A Denotational Semantics for Handel-C

Andrew Butterfield

Trinity College Dublin
Andrew.Butterfield@cs.tcd.ie

Abstract. We present a denotational semantics for a fully functional subset of the Handel-C hardware compilation language [1], 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 `prialt` (priority-alternation) construct of Handel-C. We then define the typed assertion traces over an abstract notion of actions, which we then instantiate as state-transformers. The denotational semantics is then given and some examples are discussed. As is fitting given those honoured at the Festschrift of which this paper is a part, we show how the work of both Dines Björner and Zhou Chaochen act as inspiration, from the past, into the future for this research work.

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] [1] 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 [2] and C, designed to target hardware implementations, specifically field-programmable gate arrays (FPGAs) [3]. 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.

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 [4]. 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.

[1] Handel-C is the registered trademark of Celoxica Ltd (`www.celoxica.com`)

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 ($x \in \mathit{Var}$), and we assume the existence of an expression syntax ($e \in \mathit{Exp}$) whose details need not concern us here. We also have identifiers for channels ($c \in \mathit{Ch}$), 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 ($g \in \mathit{Grd}$), 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 ($!?$).

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

A syntax of a process $p : \mathit{Proc}$ is as follows:

$$\begin{aligned} p ::= {} & \mathbf{0} \mid \mathbf{1} \mid x := e \\ & \mid p_1 \,;\, p_2 \mid p_1 \parallel p_2 \mid p_1 \lhd b \rhd p_2 \mid b * p \\ & \mid \langle g_i \rightarrow p_i \rangle_{i \in 1 \ldots n} \end{aligned}$$

The last clause is shorthand for a list of guard-process pairs.

2.2 Behaviour

We can briefly summarise the behaviour of a Handel-C process as follows: $\mathbf{0}$ does nothing, in zero time; $\mathbf{1}$ 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; $(p_1 \parallel p_2)$ runs both p_1 and p_2 in lock-step parallel, terminating when they have both finished; $(p_1 \lhd b \rhd p_2)$ evaluates b and executes p_1 immediately if b is *True*, otherwise it runs p_2; and $b * P$ tests b and if *True* it runs P and then repeats, otherwise it terminates.

The $\langle g_i \rightarrow p_i \rangle$ construct ("prialt") is an ordered sequence of guard-process pairs. Guards are either communication actions or a default guard to be activated if no communication guard is active. The default guard, if present, must be last. 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 [5].

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 [1] 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 `prialt`s 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 [6, 7]. The approach adopted was in the style of the "Irish School" of the VDM [8] which drew its inspiration from the pioneering work in VDM of Dines Björner and his colleagues [9].

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. Also, we viewed the task of developing a formal semantics for Handel-C as being an true exercise in domain modelling [10, 11], as our intention was to model an existing artefact, warts and all, rather than construct a nice simple well-behaved hardware process algebra.

A formal description of `prialt` resolution without consideration of default clauses was presented in [12]. An initial denotational semantics was developed [13] which incorporated this `prialt` resolution semantics. Then the `prialt` model of [12] was then extended to handle default clauses properly and an operational semantics for Handel-C incorporating this was developed [4, 5]. 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]. and in the more general process algebra areas [18–20]. 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 [20]

Other work involving formal techniques and Handel-C has been reported, and includes the use of the Ponder policy specification language [21] as a basis for implementing firewalls [22], as well as techniques for performing behavioural transformations from Haskell programs into Handel-C implementations [23]. Beyond the scope of Handel-C, there is considerable work on using formal techniques to develop safety-critical embedded systems, of which the languages Esterel [24–26] and Lustre [27, 28] are two key examples.

4 Overview of `prialt` Semantics

We present here a brief overview of the `prialt` semantics presented in [5], 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 `prialt`s 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 [12] 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 an *Channel-Prialt Map* (*CPMap*) and the set of `prialt`s that have remained blocked.

$$\begin{aligned} PriSet &= \mathcal{P}\, Prialt \\ CPMap &= Ch \rightarrow PriSet \\ Resltn &= CPMap \times PriSet \\ Resolve &: PriSet \rightarrow Resltn \end{aligned}$$

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 model the semantics of `prialt`, 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 an example involving default clauses in that manner that causes most semantic difficulty. This example has two `prialt`s in parallel, with the second having a default clause which itself contains a statement that subsequently invokes a `prialt`:

$$\langle\, c!66 \rightarrow \mathbf{0} \,\rangle \;||\; \langle\, d!99 \rightarrow \mathbf{0},\; !? \rightarrow (\, b * P;\; \langle c?x \rightarrow \mathbf{0}\rangle \,)\,\rangle$$

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 look like this:

$$\langle\, c!66 \rightarrow \mathbf{0} \,\rangle \;||\; \langle\, c?x \rightarrow \mathbf{0} \,\rangle$$

This requires us to lodge a new request, 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.

Prialts nested inside default clauses of other prialts may become active in the same clock cycle as those enclosing prialts, which requires us to iterate the *sel–req–res* loop several times, in any given clock cycle. Managing this micro-cycle activity severely complicates the semantics[2].

5 Semantic Framework

The "prialt-free" denotational semantics in [13], inspired by [29], was based on the notion of "branching sequences" or trees, where non-branching sequences denoted deterministic sequences of actions, and branching was used to model a choice point, such as the conditions of a while-loop or if-statement.

However, this model becomes far too complex when faced with the need to handle multiple choice points per clock-cycle, so the full semantics described here is given in terms of sets of "typed assertion traces". These are sets of sequences of

[2] Interestingly, the underlying hardware doesn't iterate, as it computes what is to be active in any given clock cycle using combinatorial logic.

actions (state-transformers), each action typed according to the phase in which it occurs (*sel*,*req*,*res*,*act*), with an assertion that indicates the conditions under which that action (and all subsequent) may proceed.

This switch also brings the semantics more in line with that of Circus [30] and its slotted variants [31], fitting in with plans to give a complete account of Handel-C and hardware compilation in the UTP framework [32].

5.1 Abstraction of Action, States and Predicates

We shall now present an abstract view of typed assertion traces, where actions $(a : Act)$ with an action merge operator $\Diamond$ form a commutative monoid, with the "null" action nop as identity.

$$\mathbf{Mon}(Act, \Diamond, \mathsf{nop})$$

We the introduce an abstract notion of a state $(s \in St)$ as something which can change as a result of actions, and denote the effect of action a on state s by $\triangle[a](s)$. The null action, not unexpectedly, brings about no change of state:

$$\triangle : Act \to St \to St \qquad \triangle[\mathsf{nop}](s) = s$$

We need predicates over states (assertions), with *true* and *false* denoting the everywhere true and false predicates respectively:

$$p \in Pred = St \to \mathbb{B}$$

We are going to capture the linkage between assertions and actions by the concept of a "guarded-action" (g), which is a predicate-action pair (p, e):

$$g, (p, a) \in GA = Pred \times Act$$

We will frequently deal with cases where either the guard is *true* or the action is nop, so we adopt the shorthands where a denotes $(true, a)$ and $\underline{p}$ denotes (p, nop). In particular we often refer to $\underline{true}$ as a null or void action.

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

$$(p_1, a_1) \Diamond (p_2, a_2) \ \widehat{=}\ (p_1 \wedge p_2, a_1 \Diamond a_2)$$

These guarded actions are the basic building blocks for "assertion traces", so the next step is to describe the typing aspects.

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.

$$\tau \in Trc = Slot^* \cup Slot^\omega$$

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 actions 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 action. We shall refer to the second component as the "final action" of the slot

$$s, (\mu, a) \in Slot = MS^* \times GA$$

A microslot (m) captures the actions in one cycle of selection-request-response and hence is is a triple of guarded actions (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:

$$m, (s, q, r) \in MS = GA^3$$

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

We need to be careful how traces and slots are interpreted: In essence, a slot where the final action 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. T

We need to be able to identify a null slot, as one with no microslots, and a null final-action:

$$\begin{aligned} \mathsf{nils} &: Slot \\ \mathsf{nils} &\mathrel{\widehat{=}} (\langle\rangle, \underline{true}) \end{aligned}$$

A trace in which no actions, 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 action is an final-action which denotes a clock-tick — we overload the notation ‡ to denote both such a clock-tick action, and the corresponding slot. We also expect that merging this action with any non-null action will result in that non-null action:

$$\begin{aligned} ‡ &: Act \qquad ‡ \Diamond a = a, \qquad a \neq \mathsf{nop} \\ ‡ &: Slot \qquad ‡ \mathrel{\widehat{=}} (\langle\rangle, ‡) \end{aligned}$$

Typing The typing of actions in slots and microslots is implicitly given by the actions' 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:

$$\begin{aligned} t \in TType &\mathrel{\widehat{=}} \{\, sel, req, res, act \,\} \\ & \quad sel < req < res < act \end{aligned}$$

We define the type of a microslot as the type of the least non-empty action present:

$$\begin{aligned} ttype_{MS} &: MS \to TType \\ ttype_{MS}(\underline{true}, \underline{true}, \_) &\mathrel{\widehat{=}} res \\ ttype_{MS}(\underline{true}, \_, \_) &\mathrel{\widehat{=}} req \\ ttype_{MS}(\_, \_, \_) &\mathrel{\widehat{=}} sel \end{aligned}$$

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

$$\begin{aligned} ttype_S &: Slot \to TType \\ ttype_S(\langle\rangle, (p, \_)) &\mathrel{\widehat{=}} act \\ ttype_S((m : \_), \_) &\mathrel{\widehat{=}} ttype_{MS}(m) \end{aligned}$$

5.3 Trace Operators

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

Building with single actions The first are a series of constructors that construct slots of the various types from a single guarded action and accompanying transition type. We shall refer to the combination of a transition type and guarded action as a *typed action*.

Given a non-*act*, non-void action, we wish to build the corresponding microslot:

$$\begin{aligned} mkm &: TType \to GA \to MS \\ mkm_{sel}(g) &\mathrel{\widehat{=}} (g, \underline{true}, \underline{true}) \\ mkm_{req}(g) &\mathrel{\widehat{=}} (\underline{true}, g, \underline{true}) \\ mkm_{res}(g) &\mathrel{\widehat{=}} (\underline{true}, \underline{true}, g) \end{aligned}$$

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

$$\begin{aligned} mks &: TType \to GA \to Slot \\ mks_{act}(g) &\mathrel{\widehat{=}} (\langle\rangle, g) \\ mks_t(g) &\mathrel{\widehat{=}} (\langle mkm_t(g)\rangle, \underline{true}) \end{aligned}$$

Lifting action Merging We want to lift the action merge operators to work with microslots.

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

$$\begin{aligned} \_ \Diamond \_ \_ &: GA \times TType \times MS \to MS \\ g \Diamond_{sel} (s, q, r) &\mathrel{\widehat{=}} (g \Diamond s, q, r) \\ g \Diamond_{req} (s, q, r) &\mathrel{\widehat{=}} (s, g \Diamond q, r) \\ g \Diamond_{res} (s, q, r) &\mathrel{\widehat{=}} (s, q, g \Diamond r) \end{aligned}$$

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

Typed Cons-ing By "typed cons-ing" (:: or $::_t$) we mean the process of placing a typed action at the start of an existing list of actions, at the microslot, slot or trace level. We first consider cons-ing a non-*act*, non-null action into a microslot or slots. If the action 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 action. This is because "consing" means pre-pending an earlier action, so if an action of type *res* (say) is being placed in front of a microslot containing *sel* or *req* actions, then it must have occurred in an earlier microslot. This is why the signature of the function indicates that merging a typed action with a microslot may result in more than one microslot as a result.

$$\begin{aligned} \_ ::_{\_} \_ &: GA \to TType \to MS \to MS^+ \\ g ::_t m &\mathrel{\widehat{=}} \textbf{if } t > ttype_{MS}(m) \\ &\quad \textbf{then } \langle mkm_t(g), m \rangle \textbf{ else } \langle g \Diamond_t m \rangle \end{aligned}$$

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

$$\begin{aligned} \_ ::_{\_} \_ &: GA \to TType \to MS^* \to MS^* \\ g ::_t \langle\rangle &\mathrel{\widehat{=}} \langle mkm_t(g) \rangle \\ g ::_t (m : \mu) &\mathrel{\widehat{=}} (g ::_t m) \frown \mu \end{aligned}$$

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

$$\begin{aligned} \_ ::_{\_} \_ &: GA \to TType \to Slot \to Slot^+ \\ g ::_{act} s &\mathrel{\widehat{=}} \langle mk_{act}(g), s \rangle \\ g ::_t (\mu, a) &\mathrel{\widehat{=}} \langle (g ::_t \mu, a) \rangle \\ \_ ::_{\_} \_ &: GA \to TType \to Trc \to Trc \\ g ::_t (s : \tau) &\mathrel{\widehat{=}} (g ::_t s) \frown \tau \end{aligned}$$

Concatenation for Microslots We can now define a form of concatenation for microslots ($\fatsemi$) 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 action in the ante-slot has a type greater than that of an action in the post-slot. We first define an operator ($\boxplus$) taking a pair of micro-slots to a sequence of same:

$$\begin{array}{rcl} \_ \boxplus \_ & : & MS^2 \rightarrow MS^* \\ (s_1, \underline{true}, \underline{true}) \boxplus (s_2, q_2, r_2) & \widehat{=} & \langle (s_1 \Diamond s_2, q_2, r_2) \rangle \\ (s_1, q_1, \underline{true}) \boxplus (\underline{true}, q_2, r_2) & \widehat{=} & \langle (s_1, q_1 \Diamond q_2, r_2) \rangle \\ (s_1, q_1, r_1) \boxplus (\underline{true}, \underline{true}, r_2) & \widehat{=} & \langle (s_1, q_1, r_1 \Diamond r_2) \rangle \\ m_1 \boxplus m_2 & \widehat{=} & \langle m_1, m_2 \rangle \end{array}$$

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

$$\begin{array}{rcl} \_ \fatsemi \_ & : & MS^* \times MS^* \rightarrow MS^* \\ \langle\rangle \fatsemi \mu_2 & \widehat{=} & \mu_2 \\ \mu_1 \fatsemi \langle\rangle & \widehat{=} & \mu_1 \\ \langle m_1 \rangle \fatsemi (m_2 : \mu_2) & \widehat{=} & (m_1 \boxplus m_2) \frown \mu_2 \\ (m_1 : \mu_1) \fatsemi \mu_2 & \widehat{=} & m_1 : (\mu_1 \fatsemi \mu_2) \end{array}$$

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* action:

$$\begin{array}{rcl} \_ :: \_ & : & Slot \times Trc \rightarrow Trc \\ s :: \langle \mathsf{nils} \rangle & \widehat{=} & \langle s \rangle \\ \mathsf{nils} :: \tau & \widehat{=} & \tau \\ (\mu, \underline{true}) :: ((\nu, a') : \tau) & \widehat{=} & ((\mu \fatsemi \nu), a') : \tau \\ s :: \tau & \widehat{=} & s : \tau \end{array}$$

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

$$\begin{array}{rcl} \_ \fatsemi \_ & : & Trc \times Trc \rightarrow Trc \\ \langle\rangle \fatsemi \tau_2 & \widehat{=} & \tau_2 \\ \langle s \rangle \fatsemi \tau_2 & \widehat{=} & s :: \tau_2 \\ (s_1 : \tau_1) \fatsemi \tau_2 & \widehat{=} & s_1 : (\tau_1 \fatsemi \tau_2) \end{array}$$

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 $\|$ for all these forms of parallel merging, except trace parallel merge which we denote by $[\!]$.

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

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

$$\begin{aligned} \_ \| \_ &: MS \times MS \to MS \\ (s_1, q_1, r_1) \| (s_2, q_2, r_2) &\mathrel{\widehat{=}} (s_1 \Diamond s_2, q_1 \Diamond q_2, r_1 \Diamond r_2) \end{aligned}$$

Merging microslot-sequences in parallel ($\|$) is done on a microslot by microslot basis, but not by merging matching pairs starting at the front of both lists, but rather by matching the ends of the lists together with the front of the longer list simply being copied to the result:

$$\begin{aligned} \_ \| \_ &: MS^* \times MS^* \to MS^* \\ \mu_1 \| \mu_2 &\mathrel{\widehat{=}} \mathtt{rev}((\mathtt{rev}\,\mu_1)\ \mathsf{mssaux}\ (\mathtt{rev}\,\mu_2)) \\ \langle\rangle\ \mathsf{mssaux}\ \mu_2 &\mathrel{\widehat{=}} \mu_2 \\ \mu_1\ \mathsf{mssaux}\ \langle\rangle &\mathrel{\widehat{=}} \mu_1 \\ (m_1 : \mu_1)\ \mathsf{mssaux}\ (m_2 : \mu_2) &\mathrel{\widehat{=}} (m_1 \parallel m_2) : (\mu_1\ \mathsf{mssaux}\ \mu_2) \end{aligned}$$

This counterintuitive notion of parallel merge ("merge from the back") was discovered as part of work animating these semantics[33] by encoding them in Haskell [34]. The reason for merging in this way is to ensure that all decisions are made as late as they possibly can be made, in particular to ensure that all the prialts involved in generating microcycles are complete before final communication resolution is done. Intuitively, this reflects how, in the real hardware implementations of Handel-C, we are waiting for combinatorial logic to settle before the clock edge marking the end of the cycle, and the occurrence of the *act*-actions.

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

$$\begin{aligned} \_ \| \_ &: Slot \times Slot \to Slot \\ (\mu_1, a_1) \| (\mu_2, a_2) &\mathrel{\widehat{=}} (\mu_1 \| \mu_2, a_1 \Diamond a_2) \end{aligned}$$

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.

$$\begin{aligned} \_ [\!] \_ &: Trc \times Trc \to Trc \\ \langle\rangle [\!] \tau_2 &\mathrel{\widehat{=}} \tau_2 \\ \tau_1 [\!] \langle\rangle &\mathrel{\widehat{=}} \tau_1 \\ (s_1 : \tau_1) [\!] (s_2 : \tau_2) &\mathrel{\widehat{=}} (s_1 \parallel s_2) : (\tau_1 [\!] \tau_2) \end{aligned}$$

Unlike the microslot-sequence case, here we do merge slot-sequences from the front.

5.5 Framework Summary

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

6 Execution State

We now turn our attention to the actions 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 `prialt` communication requests.

We define identifiers (Id) to be either variable names (Var) or one of four special identifiers $\tau, \Re, \gamma$ or B, not present in Var. We define a value space (Val) to contain integers, booleans and an error value (?), and then define a datum type as being either a value, a function Fun, or one of the three types associated with `prialt` resolution, namely $Resltn$, $CPMap$ and $PriSet$:

$$\begin{aligned} i \in Id &\mathrel{\widehat{=}} Var + \{\, \tau, \Re, \gamma, B \,\} \\ Val &\mathrel{\widehat{=}} \mathbb{Z} + \mathbb{B} + \{\, ? \,\} \\ f \in Fun &\mathrel{\widehat{=}} Var \to Val \\ d \in Datum &\mathrel{\widehat{=}} Val + Fun + Resltn + CPMap + PriSet \end{aligned}$$

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 $x : Val$, or even $x : Datum$, rather than writing the more pedantic but verbose forms of $inj_1(x) : Val$ and $inj_1(inj_1(x)) : Datum$.

We define an environment ρ as a mapping from identifiers to data, subject to the proviso that variables map only to values, $\Re$ maps only to $Resltn$s, and γ and B map respectively to the $CPMap$ and $PriGrp$ components of $\rho(\Re)$:

$$\rho \in Env \mathrel{\widehat{=}} Id \to Datum$$

We denote the updating of a map ρ so that i now maps to d by $\rho \dagger \{i \mapsto d\}$

The identifer τ 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.

Data items of type *Fun* do not form part of the state, but are used as a technical device to capture the fact that expressions in channel output guards are evaluated when that guard goes "live", if ever.

Expression evaluation w.r.t an environment is defined in the normal way, and returns a result of type *Datum* that is not itself of type *Fun*:

$$\begin{aligned} \mathcal{E} &: \mathit{Exp} \to \mathit{Env} \to \mathit{Datum} \\ \mathcal{E}[\![e]\!]\rho &\mathrel{\widehat{=}} \text{"standard" expression evaluation...} \end{aligned}$$

Note however, that the partial application $\mathcal{E}[\![e]\!]$, where e denotes a value of type *Val*, can be interpreted as a *Datum* value result of (sub-)type *Fun*.

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.

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 $\Re, \gamma$ and B.

At the start of each clock cycle, these are initialised to be empty:

$$\rho(\gamma) = \theta \qquad \rho(B) = \emptyset \qquad \rho(\Re) = (\theta, \emptyset)$$

6.4 Actions for Handel-C

We want our actions to be state-transformers, that is functions from state to state, and we need to define the null action (nop), as well as explaining how actions merge ($\Diamond$).

Actions Formally our actions are functions mapping environments into environments, and the null action is simply the identity function on environments. However, we want to capture the notion of actions that change part of the state, and to be able to merge these, and detect if they are both trying to modify the

same variable. With the action model as just described, this is hard to do, so we adopt an alternative model, were we view an action as simply being a partial environment which records the part that changes. The null action is simply the null map (θ), and two actions are merged by simply merging the maps together. Any variable conflict is recognised because the variable occurs in the domain of both maps — in this case we map the value to ? to denote the (runtime) error.

$$\begin{aligned} \mathit{Evt} &\mathrel{\widehat{=}} \mathit{Env} \\ \mathsf{nop} &\mathrel{\widehat{=}} \theta \\ e_1 \Diamond e_2 &\mathrel{\widehat{=}} e_1 \cup e_2, \\ &\qquad \textbf{if } \operatorname{dom} e_1 \cap \operatorname{dom} e_1 = \emptyset \\ \{v \mapsto e_1\} \Diamond \{v \mapsto e_2\} &= \{v \mapsto ?\} \end{aligned}$$

The one exception to map conflicts has to do with the way the communication parts are treated ($\Re, \gamma, B$). Here we find that the basic action involves lodging a `prialt` as a request, into the B component, which is a set of such `prialt`s. Multiple references to B are resolved by applying set union (remember that P_1 and P_2 are of type *PriSet*):

$$\{B \mapsto P_1\} \Diamond \{B \mapsto P_2\} = \{B \mapsto P_1 \cup P_2\}$$

The clock-tick action is simply represented by an environment where the sole identifer in its domain is τ, and the datum to which it maps is immaterial:

$$\begin{aligned} \ddagger &: \mathit{Evt} \\ \ddagger &\mathrel{\widehat{=}} \{\tau \mapsto ?\} \end{aligned}$$

State Change We use such partial maps to change the state by simply overriding the state with a mapping in which any expressions (as *Fun* in *Datum*) have been first evaluated w.r.t that state:

$$\begin{aligned} \mathit{St} &\mathrel{\widehat{=}} \mathit{Env} \\ \triangle(e_1)\rho &\mathrel{\widehat{=}} \rho \dagger \mathcal{E}'_\rho(e_1) \\ \mathcal{E}'_\rho\{v \mapsto f\} &\mathrel{\widehat{=}} \{v \mapsto f(\rho)\} \end{aligned}$$

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

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.

$$\begin{aligned} \mathit{Pred} &\mathrel{\widehat{=}} \mathit{Exp}, \qquad \text{boolean-valued} \\ e(\rho) &\mathrel{\widehat{=}} \mathcal{E}[\![e]\!]_\rho \end{aligned}$$

6.5 Fixpoints

We define an ordering $\preceq$ on traces, with $\tau_1 \preceq \tau_2$ if τ_1 is a prefix of τ_2. We note that $\langle \mathsf{nils} \rangle \preceq \tau$ for any τ. 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 τ_3 such that $\tau_1 \fatsemi \tau_3 = \tau_2$.

We extend this to an ordering $\sqsubseteq$ over sets of traces by saying that $S_1 \sqsubseteq S_2$ if for every τ_1 in S_1 there is a τ_2 in S_2 such that $\tau_1 \preceq \tau_2$. The least element in this ordering is the set $\{ \langle \mathsf{nils} \rangle \}$. Again a notion of least upper bound ($\bigsqcup$) can be defined w.r.t $\sqsubseteq$.

Our semantic domain is therefore one of trace-sets, ordered by $\sqsubseteq$, and our semantic definitions produce directed sets. We therefore handle recursion by taking the least fixed point w.r.t $\sqsubseteq$, and we can compute this as

$$\mathsf{fix}\, L \bullet F(L) = \bigsqcup_{i \in \mathbb{N}} \{ F^i \{ \langle \mathsf{nils} \rangle \} \}$$

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. Given a binary operator $*$ over values s and t of some type we assume the obvious extensions to act between sets S and T over the type, or between elements and sets as follows:

$$\begin{aligned} S * T &\mathrel{\widehat{=}} \{ s * t \mid s \in S \wedge t \in T \} \\ s * T &\mathrel{\widehat{=}} \{ s * t \mid t \in T \} \end{aligned}$$

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 actions which first distinguish them must have mutually exclusive predicates, i.e ones that are never true in the same environment (3) *Exhaustivness:* given all traces in the set with a common prefix, then all the guard predicates of the distinguishing actions must exhaust all possibilities, ie. for any environment, at least one (and only one) will return true. Conditions (2) and (3) are weakened slightly when we consider the semantics of `prialt` later on.

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

$$
\begin{aligned}
[\![\_]\!] &: \mathit{Prog} \to \mathcal{P}\,\mathit{Trc} \\
[\![\mathbf{0}]\!] &\mathrel{\widehat{=}} \{\, \langle \mathsf{nils} \rangle \,\} \\
[\![\mathbf{1}]\!] &\mathrel{\widehat{=}} \{\, \langle \ddagger \rangle \,\} \\
[\![x := e]\!] &\mathrel{\widehat{=}} \{\, \langle (\langle\rangle, \{x \mapsto e\}) \rangle \,\} \\
[\![p;\ q]\!] &\mathrel{\widehat{=}} [\![p]\!] \fatsemi [\![q]\!] \\
[\![p \parallel q]\!] &\mathrel{\widehat{=}} [\![p]\!] \,[\!]\, [\![q]\!] \\
[\![p \lhd b \rhd q]\!] &\mathrel{\widehat{=}} (\underline{b} ::_{sel} [\![p]\!]) \ \cup\ (\underline{\neg\, b} ::_{sel} [\![q]\!]) \\
[\![b * p]\!] &\mathrel{\widehat{=}} \mathsf{fix}\, L \bullet \{\, \langle mk_{sel}(\underline{\neg\, b}) \rangle \,\} \ \cup\ (\underline{b} ::_{sel} ([\![p]\!] \fatsemi L))
\end{aligned}
$$

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. Just like the conditional construct, it also ensures the exclusivity and exhaustiveness criteria are met.

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 *req*, *res* and *act*. We now introduce some extension to the language to facilitate this —note that these extensions exists 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 ($w\langle g_i \rangle$), an active guard expression ($a\langle g_i \rangle$), and a channel data expression ($\delta(c)$):

$$e \in \mathit{Exp} = \ldots \mid w\langle g_i \rangle \mid a\langle g_i \rangle \mid \delta(c)$$

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:

$$\mathcal{E}[\![w\langle g_i \rangle]\!]\rho \mathrel{\widehat{=}} \langle g_i \rangle \in \rho(B)$$

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 $w\langle g_i \rangle$ is false, and looks up the channel-prialt map

$$\begin{array}{rcl} \mathcal{E}[\![a\langle g_i \rangle]\!]\rho & \widehat{=} & \min j \\ \textbf{where } \exists\, c & \bullet & \langle g_i \rangle \in \rho(\gamma)(c) \\ & & \wedge\ \mathit{channel}(g_j) = c \end{array}$$

Here *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 [5].

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

$$p \in \mathit{Prog} ::= \ldots \mid rq\langle g_i \rangle \mid wait\langle g_i \rangle \mid e \blacktriangleright [p_i]$$

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

The case-statement $e \blacktriangleright [p_i]$ 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:

$$\begin{array}{rcl} \mathsf{act}() & : & \mathit{Grd} \rightarrow \mathit{Prog} \\ \mathsf{act}(c!e) & \widehat{=} & \mathbf{1} \\ \mathsf{act}(c?v) & \widehat{=} & v := \delta(c) \\ \mathsf{act}(!?) & \widehat{=} & \mathbf{0} \end{array}$$

We give `prialt` $\langle g_i \rightarrow p_i \rangle$ a semantics by translating it to:

$$rq\langle g_i \rangle;\ wait\langle g_i \rangle;\ a\langle g_i \rangle \blacktriangleright [\mathsf{act}(g_i);\ p_i]$$

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

We can now give the semantics of the additional constructs:

$$
\begin{aligned}
[\![rq\langle g_i\rangle]\!] &\mathrel{\widehat{=}} \{\, \langle mk_{req}(\{B \mapsto \{\, \langle g_i\rangle \,\}\})\rangle \,\} \\
[\![wait\langle g_i\rangle]\!] &\mathrel{\widehat{=}} \mathsf{fix}\ W \bullet \{\, \langle mk_{res}(\underline{\neg\, w\langle g_i\rangle})\rangle \,\} \\
&\qquad\qquad \cup \\
&\qquad\qquad (\underline{w\langle g_i\rangle} ::_{act} ([\![rq\langle g_i\rangle]\!] \fatsemi W) \\
[\![a\langle g_i\rangle \blacktriangleright [p_i]]\!] &\mathrel{\widehat{=}} \bigcup_i \{\, \underline{(a\langle g_i\rangle = i)} ::_{res} [\![p_i]\!] \,\}
\end{aligned}
$$

The request statement is simply an update of the state's "B' component, tagged as occurring during the *req* phase. The case-statement simply prepends a guarded action asserting that $e = i$ to the traces associated with process p_i, such a choice being made during the *res* phase.

The $wait\langle g_i\rangle$ statement is a looping construct, so it has a fixpoint definition as expected. It would seem obvious that $wait\langle g_i\rangle$ should be the same as $w\langle g_i\rangle * (\mathbf{1};\ rq\langle g_i\rangle)$, but it is necessary to keep it separate, because not only do the true and false branches of the *wait* statement not occur in the *sel* phase, but in fact they occur in different phases: the terminating guarded action $(\underline{\neg\ w\langle g_i\rangle})$ occurs during the *res* phase; while the continuation guarded action $(\underline{w\langle g_i\rangle})$ 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 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.

This means that the Exhaustiveness and Exclusivity healthiness conditions aren't quite adhered to at this point, as the conditions $\underline{\neg\ w\langle g_i\rangle}$ and $\underline{w\langle g_i\rangle}$ do not occur at the same point in the traces. In fact the latter is delayed until the *act* phase. The weakening that we allow is that this works because while the *act* condition occurs later in the trace, no events of any significance occur in that trace from the point in the *res* phase where $w\langle g_i\rangle$ could return *false*, up to the *act* point where the predicate can return *true*.

7.2 Examples

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

In order to keep expressions readable and manageable, we introduce the following shorthand: (i) for $(\langle\rangle, x \mapsto e)$ we simply write $\{x \mapsto e\}$; (ii) and for $mks_{req}(\{B \mapsto \{\, \langle g_i\rangle \,\}\})$ we use $req\langle g_i\rangle$. Rather than showing the slot and microcycle structure explicitly, we simply list the actions separated by commas, and use ‡ to to mark the slot boundaries (i.e clock ticks). So

$$\langle\ (\langle\rangle, y \mapsto f),\ (\langle((\underline{b}), \underline{true}, \underline{true})\rangle, x \mapsto e)\ \rangle$$

becomes $\langle \{y \mapsto f\} \ddagger \underline{b}, \{x \mapsto e\} \rangle$

Assignment, Conditional and Sequential Composition If we follow an assignment by a conditional as follows:

$$x := y + z;\ y := z \lhd (x > 0) \rhd z := y$$

then calculating this through with the semantics gives:

$$\begin{aligned}
&[\![x := y + z;\ y := z \lhd (x > 0) \rhd z := y]\!] \\
&= \{\ \langle \{x \mapsto y + z\} \ddagger \underline{x > 0}, \{y \mapsto z\} \rangle, \\
&\qquad \langle \{x \mapsto y + z\} \ddagger \underline{x \leq 0}, \{z \mapsto y\} \rangle\ \}
\end{aligned}$$

We see clearly the same starting action 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.

Parallel Assignment We can swap two variables in one clock cycle:

$$[\![x := y \parallel y := x]\!] = \{\ \{x \mapsto y, y \mapsto x\}\ \}$$

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

$$[\![x := e_1 \parallel x := e_2]\!] = \{\ \{x \mapsto ?\}\ \}$$

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:

$$\begin{aligned}
[\![b * \mathbf{1}]\!] &= \bigsqcup \{\ F^i \{\ \mathsf{nils}\ \} \mid i \in \mathbb{N}\ \} \\
&\quad \textbf{where} \qquad F(L) = \{\ \langle \underline{\neg\, b} \rangle\ \} \cup (\ (\underline{b}, \ddagger) :: L)\ )
\end{aligned}$$

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

$$\begin{aligned}
[\![b * \mathbf{1}]\!] = \{\ &\langle \underline{\neg\, b} \rangle, \\
&\langle \underline{b}, \ddagger;\ \underline{\neg\, b} \rangle, \\
&\langle \underline{b}, \ddagger;\ \underline{b}, \ddagger;\ \underline{\neg\, b} \rangle, \\
&\vdots \\
&\langle \underbrace{\underline{b}, \ddagger;\ \ldots;\ \underline{b}, \ddagger}_{i-1 \text{ times}};\ \underline{\neg\, b} \rangle, \\
&\vdots \\
&\langle \underbrace{\underline{b}, \ddagger;\ \ldots;\ \underline{b}, \ddagger \ldots}_{\infty \text{ times}} \rangle\ \}
\end{aligned}$$

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. We need to show that all this semantics describes the same language as does the operational semantics. 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 [35] or Circus [30].

Recently we have also published a "hardware semantics" for Handel-C [36], which will allow us to explore transformations that investigate the trade-off between the number of clock-cycles required to complete a task, and the length of each cycle, which depends on the complexity of the combinatorial logic that is generated. It is here that the well-known work of Zhou Chaochen on duration calculus [37] and his work on modelling synchronous circuits at switch level [38] will provide useful tools and insight for this work.

Finally, we hope to explore the embedding of these results into the UTP framework [32], as a variant of Circus [30]. This work is being funded as a three-year project by Science Foundation Ireland.

References

1. Celoxica Ltd.: Handel-C Language Reference Manual, v3.0. (2002) URL: `www.celoxica.com`.
2. Hoare, C.A.R.: Communicating Sequential Processes. Intl. Series in Computer Science. Prentice Hall (1985)
3. Page, I., Luk, W.: Compiling Occam into field-programmable gate arrays. In Moore, W., Luk, W., eds.: FPGAs, Oxford Workshop on Field Programmable Logic and Applications. Abingdon EE&CS Books, 15 Harcourt Way, Abingdon OX14 1NV, UK (1991) 271–283
4. Butterfield, A., Woodcock, J.: An operational semantics for handel-c. In Arts, T., Fokkink, W., eds.: Electronic Notes in Theoretical Computer Science. Volume 80., Elsevier (2003)
5. Butterfield, A., Woodcock, J.: prialt in Handel-C: an operational semantics. International Journal on Software Tool for Technology Transfer (STTT) (2005)
6. Butterfield, A.: Denotational semantics for `prialt`-free Handel-C. Technical Report TCD-CS-2001-53, Dept. of Computer Science, Trinity College, Dublin University (2001)

7. Butterfield, A.: Interpretative semantics for `prialt`-free Handel-C. Technical Report TCD-CS-2001-54, Dept. of Computer Science, Trinity College, Dublin University (2001)
8. Mac an Airchinnigh, M.: The irish school of vdm. In: VDM '91. Volume 552 of Lecture Notes in Computer Science. Springer Verlag (1991)
9. Björner, D., Jones, C.B.: Formal Specification & Software Development. Prentice-Hall (1982)
10. Bjørner, D.: TRain: The railway domain - A "grand challenge" for computing science & transportation engineering. In Jacquart, R., ed.: IFIP Congress Topical Sessions, Kluwer (2004) 607–612
11. Bjørner, D.: Software Engineering, vol. 3: Domains, Requirements and Software Design. Texts in Theoretical Computer Science. Springer (2006)
12. Butterfield, A., Woodcock, J.: Semantics of `prialt` in Handel-C. In Pasco, J., Welch, P., Loader, R., Sunderam, V., eds.: Communicating Process Architectures – 2002. Concurrent Systems Engineering, Amsterdam, IOS Press (2002) 1–16
13. Butterfield, A., Woodcock, J.: Semantic Domains for Handel-C. In Madden, N., Seda, A., eds.: Mathematical Foundations for Computer Science and Information Technology (MFCSIT 2003). Volume 74., Electronic Notes in Theoretical Computer Science (2002) `http:// www.elsevier.nl/ locate/ entcs` (to appear).
14. Fidge, C.J.: A formal definition of priority in CSP. ACM Transactions on Programming Languages and Systems **15** (1993) 681–705
15. Lawrence, A.E.: Cspp and event priority. In Alan Chalmers, M.M., Muller, H., eds.: Communicating Process Architectures 2001. Concurrent Systems Engineering, Amsterdam, IOS Press (2001)
16. Lawrence, A.E.: Acceptances, Behaviours and infinite activity in CSPP. In: Communicating Process Architectures – 2002. Concurrent Systems Engineering, Amsterdam, IOS Press (2002)
17. Lawrence, A.E.: HCSP and true concurrency. In: Communicating Process Architectures – 2002. Concurrent Systems Engineering, Amsterdam, IOS Press (2002)
18. Cleaveland, R., Hennessy, M.: Priorities in process algebra. In: Proceedings 3^{th} Annual Symposium on Logic in Computer Science, Edinburgh, IEEE Computer Society Press (1988) 193–202
19. Cleaveland, R., Luettgen, G., Natarajan, V., Sims, S.: Modeling and Verifying Distributed Systems Using Priorities: A Case Study. In: Second International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '96). Volume 1055 of LNCS., Passau, Germany, Springer Verlag (1996) pp287–297
20. Cleaveland, R., Luettgen, G., Natarajan, V.: Priority in process algebra. In Bergstra, J.A., Ponse, A., Smolka, S.A., eds.: Handbook of Process Algebra, Elsevier (2001) pp711–765
21. Damianou, N., Dulay, N., Lupu, E., Sloman, M.: The ponder policy specification language. Lecture Notes in Computer Science **1995** (2001) 18–??
22. Lee, T., Yusuf, S., Luk, W., Sloman, M., Lupu, E., Dulay, N.: Development framework for firewall processors. (`www.celoxica.com`, in Academic Papers section)
23. Abdallah, A.E., Hawkins, J.: Formal Behavioural Synthesis of Handel-C Parallel Hardware Implementations from Functional Specifications. In: 36th Annual Hawaii International Conference on System Sciences (HICSS'03), IEEE (2003)
24. Boussinot, F., Simone, R.D.: The ESTEREL language. Technical Report RR-1487, (Inria, Institut National de Recherche en Informatique et en Automatique)
25. Bouali, A.: Xeve: An Esterel verification environment. Lecture Notes in Computer Science **1427** (1998) 500–??

26. Tini: An axiomatic semantics for esterel. TCS: Theoretical Computer Science **269** (2001)
27. Holenderski, L.: LUSTRE. In Lewerentz, C., Lindner, T., eds.: Formal Development of Reactive Systems, Case Study Production Cell. Lecture Notes in Computer Science. Springer-Verlag (1995) 101–112
28. Andriessens, C., Lindner, T.: AL: Using FOCUS, LUSTRE, and probability theory for the design of a reliable control program. Lecture Notes in Computer Science **1165** (1996) 35–51
29. Brookes, S.D.: On the axiomatic treatment of concurrency. Lecture Notes in Computer Science **197** (1985) 1–34
30. Woodcock, J., Cavalcanti, A.: The Semantics of Circus. Volume 2272 of LNCS., 2nd International Conference of B and Z Users, Grenoble, France, Springer (2002)
31. Butterfield, A., Sherif, A., Woodcock, J.: Slotted-Circus—A UTP-Family of Reactive Theories. In Davies, J., Gibbons, J., eds.: IFM 2007. Volume 4591 of Lecture Notes in Computer Science., Springer (2007)
32. Hoare, C.A.R., Jifeng, H.: Unifying Theories of Programming. Series in Computer Science. Prentice Hall (1998)
33. Corcoran, B.J.: Testing Formal Semantics: Handel-C. M.Sc. in Computer Science, University of Dublin, Trinity College (2005) presented in partial fullfillment of the M.Sc. requirements.
34. Jones, S.P., others.: Report on the Programming Language Haskell 98. http://www.haskell.org/ (1999)
35. Schneider, S.: Concurrent and Real-time Systems — The CSP Approach. Wiley (2000)
36. Butterfield, A., Woodcock, J.: A "hardware compiler" semantics for handel-c. Electr. Notes Theor. Comput. Sci. **161** (2006) 73–90
37. Zhou, C.: Duration calculus, a logical approach to real-time systems. In Haeberer, A.M., ed.: AMAST. Volume 1548 of Lecture Notes in Computer Science., Springer (1998) 1–7
38. Zhou, C., Hoare, C.A.R.: A model for synchronous switching circuits and its theory of correctness. Formal Methods in System Design **1** (1992) 7–28