An Operational Semantics for Handel-C

Andrew Butterfield\textsuperscript{1,2}

\textit{Department of Computer Science}

\textit{Trinity College, University of Dublin}

\textit{Ireland}

Jim Woodcock\textsuperscript{3}

\textit{Computer Laboratory}

\textit{University of Kent at Canterbury,}

\textit{United Kingdom}

Abstract

We describe an operational semantics for the hardware compilation language Handel-C\textsuperscript{4} [10], which is a C-like language with channel communication and parallel constructs which compile down to mainly synchronously clocked hardware. The work in this paper builds on previous work describing the semantics of priority within Handel-C [9] and a denotational semantics for part of the language [8]. We describe a key subset of the language and show how a design decision for the real language, namely that default guards in \texttt{pria}lt\texttt{-statement} executed in “zero-time”, has consequences for the complexity of the operational semantics. We present the operational semantics, indicating clearly how it interfaces with the priority semantics of [9]. We then describe a notion of observational equivalence, and present an example illustrating how we handle the complexity of nested \texttt{pria}lt\texttt{s} in default guards.

1 Introduction

Handel-C\textsuperscript{4} [10] is a language originally developed by the Hardware Compilation Group at Oxford University Computing Laboratory, and now marketed by Celoxica Ltd. It is a hybrid of CSP [16] and C, designed to target hardware implementations, specifically field-programmable gate arrays (FPGAs) [23]. The language has sequential and parallel constructs and global variable

\textsuperscript{1} Thanks to Dean of Research Fund, TCD

\textsuperscript{2} Email: Andrew.Butterfield@cs.tcd.ie

\textsuperscript{3} Email: J.C.P.Woodcock@ukc.ac.uk

\textsuperscript{4} Handel-C is the registered trademark of Celoxica Ltd (www.celoxica.com)

\textcopyright 2004 Published by Elsevier Science B. V.
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 [22]. There is a clear need for 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.

This papers describes the current state of work being done to provide Handel-C with a formal operational semantics. We also discuss language features which prove to be problematical from an operational semantics perspective.

2 Previous and Related Work

Early work on the formal semantics of Handel-C concentrated on a subset of the language that did not contain the prialt construct. This initial exploratory work focussed on modelling the flow of control in Handel-C constructs by tracking how “execution pointers” were created, modified and merged as parallel threads were forked, executed and terminated. This led to a formal model of a Handel-C flow-of-control “interpreter” [7].

Work then started on exploring a denotational semantics for the same language subset, which was non-trivial as we had to deal with concurrency, message-passing, shared variables and the synchronous clock. Eventually, results based on branching sequences of functions mapping environments to environments were obtained [6].

At this point it became clear that prialt would have to be included. It cannot be simulated using ordinary communication and switch statements, and it has a number of effects on the overall semantics. Priority in CSP-like concurrent processes is very difficult to treat formally [15,19,20,21]. When occam [3] was developed, the only aspects of the language for which no formal semantics was provided were the constructs (PRI PAR and PRI ALT) which involved priority. One of the issues is that the asynchronous nature of CSP-like formalisms, make it very difficult to establish when prialts are “coming together”, in order for their communication requests to be resolved. However, in Handel-C, the presence of a synchronising clock makes it very easy to establish which prialts are active at any given moment, so priority handling amounts to a static problem of resolving priorities of a known collection of prialts.

In the broader arena of process algebras in general, there has been considerable work done on priorities [11,13,12]. In [12] there is an overview of this area, but there is no close match between any of the priority schemes...
described therein, and that which features in the semantics of Handel-C.

A formal description of prialt resolution without consideration of default clauses was presented in [9]. At this stage, it had become clear that prialt resolution could be treated orthogonally to most other semantic issues, and so the denotational semantics was reworked slightly to make use of the prialt semantics [8].

Other work involving formal techniques and Handel-C has been reported, and includes the use of the Ponder policy specification language [14] as a basis for implementing firewalls [22], as well as techniques for performing behavioural transformations from Haskell programs into Handel-C implementations [1]. Beyond the scope of Handel-C, there is considerable work on using formal techniques to develop safety-critical embedded systems, of which the languages Esterel [5,4,24] and Lustre [18,2] are two key examples.

3 Scope of Handel-C Semantics

Given that our final aim is a formal semantics of a real language which was itself not formally designed, we are developing our semantic framework in a manner that allows us to separate concerns as much as possible. In particular, we see the final semantics of Handel-C as having four loosely coupled components:

**types** Handel-C has a range of datatypes, all of which ultimately reduce down to specifications of bit strings of fixed length. The underlying type theory is fairly straightforward.

**synchronous “cores”** These are regions of hardware under the control of a single clock, and constitute the primary area of concern of this paper. Here we present an operational semantics for the behaviour of these cores.

**priority** The communication constructs are provided in the form of prialt-statements, which requires all choices between communication events to be prioritised.

**asynchronous “environment”** The synchronous cores communicate with each other and the external environment via asynchronous interfaces.

These four areas can be treated separately to a large degree, as the interfaces between them are simple in character. For example, the asynchronous interfaces in Handel-C involve “bus interface” constructs, rather than channels. This means that priority information and decision-making does not cross the boundaries between synchronously clocked regions.

We shall present a shorthand form, in a mathematical style, of the syntax of the Handel-C subset with which we are currently concerned. A Handel-C program \((p)\), as written, contains the following statements: a unit delay

\[^5\] Indeed it could be argued that it is the simplicity of these interfaces that has contributed to the success of Handel-C.
(1); assignment \((v := e)\); selection \((s \triangleright [p_i])\); iteration \((b * p)\); sequential composition \((p_1; p_2)\); parallel composition \((p_1 \| p_2)\); and \(\text{prialt} \ (\langle g_i : p_i \rangle)\). Here \(v\) denotes a variable, while \(e, s\) and \(b\) denote expressions, and \(g\) denotes a guard, which we describe in more detail later on. We allow the use of 0 in \(\text{prialt}\)s to indicate an empty continuation process. The language is extended with the following statements not available to programmers, namely a zero delay \((0)\) now as a fully fledged process; an ordered guard request statement \((+\langle g_i \rangle)\); and a guard action statement \((\text{act}(g))\). We also extend the language expression syntax to include two functions on guard-lists: waiting \((w\langle g_i \rangle)\) and active \((a\langle g_i \rangle)\), as well as a function on channel identifiers \((d(c))\) returning the current value being communicated on that channel.

\[
e, s, b \in E = \text{normal expression} \mid w\langle g_i \rangle \mid a\langle g_i \rangle \mid d(c)
\]

\[
p \in P ::= 1 \mid v := e \mid s \triangleright [p_i] \mid b * p \mid p_1; p_2 \mid p_1 \| p_2 \mid \langle g_i : p_i \rangle \mid 0 \mid +\langle g_i \rangle \mid \text{act}(g)
\]

We have a mix of parallel processes and global shared variables, so Handel-C has a restriction which states that no variable should ever be assigned to by two different processes during one clock cycle. It is allowable to have different processes write to the same variable on different clock cycles. A \(\text{prialt}\)-statement can be viewed as a sequence of guard-process pairs, where the guards are either communication actions like input \((c?v)\) or output \((c!e)\) or a default guard to be activated if no communication guard is active. The default guard if present must be the last element of the sequence, and we shall denote it by the shorthand \(!?\).

\[
g \in G ::= c?v \mid c!e \mid !?
\]

The sequence of guards in a \(\text{prialt}\) denotes that \(\text{prialt}\)’s priority preference, considered as relative priority — it i.e. it prefer its first guard to its second, its second to its third, and so on. A key restriction imposed by Handel-C is that during any clock-cycle, all the relative priorities of all \(\text{prialt}\)sexecuting during that cycle must be consistent with one another in that no priority cycles are introduced when all their preferences are merged. We will eventually replace all \(\text{prialt}\)s of the form: \(\langle g_i : p_i \rangle\) by the following equivalent program:

\[
+\langle g_i \rangle ; w\langle g_i \rangle * (1 ; +\langle g_i \rangle) ; a\langle g_i \rangle \triangleright [\text{act}(g_i) ; p_i]
\]

This captures the notion that a \(\text{prialt}\) acts in three stages: (i) it submits a request \((+\langle g_i \rangle)\); (ii) it waits until it become active, re-submitting the request on every clock cycle \((w\langle g_i \rangle * (1 ; +\langle g_i \rangle))\); and (iii) once waiting is over, selects and executes the active guard and process \((a\langle g_i \rangle \triangleright [\text{act}(g_i) ; p_i])\). The only uses of the request and action statements, as well as the wait, active and channel expressions are as a result of translating \(\text{prialt}\)s as just described.
4 Overview of prialt Semantics

We present here a brief overview of the prialt semantics presented in [9], with an explanation of how it can be extended to cover default clauses, and interfaced with the operational semantics described later on in this paper. If a prialt-statement is starting execution during the current clock cycle, then its guards are viewed as a prioritised sequence of requests, and if a guard \((g_i)\) for instance is deemed to be enabled or active, then it carries out its communication action during the current clock cycle. Its continuation process \((p_i)\) then starts execution at the beginning of the next clock cycle. If no communication guard is deemed active, and the prialt has a default guard \((\text{!?})\), then the corresponding continuation process \((p_n)\) starts execution immediately, i.e. during the current clock cycle. This immediate execution of a default guard’s continuation process is a major reason for the complexity of the operational semantics to be presented.

In any given clock cycle, there will be zero or more prialts commencing execution. A guard is deemed to be potentially active if elsewhere there is a complementary guard in some other prialt active during the same clock cycle. The process of determining which potentially active guards, if any, are actually active, is called Resolution. If a prialt has no active guards after resolution, and no default guard, then it is blocked, and lies idle for the rest of the current clock cycle. It then proceeds to resubmit its request in the next clock cycle. This either continues until some other request elsewhere leads to it becoming active, or runs forever if no such other request ever materialises.

Let us consider two examples, the first straightforward, the second involving default clauses in that manner that causes most semantic difficulty. The first example has three prialts executing in parallel:

\[
\langle a!1 : 0, b!2 : 0 \rangle \parallel \langle c!3 : 0, d?x : 0 \rangle \parallel \langle e!5 : 0, b?y : 0, c?z : 0 \rangle
\]

The overall priorities being expressed here can be summarized as: \(\{a, e\} < b < c < d\) where lesser values denote higher priorities. Only channels \(b\) and \(c\) are potentially active, and \(b\) has higher priority, and so will be actually active. The result is the first and third prialts execute, transferring value 2 across channel \(b\) to variable \(y\). The second prialt remains blocked, waiting on either \(c\) or \(d\) (preferring \(c\)).

The second example has two prialts in parallel, with the second having a default clause which itself is a prialt:

\[
\langle c!7 : 0 \rangle \parallel \langle d!0 : 0, \text{!?} : \langle c?v : 0 \rangle \rangle
\]

Initially we have a situation where there are no potentially active guards, so the first prialt blocks, while the second immediately activates its default clause. This introduces another prialt to the mix, and now channel \(c\) becomes active, transferring value 7 across channel \(c\) to variable \(v\). It is worth pointing out
that the above process is in fact equivalent to \(<c!7:0 || d!0:0, c?v:0⟩\), but that this simplifying law does not extend to arbitrary continuation processes. This example is used later to illustrate the operational semantics at work.

In [9] resolution is viewed formally as a process \((\text{Resolve})\) which takes a set of \(\text{Prialt Requests}\), and returns a pair called a \(\text{Resolution}\), consisting of an \(\text{Channel Map}\) and the set of requests that have remained blocked (no active communication guards).

\[
\text{Resolve} : \mathbb{P}\text{Prialt} \rightarrow (\text{Ch} \xrightarrow{m} \mathbb{P}\text{Prialt}) \times \mathbb{P}\text{Prialt}
\]

The Channel Map maps active channels to the set of requests invoking that channel which have been deemed active. A Prialt Request is simply modelled as a sequence of guards, i.e. simply as the corresponding \text{prialt}-statement with the continuation processes stripped out. A channel is deemed potentially active if it occurs in a number of \text{prialt}s, in both input and output forms. It becomes active if no other potentially active channels in its own \text{prialt}s are of higher priority. The formal semantics in [9] does not deal with default clauses, nor elaborate on how the resolution structure interfaces to the formal semantics of the rest of the language. Briefly put, the default clauses are simply handled by scanning the set of blocked requests for any that have default clauses, and simply moving them into the channel map, under a pseudo-channel called “default”.

The resolution of our first example would have got the following input:

\[
\{ \langle a!1, b!2 \rangle, \langle c!3, d?x \rangle, \langle e!5, b?y, c?z \rangle \}
\]

The resulting output would have been:

\[
( \{ b \mapsto \{ \langle a!1, b!2 \rangle, \langle e!5, b?y, c?z \rangle \}, \{ \langle c!3, d?x \rangle \} \})
\]

The rest of the interfacing is handled by three observer functions: \(W\), \(A\) and \(D\). These are represented in the (extended) syntax of Handel-C by the expressions \(w(g_i)\), \(a(g_i)\) and \(d(c)\), respectively.

\textbf{Wait} The function \(W[w(g_i)]\) takes a Resolution as argument and returns a boolean result which is \text{TRUE} if the corresponding request \(\langle g_i \rangle\) is blocked.

\textbf{Active} The function \(A[a(g_i)]\) takes a Resolution as argument and returns an integer result which indicates the index of the guard in \(g_i\) which is active. It is only defined if \(W[w(g_i)]\) returns \text{FALSE}.

\textbf{Data} The function \(D[d(c)]\) takes a Resolution as argument and returns the expression being outputted onto channel \(c\). It is only defined if channel \(c\) is in the domain of the channel map part of the Resolution, i.e. is active.
5 Operational Semantics

The operational semantics for Handel-C requires us to have an understanding of the processing that takes place during a clock-cycle. First, the atomic actions (assignment or communication) for the current clock cycle need to be determined. This involves following the flow of control from the statements which executed on the previous clock cycle, requiring the evaluation of conditional and while-loop guard expressions. We shall refer to this as the “select” (\(sel\)) phase. At this point we have identified almost all of the assignments and \(\text{prialt}\)s which are about to execute. The next phase, the “request” phase (\(req\)) involves all the selected \(\text{prialt}\)s lodging their corresponding requests with the environment. At the end of this phase, the system has global knowledge of almost all the communication requests associated with this clock cycle. The third, or “resolve” phase (\(res\)), is were the system resolves all these requests, as described in section 4 previously. The outcome of this is that we know which communication actions are going to be executed. The final or “action” phase (\(act\)) is were all the atomic assignment and active communication actions take place, all globally synchronised to the clock. It is at this stage that permanent changes are made to the system state — i.e. ones that persist across clock boundaries. The first three phases are all implemented as combinatorial logic, and are deemed to execute in “zero-time”. In other words, this logic makes these decisions well within the time allocated to the clock cycle. The last phase is effectively synchronised to the end of the cycle, and so is deemed to have taken one clock cycle to execute.

The sequencing of the phases is very important, as the outcome of resolution is not well-defined until the selection and request phases are complete. As resolution determines the final communication actions and their consequent permanent state changes, is important to get this sequencing right. In the hardware implementation, this is achieved by letting the resulting combinatorial logic have enough time to settle into a consistent state, and aided by the fact that the logic contains no “combinatorial” cycles. In our formal semantics we ensure the appropriate ordering of decisions by modelling these conceptual phases explicitly.

We now explain the frequent use in the previous few paragraphs of the phrase “almost all”. The problem here is the existence of default clauses in \(\text{prialt}\)s, which execute in “zero-time”. As a consequence of this, the corresponding continuation process starts execution in the current clock cycle. This continuation process can be any legal Handel-C process so may contain conditionals, parallel branches, while-loops and more \(\text{prialt}\)s. This means that we have to redo the select, request and resolve phases. The \(\text{prialt}\)s can be nested to an arbitrary but finite depth in this fashion (via default guards). Given a \(\text{prialt}\) nesting depth of \(d\), in general we need to iterate the first three phases \(d + 1\) times. This complication is a consequence of the decision that default guards take zero-time. The presence of arbitrary Handel-C processes
Butterfield and Woodcock

as continuations after communication guards is not a problem, as they don’t start execution until the subsequent clock cycle.

So, our operational semantics needs to model these four phases by
(i) classifying statements according to the phase in which they participate
(ii) providing transitions of different types for each phase
(iii) providing special transitions to manage the switch between phases.

5.1 Transition Types

A Transition type (TType) is one of state-selection (sel), comms-request (req), comms-resolution (res) or state update action (act), ordered as just listed.

\[ TType \triangleq \{ \text{sel, req, res, act} \} \quad \text{sel < req < res < act} \]

We classify statements by associating a transition type set (tts(p)) with them, and use ttype(p) to denote min(tts(p)):

<table>
<thead>
<tr>
<th>Statements</th>
<th>tts</th>
</tr>
</thead>
<tbody>
<tr>
<td>0, s \rightarrow [p_i], b * p, act(!?)</td>
<td>{ sel }</td>
</tr>
<tr>
<td>\langle g_i : p_i \rangle, +\langle g_i \rangle</td>
<td>{ req }</td>
</tr>
<tr>
<td>a\langle g_i \rangle \rightarrow [p_i], w\langle g_i \rangle * p</td>
<td>{ res } (w is false)</td>
</tr>
<tr>
<td>1, v := e, act(c!e), act(c?v), w\langle g_i \rangle * p</td>
<td>{ act } (w is true)</td>
</tr>
<tr>
<td>p_1;p_2</td>
<td>tts(p_1)</td>
</tr>
<tr>
<td>p_1 ∥ p_2</td>
<td>tts(p_1) \cup tts(p_2)</td>
</tr>
</tbody>
</table>

The anomalous situation regarding the classification of \( w\langle g_i \rangle * p \) is explained when the transition rules are described. In addition to the four transition types mentioned above, we have special transitions between those types: sel2req, req2res, res2sel, res2act, and act2sel (a.k.a tick).

The system state is larger than just \( p \) and \( t \), comprising a tuple \( (p, t, \rho, \gamma, R, \tau) \) where: \( p : P \) is the process state; \( t : TType \) is the current transition type; \( \rho : \text{id} \rightarrow \text{Val} \) is the variable environment; \( \gamma : \text{Ch} \rightarrow G^+ \) are the active channels; \( R : \mathbb{P}(G^+) \) are the requested/blocked priorities; and \( \tau : N \) is the current clock value. Note that \( R \) holds the resolution input data during the req phase, and \((\gamma, R)\) holds the resolution result obtained during the res phase.

Given an initial program \( p_0 \), the initial system state is \( (p_0, \text{sel}, \theta, \theta, 0, 0) \) and a program has terminated if \( p \) ever reduces to \( 0 \). To get the terminating clock transition we shall strengthen this to say that a program terminates once \( p \) is \( 0 \) and \( t \) is \text{sel}.
5.2 Transition Events

We associate events with some transitions, denoting a transition of type $t$ engaging in event $e$ by $\frac{t}{e}$, except for the special transitions were the event is fixed. All events describe some form of system state change, in addition to that already described above, as detailed below:

<table>
<thead>
<tr>
<th>transition</th>
<th>event notation</th>
<th>changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$req$</td>
<td>$+\langle g_i \rangle$</td>
<td>$\mathcal{R} \cup { \langle g_i \rangle }$</td>
</tr>
<tr>
<td>$act$</td>
<td>$x := e$</td>
<td>$\rho \uparrow { x \mapsto \varepsilon }$</td>
</tr>
<tr>
<td>$req2res$</td>
<td>$(\gamma, \mathcal{R}) := \text{Resolve}(\mathcal{R})$</td>
<td></td>
</tr>
<tr>
<td>$act2sel$</td>
<td>$(\gamma, \mathcal{R}, \tau) := (\theta, \emptyset, \tau + 1)$</td>
<td></td>
</tr>
</tbody>
</table>

Here we use the notation $\varepsilon$ as shorthand for $[e]_\rho$, i.e the evaluation of expression $e$ w.r.t. the environment $\rho$. If any variable $v$ in $e$ is not in the domain of $\rho$ (i.e not yet assigned to), the the lookup $\rho(v)$ returns zero, as this is how Handel-C initialises all variables. Note also that the $act$ action can be an empty simultaneous assignment, denoted by leaving it blank.

Given $(\gamma', \mathcal{R}') = \text{Resolve}(\mathcal{R})$ we always have that

$$\mathcal{R}' \cup \text{dom} \gamma' = \mathcal{R} \quad \mathcal{R}' \cap \text{dom} \gamma' = \emptyset$$

Formally we can view all events as functions on global state:

$$Evt \cong \text{State} \rightarrow \text{State}$$

$$[\_[.] : P \rightarrow Evt$$

$$[x := e]_{(\_, \rho, \_)} \cong (\_, \rho \uparrow \{ x \mapsto \varepsilon \}, \_)$$

$$[+\langle g_i \rangle]_{(\_, \mathcal{R}, \_)} \cong (\_, \mathcal{R} \cup \{ \langle g_i \rangle \}, \_)$$

In some rules, we merge events, using the event merge operator $\Diamond$, which is associative and commutative, with the null event as identity. In effect $\Diamond$ merges disjoint assignments into a simultaneous assignment and multiple prialt requests into one large request with a set of the prialts involved:

$$\Diamond \cong Evt \times Evt \rightarrow Evt$$

$$e_1 \Diamond e_2 \cong e_1 \circ e_2 \text{ or } e_2 \circ e_1$$

$$[x := e] \Diamond [y := f] = [x, y := e, f]$$

$$( [+\langle g_i \rangle] \Diamond [+\langle g'_j \rangle])_{(\_, \mathcal{R}, \_)} = (\_, \mathcal{R} \cup \{ \langle g_i \rangle, \langle g'_j \rangle \}, \_)$$

If assignments are not disjoint, then $\Diamond$ is not defined, but this matches Handel-C which outlaws this situation.
5.3 Transition Conditions

The conditional expressions \((s, b)\) associated with the select and while constructs are evaluated w.r.t. the environment \(\rho\). We use the same shorthand

\[ s \overset{\rho}{=} [s]_\rho \]

as is used for assignment rhs expressions.

When we evaluate the wait predicate \(w(g_i)\) and the active guard selector expression \(a(g_i)\), we do so with respect to the \(\Re\) and \(\gamma\) state components, using two of the observer functions as follows:

\[ w(g_i) \overset{\gamma, \Re}{=} \mathcal{W}[w(g_i)](\gamma, \Re) \]

\[ a(g_i) \overset{\gamma, \Re}{=} \mathcal{A}[a(g_i)](\gamma, \Re) \]

Note that a consequence of these definitions and the properties of Resolve’ mentioned above is that \(a(g_i)\) is defined only when \(w(g_i)\) is false.

We evaluate the data associated with channel identifier \(d(c)\) by using the remaining observer function to get the channel expression \(e\), and then evaluating that in the variable environment in the usual way:

\[ d(c) \overset{\gamma, \Re}{=} e \text{ where } e = D[d(c)](\gamma, \Re) \]

5.4 The Transition Relation

Given a state pair consisting of program \((p)\) and transition type \((t)\), we can define conditions when transitions are enabled:

<table>
<thead>
<tr>
<th>transition</th>
<th>enable condition</th>
<th>changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>sel</td>
<td>(t = sel \land \text{ttype}(p) = sel)</td>
<td>(p \xrightarrow{\text{sel}} p')</td>
</tr>
<tr>
<td>sel2req</td>
<td>(t = sel \land \text{ttype}(p) &gt; sel)</td>
<td>(t := \text{req})</td>
</tr>
<tr>
<td>req</td>
<td>(t = \text{req} \land \text{ttype}(p) = \text{req})</td>
<td>(p \xrightarrow{\text{req}} p')</td>
</tr>
<tr>
<td>req2res</td>
<td>(t = \text{req} \land \text{ttype}(p) &gt; \text{req})</td>
<td>(t := \text{res})</td>
</tr>
<tr>
<td>res</td>
<td>(t = \text{res} \land \text{ttype}(p) = \text{res})</td>
<td>(p \xrightarrow{\text{res}} p')</td>
</tr>
<tr>
<td>res2sel</td>
<td>(t = \text{res} \land \text{ttype}(p) = \text{sel})</td>
<td>(t := \text{sel})</td>
</tr>
<tr>
<td>res2act</td>
<td>(t = \text{res} \land \text{ttype}(p) = \text{act})</td>
<td>(t := \text{act})</td>
</tr>
<tr>
<td>act</td>
<td>(t = \text{act} \land \text{act} \in \text{tts}(p))</td>
<td>(p \xrightarrow{\text{act}} p')</td>
</tr>
<tr>
<td>act2sel</td>
<td>(t = \text{act} \land \text{act} \notin \text{tts}(p))</td>
<td>(t := \text{sel})</td>
</tr>
</tbody>
</table>

The priorities introduced by the \texttt{prialt}s are effectively “collected” together during the request phase, and then resolved all together, a process simplified
Butterfield and Woodcock

SEQ \[ p \xrightarrow{e} p' \Rightarrow p \parallel q \xrightarrow{e} p' \parallel q \]

PAR-1 \[ p \xrightarrow{e} p' \Rightarrow p \parallel q \xrightarrow{e} p' \parallel q \]

PAR-2 \[ p \xrightarrow{e} p' \Rightarrow q \parallel p \xrightarrow{e} q \parallel p' \]

MERGE-PAR \[ p \xrightarrow{e} p', q \xrightarrow{f} q' \Rightarrow p \parallel q \xrightarrow{e \in \delta f} p' \parallel q' \]

by the fact that no priority conflicts arise. None of the prialt priorities are attached to transitions or states an any obvious way, and are, as asserted earlier, treated orthogonally to the transition system being described here. Here however, we have present a second notion of priority, notably that between different types of transitions in our transition system. These priorities are similar in character to the static and global priorities of [12], except that while initially sel transitions have highest priority, and act transitions have lowest, we have a dynamic priority switch at the end of the clock cycles when act transitions become highest priority. Apart from this, the theory of [12] is not applicable here as we have no silent actions and no synchronisation between most transitions of any given type/priority.

Rules have the form:

\[
\text{NAME } [\text{cond}] \quad \text{antecedents } \Rightarrow \text{consequent}
\]

where \text{cond} and \text{antecedents}⇒ are optional. The consequent transition is enabled if and only if: (i) its transition type is enabled by the global state; (ii) the condition, evaluated against the global state, is also true; and (iii) the antecedent transitions, if any, are themselves enabled.

We first present general transition rules dealing with sequential and parallel composition, where the phase is irrelevant (Figure 1).

The basic phase-specific transition rules are shown in Figure 2, presented largely in order of transition type. The first set of rules dealing with sel transitions are PAR-END, SELECT, WHL-FALSE, WHL-TRUE, and NEXT. The rules for the conditional and while constructs are straightforward. The rules for the parallel and sequential constructs manage the fact that all parallel threads must end before a subsequent process can start execution. The 0 construct (the process that does nothing in “zero-time”) pays a key role in managing this synchronisation.

The next set of rules dealing with req transitions are PRIALT and REQUEST. The prialt-statement simply lodges its request (\(\langle g_i \rangle\)) and then evolves into the sequential composition of a wait loop and a subsequent conditional to select the active action. The wait loop resubmits the request in following clock cycles, which is why the loop body starts with 1, i.e. to force a wait until the next clock cycle. The request transition deals with the additional requests lodged by the wait loop.
The third set of transitions handle transitions which occur when evaluating the resolution observer functions $w\langle g_i \rangle$ and $a\langle g_i \rangle$. Most of these are $\text{res}$ transitions, except for $\text{Wait}$, which is an $\text{act}$ transition. This is to ensure these transitions, which initiate a one cycle wait, followed by resubmission of their request, are held over until all default clauses have been handled. If this rule was a $\text{res}$ transition, we would have a race between its firing, and the firing of a $\text{Continue}$ rule, which may be as a result of a default guard elsewhere. The result of that default guard’s execution might be a change in the status of a blocked $\text{prialt}$. It is very important that we ensure that all $\text{Continue}$ transitions occur before any $\text{Wait}$ transitions are allowed to proceed.

The final set of transitions are the $\text{act}$ transitions, all of which change the state. The delay statement state change is simply that it lets time pass. The assignment statement has the expected effect of updating the relevant variable. The output statement acts like a delay, while the input statement uses the resolution observable $d(c)$ to update its variable value.

Fig. 2. Phase-Specific Transition Rules
5.5 Observational Equivalence

An execution trace of a Handel-C program is a sequence of transitions occurring with the following structure:

$((sel^*; req^*; res^*)^*; act^+) +$

where ; denotes concatenation, and $x^*$ and $x^+$ denote zero-or-more $x$s and one-or-more $x$s, respectively. As the subsequences of act transitions can be collapsed into one such transition, using the $\Diamond$ operator, the overall effect is that of a single act transitions, one per clock cycle, separated by some mix of other transition types:

$((sel^*; req^*; res^*)^*; act)^+$

As the key function of these other transition types is to determine what act transitions occur, and as only the act transitions have persistent effects on the state, we shall choose to ignore the other transition types. Hence we shall declare two execution traces to be Observationally Equivalent if they are identical once non-act transitions are removed, and all act transitions in one clock cycle have been merged.

5.6 An example

To show the importance of requiring WAIT to be a act transition, consider the following example:

$\langle c!7 : 0 \rangle \parallel \langle d!0 : 0, !? : \langle c?v : 0 \rangle \rangle$

For brevity we shall omit 0 when it occurs either before or after “;”. Initially it engages in PRIALT transitions to become:

$w\langle c!7 \rangle * (1; +\langle c!7 \rangle); a\langle c!7 \rangle \triangleright [act(c!7)]$

$\parallel$

$w\langle d!0, !? \rangle * (1; +\langle d!0, !? \rangle); a\langle d!0, !? \rangle \triangleright [act(d!0), \langle c?v : 0 \rangle]$

If WAIT was a res transition then the top process could evolve at this point to something which would start with $1; +\langle c!7 \rangle$. This process would now wait until the mode was act, to do a DELAY transition. In particular, if events elsewhere suddenly made channel $c$ active in this clock cycle, this process would miss the event because it is no longer “listening”. Instead, the WAIT transition is blocked, but the lower process can do a CONTINUE transition to become:

$a\langle d!0, !? \rangle \triangleright [act(d!0), \langle c?v : 0 \rangle]$
A **Select-Active** transition will pick the default guard to give:

\[ \langle c?v : 0 \rangle \]

This **prialts** will then trigger another iteration of the **sel-req** transition to leave us in the following situation:

\[
\begin{align*}
   w\langle c!7 \rangle & \ast (1; +\langle c!7 \rangle); a\langle c!7 \rangle \triangleright [\text{act}(c!7)] \\
   \parallel \\
   w\langle c?v \rangle & \ast (1; +\langle c?v \rangle); a\langle c?v \rangle \triangleright [\text{act}(c?v)]
\end{align*}
\]

Now both side are enabled for **CONTINUE** transitions, so we get

\[
\begin{align*}
   a\langle c!7 \rangle & \triangleright [\text{act}(c!7)] \\
   \parallel \\
   a\langle c?v \rangle & \triangleright [\text{act}(c?v)]
\end{align*}
\]

**Select-Active** transitions reduce this to

\[ \text{act}(c!7) \parallel \text{act}(c?v) \]

Finally the **INPUT** and **OUTPUT** transitions fire, \( v \) is assigned 7, the clock is advanced, and we get \( 0 \parallel 0 \) which reduces in the next clock cycle to \( 0 \).

It is interesting to note that a **prialts** with a default guard will never wait so we can omit the wait-loop giving the law:

\[ \langle g_i : p_i,!? : p_n \rangle = +\langle g_i,!? \rangle; a\langle g_i,!? \rangle \triangleright [\text{act}(g_i); p_i, p_n] \]

### 6 Conclusions and Future Work

We have presented an operational semantics for a key subset of the Handel-C hardware compilation language. In particular our semantics handles **prialts** properly, coping with the issues raised by the zero-time execution behaviour of default clauses. Another key feature of the work presented here is that it exploits a clean separation of concerns between various key aspects of the language: types, priority, flow of control and asynchronous interfacing.

The key goal of this work is to have a formal development methodology from a suitable specification notation down to Handel-C code, with appropriate tool support. We envisage a methodology based on refinement, using an appropriate refinement calculus. To reach this goal we need to achieve a number of key milestones: an appropriate formal model of the asynchronous
interfaces, and the type system; the integration of all aspects into a unified whole; an appropriate generalisation of Handel-C to produce a specification language; and the development of refinement laws that enable correct programs to be developed. Given the need to unify the four aspects, we propose to formulate the semantics of both Handel-C and its specification language within the unifying theories framework [17]. This will facilitate refinement, as a refinement calculus (Circus) for this paradigm has been developed [25].

References


