Denotational Semantics of Handel-C Cores

Andrew Butterfield
Trinity College, University of Dublin
Dublin, Ireland
Andrew.Butterfield@cs.tcd.ie

Jim Woodcock
University of York
York, England
Jim.Woodcock@cs.york.ac.uk

ABSTRACT
We present a denotational semantics for a fully functional subset of the Handel-C hardware compilation language [9], based on the concept of typed assertion traces. We motivate the choice of semantic domains by illustrating the complexities of the behaviour of the language, paying particular attention to the priorit (priority-alternation) construct of Handel-C. We then define the typed assertion traces over an abstract notion of events, which we then instantiate as state-transformers. The denotational semantics is then given and some examples are discussed.

Categories and Subject Descriptors
F.3.2 [Logic and Meanings of Programs]: Semantics of Programming Languages—denotational semantics; B.5.2 [Register-Transfer Level Implementation]: Design Aids—hardware description languages, verification; B.6.3 [Logic Design]: Design Aids—hardware description languages, verification

General Terms
Languages, Theory

Keywords
hardware compilers, synchronous cores

1. INTRODUCTION
This paper describes a denotational semantics for Handel-C which gives a program a meaning as a set of “Typed Assertion Traces”.

Handel-C\(^1\)[9] 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 [15] and C, designed to target hardware implementations, specifically field-programmable gate arrays (FPGAs) [21]. 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 [20]. 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.

We see the final semantics of Handel-C as having four components: types; priorities; synchronous cores; and the asynchronous environment. A detailed description of these and their motivation is given in [7]. Here we simply stress that this paper is primarily concerned with the semantics of the synchronous cores, incorporating priorities. The topics of typing and the external asynchronous interface are beyond the scope of this paper.

We first introduce the language, then describe prior and related work in this area, before motivating and describing the domains used for our denotational semantics.

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

2.1 Syntax
We have variables \((x \in Var)\), and we assume the existence of an expression syntax \((e \in Exp)\) whose details need not concern us here. We also have identifiers for channels \((c \in Ch)\), and we consider all the above as having either boolean or integer type, and occasionally use \(b\) to denote a boolean-valued expression. We also have the notion of guards \((g \in Grd)\), which denote the offering and accepting of communication actions. Guards either denote a desire to perform output of an expression’s value along a channel \((c!e)\), to receive input via a channel into a variable \((c?x)\), or a skip/default guard which always succeeds (!?).

\[
g \in G ::= c?x \mid c!e \mid !?
\]
A syntax of a process $p : \text{Proc}$ is as follows:

$$
p := 0 | 1 | x := e \mid p_1 ; p_2 \mid p_1 || p_2 \mid p_1 \triangleleft b \triangleright p_2 \mid b \ast p \mid \langle g_i \rightarrow p_i \rangle
$$

We use notation like $\langle g_i \rightarrow p_i \rangle$ as a shorthand for the more explicit $\langle g_1 \rightarrow p_1, \ldots, g_n \rightarrow p_n \rangle$ where $i$ is assumed to index over $1 \ldots n$ for appropriate $n$.

### 2.2 Behaviour

We can briefly summarise the behaviour of a Handel-C process as follows:

- $0$ does nothing, in zero time;
- $1$ does nothing, but takes one clock cycle to do it; $x := e$ assigns the value of $e$ into $x$, taking one clock cycle; ($p_1 ; p_2$) first executes $p_1$, and once it has terminated immediately starts $p_2$; ($p_1 || p_2$) runs both $p_1$ and $p_2$ in lock-step parallel, terminating when they have both finished; ($p_1 \triangleleft b \triangleright p_2$) evaluates $b$ and executes $p_1$ immediately if $b$ is $\text{True}$, otherwise it runs $p_2$; and $b \ast P$ tests $b$ and if $\text{True}$ it runs $P$ and then repeats, otherwise it terminates.

The $(g_i \rightarrow p_i)$ construct ("prialt") is an ordered sequence of guard-process pairs. Guards are either communication actions or a default guard to be activated if no communication guard is active. The default guard, if present, may only be in the last element. The sequence of guards in a prialt denotes that prialt’s priority preference, considered as relative priority — i.e. it prefers its first guard to its second, its second to its third, and so on.

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 then the corresponding process is executed. 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 vice versa. The default guard $(?!)$ is always enabled. The input $(c?x)$ and output $(c!e)$ guards perform their actions taking one clock-cycle, while the default guard $(?!)$ acts in “zero-time”, so the subsequent process starts execution immediately. It is this “instant” execution of $?!$ guards that so complicates the formal semantics of Handel-C, as discussed extensively in [8].

### 2.3 Restrictions

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. The Handel-C language reference manual [9] states that different parallel processes generally should not write to the same variable, but that if they do, the programmer has a proof obligation to show that these writes never occur during the same clock cycle.

This extends to disallowing the simultaneous writing of two different values to the same channel — however having multiple readers of a channel at any one time is permitted. Another key restriction imposed by Handel-C is that during any clock-cycle, all the relative priorities of all prialts executing during that cycle must be consistent with one another in that no priority cycles are introduced when all their preferences are merged.

### 3. 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. However it soon 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. A formal description of prialt resolution without consideration of default clauses was presented in [6]. An initial denotational semantics was developed [5] which incorporated this prialt resolution semantics.

The the prialt model of [6] was then extended to handle default clauses properly and an operational semantics for Handel-C incorporating this was developed [7, 8]. The operational semantics had to introduce a notion of prioritised transitions in order to correctly capture the behaviour of default guards. This additional notion of priority was completely different and orthogonal to the priorities expressed by the prialt construct.

Priority in concurrent processes is difficult to treat formally but many examples abound, in both the CSP setting [14, 17, 18, 19], and in the more general process algebra areas [10, 12, 11]. The CSP treatment either fails to handle recursion, or is too complex and general, while the more general process algebra work is closer to what is required. Unfortunately, priority in Handel-C does not fit neatly into the priority schemes that have been considered, as described in [11].

Other work involving formal techniques and Handel-C has been reported, and includes the use of the Ponder policy specification language [13] as a basis for implementing firewalls [20], 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 Es- terel [4, 3, 23] and Lustre [16, 2] are two key examples.

### 4. OVERVIEW OF PRIALT SEMANTICS

We present here a brief overview of the prialt semantics presented in [8], with an explanation of how it can be interfaced with the denotational semantics described later on in this paper.

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 guards, if any, become active, is called Resolution.

In [6] resolution is viewed formally as a function Resolve that takes a set of Prialt Requests (PriSet), and returns a pair called a Resolution (Resltn), consisting of a Channel-Prialt Map (CPMap) and the set of prialts that have remained blocked.

$$
\text{PriSet} = \mathcal{P}\text{Prialt} \\
\text{CPMap} = \text{Ch} \to \text{PriSet} \\
\text{Resltn} = \text{CPMap} \times \text{PriSet} \\
\text{Resolve} : \text{PriSet} \to \text{Resltn}
$$

A Prialt Request is simply modelled as a sequence of guards, i.e simply as the corresponding prialt-statement with the continuation processes stripped out. The Channel-Prialt
Map identifies which channels are going to be active and maps them to those prialt{s} which will participate in communication over that channel.

In order to order to model the semantics of prialt{s}, we view a clock-cycle as being composed of four phases: selection (sel); request (req); resolve (res); and action (act). During the selection phase, flow of control decisions are taken by evaluating conditions for if-statements and while-loops. During the request phase, any prialt{s} which have been selected lodge their prioritised communication requests in a central location. Once all this has occurred, the resolve phase determines which communication requests are going to be granted. In the action phase, all the assignment statements selected earlier, and all the communication actions just resolved, are carried out simultaneously. The clock tick signals the end of the action phase.

The set of prialt{s} that are input to Resolve are those lodged centrally during the request phase. Conceptually, resolution occurs at the transition between the request and resolution phases and results in the two outputs as mentioned above. During the resolution phase, the resulting channel-map and blocked prialt{-sets} are examined to determine what activities will occur.

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

\[
\begin{align*}
&\{a1:0, b12:0\} || \{c13:0, d?x:0\} || \{e15:0, b?y:0, c?z:0\}
\end{align*}
\]

The overall priorities being expressed here can be summarized as: \(a, e \prec b < c \prec d\) 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 prialt{s} execute, transferring value \(2\) across channel \(b\) to variable \(y\). The second prialt remains blocked, waiting on either \(c\) or \(d\) (preferring \(c\)). The resolution of our first example would have got the following input:

\[
\{a11, b12\}, \{c13, d?x\}, \{e15, b?y, c?z\} \}
\]

The resulting output would have been:

\[
\{b \rightarrow \{a11, b12\}, \{e15, b?y, c?z\}\}, \{c13, d?x\} \}
\]

The second example has two prialt{s} in parallel, with the second having a default clause which itself contains a statement that subsequently invokes a prialt:

\[
\{c166 \rightarrow 0\} || \{d199 \rightarrow 0, \rightarrow (b \rightarrow P; (c?x \rightarrow 0))\}
\]

Let us consider the case where \(b\) happens to be false. 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. The while-loop has a false condition, so immediately terminates, and this introduces another prialt to the mix. At this point the program has evolved to like this:

\[
\{c166 \rightarrow 0\} || \{c?x \rightarrow 0\}
\]

This requires us to lodge a new request, in with the existing ones still in place, and to re-perform the resolution step. As a result, channel \(c\) becomes active, transferring value \(66\) across to variable \(x\).

Prialt{s} nested inside default clauses of other prialt{s} may become active in the same clock cycle as those enclosing prialt{s}, which requires us to iterate the sel{-req{-res}} loop several times, in any given clock cycle. Managing this microcycle activity severely complicates the semantics.

5. Semantic Framework

The denotational semantics in [5] was based on the notion of “branching sequences” or trees, where non-branching sequences denoted deterministic sequences of events, and branching was used to model a choice point, such as the conditions of a while-loop or if-statement. The events that populate the tree are in fact state-transformers, that is functions that describe how the state changes. When branching occurred, conditions on the program state identify which branch gets taken during a given program run. Given such a tree semantics, we could then derive a trace of a given run by picking a starting state and working along the tree choosing the appropriate branches.

Given the complexities of the decision process during a time-slot, as evidenced by the operational semantics, once the full prialt semantics is included, we propose a somewhat different semantic model for the complete denotational semantics, namely on based on sets of “typed assertion traces”.

These are sets of sequences of events (state-transformers), hence “traces”, were each event is classified according to the “type” of phase in which it occurs (sel{req{res}}act), and each event has a boolean “assertion” which indicates the conditions under which that event (and all subsequent) may proceed.

The switch from a tree-based semantic model to one based on sets of traces is motivated by two considerations:

1. The tree-based model’s major complexity occurred when defining parallel combinations of branchings. In the early semantics at most one branch per clock cycle was present so this was manageable, however we are now faced with multiple choices per cycle so the tree model becomes much more complex.

2. A trace-based model brings the semantics more in line with that of CSP and particulary Timed CSP [22], which we intend to use as the basis for modelling the asynchronous environment.

5.1 Abstraction of Event, States and Predicates

We shall now present an abstract view of typed assertion traces, where events \(e : Evt\) are simply viewed as forming a commutative monoid, formed by a notion of a null event \(\emptyset\) and an event merge operator \(\circ\), which captures the result of running both events simultaneously:

\[
\begin{align*}
\emptyset \circ e &= e \\
e_1 \circ e_2 &= e_2 \circ e_1 \\
e_1 \circ (e_2 \circ e_3) &= (e_1 \circ e_2) \circ e_3
\end{align*}
\]

We also require that merging is not a group operator, so that any merge with at least one non-null event can never result in a null event

\[\text{Interestingly, the underlying hardware doesn’t iterate, as it computes what is to be active in any given clock cycle using combinatorial logic.}\]
We introduce an abstract notion of a state as something which can change as a result of events:

\[ s \in St \]

\[ \triangle : Evt \rightarrow St \rightarrow St \]

We require that the null event results in no state change:

\[ \triangle \odot (s) = s \]

We need predicates (boolean-valued functions) over states:

\[ p \in Pred \rightarrow St \rightarrow B \]

We use `true` and `false` to denote the everywhere `true` and `false` predicates respectively:

\[ true, false : Pred \]

\[ true \equiv \lambda a . True \]

\[ false \equiv \lambda a . False \]

We are going to capture the assertions by the concept of a “guarded-event” (g), which is a predicate-event pair \((p, e)\):

\[ g, (p, e) \in GE = Pred \times Evt \]

A null guarded event (\(\nabla\)) is one where the event is nil. We distinguish two instances of \(\nabla\), namely ones with everywhere-true (1) and everywhere-false (0):

\[ \pi_2 \nabla = \emptyset \]

\[ 1 \equiv (true, \nabla) \]

\[ 0 \equiv (false, \nabla) \]

We can extend the notion of merging to guarded events in the obvious way by merging the events and taking the conjunction of the predicates:

\[ (p_1, e_1) \diamond (p_2, e_2) \equiv (p_1 \land p_2, e_1 \diamond e_2) \]

### 5.2 Typed Assertion Traces

We shall view a trace as being a non-empty sequence of slots, were each slot denotes the activity during one complete clock cycle. We allow traces to be either finite or infinite, as this is required for the semantics of any of the loop constructs.

\[ \tau \in Trc = Slot^{\infty} \]

The semantics of a Handel-C program is mapped to a set of these traces, which conform to a set of healthiness conditions to be mentioned later.

Slots have internal structure, and are divided into two components: the decision events which occur early in the clock cycle to determine the course of action to take; and the permanent state-change actions which all occur simultaneously at the end of the clock cycle. The former are modelled as sequences of “microslots” (\(MS\)), whilst the latter can simply be represented as a single (merged) guarded event. We shall refer to the second component as the “action event” of the slot

\[ s, (p, a) \in Slot = MS^\ast \times GE \]

A microslot \((m)\) captures the events in one cycle of selection-request-response and hence is a triple of guarded events \((s, q, r)\), where the first \((S)\) are of type \(sel\), the second \((q)\) are of type \(req\), and the last \((r)\) are of type \(res\):

\[ m, (s, q, r) \in MS = GE^3 \]

We expect that any microslot has at least one non-null event present.

We need to be careful how traces and slots are interpreted: In essence, a slot were the action event is null denotes the case were the clock-tick which ends the slot has yet to happen. As a consequence of this interpretation, only the last slot in a trace can be “tick-free” in this manner.

We need to be able to identify a null slot, as one with no microslots

\[ () : Slot \]

\[ () \equiv ((), 1) \]

A trace in which no events, not even a clock-tick, have occurred, is denoted by a singleton sequence consisting of one null slot. The reason for not admitting empty trace sequences is that it introduces ambiguity over interpreting null traces, and complicates the definition of various concatenation operators.

We also need to identify a slot whose only event is an action event which denotes a clock-tick — we overload the notation \(!\) to denote both that event, and the corresponding slot. We also expect that merging this event with any non-null event will result in that non-null event:

\[ ! : Evt \]

\[ ! \diamond e = e, \quad e \neq \emptyset \]

\[ ! : Slot \]

\[ ! \equiv ((), (true, !)) \]

#### 5.2.1 Typing

The typing of events in slots and microslots is implicitly given by the events’ position. We can extend the notion of typing to cover both microslots and slots themselves.

Transition types fall into four categories, with an ordering as indicated:

\[ t \in TType \equiv \{ sel, req, res, act \} \]

\[ sel < req < res < act \]

We define the type of a microslot as the type of the least non-void event present:

\[ tType_MS : MS \rightarrow TType \]

\[ tType_MS(\nabla, \nabla, \nabla) \equiv res \]

\[ tType_MS(\nabla, \nabla, \emptyset) \equiv req \]

\[ tType_MS(\emptyset, \emptyset, \emptyset) \equiv sel \]

We define the type of a Slot as the type of the first of the microslots, if present, otherwise it is \(act\).

\[ tType_S : Slot \rightarrow TType \]

\[ tType_S((), (p, \_)) \equiv act \]

\[ tType_S((m : \_), \_)) \equiv tType_MS(m) \]

### 5.3 Trace Operators

We now describe a series of operators which can be used to build and join traces and their building blocks.

#### 5.3.1 Building with single events

The first are a series of constructors that construct slots of the various types from a single guarded event and accompanying transition type. We shall refer to the combination of a transition type and guarded event as a “typed event.”
Given a non-act typed non-void event, we wish to build the corresponding microslot:

\[
\begin{align*}
mkm & : \text{TTtype} \rightarrow \text{GE} \rightarrow MS \\
mkm_{\text{act}}(g) & \equiv (g, 1, 1) \\
mkm_{\text{req}}(g) & \equiv (1, g, 1) \\
mkm_{\text{res}}(g) & \equiv (1, 1, g)
\end{align*}
\]

Given a typed event we wish to build the corresponding slot, where the event can be null only if of type act:

\[
\begin{align*}
\text{mks} & : \text{TTtype} \rightarrow \text{GE} \rightarrow \text{Slot} \\
mks_{\text{act}}(g) & \equiv (\langle \rangle, g) \\
mks_{\text{req}}(g) & \equiv (\langle \text{mkm}_{\text{act}}(g) \rangle, 1)
\end{align*}
\]

5.3.2 Lifting Event Merging

We want to lift the event merge operators to work with microslots.

We will want to merge a single guarded non-act typed event into a pre-existing microslot:

\[
\begin{align*}
\text{merge}_{\text{act}} : \text{GE} \times \text{TTtype} \times \text{MS} & \rightarrow \text{MS} \\
g \circ_{\text{act}} (s, q, r) & \equiv (g \circ s, q, r) \\
g \circ_{\text{req}} (s, q, r) & \equiv (s, g \circ q, r) \\
g \circ_{\text{res}} (s, q, r) & \equiv (s, q, g \circ r)
\end{align*}
\]

We describe the merging of two microslots later when the parallel construct is discussed.

5.3.3 Typed Cons-ing

By "typed cons-ing" we mean the process of placing a typed event at the start of an existing list of events, at the microslot, slot or trace level. We will overload.

We first consider cons-ing a non-act typed non-null event into a microslot or slots. If the event has a type greater than that of the microslot, then we have to create a new microslot immediately prior to the given one, containing the event. This is because "consing" means pre-pending an earlier event, so if an event of type req (say) is being placed in front of a microslot containing sel or req events, then it must have occurred in an earlier microslot. This is why the signature of the function indicates that merging a typed event with a microslot may result in more than one microslot as a result.

\[
\begin{align*}
\text{merge}_{\text{act}} & : \text{GE} \rightarrow \text{TTtype} \rightarrow \text{MS} \rightarrow \text{MS}^+ \\
g \circ_{\text{act}} m & \equiv \text{if } t > \text{tttype}_{\text{MS}}(m) \text{ then } \text{mkm}(g, m) \text{ else } g \circ_{\text{act}} m
\end{align*}
\]

We can extend this to work with microslot sequences in the obvious way:

\[
\begin{align*}
\text{merge}_{\text{act}} & : \text{GE} \rightarrow \text{TTtype} \rightarrow \text{MS}^+ \rightarrow \text{MS}^+ \\
g \circ_{\text{act}} \langle \rangle & \equiv \langle \text{mkm}(g) \rangle \\
g \circ_{\text{act}} (m : \mu) & \equiv (g \circ_{\text{act}} m) \circ \mu
\end{align*}
\]

We can now extend type-consing to slots and traces, in which case we can now handle act-events. Consing an act-event always creates a new slot at the front:

\[
\begin{align*}
\text{merge}_{\text{act}} & : \text{GE} \rightarrow \text{TTtype} \rightarrow \text{Slot} \rightarrow \text{Slot}^+ \\
g \circ_{\text{act}} s & \equiv \langle \text{mkm}_{\text{act}}(g), s \rangle \\
g \circ_{\text{act}} (\langle \mu, a \rangle) & \equiv \langle g \circ_{\text{act}} \mu, a \rangle
\end{align*}
\]

5.3.4 Concatenation for Microslots

We can now define a form of concatenation for microslots (\(\odot\)) which merges the last microslot of the first sequence (ante-slot) with the first microslot of the second (post-slot), if possible. This is possible when no event in the ante-slot has a type greater than that of an event in the post-slot. We first define an operator (\(\otimes\)) taking a pair of microslots to a sequence of same:

\[
\begin{align*}
\text{merge}_{\text{act}} & : \text{MS} \rightarrow \text{MS}^+ \\
\langle s_1, q \rangle \odot s_2 & \equiv \langle (s_1 \odot s_2), q \rangle \\
\langle s_1, q \rangle \odot \langle \text{req}, r \rangle & \equiv \langle (s_1, q \odot \langle \text{req}, r \rangle), q \rangle \\
m_1 \odot m_2 & \equiv \langle m_1, m_2 \rangle
\end{align*}
\]

We then define microslot-sequence concatenation using the binary merge-slot operator:

\[
\begin{align*}
\text{merge}_{\text{act}} & : \text{MS}^+ \times \text{MS}^+ \rightarrow \text{MS}^+ \\
\langle \emptyset, \mu \rangle \odot \mu & \equiv \mu \\
\mu \odot \emptyset & \equiv \mu \\
\langle m_1, \mu_2 \rangle \odot \langle m_2, \mu_2 \rangle & \equiv \langle m_1 \odot m_2, \mu_2 \rangle \\
\langle m_1, \mu_1 \rangle \odot \langle m_2, \mu_2 \rangle & \equiv \langle m_1 \odot m_2, \mu_2 \rangle
\end{align*}
\]

5.3.5 Consing Slots onto Traces

We now consider the task of cons-ing a Slot onto the start of a Trc in order to extend the Trc. Here, no type is specified, but instead is inferred from the slot contents.

The only time this differs from ordinary list cons is when the trailing trace is a singleton null slot or the slot is null or has no act event:

\[
\begin{align*}
\text{merge}_{\text{act}} & : \text{Slot} \rightarrow \text{Trc} \rightarrow \text{Trc} \\
\langle \text{null} \rangle \circ s & \equiv \langle s \rangle \\
\langle \mu, \text{TTtype} \rangle \circ (\langle \nu, \alpha \rangle : \tau) & \equiv (\langle \mu \nu \rangle, \langle \nu, \alpha \rangle : \tau) \\
\langle s, \text{tttype} \rangle \circ s & \equiv s \circ s
\end{align*}
\]

5.3.6 Catenation of Traces

We can now define trace catenation in terms of slot-consing:

\[
\begin{align*}
\text{merge}_{\text{act}} & : \text{Trc} \times \text{Trc} \rightarrow \text{Trc} \\
\langle \emptyset \rangle \circ \langle \emptyset \rangle & \equiv \langle \emptyset \rangle \\
\langle \langle s, \text{tttype} \rangle \rangle \odot \langle \emptyset \rangle & \equiv \langle s, \text{tttype} \rangle \\
\langle s_1, \text{tttype} \rangle \odot \langle s_2, \text{tttype} \rangle & \equiv \langle s_1 \odot s_2, \text{tttype} \rangle
\end{align*}
\]

Traces are non-empty, but the first clause is needed simply to handle a base case properly for the definition of the operator. We want the null trace to be an identity for trace catenation, and trace catenation to be associative.

5.4 Merging traces in parallel
Merging traces in parallel is straightforward — they are merged on a slot by slot basis, with slots merged on a micro-slot by micro-slot basis. We overload the notation \(\parallel\) for all these forms of parallel merging, except trace parallel merge which we denote by \([\parallel\]).

All these operators are associative and commutative, and the null-trace is the identity for \([\parallel\]). It is in order to get these properties that we require event merging itself to be both associative and commutative.

Merging two microslots in parallel simply involves merging the corresponding components:

\[
\begin{align*}
- \parallel - & : MS \times MS \rightarrow MS \\
(s_1, q_1, r_1) \parallel (s_2, q_2, r_2) & \equiv (s_1 \diamond s_2, q_1 \diamond q_2, r_1 \diamond r_2)
\end{align*}
\]

Merging microslot-sequences in parallel is done on a micro-slot by microslot basis, with the tail of the longer list simply being copied to the result:

\[
\begin{align*}
- \parallel - & : MS^* \times MS^* \rightarrow MS^* \\
\emptyset \parallel \mu_2 & \equiv \mu_2 \\
\mu_1 \parallel \emptyset & \equiv \mu_1 \\
(m_1 : \mu_1) \parallel (m_2 : \mu_2) & \equiv (m_1 \parallel m_2) : (\mu_1 \parallel \mu_2)
\end{align*}
\]

To parallel merge slots, we simply merge the microslots and the action events:

\[
\begin{align*}
- \parallel - & : Slot \times Slot \rightarrow Slot \\
(\mu_1, a_1) \parallel (\mu_2, a_2) & \equiv (\mu_1 \parallel \mu_2, a_1 \parallel a_2)
\end{align*}
\]

To merge a pair of traces we proceed on a slot-by-slot basis, and copy the longer tail over if the traces are of different length.

\[
\begin{align*}
- \parallel - & : Trc \times Trc \rightarrow Trc \\
\emptyset \parallel \tau_2 & \equiv \tau_2 \\
\tau_1 \parallel \emptyset & \equiv \tau_1 \\
(s_1 : \tau_1) \parallel (s_2 : \tau_2) & \equiv (s_1 \parallel s_2) : (\tau_1 \parallel \tau_2)
\end{align*}
\]

5.5 Framework Summary

We have defined a notion of guarded events, and microslots capturing sequences of \(\text{sol}, \text{req}\) and \(\text{res}\) events, as well as slots which put these before a clock-cycle terminating action event. We have defined traces as non-empty lists of such slots, with all but the last slot obliged to have an action event, and defined trace concatenation \((\langle\rangle\)) and parallel merge \([\langle\parallel\rangle\]) operators. Both have monoid properties, with the null trace as identity, and \([\langle\parallel\rangle\]) also being commutative.

6. EXECUTION STATE

We now turn our attention to the events of the previous section, and elaborate how these are in fact state-transformers. To this end, we first need to understand what is meant by the state of a Handel-C program.

6.1 Environments

We follow the classical approach for imperative languages in that the state is an “environment”: a mapping from identifiers to values. We differ in that while some identifiers denote program variables, others have special meaning and correspond to internal processing carried out during a clock-cycle, largely to do with processing \text{prialt} communication requests.

We define identifiers \(\text{Id}\) to be either variable names \(\text{Var}\) or one of four special identifiers \(\tau, \mathbb{R}, \gamma\) or \(B\), not present in \text{Var}. We define a value space \(\text{Val}\) to contain integers, booleans and an error value \((?)\), and then define a datum type as being either a value, a Handel-C expression, or one of the three types associated with \text{prialt} resolution, namely \text{Reslt}\text{n}, \text{CPMap} and \text{PriSet}:

\[
i \in \text{Id} \equiv \text{Var} + (\tau, \mathbb{R}, \gamma, B)
\]

\[
\text{Val} \equiv \mathbb{Z} + \mathbb{R} + \{?\}
\]

\[
d \in \text{Datum} \equiv \text{Val} + \text{Exp} + \text{Reslt}\text{n} + \text{CPMap} + \text{PriSet}
\]

Although we have used disjoint union or sum above, in the sequel we do not explicitly show the relevant injections, so that we interpret a value \(x : \mathbb{Z}\) as also being a value \(x : \text{Val}\), or even \(x : \text{Datum}\), rather than writing the more pedantic but verbose forms of \(\text{inj}_1(x) : \text{Val}\) and \(\text{inj}_2(\text{inj}_1(x)) : \text{Datum}\).

We define an environment \(\rho\) as a mapping from identifiers to data, subject to the proviso that variables map only to values, \(\mathbb{R}\) maps only to \text{Reslt}\text{n}s, and \(\gamma\) and \(B\) map respectively to the \text{CPMap} and \text{PriGrp} components of \(\rho(\mathbb{R})\):

\[
\rho \in \text{Env} \equiv \text{Id} \rightarrow \text{Datum}
\]

\[
i \in \text{Var} \Rightarrow \rho(i) : \text{Val}
\]

\[
\rho(\mathbb{R}) : \text{Reslt}\text{n}
\]

\[
\rho(\gamma) : \text{CPMap}
\]

\[
\rho(B) : \text{PriSet}
\]

\[
\rho(\mathbb{R}) = (\rho(\gamma), \rho(B))
\]

We denote the updating of a map \(\rho\) so that \(i\) now maps to \(d\) by \(\rho \upharpoonright (i \rightarrow d)\).

The identifier \(\tau\) is used to denote the clock-tick or clock-cycle count, so it is best viewed as mapping to an integer—however the associated value and its type is simply immaterial, as will become apparent later on.

Down items of type \text{Exp} do not form part of the state, but are used later as a technical device to capture part of the state change mechanism.

Expression evaluation w.r.t an environment is defined in the normal way, and returns a result of type \text{Datum} that is not itself of type \text{Exp}:

\[
\mathcal{E} : \text{Exp} \rightarrow \text{Env} \rightarrow \text{Datum}
\]

\[
\mathcal{E}[e]_{\rho} \equiv \text{“standard” expression evaluation…}
\]

6.2 Static State

The “static state” of a Handel-C program is that part of the state which persists across clock-cycle boundaries, and its evolution over those time-slots is what constitutes the observable behaviour of a Handel-C program.

For any Handel-C program, we simply identify all the variables used, in assignments, expressions, and channel inputs. We then tailor the environment so that its domain contains precisely those variables.

We can define a function on Handel-C processes \((\text{plds})\) which returns the set of all variables — the details are not given but it is very straightforward.

\[
\text{plds} : \text{Proc} \rightarrow \mathcal{P} \text{Var}
\]

The variable part of the environment \(\rho_p\) for a given program \(p\) is precisely that given by \(\text{plds}\):

\[
\text{dom} \rho_p = \text{plds}(p)
\]
6.3 Dynamic State

The dynamic state is that which only exists within one clock cycle, and is effectively “zeroed” at every clock tick. It contains information about communication requests and is that part of the environment accessed by the identifiers \(R, \gamma\) and \(B\).

At the start of each clock cycle, these are initialised to be empty:
\[
\rho(\gamma) = \emptyset \\
\rho(B) = \emptyset \\
\rho(R) = (\emptyset, \emptyset)
\]

6.4 Events for Handel-C

We want our events to be state-transformers, that is functions from state to state, and we need to define the null event (\(\circ\), as well as explaining how events merge (\(\circ\)).

6.4.1 Events

Formally our events are functions mapping environments into environments, and the null event is simply the identity function on environments:
\[
\text{Evt} = ? \quad \text{Env} \to \text{Env} \\
\text{\(\circ\)}(\rho) = ? \quad \rho
\]

We want to capture the notion of events that change part of the state, and we need to define the null event (\(\circ\)) which changes. The null event is simply the null map (\(\circ\)), simply being a partial environment which records the part that the basic event involves lodging a Req, and two events are merged by simply merging the maps (positions from state to state, and we need to define the null event (\(\circ\)), as well as explaining how events merge (\(\circ\)).

6.4.2 State Change

We use such partial maps to change the state by simply overriding the state with a mapping in which any expressions (as Datum) have been first evaluated w.r.t that state:
\[
\begin{aligned}
\text{St} & \equiv \text{Env} \\
\Delta(e_1) & \equiv \rho \triangleright E'_\rho(e_1) \\
E'_\rho \{ v \mapsto e \} & \equiv \{ v \mapsto E[\rho]\}
\end{aligned}
\]

It is this model of events which motivated the particular form of the abstract event model used when typed assertion traces where described previously, and why in that model we used \(\Delta\), rather than simply viewing events there directly as state-transformers themselves.

6.4.3 State Predicates

Any boolean-valued expression in Handel-C provides us with a predicate, simply by evaluating that expression against the state environment in the usual way.
\[
\text{Pred} \equiv \text{Exp}, \quad \text{boolean-valued} \\
e(\rho) \equiv E[\rho]
\]

6.5 Fixpoints

We define an ordering \(\preceq\) on traces, with \(\tau_1 \preceq \tau_2\) if \(\tau_1\) is a prefix of \(\tau_2\). We note that \(\{\emptyset\} \preceq \tau\) for any \(\tau\). A set of traces has a least upper bound w.r.t \(\preceq\) if all the traces are prefixes of some single (longest) trace, which is the shortest possible such trace. For typed assertion traces we say that \(\tau_1 \preceq \tau_2\) if there exists \(\tau_3\) such that \(\tau_1 \preceq \tau_3 \preceq \tau_2\).

We extend this to an ordering \(\subseteq\) over sets of traces by saying that \(S_1 \subseteq S_2\) if for every \(\tau_1\) in \(S_1\) there is a \(\tau_2\) in \(S_2\) such that \(\tau_1 \preceq \tau_2\). The least element in this ordering is the set \(\{\{\emptyset\}\}\). Again a notion of least upper bound (\([\bigcup\]) can be defined w.r.t \(\subseteq\).

Our semantic domain is therefore one of trace-sets, ordered by \(\subseteq\), and our semantic definitions produce directed sets. We therefore handle recursion by taking the least fixed point w.r.t \(\subseteq\), and we can compute this as
\[
\text{fix} L \bullet F(L) = \bigsqcup_{i \in \mathbb{N}} \{ F^i(\{\emptyset\}) \}
\]

7. HANDEL-C DENOTATIONAL SEMANTICS

We are now in a position to give the denotational semantics of Handel-C. First we need to introduce some shorthands to manage the complexity of the resulting expressions: we denote a guarded event (\(true, e\)) by \(\varepsilon\), and one we have a predicate and a null event (\(\emptyset, \emptyset\)) by \(\delta\). Given a binary operator \(\ast\) over type \(T\) we can extend it to act between sets of \(T\), or elements and sets of type \(T\) as follows:
\[
\begin{aligned}
\varepsilon \{\} & : PT \times PT \to PT \\
\text{S}\{\} T & \equiv \{ s \ast t | s \in S \land t \in T \} \\
\varepsilon(\delta) T & : T \times PT \to PT \\
\text{S}\{\} T & \equiv \{ s \ast t | t \in T \}
\end{aligned}
\]

The semantics of a Handel-C process is given as a set of typed assertion traces, subject to the following healthiness conditions: (1) Traces are maximal: if a trace is present, then none of its proper prefixes are; (2) Mutual Exclusivity: if two traces differ, then the pair of guarded events which first distinguish them must have mutually exclusive predicates, i.e ones that are never true in the same environment. (3) Exhaustiveness: given all traces in the set with
a common prefix, then all the guard predicates of the distinguishing events must exhaust all possibilities, i.e. for any environment, at least one (and only one) will return true.

We can now describe the semantics of all constructs except prialt in a straightforward manner as follows,

\[ \begin{align*}
[\cdot] & : \text{Prog} \rightarrow \mathcal{P} \text{Trv} \\
[0] & \equiv \{ \langle 0 \rangle \} \\
[1] & \equiv \{ ! \} \\
[x := e] & \equiv \{ \langle \langle i \rangle, \langle x \mapsto e \rangle \rangle \} \\
[p; q] & \equiv \{ p \} \{ q \} \\
[p \parallel q] & \equiv \{ p \} \{ ! \} \{ q \} \\
[p \circ e q] & \equiv \overline{\mathcal{B}_{\text{sel}}^{i+1}}[p] \cup \overline{\mathcal{B}_{\text{sel}}^{i+1}}[q] \\
[b \ast p] & \equiv \text{fix } L \bullet \{ \langle \text{mk}_{\text{sel}}(\overline{b}) \rangle \} \\
\end{align*} \]

0, 1 and assignment have a single singleton trace as semantics, being respectively the empty, clock-tick and single-variable update slots. Sequential and parallel composition simply combine all their traces with the appropriate trace operator. The conditional construct prefixes the traces of the “then” outcome with the condition as a guard predicate, while the traces of the “else” outcome have the negation of that predicate prefixed instead. It is with this construct that multiple traces are introduced, and were we ensure that the exclusivity and exhaustiveness healthiness conditions are met. The while-loop is given a fixpoint semantics, as is standard for such constructs. In effect it either immediately terminates, if the guard is false, or else the guard is true, and it then behaves like the loop-body sequentially composed with the loop itself.

### 7.1 Extending the Language

The semantics of prialt is best given by breaking the construct down into simpler components, which mainly correspond to the various phases in which prialt is active, namely \( \text{req}, \text{res} \) and \( \text{act} \). We now introduce some extension to the language to facilitate this — note that these extensions exist solely in order to elucidate the semantics, and are not available for general use by the Handel-C programmer.

We extend the expression syntax to include three special forms — a prialt-waiting predicate \( \langle w(g_i) \rangle \), an active guard expression \( \langle a(g_i) \rangle \), and a channel data expression \( \langle \delta(c) \rangle \).:

\[
e \in \text{Exp} = \ldots | w(g_i) | a(g_i) | \delta(c)
\]

The waiting predicate takes a prialt-request (guard-list) as argument, and returns true if resolution has determined that that prialt is blocked. It is evaluated, after the req phase, by looking at the \( B \) component of the state:

\[
E[w(g_i)] \rho \equiv \langle g_i \rangle \in \rho(B)
\]

The active guard expression takes a prialt-request as argument, and returns the index \( i \in \{1 \ldots n\} \) of the guard which is going to be active in this clock-cycle. It is only defined when \( w(g_i) \) is false, and looks up the channel-prialt map

\[
E[a(g_i)] \rho \equiv \min_j \text{ where } \exists c \cdot (g_i) \in \rho(\gamma(c)) \land \text{channel}(g_i) = c
\]

Here \( \text{channel} \) returns the channel associated with a guard.

The channel data expression \( \delta(c) \) returns the data expression associated with an active channel — this information can be extracted from the channel-prialt map component, as detailed in [8].

We extend the program syntax to include three new statements — a prialt-request statement \( \langle +\langle g_i \rangle \rangle \), a prialt-wait statement \( \langle \text{wait}(g_i) \rangle \), and a multi-way conditional branch (or case-statement):

\[
p \in \text{Prog} ::= \ldots | +\langle g_i \rangle | \text{wait}(g_i) | e \triangleright [p_i]
\]

The prialt-request statement simply lodges its guard-list argument into the input \( \text{PriSet} \) for resolution. In the semantics we use the \( B \) component of the state to hold both the prialt input to resolution (during the \( \text{req} \) phase) and the blocked-prialt result of resolution (available during the \( \text{res} \) and \( \text{act} \) phases).

The prialt-wait statement asks if its prialt argument is blocked. If it is, it then waits one clock cycle, then re-submits the corresponding prialt-request, before repeating itself. If the prialt is not blocked, it terminates immediately. Conceptually, \( \text{wait}(g_i) \) is the same as \( \langle w(g_i) \rangle \ast (1; +\langle g_i \rangle) \), but it is simpler to keep it separate, because the while-loop makes its decisions in a different phase \( \text{sel} \).

The case-statement \( e \triangleright [p_i] \) evaluates expression \( e \), whose value must lie in the range \( 1 \ldots n \). This value is used to select the process to execute.

We also define a function on guards which gives the underlying action as an equivalent statement:

\[
\begin{align*}
\text{act}(\cdot) & : \text{Grd} \rightarrow \text{Prog} \\
\text{act}(c|e) & \equiv 1 \\
\text{act}(c?e) & \equiv v := \delta(c) \\
\text{act}(!?) & \equiv 0
\end{align*}
\]

We give \( \text{prialt} \langle g_i \rightarrow p_i \rangle \) a semantics by translating it to:

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

This captures the notion that a prialt acts in three stages: (i) it submits a request \( +\langle g_i \rangle \); (ii) it waits until it becomes active, re-submitting the request on every clock cycle \( \text{wait}(g_i) \); and (iii) once waiting is over, selects and execute the active guard and corresponding process \( (a(g_i) \triangleright [\text{act}(g_i) ; p_i]) \).

We can now give the semantics of the additional constructs:

\[
\begin{align*}
I[+\langle g_i \rangle] & \equiv \{ \langle \text{mk}_{\text{req}}(B \rightarrow \{ g_i \}) \rangle \} \\
\text{wait}(g_i) & \equiv \text{fix } W \bullet \{ \langle \text{mk}_{\text{res}}(\overline{w(g_i)}) \rangle \} \\
& \quad \cup \langle \text{mk}_{\text{res}}^{i+1} \{ \{ v \} \} \rangle \rightarrow W \\
[a(g_i) \triangleright [p_i]] & \equiv \bigcup \{ [a(g_i) = i]\uparrow \langle p_i \rangle] \}
\end{align*}
\]

The request statement is simply an update of the state’s \( B \) component, tagged as occurring during the \( \text{req} \) phase. The case-statement simply prepends a guarded event asserting that \( e = i \) to the traces associated with process \( p_i \), such a choice being made during the \( \text{res} \) phase.

The \( \text{wait}(g_i) \) statement is a looping construct, so it has a fixpoint definition as expected. However it is important to note that the terminating guarded event \( \langle \overline{w(g_i)} \rangle \) occurs
during the res phase, while the continuation guarded event (w(g)) occurs during the act phase. The reason for this is the same as that encountered in the operational semantics, namely that the decision to end waiting can be made as soon as a prialt becomes unblocked (during some res phase), but the decision to wait until the next clock cycle to try again needs to be deferred until no more sel-req-res micro-cycles can occur, i.e. once the act phase has been reached. This is because a subsequent round of request and resolution, caused by a prialt in some default guard, may cause a blocked prialt to become unblocked. The converse never happens: once a prialt is unblocked in one microcycle, it can never become blocked again subsequently.

7.2 Examples

We now present a few small example simply to show the semantics at work.

In order to keep expressions readable and manageable, we introduce the following shorthand: (i) for ⟨⟩, we simply write {x → e}; (ii) for mkseq(⟨B → ⟨{g}⟩⟩), we use req(g); and (iii) we use undecorated {i} and [] instead of {i}() or {[]} as the types can be inferred from context. Rather than showing the slot and microcycle structure explicitly, we simply list the events separated by commas, and semicolons to mark the slot boundaries (i.e. clock ticks). So

\[(\langle\rangle, y \mapsto f), (\langle\{i\}, \nabla, \nabla\rangle, z \mapsto e)\]

becomes \{(y \mapsto f) ; \{i\}, \{z \mapsto e\}\}.

7.2.1 Assignment, Conditional and Sequential Composition

If we follow an assignment by a conditional as follows:

\[x := y + z; y := z \land (x > 0) \rightarrow z := y\]

then calculating this with the semantics gives:

\[\langle x := y + z; y := z \land (x > 0) \rightarrow z := y \rangle = \{\langle x \mapsto y + z ; \ n \geq 0, (y \mapsto z)\rangle, \langle x \mapsto y + z ; \ n \leq 0, (z \mapsto y)\rangle\}\]

We see clearly the same starting event in both traces, and then a choice based on the sign of \(x\) guarding the subsequent behaviour, each covered by one of the two traces.

7.2.2 Parallel Assignment

We can swap two variables in one clock cycle:

\[\langle x := y \parallel y := z \rangle = \{\langle x \mapsto y, y \mapsto z\rangle\}\]

This works because the expressions are evaluated first during the clock cycle, and the variables are updated simultaneously as the clock ticks. However, if we attempt to simultaneously assign two different values to one variable, the semantics flags this as an error

\[\langle x := e_1 \parallel x := e_2 \rangle = \{\langle x \mapsto ?\rangle\}\]

7.2.3 While Loop

If we consider a simply busy waiting loop (\(b\) will hopefully eventually be set by some other process), then we calculate the semantics as:

\[\langle b \ast 1 \rangle = \bigcup\{F(i) \mid i \in \mathbb{N}\}\]

where

\[F(L) = \{(\neg b) \cup (b,i)\}L\]

Evaluating this leads to the result that the set of traces are of the form:

\[\langle b \ast 1 \rangle = \{(\neg b), (b_1, \neg b), (b_1, \ldots, b_n, \neg b), \ldots \}

We have finite traces which correspond to zero or more iterations before the condition becomes true, and one infinite trace which captures the situation were \(b\) is always false — this is why we need to admit infinite traces in our semantic model.

8. CONCLUSIONS AND FUTURE WORK

We have presented a denotational semantics for Handel-C as sets of typed assertion traces, which captures all the key behaviour of the language, with particular emphasis on the proper treatment of default clauses in prialt statements.

To date we have both an operational and denotational semantics for the language, as well as a “hardware compilation” semantics currently in the pipeline. We need to show that all these semantics describe the same language, and in particular we hope to be able to prove a full abstraction theorem relating the operational and denotational forms.

The real goal is to use the denotational semantics to verify a series of algebraic laws for Handel-C, which would form the basis for a practical system for formal reasoning about such programs. We also intended to extend this to cover the notion of refinement in a Handel-C setting, linking the language to specification notations such as CSP [22] or Circus [24].

9. REFERENCES


