Semantic Domains for Handel-C

Andrew Butterfield

Department of Computer Science
Trinity College, Dublin University
Dublin, Ireland

Jim Woodcock

Computer Laboratory
University of Kent at Canterbury,
United Kingdom

Abstract

Handel-C is a programming language which is a hybrid of CSP and C, designed to target hardware implementations, specifically field-programmable gate arrays (FPGAs). The language is C-like with CSP-like parallel constructs and channel communication added. All assignments and channel communication events take one clock cycle while all expression and conditional evaluations are deemed to be instantaneous. This report presents semantic domains required to give a denotational semantics of a simplified subset of the Handel-C language. We present the key domains and equations for a denotational semantics for Handel-C. The key contribution is that our semantics deals with a concurrent, deterministic language where events occur synchronously, in the presence of global shared variables. We exploit the finite and static nature of a Handel-C program’s identifier space in order to define key concepts, such as world, change and choice. We have also demonstrated that our semantic domain is a c.p.o., with all our constructors shown to be monotonic, allowing us to assert the existence of fixpoints. General concurrency theories such as CSP and CSPP cater for a wider and more general range of circumstances than are found in Handel-C. By keeping our semantics separate and simpler, it is easier to ensure that it correctly captures the behaviour of the language. We view this work as leading towards a formal development methodology to allow the refinement of Handel-C programs from formal specifications. We hope to be able to integrate laws based on this semantics into the Circus refinement calculus framework.

1 Thanks to Dean of Research Fund, TCD
2 Email: Andrew.Butterfield@cs.tcd.ie
3 Email: J.C.P.Woodcock@ukc.ac.uk

©2003 Published by Elsevier Science B. V.
1 Introduction

Handel-C[^1] is a language originally developed by the Hardware Compilation Group at Oxford University Computing Laboratory. It is a hybrid of CSP[^2] and C, designed to target hardware implementations, specifically field-programmable gate arrays (FPGAs) [13,17,17]. The language is essentially a static subset of C, augmented with a parallel construct and channel communication, as found in CSP. The type system has been modified to refer explicitly to the number of bits required to implement any given type. The language targets largely synchronous hardware with multiple clock domains. All assignments and channel communication events take one clock cycle, with all updates synchronised with the clock edge marking the cycle end. All expression and conditional evaluations are deemed to be instantaneous, effectively being completed before the current clock-cycle ends.

This report presents semantic domains required to give a denotational semantics of a simplified subset of the Handel-C hardware compilation language. We only consider a single clock domain, and we ignore many of the detailed data manipulation features of the language. We also consider a small representative sample of the control constructs.

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.
- **synchronous “cores”** These are regions of hardware under the control of a single clock, and constitute the primary area of concern of this paper.
- **priority** The communication constructs are provided in the form of priority statements, which requires all choices between communication events to be prioritised. A formal treatment of this can be found in [2].
- **asynchronous “environment”** The synchronous cores communicate with each other and the external environment via asynchronous interfaces. The details of this is beyond the scope of this paper, but we do indicate in the conclusions section how we propose to model this, incorporating the core semantic model.

These four areas can be treated separately to a large degree, as the interfaces between them are simple in character.

2 The Language

We present a simplified form of the Handel-C language, which abstracts out the essential features. Two important aspects of communication are ignored.
Butterfield and Woodcock

here, namely default clauses in *prialt* constructs, and the distinction between internal and external channels. Both can be accommodated in the theory presented here in a straightforward manner.

We begin by introducing two identifier spaces, one ($V$) to represent regular program variables, the other ($Ch$) to represent communication channel identifiers. We also assume some syntax describing expressions ($E$) whose structure need not concern us here.

$$v \in V \quad c \in Ch \quad e \in E$$

Communication is managed by treating requests as synchronisation guards. An output guard ($c!e$) designates a channel and expression whose value is to be sent along that channel. An input guard ($c?v$) identifies a channel, and a variable into which the received value is to be placed.

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

In order to be able to describe communication easily, we shall extend the language expressions with three features, namely a wait predicate ($w$), a selector function ($s$) and 'bare' channel identifiers (not part of guards).

$$E ::= \ldots \mid w(g_0, \ldots, g_{n-1}) \mid s(g_0, \ldots, g_{n-1}) \mid c$$

These expression extensions cannot appear in Handel-C program texts, as their use is solely for the purposes of elucidating the semantics.

The program statements have the following structure:

$$s \in S ::= \begin{array}{l}
\delta^n \quad \text{Delay} \\
| v := e \quad \text{Assignment} \\
| s_1; s_2 \quad \text{Sequencing} \\
| s_1 \parallel s_2 \quad \text{Parallel} \\
| e \rightarrow s_0, \ldots, s_{n-1} \quad \text{Conditional (multi-way)} \\
| b * s \quad \text{Iteration} \\
| \langle g_0 : s_0, \ldots, g_{n-1} : s_{n-1} \rangle \quad \text{Prialt} 
\end{array}$$

A delay $\delta^n$ simply does nothing, but takes $n$ clock cycles do it. Assignment always takes exactly one clock cycle. The sequential and parallel composition are much as expected, noting that parallel processes run in lock-step with each other and the clock. We adopt a multi-way conditional, where the condition evaluates to a number which determines which alternative gets to execute. We can model a conventional binary choice by interpreting booleans $\text{TRUE}$ and
Butterfield and Woodcock

FALSE as numbers 0 and 1 respectively, which is how we shall view the condition of the iteration construct. The iteration construct evaluates its condition, and executes its body while the condition is true. The last statement above is the prialt statement which makes a communication request, considered as a priority-ordered set of guard-statement pairs. The highest-priority enabled guard is executed, and then the control flow is handed to the associated statement. The indices run from 0 to \( n - 1 \), rather than 1 to \( n \), because it makes certain book-keeping simpler, as will be discussed later on.

In a manner similar to that described above for expressions, we shall add two new statement constructs to the language, one is a communication request statement, while the second is a statement performing the action associated with a guard.

\[
S ::= \ldots | \langle g_0, \ldots, g_{n-1} \rangle | \text{Act}(g)
\]

Notably missing from the language are any form of procedural or functional abstraction. In particular there is no way in which recursive definitions can be provided. A key consequence of this is that the identifier space of a given program is finite and static.

3 Environments: Worlds and Changes

As is common in many denotational semantics of variable-based languages, we start by building a description of a variable environment, mapping identifiers to appropriate values. As we are not considering the syntax of expressions in any great detail here, we shall simply assume that they, and all program variables, are integer valued. We allow for undefined values by adding an unknown value, which we denote by \(?\), keeping the symbol \( \perp \) for another purpose. In addition to variable values, we shall also record the current time (as measured in clock cycles) as well as the set of current communication requests, each denoted by the corresponding sequence of prialt guards.

\[
\begin{align*}
\text{Val} &\triangleq \mathbb{Z} \cup \{ ? \} & \text{Time} &\triangleq \mathbb{N} \\
\text{Dir} &\triangleq \text{IN} \mid \text{OUT} & \text{PriAlt} &\triangleq G^+
\end{align*}
\]

We employ a single environment mapping to record all the above information, so we combine it in a single overarching ‘datum’ space.

\[
d \in \text{Datum} \triangleq \text{Val} \cup \text{Time} \cup \mathcal{P}\text{PriAlt}
\]

Similarly, we define a single ‘identifier’ space consisting of variables, channel names, as well as two special purpose identifiers, \( \tau \) which gives the current time value, and \( \mathcal{R} \) which maps to the current communication request state.

\[
\text{Id} \triangleq V \cup \text{Ch} \cup \{ \tau, \mathcal{R} \}
\]
As previously mentioned, the runtime variable space of a given program is finite and static, so we can define a function which returns the set of identifiers mentioned in a program statement.

\[ \text{pIds} : S \to \mathbb{P}Id \]

It is defined inductively over the syntax tree in an obvious manner, and always includes \( \tau \) and \( \mathbb{R} \).

We shall define our environments as mapping from identifiers to datum values, subject to the constraint that \( \tau \) and \( \mathbb{R} \) always map to a time and request value respectively, while all other identifiers map to ‘ordinary’ values.

\[ \rho \in \text{Env} \cong \text{Id} \to \text{Datum} \]
\[ \text{inv-Env} : \text{Env} \to \mathbb{B} \]
\[ \text{inv-Env} \rho \cong \forall [\text{ok}_\rho] (\text{dom} \rho) \]

where \( \text{ok}_\rho(i) \cong (i \in (V \cup \text{Ch}) \land \rho(i) \in \text{Val}) \)
\[ \lor (i = \tau \land \rho(i) \in \text{Time}) \lor (i = \mathbb{R} \land \rho(i) \in \mathbb{P}\text{PriAlt}) \]

Given that the identifier space is finite and static, for any program \( p \) we can classify an environment as being a “World” if it is total on this space, and as being a “Change” if partial.

\[ \omega \in \text{World}_p \cong \text{pIds}[p] \to \text{Datum} \]
\[ \delta \in \text{Change}_p \cong \text{pIds}[p]^m \to \text{Datum} \]

We can also define an initial world, as one where every variable and channel maps to the unknown value, \( \tau \) maps to 0 and \( \mathbb{R} \) to \( \emptyset \).

\[ \omega_0 : \text{World}_p \]
\[ \omega_0 \cong (\cup/ \circ \text{ival}^* \circ \text{pIds})[p] \]

where \( \text{ival}(i) \cong \begin{cases} i = \tau \to \{\tau \mapsto 0\} \\ i = \mathbb{R} \to \{i \mapsto \emptyset\} \\ i \in (V \cup \text{Ch}) \to \{i \mapsto ?\} \end{cases} \)

Here the notation \( f^* \) denotes the mapping of function \( f \) over a sequence of values, and \( */ \) denotes the reduction (or “fold”) of a sequence using the binary operator \( * \). The \( \cup \) operator denotes map extension.

Why the distinction between worlds and changes? The reason lies in the fact that in order to model a language with parallelism, sequential composition and shared variables, we need to denote the meanings of basic statements as functions over environments, which capture sequencing by sequences of such functions, rather than by simply composing them. In particular, our functions will actually map worlds into changes, which we shall refer to as “choices”:

\[ \kappa \in \text{Choice} \cong \text{World} \to \text{Change} \]

These choices will play the role of ‘events’ in our semantics.

All the Choices used in our semantics will be classified as being either “sub-Atomic” or “Atomic”. A subAtomic event is one whose execution does not
require a complete clock cycle. In Handel-C these are expression evaluations and decisions. An Atomic event is one which requires an entire clock cycle to complete—in other words the event finishes at the falling clock edge, in complete synchrony with all other atomic events during that same cycle. Atomic events include variable assignments and successful communication events. In our model, a subatomic event does not change the timestamp, while an atomic one advances it by 1.

What makes it easy to classify choices here is that all the choices we employ will be minimal and uniform. A choice is uniform if the domain of any change it produces is the same regardless of what world is input:

\[
isUniform : \text{Choice} \rightarrow \mathbb{B}
\]

\[
isUniform \kappa \equiv \forall \omega_1, \omega_2 \in \text{World} \cdot \text{dom}(\kappa(\omega_1)) = \text{dom}(\kappa(\omega_2))
\]

A choice is minimal if uniform, and the domain of any change it produces consists only of those identifiers whose values change in some world:

\[
isMinimal : \text{Choice} \rightarrow \mathbb{B}
\]

\[
isMinimal \kappa \equiv isUniform \kappa \land \forall i \in \text{dom}(\kappa(\omega_0)) \cdot \exists \omega \cdot \omega(i) \neq (\kappa(\omega)i)
\]

Given minimal choices, then atomicity is determined by the presence/absence of the timestamp identifier \(\tau\) in the resulting changes:

\[
isAtomic : \text{Choice} \rightarrow \mathbb{B}
\]

\[
isAtomic \kappa \equiv \forall \omega \in \text{World} \cdot \tau \in \text{dom} \kappa(\omega)
\]

\[
isSubAtomic : \text{Choice} \rightarrow \mathbb{B}
\]

\[
isSubAtomic \kappa \equiv \forall \omega \in \text{World} \cdot \tau \notin \text{dom} \kappa(\omega)
\]

In addition to choosing which variables change and how according to the state of the world, we also will need to select statements based on the state of the world, in order to model conditional statements. We therefore define a Selector as a total function from World to natural numbers.

\[
s \in \text{Selector} \equiv \text{World} \rightarrow \mathbb{N}
\]

In practice all Selectors we employ will have an output range which matches the number of branches in the associated conditional.

## 4 Sequence-Trees

In order to model Handel-C’s behaviour, we need to embed choices into a trace-like structure. A variety of trace models exist, ranging from prefix-closed sets of event sequences [14], through labelled transition systems [12]. Most of these models cater for nondeterminism, and support an interleaving notion of concurrency. However, Handel-C is deterministic, and has a global clock providing global synchronous communication. This means we cannot simply adopt these existing models as is. Also the notion of priority in communication guards cannot be directly modelled in CSP-like models [8,10].
Butterfield and Woodcock

We shall adopt a model which is a hybrid of sequences and trees, with two type parameters, tailored to capture the synchronous deterministic behaviour of Handel-C. Such a hybrid sequence-tree is either ‘nil’, a ‘cons’ node or a multiway ‘split’. We define such sequences as being parameterised by two other types, $A$ and $S$. These will be instantiated later using Choices and Selectors. We will also require that elements of type $A$ can be classified as either “atomic” or “sub-atomic”, which we shall distinguish below by placing a tick-mark on atomic values. So, for example, the declaration $a, b' \in A$ tells us that $a$ is sub-atomic, while $b'$ is atomic.

$$a \in A \quad s \in S \quad \sigma \in T_{A,S} ::= \Lambda \mid a : \sigma \mid s(\sigma_0|\sigma_1|\ldots|\sigma_{n-1})$$

We shall define a form of branching sequence concatenation ($\tilde{\ast}$) which is very familiar as far as the ‘nil’ and ‘cons’ constructors are concerned.

$$\tilde{\ast} : T_{A,S} \times T_{A,S} \to T_{A,S}$$

$$\Lambda \tilde{\ast} \sigma \triangleq \sigma \quad \sigma \tilde{\ast} \Lambda \triangleq \sigma \quad a : \sigma_1 \tilde{\ast} \sigma_2 \triangleq a : (\sigma_1 \tilde{\ast} \sigma_2)$$

When concatenating after a branch split, we simply (right-)distribute the concatenation through to each branch.

$$s(\sigma_0|\ldots|\sigma_{n-1}) \tilde{\ast} \sigma \triangleq s(\sigma_0 \tilde{\ast} \sigma|\ldots|\sigma_{n-1} \tilde{\ast} \sigma)$$

We also define a parallel merge operator $\parallel$. The empty branching sequence is an identity:

$$\parallel : T_{A,S} \times T_{A,S} \to T_{A,S}$$

$$\Lambda \parallel \sigma \triangleq \sigma \quad \sigma \parallel \Lambda \triangleq \sigma$$

When both start with ‘cons’ nodes, the outcome depends on the nature of the $A$ (event) value. If the two events have different atomicities, then the sub-atomic event precedes the atomic one. If both events have the same atomicity, then we merge them into a single simultaneous event with the same atomicity, using the $\sqcup$ operator (explained in more detail shortly).

$$a : \sigma_1 \parallel b : \sigma_2 \triangleq a \sqcup b : (\sigma_1 \parallel \sigma_2) \quad a : \sigma_1 \parallel b' : \sigma_2 \triangleq a : (\sigma_1 \parallel b' : \sigma_2) \quad a' : \sigma_1 \parallel b : \sigma_2 \triangleq b : (a' : \sigma_1 \parallel \sigma_2) \quad a' : \sigma_1 \parallel b' : \sigma_2 \triangleq a' \sqcup b' : (\sigma_1 \parallel \sigma_2)$$

When combining ‘cons’ and ‘split’ we wish to ensure that the decision events denoted by the split expression occur after any subatomic actions, and before any atomic actions. The split decision events ($s$) are deemed to be subatomic, so atomic events distribute through branches.

$$a' : \sigma \parallel s(\sigma_0|\ldots|\sigma_{n-1}) \triangleq s(a' : \sigma | s_0 | \ldots | a' : \sigma | \sigma_{n-1}) \quad s(\sigma_0|\ldots|\sigma_{n-1}) \parallel b' : \sigma \triangleq s(s_0 | b' : \sigma | \ldots | s_{n-1} | b' : \sigma)$$
When both start with ‘split’ nodes, the outcome is multiplicative, in that each branch of one is merged with each branch of the other:

\[ s(\sigma_0|\ldots|\sigma_{m-1}) \parallel t(s_0|\ldots|s_{n-1}) \equiv s \times_n t(\sigma_0|s_0|\ldots|\sigma_{m-1}|\sigma_0|s_{n-1}) \]

\[
\ldots \\
|\sigma_{m-1}|s_0|\ldots|\sigma_{m-1}|s_{n-1}
\]

We need some book-keeping to keep track of which branches in the result get selected. Effectively \( s \times_n t \) is an appropriate ‘lifting’ of the following operator defined on natural numbers

\[ i \times_K j \equiv Ki + j \]

The details of the lifting are specific to how \( S \) is instantiated. The simple form of this operator is a consequence of our decision to number from 0, rather than 1, in conditionals and \textit{prialt}s. Our semantic model \( \mathcal{M} \) is a branching sequence instantiated with Choices and Selectors for \( A \) and \( S \) respectively:

\[ \mathcal{M} \equiv T_{\text{Choice,Selector}} \]

Given this, we can define the ‘lifted’ \( \times_n \) operator as

\[ s \times_n t \equiv \lambda \omega \cdot s(\omega) \times_n t(\omega) \]

We can now define the appropriate instance of \( \sqcup \) (merging simultaneous events) as a lifted version of an operator \( \triangledown \) which merges Changes

\[ \kappa_1 \sqcup \kappa_2 \equiv \lambda \omega \cdot \kappa_1(\omega) \triangledown \kappa_2(\omega) \]

When merging changes, we have to consider two cases, namely (i) those identifiers affected only by one or the other change, and (ii) those identifiers mentioned in both. The result change consists of (i) plus the outcome of resolving the conflicts in (ii), using \( \triangledown' \):

\[ \triangledown' : \text{Change} \times \text{Change} \rightarrow \text{Change} \]

\[ \delta_1 \triangledown' \delta_2 \equiv [\delta_2]|\delta_1 \sqcup (\triangledown[\delta_2]|\delta_1 \triangledown' \triangledown[\delta_1]|\delta_2) \sqcup \triangledown[\delta_1]|\delta_2 \]

Simultaneous updates to variable or channel identifiers are a (runtime) error, which we handle with an unknown value:

\[ \triangledown' : \text{Change} \times \text{Change} \rightarrow \text{Change} \]

\[ \{v \mapsto e_1\} \triangledown' \{v \mapsto e_2\} \equiv \{v \mapsto ?\} \]

\[ \{c \mapsto e_1\} \triangledown' \{c \mapsto e_2\} \equiv \{c \mapsto ?\} \]

Simultaneous setting of identical time values is allowed (and is the only case that will ever arise in our model).

\[ \{\tau \mapsto t\} \triangledown' \{\tau \mapsto t\} \equiv \{\tau \mapsto t\} \]

Simultaneous requests are simply lumped together.

\[ \{\mathcal{R} \mapsto R_1\} \triangledown' \{\mathcal{R} \mapsto R_2\} \equiv \{\mathcal{R} \mapsto R_1 \cup R_2\} \]
5 Semantics

We are giving our language semantics as a function from programs to our Selector-branching Choice-sequences model:

\[ \semantics \colon S \rightarrow \mathcal{M} \]

As our basic events are functions from worlds, we adopt some shorthands in order to make our definitions more compact. In essence we drop the \( \lambda \omega \cdot \) prefixes, denote timestamp increments by primes (\( \prime \)), and use underlining of expressions to denote their evaluation w.r.t a world value. A summary of these shorthands is given in the following table:

<table>
<thead>
<tr>
<th>Longhand</th>
<th>Shorthand</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lambda \omega \cdot \theta )</td>
<td>( \theta )</td>
</tr>
<tr>
<td>( [e]_\omega )</td>
<td>( e )</td>
</tr>
<tr>
<td>( \lambda \omega \cdot { \ldots, v \mapsto [e]_\omega, \ldots } )</td>
<td>( { \ldots, v \mapsto e, \ldots } )</td>
</tr>
<tr>
<td>( \lambda \omega \cdot { \ldots, \tau \mapsto \omega(\tau) + 1, \ldots } )</td>
<td>( { \ldots, \ldots } )</td>
</tr>
</tbody>
</table>

We begin our description of the semantics with the delay statement. A zero cycle delay does nothing, and takes no time to do it. A \( n \)-cycle delay changes no variables, but advances the timestamp by \( n \).

\[
\semantics{\delta^0} \equiv \Lambda \\
\semantics{\delta^n} \equiv \theta' : \semantics{\delta^{n-1}}
\]

Note that the semantics maps \( \delta^n \) to a (branching) sequence of \( \theta' \) events of length \( n \), rather than a single choice function which increments its timestamp in one go. This ensures that we can merge these in parallel with other sequences without difficulty.

Assignment simply evaluates the expression in the current world, and then updates the variable to the new value and advances the timestamp by one.

\[
\semantics{v := e} \equiv \{ v \mapsto e \}'
\]

Note that all variable updates are deemed to occur on the falling clock edge of the current clock cycle, while all the expression evaluations precede this update time. In particular, the program \( x := y \parallel y := x \) swaps the contents of \( x \) and \( y \).

For sequential composition we simply concatenate the sequences denoting each component.

\[
\semantics{s_1; s_2} \equiv \semantics{s_1} ; \semantics{s_2}
\]

For parallel composition, we merge the component trees as described previously, noting that in any given cycle, all subatomic Choices occur first, followed by Selectors, and finishing with Atomic Choices. This is a consequence of the definition of \( \parallel \).
To evaluate a condition statement, we first evaluate the condition expression w.r.t. the world to obtain a number — this means we can interpret a conditional expression as a Selector. We then use the semantic value of each sub-statement for the appropriate branch.

\[ [e \rightarrow s_0, \ldots, s_{n-1}] \overset{\mathcal{E}}{\Rightarrow} e([s_0] | \ldots | [s_{n-1}]) \]

The semantics of iteration is given in the usual manner as the least fixed point of the appropriate functional:

\[ [b * s] \overset{\text{fix}_T(b(\theta | [s]_\mathcal{I}))}{\Rightarrow} \]

A consequence of this is that the semantics of any iterative construct is an infinitely deep (finitely) branching sequence:

\[ \mathcal{I} = b(\theta | [s]_\mathcal{I}) = b(\theta | [s]_\mathcal{I} b(\theta | [s]_\mathcal{I})) = b(\theta | [s]_\mathcal{I} b(\theta | [s]_\mathcal{I} b(\theta | [s]_\mathcal{I}))) = \ldots \]

We do not give a semantics for the \textit{prialt} statement directly. Instead we convert it into an equivalent form using the extra statement forms that were introduced when the language was described. The reason for this approach is primarily because it simplifies the description of the semantics of \textit{prialt}, by breaking it down into simpler pieces. It has no effect on the choice of semantic domains that are used, and has the advantage of conceptually matching the behaviour of the implementation. Other reasons for adopting this approach are that it facilitates future work on generalisations of the Handel-C language, as well as simplifying other semantic approaches currently under consideration, in a similar manner. A direct presentation of the \textit{prialt} semantics is possible, but would be complex and quite confusing.

The execution of a \textit{prialt} statement \( p \) is therefore broken down into:

(i) lodging a request for the communication activity \( +\langle p \rangle \)

(ii) waiting while the request is not granted \( w(p) \) — each wait consumes a clock cycle; we resubmit the request (note that all requests are automatically rescinded at the end of each clock cycle—see the ‘prune’ operator described later).

(iii) once the request is granted (if ever), we determine the enabled guard \( i = s(p) \), perform its action \( \text{Act}(g_i) \) and then continue by executing its associated statement \( s_i \).

We can summarise this translation as:

\[ \langle g_0 : s_0, \ldots, g_{n-1} : s_{n-1} \rangle \mapsto +\langle g_0, \ldots, g_{n-1} \rangle; \]

\[ w\langle g_0, \ldots, g_{n-1} \rangle * (\delta^1; +\langle g_0, \ldots, g_{n-1} \rangle); \]

\[ s\langle g_0, \ldots, g_{n-1} \rangle \rightarrow \text{Act}(g_0); s_0, \ldots, \text{Act}(g_{n-1}); s_{n-1} \]

The request is modelled as a subatomic event which simply updates the environment request variable \( \mathcal{R} \).

\[ [+\langle g_0, \ldots, g_{n-1} \rangle] \overset{\lambda \omega \cdot \omega \uparrow \{ \mathcal{R} \mapsto \{ \langle g_0, \ldots, g_{n-1} \} \}}{\Rightarrow} \]
The action of a guard depends on the direction. An output guard behaves like $\delta^1$, while an input guard behaves like an assignment where the ‘expression’ being evaluated is denoted by the channel identifier.

$$\lbrack \text{Act}(c!e) \rbrack \equiv \langle \theta' \rangle$$

$$\lbrack \text{Act}(c?v) \rbrack \equiv \{ v \mapsto \epsilon \}'$$

First note that normal Handel-C does not permit channel names to act as expressions. This is an extension we introduce to facilitate the semantics. The channel expression denotes the value of the expression associated with the active output guard on that channel. Our semantics ensure that at this point there is exactly one such, and it is elsewhere. The value is obtained by looking up the communication request data, as will be described shortly.

Why does the output guard take a passive role here? Why do we not get it to do an assignment of $e$’s value to $c$ (considered as a variable)? The reason is that we have been very careful to avoid a problem in our semantics analogous to race conditions in hardware circuits. If we treat this assignment as atomic, then it occurs simultaneously with the corresponding input guard assignment, in which case the value used by the latter will be the prior value assigned to that channel. If we treat it as subatomic, then it is an event which occurs after the Selector event $s(g_0, \ldots, g_{n-1})$ which leads to that guard being activated. Unfortunately, the input and output guards belong to distinct $\text{prialt}$s running in parallel. When doing calculations with the semantics given here, if we considered the input branch first, we would obtain an old value for $e$, rather than the value relevant to this current cycle. We do not want to have to stipulate an order in which to evaluate the semantics of components of parallel streams, especially as that order can only be determined at ‘runtime’. This is not compatible with the notion of denotational semantics, that the semantic model should be compositional in character. Accordingly, we have been very careful, in our ordering of: subatomic changes (lodging requests into environment); then subatomic selections (judging requests); and finally atomic changes (assignments and channel communications), to ensure all calculations occur in a safe order.

Also, it is not possible to work out a channel’s value before selections are done as there may be many possible expressions associated with any given channel at any given time in different $\text{prialt}$s. Only one of these, at most, will get selected. Given all of these considerations, it is simpler to let the output guard ‘do nothing’ while the input guard uses the channel and communication request portion of the environment to determine the expression value.

The determination of which $\text{prialt}$ statements are going to run in any given cycle, and which guards therein, is determined by a $\text{prialt}$ resolution function $\mathcal{R}$. 
This takes a set of requests (guard sequences) and returns two components 
(\(\gamma, B\)): a map from channels (\(\gamma\)) deemed to be active to the two 
prialts involved, and the set of prialts (\(B\)) which have not been enabled (‘blocked’) 
this time around.

\[
\mathcal{R} : \mathcal{P}(G^+) \rightarrow (Ch \rightarrow \mathcal{P}(G^+)) \times \mathcal{P}(G^+)
\]

Space does not permit us to describe the behaviour of this function here, 
but it is described at length in [2]. Instead we simply note that all the sets 
comprising the range of \(\gamma\) along with \(B\) constitute a partitioning of the input set. 
Also, if an output guard \(cle\) in prialt \(p_i\) is enabled, along with a 
number of input guards \(c?v_i\) in \(p_i\), for \(i \in 2 \ldots n\), then \(\gamma\) maps \(c\) to precisely 
\(\{p_1, p_2, \ldots, p_n\}\). As already mentioned in the introduction, this part of our 
separation of concerns. The priority resolution semantics can be treated separately in this fashion precisely because this is a synchronous language, and 
as a consequence, both the time when priority is resolved and the identities 
of the participating prialts is well-defined.

We can now describe how Handel-C expressions are evaluated, paying specific attention to the extra expression forms we introduced.

\[\lfloor . \rfloor : E \rightarrow World \rightarrow Datum\]

Regular Handel-C expressions are evaluated w.r.t the environment in the 
‘traditional’ manner:

\[\lfloor e \rfloor_\omega \doteq \ldots\]

The ‘wait’ expression has a guard sequence as argument, and returns true 
if that guard is reported as un-enabled when all the requests lodged in the environment are resolved.

\[\lfloor w\langle g_0, \ldots, g_{n-1} \rangle \rfloor_\omega \doteq \text{let } (\gamma, B) = \mathcal{R}(\omega(\emptyset)) \text{ in } \langle g_0, \ldots, g_{n-1} \rangle \in B\]

The ‘select’ expression has a guard sequence as argument, and is only ever evaluated if the corresponding ‘wait’ expression returns \textit{False}. In this case it looks up the resolution result and searches the channel map to find the guard sequence, and returns the index in that sequence of the enabled channel:

\[\lfloor s\langle g_0, \ldots, g_{n-1} \rangle \rfloor_\omega \doteq \text{let } (\gamma, B) = \mathcal{R}(\omega(\emptyset)) \text{ and } \{ c \} = \gamma^{-1}\{ \langle g_0, \ldots, g_{n-1} \rangle \} \text{ in indexof}[c]\langle g_0, \ldots, g_{n-1} \rangle\]

Treating a channel identifier as an expression only occurs when resolution returns it as enabled, and hence mapping to several guard sequences. We simply look the channel up in \(\gamma\), convert both sequences to sets and merge them and look for the single occurrence of that channel in an input guard, and return the value of the associated expression

\[\lfloor c \rfloor_\omega \doteq \text{let } (\gamma, B) = \mathcal{R}(\omega(\emptyset)) \text{ and } P = \gamma(c) \text{ in } \lfloor (\text{extract}[c] \circ \text{elems}) P \rfloor_\omega\]

\[\text{extract}[c]\{ cle, \ldots \} \doteq e\]
5.1 Restoring Determinism

At this point we have given a semantics to a Handel-C program as a branching sequence which captures all possible execution paths. It is useful however, to define a pruning operation which takes such a tree, and effectively determines which choices are actually made.

\[
\text{prune} : \text{World} \rightarrow \mathcal{M} \rightarrow \text{World}^* \times \mathcal{M}
\]

If we assume a closed world, with no external communication, then the program is deterministic, and can be reduced to a sequence of worlds, by starting with the initial world and computing through all the choices. Even with external communication, we can still view the program as deterministic in between external interactions. Furthermore, the fact that a “core” is synchronously clocked means it is very easy to give time-stamps to the moments when external interactions occurs. This points towards the use of timed-CSP \cite{15} as a formalism for describing how “cores” interact with each other and the external environment. We prefer to use a CSP-base formalism, as these admit refinement in a natural manner, (indeed have been designed with refinement in mind). This clearly continues as long as there is a branching sequence to be processed

\[
\text{prune}[\omega] \Lambda \triangleq (\Lambda, \Lambda)
\]

If we have a subatomic event at the head, we simply perform it and use the updated environment to continue, effectively merging all consecutive subatomic events into one.

\[
\text{prune}[\omega](\kappa : \sigma) \triangleq \text{let } \varpi = \omega \uparrow \omega(\kappa) \text{ in } \text{prune}[\varpi]\sigma
\]

If we have an atomic event, we perform it, which results in a modified world. We then start the next clock cycle using the new world and an empty communication request state.

\[
\text{prune}[\omega](\kappa' : \sigma) \triangleq \text{let } \varpi = \omega \uparrow \omega(\kappa) \text{ and } (\Omega, \Upsilon) = \text{prune}[\varpi \uparrow \{R \mapsto \emptyset\}]\sigma \text{ in } (\varpi : \Omega, \Upsilon)
\]

When we encounter a branch, we evaluate the selector to determine which branch to use (noting that the world is left unchanged)

\[
\text{prune}[\omega](s(\sigma_0 | \ldots | \sigma_{n-1})) \triangleq \text{let } i = s(\omega) \text{ in } \text{prune}[\omega]\sigma_i
\]

The prune operator is important, as it is the appropriate basis for defining program equivalence. We shall deem \(p_1\) and \(p_2\) to be equivalent if they have the same identifier spaces, and pruning their semantic models w.r.t. to any arbitrary starting world always gives the same result

\[
p_1 \equiv p_2 \triangleq \forall \omega \in \text{World}_I \bullet \text{prune}[\omega][p_1] = \text{prune}[\omega][p_2]
\]

where \(I = \text{pIds}[p_1] = \text{pIds}[p_2]\).

While this notion of equivalence seems quite parochial, being limited to discussing alternative descriptions of a single “core”, it is still useful as a basis...
for reasoning about the design and implementation of same. This is a task which we will want to keep separate from the more global concerns to do with asynchronous interactions between cores.

We have not discussed the distinction between internal and external channels. In order to handle external channels, we need to be able to distinguish them from internal channels, and to flag when they are involved in communication requests. We would then use a modified prune operator, which halts whenever an external channel is among those being requested. The semantic model dealing with the asynchronous environment would determine how to proceed. The details of this will not have a major impact on the rest of the semantic model, as only pruning is affected in any deep way. This is one beneficial consequence of our separation of concerns.

6 Fixpoints

We have used fixpoints in our semantics, so we need to ensure that these exist. We shall revisit branching sequences, extending the structure with an explicit bottom element ($\bot$).

$$\sigma \in T ::= \bot | \Lambda | a : \sigma | s(\sigma_0|\ldots|\sigma_{N-1}) \quad \text{where} \quad N > 0$$

The parameter type $S$ (selector) will be considered as having extra structure, defined in terms of a basic selector type $B$. An element of a basic selector type $B$ is something which “determines” a natural number in the range $0\ldots N-1$, where $N > 1$ is the “size” of the selector. For a selector $b \in B$ we write $[b] : B \to \mathbb{N}$ to denote the resulting value. We use $\text{size}(b)$ to denote a selector’s size, and $\text{range}(b) = \{0\ldots\text{size}(b) - 1\}$ to denote the range of possible selector values. Often we wish to denote a selector type of a given size, so we use notations $2, 3, \ldots, N$ to denote the sets of all selectors of sizes $2, 3, \ldots,$ and $N$ respectively. An important structural constraint on $T$ is that in $b(\sigma_0|\ldots|\sigma_{N-1})$ we always have $\text{size}(b) = N$. The domain $S$ is built from $B$ as follows:

$$S = B + S \times \mathbb{N} \times S$$

$$s \in S ::= b \mid s_1 \times_N s_2 \quad \text{where} \quad N = \text{size}(s_2)$$

We have the property that size is multiplicative:

$$\text{size}(s_1 \times_N s_2) = \text{size}(s_1) \cdot \text{size}(s_2)$$

and we extend $[\_]$ to act over $\times_N$ as follows:

$$[s_1 \times_N s_2] \equiv N \cdot [s_1] + [s_2]$$

In general we shall write $s_N(\sigma_i | \ldots | \sigma_i | \ldots | \sigma_{N-1})$, $i \in 0\ldots N - 1$. We also often omit the $N$ subscript when irrelevant or clear from context. We assume that the domains $A$ and $B$ described above all contain
finite elements only, and equality is decidable in every case. This is the case with our instantiations for \( A \) and \( B \), namely \textit{Choice} and \textit{Selector}.

We define the ordering as the smallest relation \((\preceq)\) satisfying:

\[
\bot \preceq \sigma
\]

\[
a : \sigma \preceq a : \tau \quad \equiv \quad \sigma \preceq \tau
\]

\[
s_N(|\sigma_i|) \preceq s_N(|\tau_i|) \quad \equiv \quad \sigma_i \preceq \tau_i, \quad \text{for all } i \in \{0 \ldots N - 1\}
\]

\[
\sigma \preceq \tau \quad \equiv \quad \sigma = \tau
\]

See Figure 1 for a portion of the Hasse Diagram. All directed sets are chains, either of the form

\[
\{ \bot, a : \bot, a : b : \bot, a : b : c : \bot, \ldots \}
\]

or the form

\[
\{ \bot, s_1(\bot, \ldots, \bot), s_1(s_2(\bot, \ldots, \bot), \ldots, \bot), \ldots \}
\]

or a mixture of the two. In order to obtain a complete partial order, we need to admit infinitely long chains, also required by our semantics for iteration. Technically we can do this by taking the definition

\[
T = 1 + 1 + A \times T + S \times T^*
\]

as defining a co-algebra.
To establish the existence of fixpoints, we also need to show that the operators and functions in our semantics are monotonic. In what follows we state all relevant theorems and give sketch proofs where those are straightforward. For more complex proofs we go into more detail.

**Theorem 6.1** The ‘cons’ constructor is monotonic

\[ \sigma \preceq \sigma' \Rightarrow a : \sigma \preceq a : \sigma' \]

**Proof.** Immediate by definition of ordering, which strengthens above implication to an equivalence. \( \square \)

**Theorem 6.2** The ‘split’ constructor is monotonic

Show, for \( i \in 0 \ldots N - 1 \):

\[ \sigma \preceq \sigma' \Rightarrow s(\sigma_0 | \ldots | \sigma_{i-1} | \sigma | \ldots | \sigma_{N-1}) \preceq s(\sigma_0 | \ldots | \sigma_{i-1} | \sigma' | \ldots | \sigma_{N-1}) \]

**Proof.** Immediate by definition of ordering, which strengthens above implication to an equivalence, and noting that \( \sigma \preceq \sigma \) for all \( \sigma \). \( \square \)

**Definition 6.3** We extend the definition of \( \circ \) to handle \( \bot \), treating it as a left-zero.

\[ \bot \circ \sigma \equiv \bot \]

**Theorem 6.4** \( \circ \) is monotonic in its 2nd argument

\[ \sigma \preceq \sigma' \Rightarrow \tau \circ \sigma \preceq \tau \circ \sigma' \]

**Proof.** By induction on structure of \( \tau \). \( \square \)

**Theorem 6.5** \( \circ \) is monotonic in its 1st argument

\[ \sigma \preceq \sigma' \Rightarrow \sigma \circ \tau \preceq \sigma' \circ \tau \]

**Proof.** By induction over the clauses of the definition of \( \preceq \) (which can be converted systematically into a double induction over \( \sigma \) and \( \sigma' \), noting that any such pair not matching a clause of said definition will return a false antecedent, so making the property true as \( \text{FALSE} \Rightarrow P \equiv \text{TRUE} \)). Note also that the three cases below subsume that where \( \sigma = \sigma' \).

**Base Case 1** (\( \sigma = \bot \)): Trivial.

**Inductive Step 1** (\( \sigma, \sigma' \) = (\( a : \varsigma, a : \varsigma' \))):

We assume it is true for (\( \varsigma, \varsigma' \)) (Inductive Hypothesis)

\[ a : \varsigma \circ \tau \preceq a : \varsigma' \circ \tau \]

\[ = \langle \text{defn } \circ \rangle \]
\[
\begin{align*}
\text{Base Case (}) & : (ς \vartriangleright \tau) \preceq a : (ς' \vartriangleright \tau) \\
& = \langle \text{defn } \preceq \rangle \\
& \varsigma \vartriangleright \tau \preceq \varsigma' \vartriangleright \tau \\
& = \langle \text{assumption, inductive hypothesis} \rangle \\
& \text{TRUE}
\end{align*}
\]

**Inductive Step 2** \((\sigma, \sigma') = (s(ς_0|...|ς_{N-1}), s(ς'_0|...|ς'_{N-1}))\):

We assume it is true for each \((ς_i, ς'_i)\) (Inductive Hypothesis)

\[
\begin{align*}
\sigma \vartriangleright \sigma' & \Rightarrow \sigma(\tau) \preceq \sigma'(\tau) \\
& = \langle \text{defn } \preceq \rangle \\
& s(ς_0|...|ς_{N-1}) \vartriangleright \sigma(\tau) \preceq s(ς'_0|...|ς'_{N-1}) \vartriangleright \tau \\
& = \langle \text{defn } \preceq \rangle \\
& \bigwedge_{i=1...N}(ς_i \vartriangleright \varsigma'_i \vartriangleright \tau) \\
& = \langle \text{assumption, inductive hypothesis} \rangle \\
& \text{TRUE}
\end{align*}
\]

\[\square\]

**Definition 6.6** We extend the definition of \(\parallel\) to handle \(\perp\), treating it as a zero:

\[
\perp \parallel σ \equiv \perp \equiv σ \parallel \perp
\]

**Theorem 6.7** \(\parallel\) is monotonic in its 1st argument

\[
\sigma \preceq \sigma' \Rightarrow σ(\tau) \preceq σ'(\tau)
\]

**Proof.** By induction on the case definitions of \(\preceq\).

**Base Case** \((\sigma = \perp): \text{ Trivial.}\)

**Inductive Step 1** \((σ, σ') = (a : ς, a : ς'): \text{ We assume it is true for } (ς, ς') \text{ (Inductive Hypothesis)} \)

Assuming \(ς \preceq ς'\) \text{ (Inductive Hypothesis)}

Show
\[
\begin{align*}
a : ς \preceq a : ς' & \Rightarrow a : ς(\tau) \preceq a : ς'(\tau) \\
\end{align*}
\]

As \(A \Rightarrow (B \Rightarrow C) \equiv A \land B \Rightarrow C\) we change the problem to that of assuming:

\(ς \preceq ς'\) \Rightarrow \(ς(\tau) \preceq ς'(\tau)\) \text{ and } \(a : ς \preceq a : ς'\)

in order to show
\[
\begin{align*}
a : ς(\tau) \preceq a : ς'(\tau)
\end{align*}
\]

This can be proved by an induction on the structure of \(\tau\). The only point worthy of note is that one of the inductive cases involves a three-way case-split. We have to show

\[
\begin{align*}
a : ς(\tau) \preceq a : ς'(b : τ')
\end{align*}
\]

17
We eventually get a case split according to the atomicity of \(a\) and \(b\) — both of same atomicity; \(a\) atomic with \(b\) subatomic; and finally \(a\) subatomic with \(b\) atomic. Each case split is straightforward.

**Theorem 6.8** \(\parallel\) is monotonic in its 2nd argument

\[
\sigma \preceq \sigma' \Rightarrow \tau \parallel \sigma \preceq \tau \parallel \sigma'
\]

**Proof.** Similar to 1st argument proof

7 Related and Future Work

Concurrency with interleaving semantics and asynchronous communication is comprehensively dealt with in the theory of CSP \[14\]. Extensions to that theory have been developed by Adrian Lawrence to deal with priority (\(CSPP\)) \[9,8,10\] and hardware clocks (\(HCSP\)) \[11\]. However, the \(CSPP\) and \(HCSP\) theories are complex as they cater for a wider and more general range of circumstances than are found in Handel-C. By keeping our semantics separate and simpler, it is easier to ensure that it correctly captures the behaviour of the language. It would be instructive however, to re-encode the semantics using \(CSPP/HCSP\) for comparative purposes.

However the main goal of this work is to arrive at a formal development methodology to allow the formal developments of Handel-C programs from formal specifications in some appropriate notation. To this end, the next step to be taken is to provide an axiomatic semantics for Handel-C, i.e. a set of laws of Handel-C programming. The denotational semantics will be used as a model for the axiomatic laws. For example, we could prove the algebraic (axiomatic) law

\[
p_1 \parallel p_2 = p_2 \parallel p_1
\]

by showing, using our denotational model that:

\[
p_1 \parallel p_2 \equiv p_2 \parallel p_1
\]

as defined in the semantics section. In general a law \(p = q\) can be proved as \(p \equiv q\) or \(\text{prune}[\omega][p] = \text{prune}[\omega][q]\), for all possible relevant worlds \(\omega\). We hope to be able to integrate those laws into the Circus refinement calculus framework \[18,9\] in order to provide the desired formal methodology, paying particular attention to using a timed model for same \[16\], in order to incorporate the asynchronous interfaces.

8 Conclusions

We have presented the key semantic domains and semantic equations for a denotational semantics for a subset of Handel-C dealing with synchronously clocked “cores”. The key contribution is that our semantics deals with a
Butterfield and Woodcock concurrent, deterministic language where events occur synchronously, in the presence of global shared variables. Our semantic model has been designed to exploit any features of Handel-C which make the model simpler. An important aspect of our work is the separation of the semantic model into relatively independent parts. The interface between the “core” semantics presented here, and the prialt semantics in [2] is very simple, and facilitates not only the denotational semantics presented here, but also the axiomatic and operational semantics that are preparation.

We are adopting an approach to semantics that fits in with the CSP style of modelling concurrency. This is motivated by our desire to incorporate this theory into the UTP framework of Hoare and He [6]. We observe that the semantics of CSP has already been expressed in UTP [6, pp207–216], and refinement calculus exists [18,3], for which a timed variant is under development [16]. This ensure a close fit with our current approach, and with our intention to used timed-CSP to capture the asynchronous/external aspects of the semantics of Handel-C.

References


Butterfield and Woodcock


