ [\ell::(p::P ; \ell::\text{Skip})] \]
\[ \cong done_p = P[\text{start}_p]; \; done_\ell = \text{start}_t \]
\[ \text{start}_p = \text{start}_\ell; \; \text{start}_t = \text{done}_p; \; \text{done}_\ell = \text{done}_t \]

How do we reconcile these three? We have a label \( s \) in one, but a label \( t \) in another, which are not equivalent. The key is to define the concept of a degenerate equation as one which simply equates two variables. We then add in the concept of a degenerate label by defining such as a label for which every equation in which it appears is degenerate, and that all variables referencing it occur as the righthand-side of at least one such degenerate equation. Careful examination of the equations above finds that labels \( s \) and \( t \) are degenerate. Label \( \ell \) is not degenerate, because \( \text{start}_\ell \) does not appear on any equation righthand-side.

We shall simply use appropriate equation substitution to eliminate degenerate labels — for example if \( n \) is degenerate below then
\[ x_n = y_m; \; z_p = f(\ldots x_n \ldots) \]
can be safely replaced by

\[ z_p = f(\ldots y_m \ldots) \]

We can safely do this as it has no effect on the underlying hardware — in effect degenerate equations simply indicate a situation where wires have multiple names, and a degenerate label is one whose sole use is in the provision of one of these aliases. Removing them makes no difference to the underlying hardware.

If we now strip \( t \) and \( s \) out

\[
\begin{align*}
\ell &:: (p::P)\\
\equiv &\begin{cases} &\text{start}_p = \text{start}_t; \text{done}_t = \text{done}_p; \text{done}_p = P[\text{start}_p] \\
&[t::(p::P); t::\text{Skip}]\\
\equiv &\begin{cases} &\text{start}_p = \text{start}_t; \text{done}_t = \text{done}_p; \text{done}_p = P[\text{start}_p] \\
\end{cases}
\end{cases}
\]

We see all three sets of equations are now identical. \( \square \)

4.2 Proving commutativity of ||

We now consider the issues surrounding a proof of the commutativity of parallel composition:

\[ P || Q = Q || P \]

We label both sides as follows:

\[
\begin{align*}
t &:: ((p::P) || (q::Q)) \\
&\equiv [p::P] \uplus [q::Q] \uplus \\
&\begin{cases} &\text{start}_p = \text{start}_q; \text{start}_q = \text{start}_t; \text{done}_t = \text{sync}[\text{done}_p, \text{done}_q] \\
\end{cases}
\end{align*}
\]

Before expanding out \([p::P]\) and \([q::Q]\), we note that in general these produce not just one equation, but many, and that these may refer to common variables. However, as the \( \uplus \) operator is associative and commutative, we do not need to deal with this explicitly, so we can complete the expansion of the lefthand-side as:

\[
\begin{align*}
[t::((p::P) || (q::Q))] \\
\equiv &\begin{cases} &\text{start}_p = \text{start}_q; \text{start}_q = \text{start}_t; \text{done}_t = \text{sync}[\text{done}_p, \text{done}_q] \\
\end{cases}
\end{align*}
\]

This works fine, but we need to keep in the back of our minds that \( P \) and \( Q \) in the semantics also stand for zero or more additional hidden equations.

We expand out the righthand-side:
Butterfield and Woodcock

\[
\begin{align*}
\ell :: (q::Q || (p::P)) \\
= \text{done}_q = Q[\text{start}_q] \uplus \text{done}_p = P[\text{start}_p] \uplus \\
\text{start}_q = \text{start}_\ell; \quad \text{start}_p = \text{start}_\ell; \quad \text{done}_\ell = \text{sync}[\text{done}_q, \text{done}_p]
\end{align*}
\]

The ordering of equations is irrelevant, and the only difference between the two forms is the equation for \( \text{done}_\ell \). To complete the proof we require that the \( \text{sync} \) function be invariant on any re-ordering of its inputs:

\[
\begin{align*}
\text{sync}[a, b] = \text{sync}[b, a]
\end{align*}
\]

If we assume that \( \text{sync} \) has this property then our proof is complete. In fact, we take this property of \( \text{sync} \) as a specification that \( \text{sync} \) must satisfy.

The proof that sequential composition is associative is similar to that showing that \( \text{Skip} \) is a unit for composition, and requires eliminating degenerate variables in the same way. The (perhaps unsurprising) result of that proof is the following:

\[
\begin{align*}
\ell :: (p::P; \quad m::(q::Q; \quad r::R)) \\
≡ \quad \ell :: (n::(p::P; \quad q::Q); \quad r::R)) \\
≡ \quad \text{start}_p = \text{start}_\ell; \quad \text{done}_p = P[\text{start}_p] \\
\text{start}_q = \text{done}_p; \quad \text{done}_q = Q[\text{start}_q] \\
\text{start}_r = \text{done}_q; \quad \text{done}_r = R[\text{start}_r] \\
\text{done}_\ell = \text{done}_r
\end{align*}
\]

which is the obvious way one would define the semantics of the three-way sequential composition construct:

\[
\ell :: (p::P; \quad q::Q; \quad r::R)
\]

We have seen that in order to prove some laws we need lemmas regarding properties of building blocks, such as \( \text{sync} \). Proving \( b * P = P \); \( b * P < b * \text{Skip} \) requires a much more complex result: namely the Hardware Cloning Lemma.

The Hardware Cloning Lemma states that if we clone a piece of control hardware, and occasionally run that instead of the original, that the switch is unobservable. Note that only the control hardware is cloned — a reference to a variable \( x \) or channel \( c \) denotes the same hardware elements in both the original and cloned hardware.

Let \( \text{done}_p = P[\text{start}_p] \) denote the hardware generated for program \( p::P \). The cloned hardware needs to have labels distinct from those of the original, so let \( \text{done}_r = R(P)[\text{start}_r] \) denote the cloned hardware, where \( R \) is a relabelling function, that maps label \( p \) to \( r \), and maps other labels to new values. We can state the Lemma as

\[
P[\text{start}_p] \lor R(P)[\text{start}_r] \equiv P[\text{start}_p \lor \text{start}_r]
\]

Here \( \text{start}_p \) is true when we plan to run the original, and \( \text{start}_r \) is true when we want to run the clone. Starting \( P \) with \( \text{start}_p \lor \text{start}_r \) corresponds to running
Butterfield and Woodcock

\[
\begin{align*}
\text{wait}[x] \lor \text{wait}[y] &= \text{wait}[x \lor y] & \text{(Wait)} \\
mux_2(c, d)(x, x) &= mux_1(c \lor d)(x) & \text{(Mux)} \\
sync_k[\text{TRUE, ..., TRUE}] &= \text{TRUE} & \text{(SyncNow)} \\
sync_k[\text{wait[TRUE], ..., wait[TRUE]}] &= \text{wait[TRUE]} & \text{(SyncNext)} \\
sync_2[m, n] \lor \text{sync}_3[s, t] &= \text{sync}_2[m \lor s, n \lor t] & \text{provided } \neg((m \lor n) \land (s \lor t)) & \text{(SyncClone)}
\end{align*}
\]

Fig. 6. Building Block Properties

the original in all cases. We assert that we cannot distinguish these two cases. The proof is a long induction over the abstract syntax structure of processes, which we omit.

Why do we need the Lemma to show \( b \ast P \equiv (P; b \ast P) \triangleleft b \triangleright \text{Skip} \)? Simply because the lefthand-side mentions \( P \) once, but the righthand-side mentions it twice. The proof also requires the properties of building blocks shown in Figure 6. Property \( Mux \) is combinatorial and easy to prove, and captures the fact that consistent re-ordering of controls and inputs does not alter the behaviour. Property \( SyncClone \) arises because the \( \parallel \) case in the proof exposes a key assumption required in order for the cloning lemma to hold, namely that for any language construct, once \( \text{start} \) is asserted, it must remain false on subsequent cycles until \( \text{done} \) is asserted. In other words, we cannot re-start hardware until it is done. We shall refer to this as the “No-Pipeline principle” \(^9\), which is captured by the side-condition \( \neg((m \lor n) \land (s \lor t)) \).

We consider this a nice example of how a formal theoretical analysis of an artifact (Handel-C hardware in this case) exposes a key underlying principle about how such an artifact should be operated.

This and the other properties require a formal model that captures time in order to be proven. It is to this that next turn our attention.

5 Register Transfer Notation

To capture time, we need to be very clear about when signals are latched into registers, something about which the equations are somewhat vague. The equation \( x = g[y, z] \) indicates that clocked storage is used by \( g \) but is unclear about precise timings. We shall define a subset of our hardware equation notation, called \( \text{Reg-} \)

\(^9\) This should not be interpreted as meaning that it is not possible to build pipe-lined architectures using Handel-C — this is very feasible, by ensuring each pipeline stage is a separate block running in parallel with other stages.
**Butterfield and Woodcock**

\[
y = \text{wait}[x] \mapsto y := x
\]

\[
x = \text{register}[\text{load}, d] \mapsto x := d < \text{load} > x
\]

\[
\text{alldone} = \text{sync}_k[\text{done}_1, \ldots, \text{done}_k] \mapsto \text{alldone} = \bigwedge \{ \text{isdone}_1, \ldots, \text{isdone}_k \}
\]

\[
\text{isdone}_i = \text{done}_i \lor \text{cmpl}_i
\]

\[
\text{cmpl}_i := \neg \text{alldone} \land (\text{cmpl}_i \lor \text{done}_i)
\]

Fig. 7. Sequential Building Blocks in RTN

**Register Transfer Notation** (RTN) that explicitly defines which lefthand-side variables denote registers. The combinatorial expressions remain unchanged, but the sequential ones must now be “implemented” in terms of a single store primitive \(\text{store}[\text{in}]\) which stores its \(\text{in}\) value (boolean or integer) at every clock edge. We insist that a sequential statement can only consist of a single use of \(\text{store}\), so must be of the form: \(x = \text{store}[\text{data}]\) which we shall simplify with the shorthand \(x := \text{data}\). These latter equations are now referred to as storage equations. We show the implementation of the sequential building blocks in terms of RTN in Figure 7. It is worth noting that the \(\text{wait}\) building block is in fact exactly the same as the \(\text{store}\) block just introduced.

We can give a formal semantics to RTN by translating it into state machines. Given combinatorial-cycle free RTN equations:

\[
w_1 = \text{expr}_1; \ldots; w_m = \text{expr}_m; \ldots; v_1 := \text{expr}_{m+1}; \ldots; v_n := \text{expr}_{m+n}
\]

where \(\text{expr}_i\) ranges over \(v_1, \ldots, v_n\) and \(x_1, \ldots, x_t\), we define the state to be the vector \(s = (v_1, \ldots, v_n) \in S\), the output vector to be \(\mathbf{o} = (w_1, \ldots, w_m) \in O\), and the input vector to be \(\mathbf{i} = (x_1, \ldots, x_t) \in I\). In effect the lefthand-sides of the storage equations constitute the state, and outputs are characterised by being the lefthand-sides of the other equations. If we want to output a state component directly (\(v_i\) say), then we add a (degenerate) equation \(w_{m+1} = v_i\) to signal this.

Given the vectors \(\mathbf{i}, \mathbf{s}\) and \(\mathbf{o}\), we can then summarise the equations as:

\[
\mathbf{o} = (\text{expr}_1(\mathbf{s}, \mathbf{i}), \ldots, \text{expr}_m(\mathbf{s}, \mathbf{i}))
\]

\[
\mathbf{s} := (\text{expr}_{m+1}(\mathbf{s}, \mathbf{i}), \ldots, \text{expr}_{m+n}(\mathbf{s}, \mathbf{i}))
\]

These can be interpreted as representing the next-state and output functions of a state machine.

A state machine with input \(i : I\), state \(s : S\), output \(o : O\), next-state \((\text{ns} : I \rightarrow S \rightarrow S)\) and output \((\text{op} : I \rightarrow S \rightarrow O)\) functions has behaviour:

\[
\text{run}_{\text{ns}, \text{op}} : \text{seq } I \rightarrow S \rightarrow S \times \text{seq } O
\]

\[
\text{run}_{\text{ns}, \text{op}}(\langle \rangle)s_0 \doteq (s_0, \langle \rangle)
\]

\[
\text{run}_{\text{ns}, \text{op}}(i : \text{is})s_0 \doteq (s', (\text{op}(i)s_0) : \text{os}')
\]

where \((s', \text{os}') = \text{run}_{\text{ns}, \text{op}}(\text{is})(\text{ns}(i)s_0)\)
By encoding hardware equation laws as state machines, we can use run to show that both sides have the same outcome.

Consider proving that \( \text{wait}[x] \lor \text{wait}[y] \equiv \text{wait}[x \lor y] \). We define state-machines for both sides of the equivalence as follows: For \( w_1 = \text{wait}[x] \lor \text{wait}[y] \) we get equations \( w = x \lor y \) and state machine:

\[
\begin{align*}
s_1 &= (s_a, s_b); \quad \text{run}_1 = \text{run}_{\text{ns}_1, \text{op}_1} \\
\text{ns}_1(x, y)(s_a, s_b) &= (x, y); \quad \text{op}_1(x, y)(s_a, s_b) = s_a \lor s_b
\end{align*}
\]

For \( w_2 = \text{wait}[x \lor y] \) we obtain equations \( s_2 := x \lor y \); \( w_2 = s_2 \) and machine:

\[
\begin{align*}
s_2 &= s_0 \lor s_b; \quad \text{run}_2 = \text{run}_{\text{ns}_2, \text{op}_2} \\
\text{ns}_2(x, y)(s_2) &= x \lor y; \quad \text{op}_2(x, y)(s_2) = s_2
\end{align*}
\]

We cannot prove that \( \text{run}_1 = \text{run}_2 \) because the states have different types. Instead we prove that outputs are identical for given inputs, and corresponding initial states:

\[
\pi \text{run}_1 = \pi \text{run}_2 (\pi_0, \pi_y)
\]

**Proof:** by induction on (length of) is. The base case is straightforward. The inductive step requires following two lemmas:

\[
\begin{align*}
\text{run}_1((x, y) : is)(s_x, s_y) &= (s''', (s_x \lor s_y) : os''') \quad \text{where} \quad (s'', os'') = \text{run}_1(is)(x, y) \\
\text{run}_2((x, y) : is)(s) &= (s'', s : os'') \quad \text{where} \quad (s'', os'') = \text{run}_2(is)(x \lor y)
\end{align*}
\]

The Inductive Step:

\[
\begin{align*}
\pi \text{run}_1 &= \pi \text{run}_2 (\pi_0, \pi_y) \\
&= \text{" Lemma for run}_1 " \\
\pi && \pi \text{run}_2 ((s'', (x_0 \lor y_0) : os'') \quad \text{where} \quad (s'', os'') = \text{run}_1(is)(x, y)) \\
&= \text{" defn. } \pi \text{run}_2 (\pi_0, \pi_y) \\
&= \text{" inductive step } \pi \text{run}_2 ((x_0 \lor y_0) : os'') \quad \text{where} \quad (s'', os'') = \text{run}_2(is)(x \lor y)) \\
&= \text{" defn. } \pi \text{run}_2 (\pi_0, \pi_y) \\
&= \text{" Lemma for run}_2 " \\
&= \pi \text{run}_2 ((x, y) : is)(x_0 \lor y_0)
\end{align*}
\]

State machines can be coded up in the UTP framework [6] (see Appendix A) and similar proofs can be performed in that setting.
6 Conclusions

We have presented a formal semantics for Handel-C which is compositional and expresses how a process denotes a chunk of hardware described by a set of equations. We have also given examples of how we can use this semantics to prove various laws regarding Handel-C, and we have sketched out a more complex result which requires a specific operating principle (No-Pipelining) to hold in order for the underlying hardware to have the correct behaviour.

We have also built a semantic bridge from Handel-C to UTP, as we can formulate a theory in UTP about state machines (see Appendix A). This results in the first UTP semantics for Handel-C.

Has the Hardware Semantics provided any insight into an “improved” Operational Semantics? This is not immediately clear or obvious, and as the issue has to do with prialt guards and default clauses, we should consider a relevant example — namely the one referred to earlier in this paper:

\( \langle c!99 \rightarrow \text{Skip} \rangle \mid \{ d!66 \rightarrow \text{Skip}, !? \rightarrow (\text{false} \ast \text{Delay}; \langle c?x \rightarrow \text{Skip} \rangle) \}\)

We attach labels to obtain:

\[
\text{done}_0 = (\text{start}_5 \lor \text{cmpl}_1) \land (\text{start}_17 \lor \text{cmpl}_2) \\
\text{cmpl}_1 := \neg \text{done}_0 \land (\text{start}_5 \lor \text{cmpl}_1) \\
\text{cmpl}_2 := \neg \text{done}_0 \land (\text{start}_8 \lor \text{start}_17 \lor \text{cmpl}_2) \\
\text{start}_5 := \text{start}_0; \quad \text{start}_8 := \text{false}; \quad \text{start}_17 := \text{start}_0 \\
x := \text{mux}_1(\text{start}_0)(99) \prec \text{start}_0 \succ x
\]

The resulting hardware semantics, simplified by removing degenerate equations and dangling variables, is:

\[
done_0 = (\text{start}_5 \lor \text{cmpl}_1) \land (\text{start}_17 \lor \text{cmpl}_2) \\
cmpl_1 := \neg \text{done}_0 \land (\text{start}_5 \lor \text{cmpl}_1) \\
cmpl_2 := \neg \text{done}_0 \land (\text{start}_8 \lor \text{start}_17 \lor \text{cmpl}_2) \\
\text{start}_5 := \text{start}_0; \quad \text{start}_8 := \text{false}; \quad \text{start}_17 := \text{start}_0 \\
x := \text{mux}_1(\text{start}_0)(99) \prec \text{start}_0 \succ x
\]

The process of selecting which guards are active has been effectively “calculated out” by the process of simplification. It is not clear how this could reflect back into operational semantics: There are chains of equations linking prialt guards in order, and cross-linking to other prialt guards, but there are difficult to see, even with a global overview!

However, the Hardware Semantics is interesting in its own right, as it exposes clearly how a Handel-C program is really a description of a finite state machine. It also exposed the “No-Pipeline” principle, which suggests experimenting with pipelining language constructs.
We need to complete ongoing work to fully formalise the linkages between hardware equations, RTN, the state machines and the UTP semantics. It also needs to be seen what is the full range of Handel-C laws that can be verified using this hardware semantics.

References


A Unifying Theories of Programming

We can encode state machine semantics in the Unifying Theories of Programming framework (UTP) [6]. This describes systems via alphabetised relational predicates which relate the values of observational variables before some action to their values after the action has run. This framework has been used to model sequential programs, concurrent process formalisms such as CSP, logic programming, and implementation techniques such a assembly language, among others.

Given a state machines with input: \( \text{in} : I \), output \( \text{out} : O \), state \( \text{test} : S \), next-state function \( \text{ns} : I \rightarrow S \rightarrow S \) and output function \( \text{ns} : I \rightarrow S \rightarrow O \), we establish our UTP observation variables as being: the inputs seen so far: \( \text{ins} : \text{seq} I \); the current state: \( \text{st} : S \) and the outputs generated to date: \( \text{outs} : \text{seq} O \). A State machine
$\text{init}(s_0) \triangleq \text{ins}' = \langle \rangle \land s' = s_0 \land \text{outs'} = \langle \rangle$

$M_1; M_2 \triangleq \exists \text{ins}_0, \text{st}_0, \text{outs}_0 \bullet$

$M_1[\text{ins}_0, \text{st}_0, \text{outs}_0/\text{ins}', \text{st}', \text{outs}']$

$\models \text{ins}' = \text{ins} \land \text{st}' = \text{st} \land \text{outs}' = \text{outs}$

$\text{step}_{\text{ns}, \text{op}}(i) \triangleq \text{ins}' = \text{ins} \cap \langle i \rangle \land \text{st}' = \text{ns}(i)\text{st} \land \text{outs}' = \text{outs} \cap \langle \text{op}(i)\text{st} \rangle$

$\text{run}_{\text{ns}, \text{op}}(\langle \rangle) \triangleq \models$

$\text{run}_{\text{ns}, \text{op}}(i : \text{is}) \triangleq \text{step}_{\text{ns}, \text{op}}(i); \text{run}_{\text{ns}, \text{op}}(\text{is})$

Fig. A.1. UTP State-Machine Predicates

The predicate relates the values of these variables before a run (ins, st, outs) to their values once the run has completed (ins', st', outs').

We define four actions on state-machines as UTP predicates in Figure A.1.

The notation $P[a, b, c/x, y, z]$ denotes the simultaneous substitution of $a$, $b$ and $c$ for all free occurrences of $x$, $y$ and $z$ respectively.

The predicate $\text{init}(s_0)$ indicates that after a state-machine is initialised, its state is $s_0$ and the input and output sequences are empty, regardless of prior values. Sequential composition (;) is straight from standard UTP theory[6], and here is given a definition tailored to state-machine observables. The predicate $\equiv$ (Skip) simply describes a situation where nothing happens — it is useful as an identity for sequential composition. The predicate $\text{step}_{\text{ns}, \text{op}}(i)$ describes the effect of stepping a state-machine over one input $i$. The predicate $\text{run}_{\text{ns}, \text{op}}(\text{is})$ describes the effect of stepping a state-machine over the input sequence $\text{is}$. It is defined in terms of skip, step and sequential composition. We can easily show the following laws to hold true:

$\equiv; P \equiv P \equiv P; \equiv$

$\text{run}_{\text{ns}, \text{op}}(\text{is}_1); \text{run}_{\text{ns}, \text{op}}(\text{is}_2) \equiv \text{run}_{\text{ns}, \text{op}}(\text{is}_1 \cap \text{is}_2)$