UTP Semantics for Handel-C

Juan Ignacio Perna and Jim Woodcock

Computer Science Department
The University of York
York - United Kingdom
{jiperna,jim}@cs.york.ac.uk

Abstract. Only limited progress has been made so far towards an axiomatic semantics or discovering the algebraic rules that characterise Handel-C programs. In this paper we present a UTP semantics together with extensions we needed to include in order to express Handel-C properties that were not addressable with standard UTP. We also show how our extensions can be abstracted to a more general context and prove a set of algebraic rules that hold for them. Finally, we use the semantics to prove some properties about Handel-C constructs.

1 Introduction

Handel-C [10] is a Hardware Description Language (HDL) based on the syntax of the C language extended with constructs to deal with parallel behaviour and process communications based on CSP [11]. The language is designed to target synchronous hardware components with multiple clock domains, usually implemented in Field Programmable Gate Arrays (FPGAs).

In this paper we present a denotational semantics for a subset of Handel-C. Our semantics is based on the theory of designs as presented in the Unifying Theories of Programming (UTP) [12]. Special attention is paid to the way in which parallelism is captured, as the UTP model for parallel composition is more restrictive than the one used in Handel-C. The major difference between the two parallel models lies in the fact that the shared-variable parallel model presented in UTP is based on the parallel processes terminating at the same time. As this restriction does not hold for Handel-C programs, we propose an extension of this UTP theory that is capable of handling the kind of parallelism we required. We also used the semantics to prove a set of algebraic rules about Handel-C programs.

We also generalise the notions in our parallel operator for Handel-C and provide a more general parallel operator that is able to handle processes that may take a different amount of clock cycles to finish. We also address the algebraic laws of our operator together with the healthiness conditions that it preserves.

The rest of this paper is organised as follows: section 2 presents the syntax of the subset of Handel-C we address in this work together with an informal account of its semantics. Section 3 presents our parallel-by-merge operator for Handel-C that handles parallel composition of processes of different length. This section

A. Butterfield (Ed.): UTP 2008, LNCS 5713, pp. 142–160, 2010.

also covers the algebraic laws we have proved about the operator together with the healthiness conditions the operator preserves. Section 4 presents the UTP semantics for Handel-C and motivates the changes we introduced in UTP in order to be able to capture Handel-C's timing model and restrictions. This section also includes a set of algebraic laws we have proved from the semantics together with examples of the semantics in action. In section 5 we propose an abstraction of our parallel-by-merge operator suitable for more general synchronous environments. Finally, section 6 presents the related research and section 7 the conclusions and future extensions of this work.

2 Handel-C in More Detail

In order to provide semantics for the language, a simplified subset that captures the major constructs in the Handel-C language is being used. Most constructs in the language can be built by combining constructs in this subset, with exception of the prioritised choice construct and function calls. Our subset of Handel-C constructs is presented in figure 1.

$$
\begin{aligned}
\langle \text{program} \rangle ::= \;& \textbf{main } \{ \langle \text{statements} \rangle \} \\
\langle \text{statements} \rangle ::= \;& \langle \text{statement} \rangle \fatsemi \langle \text{statements} \rangle \mid \langle \text{statements} \rangle \parallel_{\mathsf{HC}} \langle \text{statements} \rangle \mid \langle \text{statement} \rangle \\
\langle \text{statement} \rangle ::= \;& \textbf{if } \langle \text{boolean expression} \rangle \textbf{ then } \langle \text{statements} \rangle \textbf{ else } \langle \text{statements} \rangle \\
& \mid \textbf{while } \langle \text{boolean expression} \rangle \textbf{ do } \langle \text{statements} \rangle \\
& \mid \langle \text{variable list} \rangle :=_{\mathsf{HC}} \langle \text{expression list} \rangle \mid \delta^{\mathsf{HC}} \mid \Pi_{\mathsf{HC}} \\
& \mid \langle \text{channel name} \rangle ? \langle \text{variable name} \rangle \mid \langle \text{channel name} \rangle ! \langle \text{expression} \rangle
\end{aligned}
$$

Fig. 1. Restricted syntax for Handel-C programs

As described in the language documentation [10], programs are comprised of at least one **main** function and, possibly, some additional functions. Multiple main functions (within the same file) produces the parallel execution of their bodies under the same clock domain. It is possible to produce the same effect in our reduced subset by means of the parallel operator.

All C-based constructs in Handel-C behave as defined in ANSI-C [14] but with some additional restrictions regarding the clock-based, synchronous nature of the language. In this sense, the evaluation of expressions is performed by means of combinatorial circuitry and it is completed within the clock cycle in which it is initiated (expressions are considered to be evaluated "for free" [10] due to this semantic interpretation).

This way of evaluating conditions affects the timing of all the constructs in the language. In the case of selection, the branch selected for execution (depending on the condition) will start execution within the same clock cycle in which the whole construct is initiated. The **while** construct behaves in a similar way when

its condition is true (i.e., it starts its body in the same clock cycle in which its condition is evaluated) and, because of the same reason, terminates within the same clock cycle in which its condition becomes false. Assignment, on the other hand, happens at the end of the clock cycle. This definition of the assignment construct allows swapping of variables without the need of temporary variables.

From the remaining non-C constructs, parallel composition of statements executes in a *real* parallel fashion as it refers to independent pieces of hardware running in the same clock domain. Delay leaves the state unchanged but takes a whole clock cycle to finish and $\mathit{\Pi}_{\mathsf{HC}}$ leaves the state unchanged and finishes immediately (in fact, no hardware is generated for it).

Finally, input and output have the standard blocking semantics: if the two parts are ready to communicate, the value outputted at one end is assigned to the variable associated with the input side. Both sides of the communication take one full clock cycle to successfully communicate. A process trying to communicate over a channel without the other side being ready will block (delay) for a single clock cycle and try again.

3 Extended Parallel by Merge

As mentioned before, we intend to define the semantics of Handel-C constructs in terms of synchronous UTP designs. The first problem we faced in this context is the fact that the parallel-by-merge approach used in UTP (see [12] chapter 7), is only applicable to parallel processes that take the same amount of time to terminate. This is a very strong restriction, especially in the context of Handel-C where parallel composition is unrestricted in this sense.

The rest of this section outlines the definitions and algebraic laws that hold for a new parallel-by-merge operator that can handle processes that do not necessarily take the same amount of time to finish.

3.1 The Merge Predicate

The first step towards the definition of our operator is to instantiate the merge predicate M that will join the results of two *single-step* parallel process. By *single-step* we mean a process that performs all its actions in a single time unit (e.g., a single clock cycle in the context of synchronous hardware). The intuition behind our definition is that M will update the shared variables to the value of the process that has modified it or will leave it unchanged if none of the parallel processes modified it. More precisely, we define M as follows:

$$
\begin{aligned}
&M(ok, m, 0.m, 1.m, m', ok') = \\
&\quad ok \Rightarrow ok' \wedge \\
&\qquad\qquad ((m' = m) \lhd m = 1.m \rhd (m' = 1.m)) \\
&\qquad\qquad \lhd m = 0.m \rhd \\
&\qquad\qquad ((m' = 0.m) \lhd m = 1.m \rhd (m' = 1.m \sqcap m' = 0.m))
\end{aligned}
$$

Handel-C semantics allows at most one write to any shared variable per clock cycle. In this context, our definition for M behaves as expected as it will be applied at the end of each clock cycle where we know that, at most, one of the processes has changed the value in its local copy of m.

We "totalised" the definition of M in order to cover the (impossible) case where the two parallel processes modify m, as we needed M to be symmetric in order to prove our operators associative later in this section. In this context, the result of multiple assignment to the same variable during the same clock cycle is the internal choice of updating the store with either of the values being assigned. This unexpected non-determinism can be explained at the hardware level by the unpredictable value that will be stored in a register when it is fed with more than one value at the same time.

Following Hoare and He [12], we define the single-step parallel composition operator $\|_M$ as $P \|_M Q =_{df} ((P; U0) \parallel (Q; U1)); M$. Here $U0$ and $U1$ are separating simulations that will generate the local copies of the shared state.

We are interested in proving some standard algebraic laws about our merge predicate:

L1 $P \|_M Q = Q \|_M P$ — $\|_M$-comm

L2 $P \|_M (Q \|_M R) = (P \|_M Q) \|_M R$ — $\|_M$-assoc

L3 $(I\!I_X \|_M I\!I_Y) = I\!I_{X \cup Y}$ — $\|_M$-$I\!I$

L4 $\mathbf{true} \|_M P = \mathbf{true}$ — $\|_M$-**true**

L5 $(P \lhd b \rhd Q) \|_M R = ((P \|_M R) \lhd b \rhd (Q \|_M R))$ — $\|_M$-$\lhd\rhd$

L6 $(P \sqcap Q) \|_M R = (P \|_M R) \sqcap (Q \|_M R)$ — $\|_M$-$\sqcap$

L7 $(\bigsqcup S) \|_M R = \bigsqcup_n (S_n \|_M R)$ — $\|_M$-$\bigsqcup$

for any descending chain $S = \{S_n \mid n \in \mathcal{N}\}$

L8 $(x := e; P) \|_M Q = (x := e); (P \|_M Q)$

provided that $x := e$ does not mention m

Instead of proving all these laws for our operator, we can take advantage of an already proved result from UTP that guarantees properties **L1** - **L7** above to hold iff M is a *valid merge*. We proved M to be valid by showing it satisfies:

V1 $(0.m, 1.m := 1.m, 0.m); M = M$ — M is symmetric

V2 $(0.m, 1.m, 2.m := 1.m, 2.m, 0.m); M3 = M3$ — M is associative

where $M3 = \exists x, t \bullet M(ok, m, 0.m, 1.m, x, t) \wedge M(t, m, x, 2.m, m', ok')$

V3 $(\mathbf{var}\ 0.m, 1.m := m, m; M) = I\!I$

We were also able to prove two expected properties from our definition of M: if one of the branches remains idle (i.e., does not modify the shared variable), then the shared variable will be updated according to the other branch (M-unit); and if the two processes modify the variable in the same way then the shared variable will be updated to that value (M-idemp). More formally stated:

$$(0.m = v); M(v, 0.m, 1.m, m') = (m' = 1.m) \qquad M\text{-unit}$$
$$(0.m, 1.m := v, v); M(m, 0.m, 1.m, m') = m' = v \qquad M\text{-idemp}$$

We also proved that M preserves healthiness conditions **H1** to **H4**, by proving (again, by a result from UTP) that $\|_M$ is **H1** to **H4**.

3.2 The Final Merge Predicate

In the context of UTP synchronous parallel process, time is captured by a global counter c and each parallel process has its own copy of the store. Each process has access to the global state by means of a pair of vectors indexed by time: *in* and *out* that can be interpreted as the values of the global variables at the beginning and end of the clock cycle respectively. Processes behave independently from each other, signalling the end of their actions at each clock cycle by performing a **sync** action. The merge predicate is then used to calculate the global value of the store for that clock cycle and to propagate the value to the processes through the *in* observation.

So far we have defined how to merge the result of a single step in the computations of parallel processes. The next step is to define a final merge predicate $\hat{M}$ (i.e., a predicate that will take the result of two arbitrary processes and will compute the final outcome of their parallel execution) that is capable to handle different-length parallel processes. The main issue when trying to define such an operator is how to state that if one of the processes takes less clock cycles to finish than the other one, then it should do nothing but wait. More important, how to produce this "missing behaviour" while preserving properties **L1** to **L8** from the previous section.

The above idea could be expressed in the UTP by forcing the shorter process to perform the missing **sync** actions it is not doing (i.e., advancing the local counter c and updating out_c and the shared resource m appropriately). There are several alternative ways to achieve this effect but, even though all of them are operationally correct, they fail when trying to prove some of the desired properties for the parallel merge operator. The main reason for this being the *behavioural padding* we are using to generate the missing behaviour for the shorter process not being associative and not distributing over $\hat{M}$.

The evidence above suggests that we need a way of denoting the padding in a less explicit way. In fact, we need to find a way to establish the right values in the variables used to control the parallel execution for the shorter processes and to denote the fact that the local copy of the shared resource m is keeping its previous value while the clock counter is advancing.

To achieve this effect we first introduce a new variable f recording the clock cycle count in which the whole program finishes. In this way, we keep the local copies of the counter c to the actual termination times for each branch while we are able to express actions for the whole duration of the program. We also introduce the $0.m.in$ inspired after the *in* vector in the UTP formulation. We initialise $0.m.in$ to behave like the standard feedback loop in a flip-flop (at each clock cycle, it holds the same value it had during the previous clock cycle). In

this way, we are avoiding an explicit mention of how the variable is preserving its previous value during the cycles in which the process is inactive.

We also need to account for the communication primitives and how our parallel operator handles them. We define the input and output commands to rely upon a set of special variables that are not included in the list of program variables. The special set of variables associated to a given channel ch include $ch?, ch!$ and ch standing, respectively, for the requests for inputting, outputting and the value to be transmitted over ch. We also assume that $ch?, ch!$ (the requests for communication) will remain in the logical value $false$ unless they are used. This assumption is consistent with the hardware implementation of communications, where the requests are wires that remain in a "low state" unless they are explicitly fed with current when the request is done.

Finally, we introduce the fixed, but arbitrary value `ARB`. As with the $false$ logical value for the communication requests, this value will be the default value for all channels when they are not being used. This is a refinement of what happens at the hardware level where the value of this kind of buses is left unconstraint when they are not being used.

We now extend the standard definition of the separating simulation $U0$ (and similarly $U1$) to include $m.in$ together with the channel request wires:

$$\begin{aligned} U0 =_{df}\ & \mathbf{var}\ 0.m.in, 0.c, 0.ch?, 0.ch!, 0.ch := m.in, c, ch?, ch!, ch; \\ & \mathbf{end}\ m.in, c, ch?, ch!, ch \end{aligned}$$

With these definitions in place we now define the final merge predicate:

$$\begin{aligned} \hat{M} =_{df}\ & (c := max(0.c, 1.c) \parallel \\ & \{M(m_{i-1}, 0.m.in_i, 1.m.in_i, m.in'_i) | c \leq i \leq f\} \parallel \\ & \{M(false, 0.ch_i, 1.ch_i, ch'_i) | c \leq i \leq f\} \parallel \\ & \{M(false, 0.ch?_i, 1.ch?_i, ch?'_i) | c \leq i \leq f\} \parallel \\ & \{M(\mathtt{ARB}, 0.ch!_i, 1.ch!_i, ch!'_i) | c \leq i \leq f\}); \\ & \mathbf{end}\ 0.c, 1.c, 0.ch?, 1.ch?, 0.ch!, 1.ch!, 0.ch?, 1.ch?, 0.ch, 1.ch; \\ & \{I\!I_{m.in_i, m_i, \mathsf{com}(ch)_i} | i < c\} \end{aligned}$$

There are several aspects of this definition that are worth noticing:

- Even though the introduction of f in our model allows the local counters ($0.c$ and $1.c$) to be different, we know one of them (the bigger one) matches the actual cycle count for the parallel execution of both processes. We choose the longest execution time to update the global cycle counter.
- All our updates to the shared store (generically referred to as m) are based on the value of the resource being updated at the previous clock cycle. Updates to the communication requests and bus values, on the other hand, are based on their respective default values.

The rationale behind the behaviour for the store is that we are modelling sequential hardware, where the next value of registers (variables) will depend on the state of the machine in the previous clock cycle. A similar explanation holds for communication requests and buses, where the default values are used to detect if any of the processes has changed them. In both cases, M acts as a multiplexer that selects between preserving the old/default value or routing the updated value.

– Regarding Hoare and He's initial formulation, we removed the presence of the *out* sequence in our model. In UTP, the *out* vector is used to record the intermediate results produced by the process over the shared variables and avoid variable capture. The fact that our variables are themselves sequences allows us to remove *out* and reuse the local copy of m for this purpose.

Finally, we define the parallel-by-merge operator as:

$$P \parallel_{\hat{M}} Q =_{df} ((P; U0) \parallel (Q; U1)); \hat{M}$$

3.3 Algebraic Laws and Healthiness Conditions

In this section we provide the set of laws we proved about our parallel-by-merge operator. Most of the laws are similar to the ones presented earlier in the paper for the $\parallel_M$ operator, but we recast them here for clarity.

L1 $P \parallel_{\hat{M}} Q = Q \parallel_{\hat{M}} P$ $\quad \parallel_{\hat{M}}$-comm

L2 $P \parallel_{\hat{M}} (Q \parallel_{\hat{M}} R) = (P \parallel_{\hat{M}} Q) \parallel_{\hat{M}} R$ $\quad \parallel_{\hat{M}}$-assoc
provided that P, Q and R are **H4**

L3 $(I\!I \parallel_{\hat{M}} P) = P$ $\quad \parallel_{\hat{M}}$-$I\!I$

L4 $\mathbf{true} \parallel_{\hat{M}} P = \mathbf{true}$ $\quad \parallel_{\hat{M}}$-**true**

L5 $(P \lhd b \rhd Q) \parallel_{\hat{M}} R = ((P \parallel_{\hat{M}} R) \lhd b \rhd (Q \parallel_{\hat{M}} R))$ $\quad \parallel_{\hat{M}}$-$\lhd\rhd$

L6 $(P \sqcap Q) \parallel_{\hat{M}} R = (P \parallel_{\hat{M}} R) \sqcap (Q \parallel_{\hat{M}} R)$ $\quad \parallel_{\hat{M}}$-$\sqcap$

L7 $(\bigsqcup S) \parallel_{\hat{M}} R = \bigsqcup_n (S_n \parallel_{\hat{M}} R)$ $\quad \parallel_{\hat{M}}$-$\bigsqcup$
for any descending chain $S = \{S_n \mid n \in \mathcal{N}\}$

L8 $x := e; (P \parallel_{\hat{M}} Q) = (x := e; P) \parallel_{\hat{M}} Q$ $\quad :=$-$\parallel_{\hat{M}}$

L9 $(P \parallel_{M_{\{m,ch\}}} Q); \mathbf{tick}; (R \parallel_{\hat{M}} S) = (P; \mathbf{tick}; R) \parallel_{\hat{M}} (Q; \mathbf{tick}; S)$ $\quad \parallel_M$-$\parallel_{\hat{M}}$
provided that P and Q do not perform any **tick** event
where $\parallel_{M_{\{m,ch\}}} =_{df} M(m, 0.m.in, 1.m.in, m.in') \parallel$
$M(\mathsf{com}(ch), 0.\mathsf{com}(ch), 1.\mathsf{com}(ch), \mathsf{com}(ch)')$

We start by proving two of the three validity properties of the $\|_{\hat{M}}$ operator by showing:

$$(0.st, 1.st := 1.st, 0.st); \hat{M} = \hat{M} \qquad \hat{M}\text{-symmetric}$$
$$\text{where } st =_{df} m_{0..f}, c, ch?, ch!, ch$$
$$(0.st, 1.st, 2.st := 1.st, 2.st, 0.st); \hat{M}3 = \hat{M}3 \qquad \hat{M}\text{-associative}$$
$$\text{where } \hat{M}3 =_{df} \exists x.st \bullet \hat{M}(st, 0.st, 1.st, x.st) \wedge \hat{M}(st, x.st, 2.st, st')$$

With these results, we easily proved ($\|_{\hat{M}}$-comm) and ($\|_{\hat{M}}$-assoc).

The key result regarding our parallel-by-merge operator's capability to handle processes of different length lies in property 3.3**L3**, as the spreadsheet principle (3.3**L9**) will eventually reduce the shorter process to $I\!I_{\mathsf{HC}}$.

Proof of 3.3**L3**: For the proof, consider:

$$\begin{aligned} P =_{df}\ & c, m.in, ch, ch?, ch! := \\ & c + t, P.m.in \frown \langle m.in_{c+t+1}, ..., m.in_f \rangle, P.ch_{c..c+t} \frown \langle \texttt{ARB}, ..., \texttt{ARB} \rangle, \\ & P.ch?_{c..c+t} \frown \langle false, ..., false \rangle, P.ch!_{c..c+t} \frown \langle false, ..., false \rangle)) \end{aligned}$$

Then we have:

$$\begin{aligned} & I\!I_{\{m.in, \mathsf{com}(ch), c\}} \|_{\hat{M}} P \\ = {} & [\|_{\hat{M}}\text{'s definition, } U0 \text{ and } U1 \text{ definition and predicate calculus}] \\ & ((0.c, 0.m.in_{c..f}, 0.ch_{c..f}, 0.ch?_{c..f}, 0.ch!_{c..f} := \\ & \quad c, \langle m.in_{c-1}, ..., m.in_{f-1} \rangle, \langle \texttt{ARB}, ..., \texttt{ARB} \rangle, \langle false, ..., false \rangle, \langle false, ..., false \rangle) \| \\ & (1.c, 1.m.in_{c..f}, 1.ch_{c..f}, 1.ch?_{c..f}, 1.ch!_{c..f} := \\ & \quad c + t, P.m.in \frown \langle m.in_{c+t+1}, ..., m.in_f \rangle, P.ch_{c..c+t} \frown \langle \texttt{ARB}, ..., \texttt{ARB} \rangle, \\ & \quad P.ch?_{c..c+t} \frown \langle false, ..., false \rangle, P.ch!_{c..c+t} \frown \langle false, ..., false \rangle)); \hat{M} \\ = {} & [\hat{M}\text{-unit}] \\ & (c, m.in_{c..f}, ch_{c..f}, ch?_{c..f}, ch!_{c..f} := \\ & \quad c + t, P.m.in \frown \langle m.in_{c+t+1}, ..., m.in_f \rangle, P.ch_{c..c+t} \frown \langle \texttt{ARB}, ..., \texttt{ARB} \rangle, \\ & \quad P.ch?_{c..c+t} \frown \langle false, ..., false \rangle, P.ch!_{c..c+t} \frown \langle false, ..., false \rangle) \\ = {} & [\text{Definition of } P] \\ & P \end{aligned}$$

Laws **L4**-**L8** can be easily proved from the fact that $\hat{M}$ is defined in terms of $\|$ and these properties hold for the disjoint-alphabet parallel operator. **L9** can be proved following the proof sketched in [12].

Regarding the healthiness conditions and their preservation through the $\|_{\hat{M}}$ operator, we begin by observing that even though we have not explicitly stated that $\hat{M}$ is a design, this can be easily shown if we first note that all the parallel elements in its definition are designs:

$$
\begin{aligned}
& \hat{M} \\
= & \ [\hat{M}\text{'s definition}] \\
& (\mathbf{true} \vdash c := max(0.c, 1.c)) \parallel \\
& \{(\mathbf{true} \vdash M(m_{i-1}, 0.m_i, 1.m_i, m.in'_i)) | c \leq i \leq f\} \parallel \\
& \{(\mathbf{true} \vdash M(ch_{i-1}, 0.ch_i, 1.ch_i, ch_i)) | 0 < i < f\} \parallel \\
& \{(\mathbf{true} \vdash M(ch?_{i-1}, 0.ch?_i, 1.ch?_i, ch?_i)) | 0 < i < f\} \parallel \\
& \{(\mathbf{true} \vdash M(ch!_{i-1}, 0.ch!_i, 1.ch!_i, ch!_i)) | 0 < i < f\} \parallel \\
& \mathbf{end}\ 0.c, 1.c, 0.ch?, 1.ch?, 0.ch!, 1.ch!, 0.ch?, 1.ch?, 0.ch, 1.ch \\
= & \ [\parallel \text{ composition of designs, } \mathbb{M} \text{ for } \hat{M}\text{'s body}] \\
& (\mathbf{true} \vdash \mathbb{M})
\end{aligned}
$$

By being a design, $\hat{M}$ satisfies **H1** and **H2**. $\hat{M}$'s simple assumption (**true**) makes it trivial to prove that it also satisfies **H3**. Finally, **H4** follows naturally from M being **H4**. We use these results together with the fact that our definition of $\|_{\hat{M}}$ follows the UTP parallel-by-merge template to ensure that $\|_{\hat{M}}$ is implementable and preserves the four healthiness conditions.

4 Handel-C Semantics

In this section we present the semantic expressions that give meaning to Handel-C constructs. The first problem we face when trying to produce a UTP-based semantics is the property of the assignment design that allows us to flatten a sequence of assignments to a single (possibly multiple) assignment (law 3.1.**L2** in UTP). For example, UTP algebraic laws for assignment and sequential composition allow us to reduce $(x := 1; x := x + 1)$ to $x := 2$. Even though the equivalent Handel-C program also finishes by storing the value 2 in x, it does so after two clock cycles and we are interested in preserving the information about $x = 1$ for a whole clock cycle before changing into its final value (this is fundamental when parallel composition is taken into account).

We address this problem by turning the variables in the program into sequences of values indexed by clock cycle. In this way, it does not hold that $(x \overset{:=}{\mathsf{HC}} 1 \fatsemi x \overset{:=}{\mathsf{HC}} x + 1) = x \overset{:=}{\mathsf{HC}} 2$. For this idea to work we need to introduce a way to keep track of the current clock cycle and how each construct behaves with respect to it. With this in mind, we extend the scope of the observational variable c from just parallel regions to the full scope of the program. We also add a single action capturing the notion of the clock *ticking*:

$$\mathbf{tick} =_{df} c := c + 1$$

We also take advantage of the *in* vector as defined in section 3.2. In the context of the semantics, it plays a key role because it allows us to unify the sequential and parallel worlds and preserve the compositionality of the approach (we will address this issue in more detail at the end of this section).

To keep the presentation compact, we introduce the notation $\mathsf{com}(ch)$ to stand for all the variables associated to channel ch ($ch?$,$ch!$,ch) and $\mathsf{com}_{\mathsf{idle}}$ to the set of values associated to a channel when it is idle (i.e., it's default values).

In these terms, the semantics of assignment, the one-clock-cycle delay and $I\!I_{\mathsf{HC}}$ can be stated as follows:

$$[\![x :=_{\mathsf{HC}} e]\!] =_{df} (x.in_c, v.in_c, \mathsf{com}(c) := [\![e]\!], v_{c-1}, \mathsf{com}_{\mathsf{idle}}); \mathbf{tick}$$
$$[\![\delta^{\mathsf{HC}}]\!] =_{df} (v.in_c, \mathsf{com}(c) := v_{c-1}, \mathsf{com}_{\mathsf{idle}}); \mathbf{tick}$$
$$[\![I\!I_{\mathsf{HC}}]\!] =_{df} I\!I_D$$

Here v stands for the remaining variables in the state space of the program. Thus, v_c refers to the values of the variables mentioned in v at clock cycle c and $v.in_{c+1}$ to the value for the in vectors associated to each of them at clock cycle $c+1$. On the other hand, the semantics of an expression e are defined in the usual way with the exception that variable accesses (i.e., reads) are indexed by the clock cycle in which they happen.

The basic sequential constructs of the language can be given semantics by their UTP counterparts:

$$[\![P \fatsemi Q]\!] =_{df} [\![P]\!]; [\![Q]\!]$$
$$[\![\mathbf{if}\ c\ \mathbf{then}\ P\ \mathbf{else}\ Q]\!] =_{df} [\![P]\!] \lhd [\![c]\!] \rhd [\![Q]\!]$$
$$[\![\mathbf{while}\ c\ \mathbf{do}\ P]\!] =_{df} \mu X \bullet ([\![P]\!]; X) \lhd [\![c]\!] \rhd I\!I$$

We use the communication requests introduced in section 3.2 and include three new signals $\overleftarrow{ch}, \overrightarrow{ch}$ and $\overleftrightarrow{ch}$ standing for the granted request for input and output over ch together with the actual value transmitted over the bus ch. In this context, the semantics of the input/output primitives can be stated as follows:

$$\begin{aligned}
&[\![ch?m]\!] =_{df} \\
&\quad \mu X \bullet ch?_c := true; \\
&\qquad ((m.in_c, v.in_c, ch!_c, ch_c := \overleftrightarrow{ch}'_c, v_{c-1}, false, \mathtt{ARB}; \mathbf{tick}) \\
&\qquad \lhd \overrightarrow{ch}'_c = true \rhd \\
&\qquad (v.in_c, ch!_c, ch_c := v_{c-1}, false, \mathtt{ARB}; \mathbf{tick}; X)) \\
&[\![ch!x]\!] =_{df} \\
&\quad \mu X \bullet ch!_c, ch_c := true, [\![x]\!]; \\
&\qquad ((v.in_c, ch?_c := v_{c-1}, false; \mathbf{tick}) \\
&\qquad \lhd \overleftarrow{ch}'_c = true \rhd \\
&\qquad (v.in_c, ch?_c := v_{c-1}, false; \mathbf{tick}; X))
\end{aligned}$$

It is worth noticing that none of the *granted-request* variables are modified by the communicating processes (i.e., they do not appear in the output alphabet

of the processes). In this way, the same variable can be mentioned in multiple parallel without risking to interfere with each other.

The semantics for parallel composition is defined in terms of the $\|_{\hat{M}}$ operator:

$$[\![P \parallel_{\mathsf{HC}} Q]\!] =_{df} [\![P]\!] \|_{\hat{M}} [\![Q]\!]$$

We have addressed $\|_{\hat{M}}$ in full detail in section 3.2. For the present discussion it is relevant to highlight that it produces local copies of the state and the channels (thanks to the separating simulations) that each process will access and modify. The merge predicate $\hat{M}$ is then be used to merge these copies back into the original shared variables.

Finally, we use the top-level **main** function to introduce the clock cycle count c together with the traces for the store, their associated in variables and the channel request/granted signals. We also initialise the shared variables (with their corresponding in vectors) to behave like a flip-flop. We apply a similar technique to establish the default value for channel requests and to set the default value transmitted over the channels to `ARB`. In this way, we satisfy the assumptions about default values we made when defining $\|_{\hat{M}}$ in section 3.2.

$$
\begin{array}{l}
[\![\textbf{main } \{P\}]\!] =_{df} \\
\quad \textbf{var } c, m, m.in, f, ch?, ch!, ch, \overleftarrow{ch}, \overrightarrow{ch}, \overleftrightarrow{ch}; ch?, ch!, ch; \\
\qquad c, m_{0..f}, m.in_{0..f} := 1, \lambda c \bullet \texttt{ARB} \lhd c = 0 \rhd m_{c-1}, \lambda c \bullet \texttt{ARB} \lhd c = 0 \rhd m_{c-1}; \\
\qquad ch?_{0..f}, ch!_{0..f}, ch_{0..f} := \lambda c \bullet false; \lambda c \bullet false; \lambda c \bullet \texttt{ARB}; \\
\qquad [\![P]\!] \wedge (m = m.in') \wedge (f = c') \wedge (\overrightarrow{ch}' = ch!') \wedge (\overleftarrow{ch}' = ch?') \wedge (\overleftrightarrow{ch}' = ch'); \\
\quad \textbf{end } m.in, f, ch?, ch!, ch, \overleftarrow{ch}, \overrightarrow{ch}, \overleftrightarrow{ch}
\end{array}
$$

It is worth noting the mapping we are producing between m and $m.in'$. In this way, the register storing m is copying what is fed to it through the in channel at every clock cycle. The simple relation this equation establishes is the key for the compositionality of the approach. In the context of sequential fragments, each sub-process will modify in according to its needs and this will be reflected in m. In the context of parallel processes, the in variable will be replicated (i.e., locally copied), generating multiple inputs to the same register. The $\hat{M}$ operator will appropriately merge (select) the right one and transfer the final value to the global in, ensuring homogeneous operation and compositionality.

We also constrain the value of f to the final value of the clock counter, making it consistent with our requirements in section 3. Regarding granted/request signals, they are used to avoid variable-capture when producing the local copies of the state within the parallel operator. The restrictions imposed here to keep them equal to the communication requests at all times, allows the feedback of the merged result (captured in the primed version of the requests) to the recursive equations used in the communication.

4.1 Properties about the Semantics

So far we have introduced a way to express the semantics for Handel-C in the theory of designs in UTP. At this point we are interested in using the semantics to find out which properties hold true for Handel-C syntactic constructs. We devote the rest of this section to describe the results we have proved so far towards this goal.

L1 $P \fatsemi (Q \fatsemi S) = (P \fatsemi Q) \fatsemi S$ $\fatsemi$-assoc

L2 $P \parallel_{\mathsf{HC}} Q = Q \parallel_{\mathsf{HC}} P$ $\parallel_{\mathsf{HC}}$-comm

L3 $(P \parallel_{\mathsf{HC}} Q) \parallel_{\mathsf{HC}} R = P \parallel_{\mathsf{HC}} (Q \parallel_{\mathsf{HC}} R)$ $\parallel_{\mathsf{HC}}$-assoc

L4 $P \fatsemi I\!I_{\mathsf{HC}} = P = I\!I_{\mathsf{HC}} \parallel_{\mathsf{HC}} P$ $\fatsemi$-skip

L5 $I\!I_{\mathsf{HC}} \parallel_{\mathsf{HC}} P = P$ $\parallel_{\mathsf{HC}}$-$I\!I_{\mathsf{HC}}$

L6 $x :\!=_{\mathsf{HC}} e \fatsemi (P \parallel_{\mathsf{HC}} Q) = (x :\!=_{\mathsf{HC}} e \fatsemi P) \parallel_{\mathsf{HC}} (x :\!=_{\mathsf{HC}} e \fatsemi Q)$ $\parallel_{\mathsf{HC}}$-$:\!=_{\mathsf{HC}}$

L7 $x :\!=_{\mathsf{HC}} e \fatsemi (P \parallel_{\mathsf{HC}} Q) = (x :\!=_{\mathsf{HC}} e \fatsemi P) \parallel_{\mathsf{HC}} (\delta^{\mathsf{HC}} \fatsemi Q)$ $\parallel_{\mathsf{HC}}$-$:\!=_{\mathsf{HC}}$-δ^{HC}

L8 $x, y :\!=_{\mathsf{HC}} e_1, e_2 \fatsemi (P \parallel_{\mathsf{HC}} Q) = (x :\!=_{\mathsf{HC}} e_1 \fatsemi P) \parallel_{\mathsf{HC}} (y :\!=_{\mathsf{HC}} e_2 \fatsemi Q)$ $\parallel_{\mathsf{HC}}$-multiple-$:\!=_{\mathsf{HC}}$

L9 $(ch?x \fatsemi P) \parallel_{\mathsf{HC}} (ch!e \fatsemi Q) =$
$(x, ch?, ch!, ch :\!=_{\mathsf{HC}} e, true, true, e) \fatsemi (P \parallel_{\mathsf{HC}} Q)$?!-$:\!=_{\mathsf{HC}}$
Provided that $(ch?', ch!', ch' = \overleftarrow{ch'}, \overrightarrow{ch'}, \overleftrightarrow{ch'})$

L10 $(ch?x \fatsemi P) \parallel_{\mathsf{HC}} (ch!e \fatsemi Q) \parallel_{\mathsf{HC}} (ch?y \fatsemi R) =$
$(x, y, ch?, ch!, ch :\!=_{\mathsf{HC}} e, e, true, true, e) \fatsemi (P \parallel_{\mathsf{HC}} Q \parallel_{\mathsf{HC}} R)$?!-multiple-readers
Provided that $(ch?', ch!', ch' = \overleftarrow{ch'}, \overrightarrow{ch'}, \overleftrightarrow{ch'})$

L11 $(ch?x \fatsemi P) \parallel_{\mathsf{HC}} Q =$
$((ch?, ch!, ch :\!=_{\mathsf{HC}} true, false, \mathtt{ARB}) \fatsemi ch?x \fatsemi P) \parallel_{\mathsf{HC}} Q$?-copy-rule
Provided that there is no process writing into ch during the first clock cycle in the execution of the parallel region

L12 $(ch!e \fatsemi P) \parallel_{\mathsf{HC}} Q =$
$((ch?, ch!, ch :\!=_{\mathsf{HC}} false, true, e) \fatsemi ch?x \fatsemi P) \parallel_{\mathsf{HC}} Q$!-copy-rule
Provided that there is no process reading from ch during the first clock cycle in the execution of the parallel region

The proofs for **L1** to **L5** are straightforward from our definition of the semantics and the properties of the underlying sequential and parallel composition operators. In particular, **L4** holds because the semantics of all our constructs in the language can be expressed as designs ($I\!I$ is a left unit) that are also **H3** healthy ($I\!I_D$ is a right unit).

Proof of **L6** (the proofs of **L7** and **L8** follow the same proof outline).

$$
\begin{array}{ll}
 & (x \mathrel{:=_{\mathsf{HC}}} e \fatsemi P) \parallel_{\mathsf{HC}} (x \mathrel{:=_{\mathsf{HC}}} e \fatsemi Q) \\
= & \text{[Semantics of } :=_{\mathsf{HC}}\text{, } \fatsemi \text{ and } \parallel_{\mathsf{HC}}\text{]} \\
 & (x_c, v_c, \mathsf{com}(c) := e, v_{c-1}, \mathsf{com}_{\mathsf{idle}}; \mathbf{tick}; P) \parallel_{\hat{M}} \\
 & \quad (x_c, v_c, \mathsf{com}(c) := e, v_{c-1}, \mathsf{com}_{\mathsf{idle}}; \mathbf{tick}; Q) \\
= & [\parallel_M\text{-}\parallel_{\hat{M}}] \\
 & ((x_c, v_c, \mathsf{com}(c) := e, y_{c-1}, \mathsf{com}_{\mathsf{idle}}) \parallel_{M_{\{x,v,\mathsf{com}(ch)\}}} \\
 & \quad (x_c, v_c, \mathsf{com}(c) := e, v_{c-1}, \mathsf{com}_{\mathsf{idle}})); \mathbf{tick}; (P \parallel_{\hat{M}} Q) \\
= & [M\text{-idemp}] \\
 & (x_c, v_c, \mathsf{com}(c) := e, v_{c-1}, \mathsf{com}_{\mathsf{idle}}); \mathbf{tick}; (P \parallel_{\hat{M}} Q) \\
= & \text{[Semantics of } :=_{\mathsf{HC}}\text{, } \fatsemi \text{ and } \parallel_{\mathsf{HC}}\text{]} \\
 & x \mathrel{:=_{\mathsf{HC}}} e \fatsemi (P \parallel_{\mathsf{HC}} Q)
\end{array}
$$

4.2 The Semantics in Action

In this section we present two simple cases to illustrate the way the semantics work on an environment of shared variables. The first example shows a program that first initialises one of the shared variables to them modify them in an uneven-length parallel subprocess:

$$
\begin{array}{ll}
 & \mathbf{main}\ \{x \mathrel{:=_{\mathsf{HC}}} 8 \fatsemi ((x \mathrel{:=_{\mathsf{HC}}} x + 1) \parallel_{\mathsf{HC}} (y \mathrel{:=_{\mathsf{HC}}} 1 \fatsemi x \mathrel{:=_{\mathsf{HC}}} x + y + 1))\} \\
= & [\parallel_{\mathsf{HC}}\text{-multiple-} :=_{\mathsf{HC}}] \\
 & \mathbf{main}\ \{x \mathrel{:=_{\mathsf{HC}}} 8 \fatsemi (x, y \mathrel{:=_{\mathsf{HC}}} x + 1, 1) \fatsemi (\Pi_{\mathsf{HC}} \parallel_{\mathsf{HC}} x \mathrel{:=_{\mathsf{HC}}} x + y + 1)\} \\
= & [\parallel_{\mathsf{HC}}\text{-}\Pi_{\mathsf{HC}}] \\
 & \mathbf{main}\ \{x \mathrel{:=_{\mathsf{HC}}} 8 \fatsemi (x, y \mathrel{:=_{\mathsf{HC}}} x + 1, 1) \fatsemi x \mathrel{:=_{\mathsf{HC}}} x + y + 1\}
\end{array}
$$

As expected, the program can be flattened into a sequence of parallel assignments. We can apply the semantic expressions for the constructs in Handel-C to obtain the trace:

$$\mathbf{var}\ c, x, y := 3, \langle \mathtt{ARB}, 8, 9, 11\rangle, \langle \mathtt{ARB}, \mathtt{ARB}, 1, 1\rangle$$

Our next example addresses the case where one process is trying to communicate with another one that is not ready:

$$
\begin{array}{ll}
 & \mathbf{main}\ \{(ch?x) \parallel_{\mathsf{HC}} (y := 10 \fatsemi ch!y)\} \\
= & \text{[?-copy-rule]} \\
 & \mathbf{main}\ \{(ch?, ch!, ch \mathrel{:=_{\mathsf{HC}}} true, false, \mathtt{ARB} \fatsemi (ch?x)) \parallel_{\mathsf{HC}} (y := 10 \fatsemi ch!y)\} \\
= & [\parallel_{\mathsf{HC}}\text{-multiple-} :=_{\mathsf{HC}}] \\
 & \mathbf{main}\ \{(y, ch?, ch!, ch \mathrel{:=_{\mathsf{HC}}} 10, true, false, \mathtt{ARB}) \fatsemi (ch?x \parallel_{\mathsf{HC}} ch!y)\}
\end{array}
$$

$$
\begin{aligned}
&= [?!\text{-}\,{}^{:=}_{\mathsf{HC}}] \\
&\quad \textbf{main } \{(y, ch?, ch!, ch\,{}^{:=}_{\mathsf{HC}}10, true, false, \texttt{ARB}) \fatsemi \\
&\qquad\qquad (x, ch?, ch!, ch\,{}^{:=}_{\mathsf{HC}}y, true, true, y) \fatsemi (\Pi_{\mathsf{HC}} \,{}^{\parallel}_{\mathsf{HC}}\, \Pi_{\mathsf{HC}})\} \\
&= [{}^{\parallel}_{\mathsf{HC}}\text{-}\Pi_{\mathsf{HC}},\ \fatsemi\text{-skip}] \\
&\quad \textbf{main } \{(y, ch?, ch!, ch\,{}^{:=}_{\mathsf{HC}}10, true, false, \texttt{ARB}) \fatsemi (x, ch?, ch!, ch\,{}^{:=}_{\mathsf{HC}}y, \\
&\quad true, true, y)\}
\end{aligned}
$$

From the final equation above, it is easy to see that there was a failed attempt of communication during the first clock cycle, and that the communication was carried out during the following clock cycle. Expanding the semantics of the **main** function and assignment we can get the actual trace of the program:

$$
\begin{aligned}
\textbf{var } c, x, y, ch?, ch!, ch := 2, &\langle \texttt{ARB}, \texttt{ARB}, 10\rangle, \langle \texttt{ARB}, 10, 10\rangle, \\
&\langle false, true, true\rangle, \langle false, false, true\rangle, \langle \texttt{ARB}, \texttt{ARB}, 10\rangle
\end{aligned}
$$

5 Generalising the Parallel by Merge Operator

Up to this point we have presented an extension of the parallel-by-merge theory presented in [12] that is able to handle different-length parallel processes in the context of the semantic expressions we are generating for Handel-C.

In this section we explore the possibilities of extending this notion to a more general case in order to make our results available to a broader application domain.

For the remainder of this section, we return to the framework in which this theory was initially developed by assuming a context in which inter-process communication, as described in earlier sections of this paper, is not required[1]. We are also going to remove the need to use sequences to represent the store, as it was introduced because of a particular need of the semantics for Handel-C.

Recasting from the previous section, we need to establish the properties that the merge predicate M must satisfy. Apart from being a valid merge (to guarantee properties 3.1.L1, 3.1.L2 and 3.1.L4 to 3.1.L7) we also require M to satisfy M-unit and M-idemp. We need the former to ensure that Π is the unit for parallel composition inside shared regions and the later to prove that equality distributes over parallel composition with final merge.

We can interpret (M-unit) as defining the behaviour of the merge predicate when one of the parallel processes is idle. As the $\hat{M}$ operator is based on M, we can easily lift the property to $\hat{M}$ and prove:

$$
\begin{aligned}
&(0.st = st); \hat{M}(st, 0.st, 1.st, st') = (st' = 1.st) && \hat{M}\text{-unit} \\
&\quad \text{provided that } 1.c = j,\ f > c \text{ and } j > 0
\end{aligned}
$$

Based on M satisfying the properties above, we intend to produce a final-merge operator that satisfies the laws 3.3.L1 to 3.3.L8 in this paper. In this

[1] We also assume the reader is familiar with the contents in chapter 7: *Concurrency* of UTP.

sense, we still need to provide a *valid* $\hat{M}$ predicate and, hence, we still have the problem of handling the behavioural padding of the shorter processes in the parallel composition. We take advantage of the f variable introduced earlier to deal with this problem and define our more general formulation as:

$$
\begin{aligned}
U0(m) =_{df}\ & \textbf{var}\ 0.out, 0.c, 0.m; \\
& 0.c := c; \\
& \{0.out_i := in_{i-1} | 0 < i \leq f\}; \\
& 0.out = 0.out \oplus out; 0.out_{0.c} = m; && (1) \\
& 0.m = 0.out_f; && (2) \\
& \textbf{end}\ out, c, m
\end{aligned}
$$

Not surprisingly, we needed to re-introduce the *out* variable and we use the same trace-like approach we defined before to perform the behavioural padding. We also keep the same overriding behaviour we used before (line (1)) but we also include the final value of the local copy of m at the end of *out* (note that process 0 modifies the *out* sequence only within the index range $[0..(0.c - 1)]$). The reason for transferring the value of m to the *out* sequence is to cover the case where $0.c < f$ (the process finishes earlier than other processes in the parallel composition). In this context, the value of the local copy of m should be merged with the corresponding outcome of the other processes at clock cycle $0.c$, and these values are stored in the corresponding copies of *out* at this particular index (clock cycle).

Finally, line (2) sets the value of the local copy of m to the outcome of the current process at clock cycle f. In this way, we make $0.m$'s value independent of the actual execution time for process 0 (we will take advantage of this fact to define an associative $\hat{M}$ operator).

We are now ready to define the final-merge operator $\hat{M}$ as:

$$
\begin{aligned}
\hat{M} =_{df}\ & c' = max(0.c, 1.c) \parallel \\
& M(m \lhd f = c \rhd in_{f-1}, 0.m, 1.m, m') \parallel && (3) \\
& \{M(m \lhd i = c \rhd in_{i-1}), 0.out_i, 1.out_i, out'_i | c \leq i < f\} \parallel && (4) \\
& \{\mathbb{I}_{\{out_i\}} | i < c\}; \\
& \textbf{end}\ 0.c, 1.c, 0.out, 1.out
\end{aligned}
$$

Apart from the change in the way the clock is handled (already introduced in the previous section), the main point to be noted here is that we changed Hoare and He's initial formulation by replacing m with $m \lhd f = c \rhd in_{f-1}$ as the first argument for M. The reason for this change is the fact that the initial formulation by Hoare and He will ignore the intermediate changes to the shared store and will calculate m's final value based on its value before the parallel branches started executing (i.e., the value in m).

As mentioned earlier, we are interested in a clock-wise update of the shared variable. To achieve this goal, assignments consume a clock cycle (i.e., they

produce a **sync** event) so synchronisation (and, hence, *merging*) happens on every clock cycle.

Moreover, our definition of M calculates the next value of m based on the local copy of the store that changed during the previous clock cycle. Thus, the final value for m should be calculated from the last update (stored in the in sequence at the current clock cycle minus one $0.c-1$) rather than based on the value of m before the execution of the parallel processes (as several changes from that value may have happened to m since the parallel composition started and it would be impossible to find out which process made a modification during the last clock).

Finally, we need to define the way in which f is introduced (and calculated). As it only makes sense to mention f in the context of parallel processes sharing variables, we add it to the set of variables introduced in the **shared** declaration. In turn, we use the same "loop-back" approach used by Hoare and He to feedback the *out* values produced by the parallel composition into the in vector to update f and define:

$$\begin{aligned}(\mathbf{shared}\ m \fatsemi P \fatsemi \mathbf{end}\ m) =_{df}\ & \mathbf{var}\ c, in, out, f \fatsemi \\ & (c := 0) \fatsemi (P \wedge (in = out') \wedge (f = c')) \fatsemi \\ & \mathbf{end}\ c, in, out, f\end{aligned}$$

5.1 Validity, Algebraic Laws and Healthiness Conditions

Based on the properties we assumed for the merge predicate together with the associativity and commutativity of the *max* function, it is easy to show that our definition of $\hat{M}$ satisfies the symmetric and associative properties from the valid merge definition. Regarding the last valid property, the presence of f in the definition makes it impossible to be proved unless M satisfies (M-idemp).

Even though we can prove the third property in the *valid merge* definition, this result is not useful in the proof we intend to conduct. Instead, the proof relies on M satisfying (M-unit) as defined at the beginning of this section.

With the results above together with the laws for $\|$ we can prove that our general $\|_{\hat{M}}$ satisfies 3.3.L1 to 3.3.L7.

The proof of 3.3.**L8** relies in the following additional results we have proved about the UTP:

$$(v, m := x, v') = (v := x; m := v) \qquad \text{Primed assignment unfold}$$

$$\mathbf{var}\ v; v_{0..j} := \langle v_0, v_1, ..., v_{j-1}, v_j \rangle; P(v_{j-1}); \mathbf{end}\ v = \qquad \text{Partial end of scope}$$

$$\mathbf{var}\ v; v_{0..j-1} := \langle v_0, v_1, .., v_{j-1} \rangle \wedge P(v_{j-1}); \mathbf{end}\ v$$

Regarding the healthiness of our operator, we follow the same approach we used for the parallel-by-merge operator we defined for Handel-C. By a similar argument, our general final merge predicate can also be expressed as a design with trivial precondition **true**. In this way, we are sure it is **H1** to **H3** healthy. The proof of **H4** is based on the fact that $\hat{M}$ is a design and that M is also **H4**.

6 Related Work

Operational [7] and denotational [6,4,5] semantics have been proposed for Handel-C, providing interpretations for most constructs, ranging from simple assignments to prioritised choices (priAlts). Denotational semantics have also been proposed for the compilation into hardware [15] and used to formally verify some correctness properties of the generated hardware [16]. All these papers describe works based either in a branching-sequences semantic domain or a flattened version of the branching structure based on merge functions. In general, all these works were based on the notion of state-transformers, were each step in the semantics was expressing the effect of the construct over the state space of the program. The time model of Handel-C also directed all these works towards adopting a clock cycle as a unit and to split it into two disjoint sets of actions (i.e., combinatorial and sequential actions). The complexity of this kind of semantic domains made it quite difficult to use the semantics to validate/discover algebraic laws about Handel-C programs. In fact, only [8] used the semantics to prove some standard algebraic properties that also hold for Handel-C (e.g., [$I\!I$-; unit], [$\|$-assoc], *etc*).

In [9], initial steps towards the unification of most of these works in semantics are presented. The goal of this work is to provide a framework where a timed version of *Circus* [18] can be used as the specification language and several lower level languages (Handel-C among them) can be used to implement such a specification. The work is based on the reactive processes model provided in UTP. The rationale behind the selection of a reactive processes formalism lies in the need to cope with nondeterminism and refusals (present in the Circus language). The expressiveness of the acceptance-refusals model underlying the reactive theory in UTP is also likely to allow this framework to cover the recent trend in hardware design of interconnecting hardware working at different clock speeds (multiple clock domains). The price to be paid for this richness in expressivity is a more complicated theory, where it is necessary to deal with several intermediate observation points during each process' execution.

Our work is similar to [9] in the sense that it tends towards unifying the existing semantics for Handel-C and is oriented towards the algebraic rules satisfied by Handel-C programs. On the other hand, we have based our work on the theory UTP designs, preempting us from covering multiple clock domains but allowing a more compact and elegant representation of Handel-C programs aiming at single-clocked domains. We believe we will be able to profit from the elegance of our model when trying to prove algebraic laws about Handel-C operators/constructs.

UTP denotational semantics has also been been proposed for a subset of Verilog [13] that is similar to ours but includes guarded events (a non existing feature in Handel-C) and excludes recursion. The semantics are derived from an operational semantics model and they also include some algebraic reduction rules for parallel composition. The work is based in the reactive-processes theory of UTP (a subset of it, as they avoid healthiness condition **R2**). Our work is based on the simpler theory of designs and our focus is not in the derivation of

the semantics from existing operational ones but in finding a comprehensive set of deduction rules for Handel-C.

Regarding other HDL languages (such as VHDL or SystemC), their semantics are informally provided in terms of a simulator [1,2]. Most works on the semantics of these languages follow these simulation models [3,17], making them quite different in purpose in comparison to our work.

7 Conclusions and Future Work

We have presented semantics for a subset of Handel-C including parallelism and communication. We have done so by using UTP's theory of designs as the semantic domain. The main contribution of this work is a denotational semantics for Handel-C that is well suited for reasoning and finding properties about the constructs of the language. Our usage of the theory of designs to describe the semantics for a HDL is also novel, as all existing works in the field address the semantics from the more powerful, yet more complex, theory of reactive processes (or a subset of it).

In the process of capturing the semantics of Handel-C in UTP we found several points in which we needed to extend or modify some aspects of UTP. The most interesting of these extensions is a parallel-by-merge operator that can handle parallel processes of uneven length. We have provided such an operator for the context of the semantics and proved a significant set of algebraic laws and healthiness conditions about it.

We also abstracted the key features of our parallel-by-merge operator and provided a more general formulation that we expect to be useful in a larger application domain. We also summarised the additional constraints that has to be satisfied by the single-step merge predicate in order for the general parallel merge to satisfy additional rules.

Finally, we have been able to take advantage of existing algebraic laws from UTP together with the rules provided in this work to easily prove an interesting set of algebraic laws about Handel-C programs. Some of these laws have been used to derive the semantics of example programs involving fixed-points in a few steps.

As future work we intend to keep on exploring the set of algebraic laws we can prove about the semantics. We are also interested in completing our work on semantics for Handel-C by covering priorities and procedure calls.

References

1. Multivalue Logic System for VHDL Model Interoperability (Std_logic_1164). IEEE Standard 1164-1993 (1993)
2. Standard SystemC Language Reference Manual (LRM). IEEE Standard 1666-2005 (2005)
3. Breuer, P.T., Fernández, L.S., Kloos, C.D.: Proof theory and a validation condition generator for VHDL. In: Euro-VHDL '94, pp. 512–517 (1994)

4. Butterfield, A.: Denotational semantics for prialt-free Handel-C. Technical report, The University of Dublin, Trinity College (December 2001)
5. Butterfield, A., Woodcock, J.: Semantic domains for Handel-C. Electronic Notes in Theoretical Computer Science, vol. 74 (2002)
6. Butterfield, A., Woodcock, J.: Semantics of prialt in Handel-C. In: Concurrent Systems Engineering. IOS Press, Amsterdam (2002)
7. Butterfield, A., Woodcock, J.: Prialt in handel-c: an operational semantics. International Journal on Software Tools Technology Transfer 7(3), 248–267 (2005)
8. Butterfield, A., Woodcock, J.: A Hardware Compiler Semantics for Handel-C. In: MFCSIT 2004, Dublin, Ireland, August 2006. ENTCS, vol. 161, pp. 73–90 (2006)
9. Butterfield, A., Sherif, A., Woodcock, J.: Slotted-circus. In: IFM, pp. 75–97 (2007)
10. Celoxica Ltd. DK3: Handel-C Language Reference Manual (2002)
11. Hoare, C.A.R.: Communicating sequential processes. Commun. ACM 26(1), 100–106 (1983)
12. Hoare, C.A.R., Jifeng, H.: Unifying Theories of Programming. Prentice-Hall, Englewood Cliffs (1998)
13. Huibiao, Z., Bowen, J.P., Jifeng, H.: From operational semantics to denotational semantics for verilog. In: Margaria, T., Melham, T.F. (eds.) CHARME 2001. LNCS, vol. 2144, pp. 449–471. Springer, Heidelberg (2001)
14. Kernighan, B.W.: The C Programming Language. Prentice Hall Professional Technical Reference (1988)
15. Perna, J.I., Woodcock, J.: A denotational semantics for Handel-C hardware compilation. In: ICFEM, pp. 266–285 (2007)
16. Perna, J.I., Woodcock, J.: Wire-Wise Correctness for Handel-C Synthesis in HOL. In: ETAPS'08 - Seventh International Workshop on Designing Correct Circuits (DCC), March 2008, pp. 86–100 (2008)
17. Salem, A.: Formal semantics of synchronous systemc. In: DATE '03: Proceedings of the conference on Design, Automation and Test in Europe, Washington, DC, USA, p. 10376. IEEE Computer Society, Los Alamitos (2003)
18. Woodcock, J., Cavalcanti, A.: A concurrent language for refinement. In: Butterfield, A., Strong, G., Pahl, C. (eds.) IWFM, Workshops in Computing. BCS (2001)