Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 240 (2009) 201–219
www.elsevier.com/locate/entcs

Mechanised Wire-wise Verification of Handel-C Synthesis

Juan Ignacio Perna[1]

Computer Science Department
The University of York
York, United Kingdom

Jim Woodcock[2]

Computer Science Department
The University of York
York, United Kingdom

Abstract

The compilation of Handel-C programs into net-list descriptions of hardware components has been extensively used in commercial tools but never formally verified. In this paper, we first introduce a variation of the existing semantic model for Handel-C compilation that is amenable for mechanical proofs and detailed enough to analyse properties about the generated hardware. We use this model to prove the correctness of the wiring schema used to interconnect the components at the hardware level and propagate control signals among them. Finally, we present the most interesting aspects of the mechanisation of the model and the correctness proofs in the HOL theorem prover.

Keywords: Handel-C synthesis, mechanical verification, HOL

1 Introduction

Handel-C [6] is a Hardware Description Language (HDL) based on the syntax of the C language extended with constructs to deal with CSP-based [4] parallel behaviour and process communications.

Handel-C's compilation into hardware components (synthesis) was initially formulated to be used in a hardware/software co-design project [9] and later adopted by commercial tools [7]. The compilation is based on the fact that the control of the generated circuits is handled by a pair of handshake signals (*start* and *finish*). The semantics is based on the assumption that the *start* signal will be given to a

[1] Email: jiperna@cs.york.ac.uk
[2] Email: jim@cs.york.ac.uk

doi:10.1016/j.entcs.2009.05.053

circuit only if all the previous circuits have already finished. A formal model of the compilation and denotational semantics for the generated hardware has been proposed as a first step towards verifying the compilation scheme [10]. The semantics relies on the concept of state-transformers and uses branching sequences of atomic hardware actions as its semantic domain.

The aim of this paper is to give an accessible account of the work we have carried out on wire correctness for Handel-C [11]. By doing so, we deliberately omit all proofs and consequently many technical details. Given the fact that we focus on the correctness of the synthesis process, we also exclude any efficiency/optimisation analysis from this work. The rest of the paper is organised as follows: we first propose a reformulation of the semantic domain in [10] in terms of a slightly higher level semantic domain that allows us to capture parallelism at the sequence level. We then redefine the semantics in terms of the new domain and prove the existence of fix-point solutions for the recursive equations. Finally, we define the concept of wire-satisfiability in our model and prove the correctness of the wiring schema used in the compilation.

2 The semantic model

The existing denotational semantics for the synthesis process [10] is based on the notion of "branching sequences", where non-branching sequences denote the execution of actions in which no information from the persistent part of the state (i.e., memory locations) is required. Branching nodes, on the other hand, model a choice point (such as the condition in a `while` construct), where the state must be accessed in order to evaluate the condition. The actual *trace* of the program is obtained by keeping track of the updates over the environment and by pruning the branches that do not get executed. Even though this semantic model was successfully implemented and tested against existing semantics for Handel-C [1,2], our goal is to achieve a higher degree of confidence in it by means of proving its correctness.

When trying to prove correctness properties about the generated hardware from the semantics, we observed that the semantics fails to capture the hardware parallelism at the branching sequence level (parallel actions performed combinatorially in the hardware are "linearised" in the branching sequences). The main drawback of this feature of the semantics is that most of the properties we are interested in proving hold at the end of clock cycles. In this context, proving relatively simple properties (such as that parallel composition is commutative if only state updates are considered [3]) showed itself to be very complicated, given the need to establish the equivalence at synchronous points (i.e., clock cycle edges) that did not occur at the same depth in the different parallel branches.

To overcome this problem, we observed that Handel-C's synchronous time model allowed us to group the actions performed on each clock cycle into two disjoint sets: *combinatoric actions* (performed before the end of the clock cycle) and *sequential actions* (performed at the end of the clock cycle). The idea of grouping similar

[3] The wiring of $c_1 || c_2$ is different than the one for $c_2 || c_1$, but their effect over the state is the same.

actions together is not fully compatible with the tree-like structure used in branching sequences. In particular, branching conditions are combinatoric actions and we want to include them in the corresponding set of actions, rather than having them branching the sequences' structure. We address this problem by formulating branching sequences of the form $S \boxplus cond \rightarrow S_1 \mid S_2$ (where $\boxplus$ stands for the concatenation operation over Seq) by factorising them into two sequences: $S \boxplus (cond)^{\perp} : S_1$ and $S \boxplus (\neg cond)^{\perp} : S_2$. This observation allows us to turn the semantic domain into a set of finite-length linear sequences (this view of the selection construct is consistent with the one used in other formalisms such as [5]).

A similar problem is encountered when constructs are composed in parallel as a structured node was generated in the sequence, preempting the actions in the parallel branches to be collected together. We overcame this problem by introducing a merge operator that joins the pair of sequences being composed in parallel.

Our new semantic domain is the powerset of sequences of the type:

Definition 2.1

$$
\begin{array}{rll}
e \in Seq ::= & \texttt{Empty} & \\
\mid & \texttt{Cb}\ \mathscr{P}(Action) \rightarrow Seq & \text{combinatoric behaviour} \\
\mid & \texttt{Ck}\ \mathscr{P}(Action) \rightarrow Seq & \text{sequential behaviour}
\end{array}
$$

The *Action* type captures the notion of actions that the hardware can perform together with assertion-like extensions (in the sense of [3]) in order to allow the verifications we need over the hardware. More precisely, we define the *Action* type as:

Definition 2.2

$$
\begin{array}{rll}
e \in Action ::= & w & w \text{ is set to the } \textit{high} \text{ value} \\
\mid & w_1 \multimap w_2 & \text{the value from } w_2 \text{ is transferred to } w_1 \\
\mid & w_1 \wedge w_2 & \text{the logical } \textbf{and} \text{ of values in } w_1 \text{ and } w_2 \\
\mid & var \leftarrow val & \text{the store location } var \text{ gets the value } val \\
\mid & (cnd)^{\perp} & cnd \text{ must be satisfied in the current clock cycle}
\end{array}
$$

2.1 Domain operations

We use a trivial extension of the standard concatenation operator $\boxplus$ that is capable of handling our heterogeneous sequences. This definition allows consecutive nodes based on the same constructor to be put in sequence and this kind of behaviour is not suitable in our semantics (we would expect the actions in the two consecutive nodes of the same kind to be collected in a single set of actions). We solve this problem by ensuring that the sequences being concatenated using the $\boxplus$ operator avoid the problematic cases (see section 3 for further details).

For the parallel-merge operator $\uplus$, the major complication arises from trying to merge *out-of-phase* sequences (i.e. a pair of sequences in which the structure is not

node-wise equal). Even though it would be possible to implement a merge operator that accounts for this problem (using priorities for example), an operator defined in such terms will be very difficult to reason about. On the other hand, the definition of the function becomes completely symmetric if we can ensure that the sequences are in-phase:

Definition 2.3

$$S_1 \uplus [] = S_1$$
$$[] \uplus S_2 = S_2$$
$$(\texttt{Cb}\ a_1\ s_1) \uplus (\texttt{Cb}\ a_2\ s_2) = (\texttt{Cb}\ a_1 \cup a_2\ (s_1 \uplus s_2))$$
$$(\texttt{Ck}\ a_1\ s_1) \uplus (\texttt{Ck}\ a_2\ s_2) = (\texttt{Ck}\ a_1 \cup a_2\ (s_1 \uplus s_2))$$
$$\textbf{pre}\ inPhase(S_1, S_2)$$

2.2 Fix-points

As our semantic function is going to use recursive equations, it is necessary to assure the existence of an appropriate semantic domain with fix-point solutions to them. To achieve this goal we extend Seq with a bottom element $\perp$ and order our lifted semantic domain $Seq_\perp$ by the relation $\preceq$ below. We prove this ordering to be a partial order and that the constructors are monotonic with respect to it.

Definition 2.4

$$\perp \preceq s \wedge [] \preceq [] \wedge$$
$$(\texttt{Cb}\ a\ s_1) \preceq (\texttt{Cb}\ a\ s_2) \Leftrightarrow s_1 \preceq s_2 \wedge$$
$$(\texttt{Ck}\ a\ s_1) \preceq (\texttt{Ck}\ a\ s_2) \Leftrightarrow s_1 \preceq s_2$$

We have now to verify that the concatenation function is also monotonic with respect to $\preceq$. We first extend $\boxplus$ to treat $\perp$ as left and right zero. With this extended definition, we can easily prove $\boxplus$ to be monotonic on its first and second arguments (by structural induction on s_1, s_2 and s).

We extend the parallel-merge operator to treat $\perp$ as zero when it appears in any of its arguments. We also use $\perp$ as the result for the function when the arguments are out-of-phase, as we need to totalise the functions in order to be able to encode them in the HOL theorem prover. The proof of right monotonicity for $\uplus$ is done by case analysis on the result of $\uplus$'s application (after using the right kind of induction).

The proof of $\uplus$'s left monotonicity, on the other hand, cannot be performed by structural induction because the sequences cannot be handled as a pair (the induction principle must be applied to individual sequences in a sequential way). The solution is to make the proof by complete induction over the sum of the lengths of the sequences being merged. This way of proving the theorem is quite laborious, but allows us to instantiate the inductive hypothesis to the sequences of the right shape when needed. Following this approach combined with the case analysis mentioned above, we prove $\uplus$ left monotonic.

Having shown a suitable partial order over the semantic domain and proved that all the operators preserve that ordering, we can guarantee that fix-point solutions

to the recursive equations introduced in the next section exist in our model.

3 Compilation semantics

Using the sequence-based domain defined in the previous section, a denotational semantics can be given to the translation for any given Handel-C syntactic construct described in [9]. The semantics is going to be described as a set containing all possible execution traces for the program being synthesised.

As our hardware model captures the semantics of circuits with loop-back connections by replicating the appropriate behaviour (i.e., by means of successive syntactic approximations), we also need to account for the possibility of having an infinite set of traces. This feature clearly preempts any form of explicit description of the semantic set, especially if we consider that we are also aiming at mechanically verifying our approach.

The obvious solution is to find a predicate *smPred* such that we can define the semantic function `Sm` as $(\texttt{Sm}\ c) = \{s : Seq_\perp \mid smPred(c)\}$. In this context, *inductively defined relations* [8] are predicates defined by a set of rules for generating their elements.We adopt this approach to define our semantic predicate, taking the (informal) rules in the compilation schema as the basis for defining the set of rules for our semantic predicate *smPred*.

We also need for a way to incorporate the unique pair of wires (*start* and *finish*) generated for each statement in the program by the compilation process. We do so by means of the pair of functions $\pi_s(c)$ and $\pi_f(c)$ returning, respectively, the *start* and *finish* identifiers (belonging to the *wireId* type) for a given circuit c.

3.1 The semantic predicate

The hardware components generated by the synthesis process start their execution by performing a set of combinatoric actions (essentially to propagate their start signal to their constituent constructs) and also finish by carrying out a set of combinatoric actions (to propagate the finish signal appropriately). Even more, these "before" and "after" combinatoric actions performed by all the circuits are likely to be executed during the same clock cycle in which the previous circuit was terminating (or the next one is starting). This suggests that these are points in which the merging of action sets should take place to condense actions of the same type that happen in the same clock cycle in a single node.

We capture this notion by isolating these special points in the semantics, allowing us to have greater control over the structure of the sequences and the way in which they get concatenated/merged. We redefine our semantic predicate to relate a circuit c with two sets of combinatoric actions: *prologue* and *epilogue* (accounting respectively for actions at the beginning and end of the execution of the circuit) and a *behavioural sequence* (capturing the actions being executed in between these two combinatoric fragments).

On the other hand, the fact that the `while` construct reduces to pure combinatoric actions in the case where its condition is false makes its behaviour different

from the the rest of the constructs (that take at least one clock cycle to finish their execution). In this sense, we allow *smPred* to produce two kind of results: *mixed*, to capture the semantics of construct involving both combinatoric and sequential actions (this is our formulation based in the prologue, behavioural sequences and epilogue mentioned above); and *combinatoric*, to encode the case in which the semantics involve only actions that are performed within the current clock cycle. We split the actions in this last type of semantic expression into two, allowing us to have a prologue and epilogue when we produce the combinatoric type of result. We formalise this notion by defining the result type:

Definition 3.1

$$\begin{aligned} e \in smPredResult ::= \ & \texttt{Mixed}\ \mathscr{P}(Action) \to Seq \to \mathscr{P}(Action) \\ & \mid \texttt{Combin}\ \mathscr{P}(Action) \to \mathscr{P}(Action) \end{aligned}$$

For the remainder of the paper, we will use $[]$ and $\langle\rangle$ as short-hands for the `Mixed` and `Combin` constructors respectively. In these terms, the semantics of the `delay` construct state that its combinatoric prelude only includes a verification for the *start* wire, while its combinatoric prologue just sets its finish wire to the high value. The behavioural part of the circuit just states that it delays its execution for a single clock cycle (definition 3.2).

Definition 3.2 $smPred(\texttt{delay}, \left[\ \{(\pi_s(\delta))^{\perp}\}\ (\texttt{Ck}\ \{skip\})\ \{\pi_f(\delta)\}\right])$

The semantics for the assignment construct is very similar but the behavioural component is modified to capture the update to the store:

Definition 3.3

$$smPred(var = val, \left[\ \{(\pi_s(var = val))^{\perp}\}\ (\texttt{Ck}\ \{var \leftarrow val\})\ \{\pi_f(var = val)\}\right])$$

In the case of constructs c_1 and c_2 being sequentially composed, the prelude transfers the *start* signal from the sequential composition circuit to c_1's *start* and also includes c_1's combinatoric prelude (notice here that we are joining at this point the sequential composition's and c_1's preludes). The behavioural part comprises c_1's behaviour followed by a combinatoric set of actions turning the finish signal of c_1 into c_2's start and performing c_2's prelude, to conclude with c_2's behaviour. Finally, the sequential composition's prologue is composed of c_2's prologue and combinatoric hardware to propagate c_2's finish signal as the sequential composition's one. Definition 3.4 presents a more formal description of these actions.

Definition 3.4

$$\begin{array}{l}\forall\, c_1, init_{c_1}, seq_{c_1}, link_{c_1}, c_2, init_{c_2}, seq_{c_2}, link_{c_2} \bullet \\ \quad smPred(c_1,\ \lceil init_{c_1}\ seq_{c_1}\ link_{c_1}\rceil) \wedge smPred(c_2, \lceil\ init_{c_2}\ seq_{c_2}\ link_{c_2}\rceil) \Rightarrow \\ \qquad smPred(c_1 \fatsemi c_2, \\ \qquad\qquad \lceil init_{c_1} \cup \{(\pi_s(c_1 \fatsemi c_2))^{\perp}\,;\pi_s(c_1) \longleftarrow\!\circ\ \pi_s(c_1 \fatsemi c_2)\} \\ \qquad\qquad\ seq_{c_1} \boxplus (\mathtt{Cb}\ (link_{c_1} \cup \{\pi_s(c_2) \longleftarrow\!\circ\ \pi_f(c_1)\} \cup init_{c_2})\ seq_{c_2}) \\ \qquad\qquad (link_{c_2} \cup \{\pi_f(c_1 \fatsemi c_2) \longleftarrow\!\circ\ \pi_f(c_2)\})\rceil)\end{array}$$

We also need to add rules to handle the cases in which one of the constructs (or both) being composed sequentially only performs combinatoric actions. As an example, the case in which the first construct terminates "immediately" is described by definition 3.5.

Definition 3.5

$$\begin{array}{l}\forall\, c_1, init_{c_1}, link_{c_1}, c_2, init_{c_2}, seq_{c_2}, link_{c_2} \bullet \\ \quad smPred(c_1, \langle init_{c_1}\ link_{c_1}\rangle) \wedge smPred(c_2, \lceil init_{c_2}\ seq_{c_2}\ link_{c_2}\rceil) \Rightarrow \\ \qquad smPred(c_1 \fatsemi c_2, \lceil init_{c_1} \cup \{(\pi_s(c_1 \fatsemi c_2))^{\perp}\,;\pi_s(c_1) \longleftarrow\!\circ\ \pi_s(c_1 \fatsemi c_2)\} \cup \\ \qquad\qquad\qquad\qquad link_{c_1} \cup \{\pi_s(c_2) \longleftarrow\!\circ\ \pi_f(c_1)\} \cup init_{c_2} \\ \qquad\qquad\qquad seq_{c_2} \\ \qquad\qquad\qquad (link_{c_2} \cup \{\pi_f(c_1 \fatsemi c_2) \longleftarrow\!\circ\ \pi_f(c_2)\})\rceil)\end{array}$$

The symmetric case (the second construct terminates within the same clock cycle in which it was started) is described in a similar way and we do not show it here. The case in which the two constructs being composed in sequence terminate in the same clock cycle also terminates in a single clock cycle (definition 3.6).

Definition 3.6

$$\begin{array}{l}\forall\, c_1, init_{c_1}, link_{c_1}, c_2, init_{c_2}, link_{c_2} \bullet \\ \quad smPred(c_1, \langle init_{c_1}\ link_{c_1}\rangle) \wedge smPred(c_2, \langle init_{c_2}\ link_{c_2}\rangle) \Rightarrow \\ \qquad smPred(c_1 \fatsemi c_2, \langle init_{c_1} \cup \{(\pi_s(c_1 \fatsemi c_2))^{\perp}\,;\pi_s(c_1) \longleftarrow\!\circ\ \pi_s(c_1 \fatsemi c_2)\} \cup \\ \qquad\qquad\qquad\qquad link_{c_1} \cup \{\pi_s(c_2) \longleftarrow\!\circ\ \pi_f(c_1)\} \cup init_{c_2} \\ \qquad\qquad\qquad link_{c_2} \cup \{\pi_f(c_1 \fatsemi c_2) \longleftarrow\!\circ\ \pi_f(c_2)\}\rangle)\end{array}$$

In the case of the parallel composition of c_1 and c_2, the combinatoric prelude propagates the parallel composition's *start* signal to c_1 and c_2 *start* wires and brings together their combinatoric preludes. The behavioural component of the semantics is just constructed by parallel merging c_1 and c_2's behavioural sequences. Finally, the actions in the prologue include the prologues of both c_1 and c_2, together with combinatoric logic to generate the finish signal for the parallel composition when $\pi_f(c_1)$ and $\pi_f(c_2)$ are in *high*.

As with the sequential composition construct, it is also necessary to address the cases involving the instantaneous termination of the constructs being composed in

parallel. We omit these rules (together with the ones handling the selection and communication constructs) due to space constraints.

The semantics for the `while` construct needs to provide rules for handling the two possible outcomes of the evaluation of the condition. The first rule accounts for the case when the condition is false and the `while` terminates immediately (definition 3.7).

Definition 3.7

$$\forall\, c, body \bullet smPred(c * body, \langle \{(\pi_s(c * body))^{\perp}\,;((\neg c))^{\perp}\}\{\pi_f(c * body)\}\rangle)$$

When the condition holds, the traditional notion of syntactic approximations [12] is applied. We provide two rules for the approximation: a base case capturing the first approximation to the solution (by means of a single execution of the `while`'s body) and another one to capture the way in which we can construct a longer approximation from an existing one.

The first approximation calculates the semantics of the while's body, assumes that the looping-condition does not hold and uses its prologue to signal the `while`'s termination (definition 3.8).

Definition 3.8

$$\begin{array}{l}\forall\, b, init_b, seq_b, link_b, c \bullet \\ \quad smPred(b, [init_b\ seq_b\ link_b]) \Rightarrow \\ \qquad smPred(c * b, [\{(\pi_s(c * b))^{\perp}\,;(c)^{\perp}\,;\pi_s(b) \longleftarrow \pi_s(c * b)\} \cup init_b \\ \qquad\qquad\qquad\qquad seq_b \\ \qquad\qquad\qquad\qquad \{\pi_s(c * b) \longleftarrow \pi_f(b)\} \cup link_b \cup \\ \qquad\qquad\qquad\qquad \{(\neg c)^{\perp}\,;(\pi_s(c * b))^{\perp}\,;\pi_f(c * b) \longleftarrow \pi_s(c * b)\}])\end{array}$$

The final rule for the `while` construct is meant to extend an existing approximation (generated either by the basic rule above or by previous applications of itself). The approximation is constructed by appending one expansion of the body and the proper linking combinatoric action in front of the approximation's behavioural sequence (definition 3.9).

Definition 3.9

$$\begin{array}{l}\forall\, b, init_b, seq_b, link_b, init_{approx}, seq_{approx}, link_{approx}, c \bullet \\ \quad smPred(b, [init_b\ seq_b\ link_b]) \wedge \\ \quad smPred(c * b, [init_{approx}\ seq_{approx}\ link_{approx}]) \Rightarrow \\ \qquad smPred(c * b, [init_{approx} \\ \qquad\qquad\qquad seq_{approx} \boxplus (\texttt{Cb}\ (\{(c)^{\perp}\,;(\pi_s(c * b))^{\perp}\,;\pi_s(b) \longleftarrow \pi_s(c * b)\,; \\ \qquad\qquad\qquad\qquad\qquad \pi_s(c * b) \longleftarrow \pi_f(b)\} \cup link_{approx} \cup init_b)\ seq_b) \\ \qquad\qquad\qquad link_{approx}])\end{array}$$

We use the approximation-based principle defined above to give semantics to the `input` and `output` primitives. In this particular case, we need two rules per constructor because the approximation part of `inputs` and `outputs` only requires waiting for a clock cycle and retrying the communication (the equations for while are more complicated given the fact that we require to include the body's semantics on each approximative step). In providing the semantics for `input` and `output`, we need to address the different parts involved in the communication. We use $ch?$, $ch!$ and ch to denote the reading, writing and bus (the physical means by which the value being communicated is transmitted) components in the communication over channel ch.

In particular, the base case for the inputting construct uses its prelude to state the presence of the reading component and requests the presence of the writing part (by means of an assertion). If this is the case, the behavioural part of the semantics updates the store according to the value being transmitted over the channel. The actions in the prologue are used to establish the generation of the finish signal (definition 3.10).

Definition 3.10

$$
\begin{array}{l}
(\forall\, chan, var \bullet \\
\quad smPred(chan?\, var, \\
\qquad \big[\{(\pi_s(chan?\, var))^{\perp}; chan?\,; (chan!)^{\perp}\} \\
\qquad\ (\texttt{Ck}\ \{var \leftarrow chan\}) \\
\qquad\ \{\pi_f(chan?\, var)\}\big]))
\end{array}
$$

The rule defining the approximations when the writer is not ready to communicate just assert the proper condition in the combinatoric prelude ($(\neg chan!)^{\perp}$) and adds a one clock delay and appropriate combinatoric actions in front of the existing approximation (definition 3.11).

Definition 3.11

$$
\begin{array}{l}
(\forall\, chan, var, init_{approx}, seq_{approx}, link_{approx} \bullet \\
\quad smPred(chan?\, var, \big[init_{approx}, seq_{approx}, link_{approx}\big]) \Rightarrow \\
\qquad smPred(chan?\, var, \\
\qquad\qquad \big[init_{approx} \\
\qquad\qquad\ (\texttt{Ck}\ \{\}\ (\texttt{Cb}\ init_{approx} \cup \{\pi_s(chan?\, var)\}\ seq_{approx})), \\
\qquad\qquad\ link_{approx}\big]))
\end{array}
$$

Note that all the approximations constructed by this rule conclude with a successful communication (to see why, consider that the base case for the communication primitives is a successful one, and the fact that all approximations will just add "failing" communication behaviour in front of this successfully terminating trace). This fact is the key factor we use to order approximations (see section 2.2) as the "successful termination" part of the sequence is turned into $\perp$ by the prune function if the construct should not terminate its execution at this moment of execution

(i.e., the approximation has not yet reached the fix-point solution and, hence, the assertion stating the terminating condition to hold cannot be satisfied).

Finally, the base case for the `output` construct is similar to the one for `input`, but it inverts the roles in the combinatoric prelude (it establishes the presence of the writer and asserts the readiness of the reader). Its behavioural part is also different as it has to assign the value being transmitted to the appropriate channel (definition 3.12).

Definition 3.12

$$
\begin{array}{l}
(\forall\, chan, val \bullet \\
\quad smPred(chan!\, val, \\
\qquad \big[\{(\pi_s(chan!\, val))^{\perp}; chan!\,; (chan?)^{\perp}\,; chan \leftarrow val\} \\
\qquad\ (\texttt{Ck}\ \{Skip\}) \\
\qquad\ \{\pi_f(chan!\, val)\}\big]))
\end{array}
$$

The rule capturing the inductive approximations to the `output` construct using the same approach introduced for the `input` approximative solution is as described by definition 3.13.

Definition 3.13

$$
\begin{array}{l}
(\forall\, chan, val, init_{approx}, seq_{approx}, link_{approx} \bullet \\
\quad smPred(chan!\, val, \big[init_{approx}\ seq_{approx}\ link_{approx}\big]) \Rightarrow \\
\qquad smPred(chan!\, val, \\
\qquad\qquad \big[init_{approx} \\
\qquad\qquad (\texttt{Ck}\ \{Skip\}\ (\texttt{Cb}\ init_{approx} \cup \{\pi_s(chan!\, val)\}\ seq_{approx})), \\
\qquad\qquad link_{approx}\big]))
\end{array}
$$

Having reformulated the semantic predicate *smPred*, we still need to provide a way to use it in the definition of our semantic function `Sm`. Considering the fact that the prelude and prologue components of the semantic predicate are just sets of combinatoric actions, we define the semantic function for a given syntactic construct c as:

Definition 3.14

$$
\begin{array}{l}
\texttt{Sm}\ c = \{s \mid \exists init_c, seq_c, link_c \bullet \\
\qquad (smPred(c, \big[init_c\ seq_c\ link_c\big]) \Rightarrow (s = (\texttt{Cb}\ init_c\ seq_c) \boxplus (\texttt{Cb}\ link_c))) \wedge \\
\qquad (smPred(c, \langle init_c\ link_c \rangle) \Rightarrow (s = (\texttt{Cb}\ init_c \cup link_c)))\}
\end{array}
$$

3.2 Consistency considerations

Our definition of the semantic function relies on the sequences produced to have certain properties in order to satisfy the preconditions imposed by the operations in our semantic domain. The concatenation function requires the behavioural sequences produces by *smPred* to have the pattern $(\texttt{Cb}\ a_1\ s) \boxplus (\texttt{Cb}\ a_n)$ for sets of

combinatoric actions a_1 and a_n and a sequence s. Even more, we need to ensure that the sequences produced by the semantics are $inPhase$ to ensure a safe application of the merge operator.

The methodology we use to ensure the satisfaction of these requirements is as follows: we first define a suitable subset of the sequences in our semantic domain and prove that the properties mentioned above hold for any element in this subset. Then we prove that all the sequences produced by the semantic predicate belong to this subset, ensuring that the semantic function also satisfies the required conditions.

3.2.1 Clock-Bound sequences.

Given the properties we need to satisfy, we observed that all instances of s above should have their *boundaries* (i.e., first and last elements) based on the `Ck` constructor. Moreover, s should *alternate* (there shouldn't be two consecutive applications of the same constructor). The predicate $ckBnd$ captures this notion by means of the inductive definition:

Definition 3.15

$$(\forall\, cka \in \mathscr{P}(Action) \bullet ckBnd(\texttt{Ck}\ cka) \land$$
$$(\forall\, cka, ca \in \mathscr{P}(Action), s \in Seq \bullet ckBnd(s) \Rightarrow ckBnd(\texttt{Ck}\ cka\ (\texttt{Cb}\ ca\ s)))$$

We also need to show that any pair of sequences satisfying the $ckBnd$ property is also $inPhase$. The *alternation*-nature of $ckBnd$ sequences also guarantees the satisfaction of this requirement, allowing us to prove:

Lemma 3.16

$$\forall\, s_1, s_2 \in Seq \bullet ckBnd(s_1) \land ckBnd(s_2) \Rightarrow inPhase(s_1, s_2)$$

3.2.2 The semantic predicate only generates clock-bounded sequences.

We need to show that all behavioural sequences generated by $smPred$ belong to $ckBnd$ subtype. To do so, we first need two lemmas proving that the semantic domain operators preserve the $ckBnd$ property. Regarding the application of $\boxplus$ to clock-bounded sequences, it is not possible to prove that the concatenation of two $ckBnd$ sequences is still a clock-bounded sequence (it is not even possible to apply $\boxplus$ as the arguments do not satisfy its precondition). On the other hand, it is possible to prove that:

Lemma 3.17

$$\forall\, s_1, s_2 \in Seq, a \in \mathscr{P}(Action) \bullet$$
$$ckBnd(s_1) \land ckBnd(s_2) \Rightarrow ckBnd(s_1 \boxplus (\texttt{Cb}\ a\ s_2))$$

This result is strong enough to aid us in the proof of $smPred$'s preservation of the $ckBnd$ property. In fact, it is easy to observe that the way in which the concatenation function is applied in the above lemma is the only way in which the function is applied in $smPred$'s definition.

The proof of the lemma stating $\uplus$'s monotonicity with respect to the $ckBnd$

property is very complicated. The complication arises because of the way in which the *ckBnd*'s induction principle has to be applied (i.e., in sequential order) together with $\uplus$'s definition. In order to overcome this problem, we capture $\uplus$'s behaviour in the inductive predicate $parMerge$ and prove it equivalent to $\uplus$.

The main advantage of the predicate-based form of $\uplus$ is that it provides an induction principle, allowing us to conduct proofs by induction on the merge operator rather than on sequences or its properties. We then state that $parMerge$ preserves the $ckBnd$ property:

Lemma 3.18

$$\forall\, s_1, s_2, res \in Seq \bullet \\ ckBnd(s_1) \land ckBnd(s_2) \land parMerge(s_1, s_2, res) \Rightarrow ckBnd(res)$$

We use our equivalence result to show the lemma above also holds for $\uplus$:

Lemma 3.19

$$\forall\, s_1, s_2, res \in Seq \bullet ckBnd(s_1) \land ckBnd(s_2) \Rightarrow ckBnd(s_1 \uplus s_2)$$

The two main lemmas of this section allow us to prove that the behavioural sequences generated by $smPred$ belong to the subset of Seq induced by $ckBnd$:

Theorem 3.20

$$ckBnd_{seq_c} \vdash \forall\, c \in Contructs; init_c, link_c \in \mathscr{P}(Action); seq_c \in Seq \bullet \\ smPred(c, (\lbrack\!\lbrack (init_c\ seq_c\ link_c) \rbrack\!\rbrack)) \Rightarrow ckBnd(seq_c)$$

This final result ensures that the operators in the semantic domain are always applied within their definition domain by the semantic predicate.

3.3 Pruning

So far we have described the semantics of the translation from Handel-C into netlists as a (possibly infinite) set of finite-length sequences. In order to complete the semantic description of the generated circuits, we need to find (if it exists) a single sequence that specifies the actual execution path and outcome of the program being synthesised.

As in [10], we define two auxiliary functions: $\Delta Env : \texttt{env} \rightarrow Action \rightarrow \texttt{env}$ and $flattenEnv : \texttt{env} \rightarrow \texttt{env}$. The former updates the environment according to the action passed as argument by means of rewriting the appropriate function using λ-abstractions. In the particular case of *skip*, ΔEnv treats it as its unit value and returns the same environment. On the other hand, $flattenEnv$ is meant to be used to generate a new environment after a clock cycle edge. In particular, it flattens all wire values (to the logical value false), resets the channel values to the undefined value and advances the time-stamp by one unit.

We define the execution of sets of hardware actions by means of the predicate $exec : \texttt{env} \times \mathscr{P}(Assertion) \rightarrow (\texttt{env}, \mathscr{P}(Assertion))$ by means of the following rules:

$$\frac{s = \{\}}{exec(e, s) = (e, \{\})} \qquad \frac{s \neq \{\} \land a \in s}{exec(e, s) = exec(\Delta Env(e, a), s - \{a\})}$$

We also need to be able to handle assertions, so we introduce the function $sat^{\perp} : \texttt{env} \times \mathscr{P}(Assertion) \rightarrow bool$ defined as $sat^{\perp}(e, set) = \forall\, a \in set \bullet holds(a, e)$, where $holds(a, e)$ is true iff the assertion a is true in the environment e.

As we are dealing with sets of actions and assertions on each node of our sequences, we need to define the collective effect of this heterogeneous set of actions over the environment. The first difficulty we face when defining how a set of actions is going to be executed is that the initial order between actions and conditions has been lost. This is, however, not a problem if we consider that assertions and control flow conditions refer only to the present value of the memory and all variables preserve their values during the whole clock cycle. This fact makes the evaluation of assertions and control flow decisions independent of the combinatoric actions performed in parallel with them and they can be evaluated at any time.

From the observation above, we split the set of actions into the disjoint sets of assertions (As) and the remaining "unconditional actions" (HAs). The partition into As/HAs is induced over the set of actions by means of the functions ∇_{As} and ∇_{HAs}. Also from the observations above, we know that control-flow assertions can be evaluated at any time, and that wire-correctness assertions must be evaluated after HAs, allowing us to establish the following order of evaluation: HAs $\prec$ As.

Taking advantage of the execution order outlined above, we can introduce the function $setExec : \texttt{env} \times \mathscr{P}(Action) \rightarrow (\texttt{env} \cup \perp)$ defined by the following rule[4]:

$$\frac{e_{new} = exec(e, \nabla_{\text{HA}}(s)) \land sat^{\perp}(e_{new}, \nabla_{\text{A}}(s))}{setExec(e, s) = e_{new}}$$

In turn, the above functions can be used to define a single-node execution function for sequences $seqExec : \texttt{env} \times Seq \rightarrow ((\texttt{env}, Seq_{\perp}) \cup \{(\texttt{env}, \checkmark)\})$. The simplest case is successful termination (i.e., when the sequence we are trying to execute is empty), captured by the rule:

$$\frac{seq = []}{seqExec(e, seq) = (e, \checkmark)}$$

The next case captures the execution of a sequence that begins with a set of actions containing unsatisfiable conditions (the symmetric case is similar and we omit it here):

$$\frac{\exists\, ca \in \mathscr{P}(Action), s_1 \in Seq \bullet seq = (\texttt{Cb}\ ca\ s_1) \land setExec(e, ca) = \perp}{seqExec(e, seq) = (e, \perp)}$$

The case in which it is possible to perform all actions and satisfy all tests within a combinatoric node is described by the following rule:

[4] To keep the presentation compact, we omit the counterpart of this rule that maps all the cases when the antecedent does not hold to the $\perp$ value.

$$\frac{\exists\, ca \in \mathscr{P}(Action), s_1 \in Seq \bullet seq = (\texttt{Cb}\ ca\ s_1) \land setExec(e, ca) = e_{new}}{seqExec(e, seq) = (e_{new}, s_1)}$$

The counterpart of the above rule (dealing with sequences starting with a clock-edged block) is as follows:

$$\frac{\exists\, ca \in \mathscr{P}(Action), s_1 \in Seq \bullet seq = (\texttt{Ck}\ ca\ s_1) \land setExec(e, ca) = e_{new}}{seqExec(e, seq) = (flattenEnv(e_{new}), s_1)}$$

Note that the environment needs to be *flattened* after all actions at the clock edge have taken place. The flattening can take place only at this point because of the possibility of having a value being transmitted over a bus (we will lose the value being transferred if we flatten the environment before updating the store with it).

In order to get the actual execution path (for the case in which the program terminates) we define the operator $prune : \texttt{env} \to \mathscr{P}(Seq) \to (\texttt{env}, \mathscr{P}(Seq_\bot)) \cup \{\texttt{env}, \checkmark\}$ that advances one step at a time over all sequences in the set (using the function $seqExec$ defined above), updating the environment accordingly and removing unsatisfiable sequences. To deal with the infiniteness of the set, we need to observe that the sequences in the set can be partitioned into equivalence classes, grouping together sequences that share the same head. In particular, the "infiniteness" is brought to the set by the approximation chains used in the semantics for circuits with loop-back wiring. The way in which the approximations are constructed forces all of them to share the same trace of actions and to differ only at the very last node of each of them. In this way, we have only a finite number of equivalence classes (the amount of classes is directly proportional to the branching in the control flow of the program, which is known to be finite) at any given time and the effect of all the sequences in a given class over the environment in the current clock cycle is the same. Moreover, as Handel-C's control flow is governed by boolean conditions, only one of the possible branches is executable at any given time, making our semantic traces *mutually exclusive*. From this observation, it follows that only one of the equivalence classes will remain in the set of traces after the execution of the combinatoric header (all the other traces with unsatisfiable conditions will reduce to bottom and will be removed from the set).

4 Wire-wise Correctness

We are now in a good position to verify the correctness of the wiring schema used to link the different components used at the hardware level. In particular, we are interested in proving the wire-correctness of the generated hardware by means of verifying whether (a) the activation signal is propagated from the *finish* signal of the previous circuit; (b) the start signal is given to each component at the right clock cycle; and (c) the internal wiring of each circuit propagates the control signals

in the right way and produces the *finish* pulse at the right time.

It is worthwhile noting that part of the verification is straightforward from the way in which the compilation is done. In particular, each construct is compiled into a *black box* and it is interfaced only through its *start* and *finish* wires. In this sense, it is impossible for a circuit to be started by any component but the circuit containing it, preempting the chance of a component c being activated by a random piece of hardware. After this observation, to prove (a) we only need to prove that the *finish* wire of the appropriate circuit is set to *high* by the time the subsequent circuit is started.

Regarding the verification of the *start* signal given at the right time (condition b), we have already included assertions regarding the *start* signal in the combinatoric prelude of all constructs in order to make sure that the circuit receives a *start* pulse during the first clock cycle of its execution. The remaining aspect of this question is whether our semantic model of the hardware activates the circuits at the right clock cycle. Towards this question, the synchronous-time model used in Handel-C together with the component-based approach used in the compilation allows us to verify that the timing in our semantic model is equivalent to the one of the generated hardware. In this context, assuming that the generated hardware implements the timing model correctly, we only need to verify that the wire-related assertions are satisfied in order to verify (b).

The rest of this section is devoted to verifying the wire-correctness of the hardware based on the observations above. We first define a way of calculating if a given wire is *high* within the current clock cycle. We then define the concept of wire-satisfiability capturing the notion of wire-based assertions being true in a given set of combinatoric actions and use it to prove all the circuits are given the start signal in the clock cycle they are supposed to be started.

4.1 Wire-transfer closure

The fact that all the combinatoric actions happening at a given clock cycle are collected together in a set provides enough information to "calculate" which wires hold the *high* value in that clock cycle. In fact, the information about a wire holding the *high* value can be given either by a single formula (such as $\pi_s(c)$) or by a chain of value transfers from other wires (such as $\{w_1 \longleftarrow\!\circ\ w_2\,;w_2\}$). In this context, we define the notion of a wire in *high* by means of the $isHigh$ predicate:

Definition 4.1

$$\begin{aligned}
&\forall\, w_1, set \bullet (w_1 \in set) \Rightarrow isHigh(w_1, set)) \wedge \\
&\forall\, w_1, w_2, w_3, set \bullet isHigh(w_2, set) \wedge isHigh(w_3, set) \wedge \\
&\qquad\qquad (w_1 \longleftarrow\!\circ\ (w_2 \wedge w_3) \in set) \Rightarrow isHigh(w_1, set) \wedge \\
&\forall\, w_1, w_2, set \bullet isHigh(w_2, set) \wedge (w_1 \longleftarrow\!\circ\ w_2 \in set) \Rightarrow isHigh(w_1, set)
\end{aligned}$$

The predicate $isHigh$ captures the notion of a wire w holding the high value provided the actions in *set* are executed. From this definition we were able to prove some lemmas that will be necessary in the following sections. In particular, if a

given wire w holds the high value in a given set s, then it still does so in a bigger set:

Lemma 4.2 $isHigh(w, s) \Rightarrow isHigh(w, (s \cup s_1))$

It is also possible to prove that any explicit action in the set of actions setting up a wire w_1 to the *high* value can be replaced by a pair of actions, one setting up a new wire w_2 to the *high* value and another one to propagate its value into w_1:

Lemma 4.3

$$isHigh(w, s \cup \{w_1\}) \Rightarrow isHigh(w, s \cup \{w_2 \,;\, w_1 \longleftarrow\!\circ\; w_2\})$$

With these results we are able to prove that for any given construct c the prologue in the semantics always sets its finish wire $\pi_f(c)$ to *high*:

Theorem 4.4

$$smPred(c, (\lbag\, init_c\ seq_c\ link_c \,\rbag)) \Rightarrow isHigh(\pi_f(c), link_c) \wedge$$
$$smPred(c, (\langle init_c\ link_c \rangle) \Rightarrow isHigh(\pi_f(c), link_c)$$

This is the first result towards proving (a) in the introduction of this section. In order to complete our verification of (a) we introduce a new assertion type $(w_1 \leftrightsquigarrow w_2)^{\perp}$ defined to hold iff $isHigh(w_1) \wedge isHigh(w_2)$ holds in a given set of combinatoric actions. We then modify our semantic predicate to include this new type of assertions at the places where condition (a) is expected to hold. As an example, we show the updated version of the semantics for the sequential composition construct:

Definition 4.5

$$\begin{aligned}
&\forall\, c_1, init_{c_1}, seq_{c_1}, link_{c_1}, c_2, init_{c_2}, seq_{c_2}, link_{c_2} \bullet \\
&\quad smPred(c_1, \lbag init_{c_1}\ seq_{c_1}\ link_{c_1} \rbag) \wedge smPred(c_2, \lbag init_{c_2}\ seq_{c_2}\ link_{c_2} \rbag) \Rightarrow \\
&\qquad smPred(c_1 \fatsemi c_2, \lbag init_{c_1} \cup \{\pi_s(c_1) \longleftarrow\!\circ\; \pi_s(c_1 \fatsemi c_2)\} \\
&\qquad\qquad seq_{c_1} \boxplus (\mathtt{Cb}\ (link_{c_1} \cup init_{c_2} \cup \{\pi_s(c_2) \longleftarrow\!\circ\; \pi_f(c_1)\} \cup \\
&\qquad\qquad\qquad \{(\pi_f(c_1) \leftrightsquigarrow \pi_s(c_2))^{\perp}\}\ seq_{c_2}) \\
&\qquad\qquad link_{c_2} \cup \{\pi_f(c_1 \fatsemi c_2) \longleftarrow\!\circ\; \pi_f(c_2)\}) \rbag)
\end{aligned}$$

4.2 Assertion satisfiability

Having defined the concept of wire being set to *high*, we need a way to capture the idea of satisfaction of our assertions regarding wires. In particular, we say that all the wire-related assertions in a set are *satisfied* iff the predicate $wireSAT$ holds[5]:

Definition 4.6

$$wireSAT(s) = \forall\, w \in WireIds \bullet (w)^{\perp} \in s \Rightarrow isHigh((w)^{\perp}, s)$$

[5] We replace assertions of the form $(w_1 \leftrightsquigarrow w_2)^{\perp}$ by the equivalent set of assertions $\{(w_1)^{\perp} \,;\, (w_2)^{\perp}\}$, allowing us to use the simple form of satisfiability defined before.

Having the definition of wire-satisfiability we can prove that if the hardware generated from any given syntactic construct c is started, then all the wire-related assertions in its combinatoric prelude hold:

Theorem 4.7

$$smPred(c, \lceil init_c\ seq_c\ link_c \rceil) \Rightarrow wireSAT(init_c \cup \{\pi_s(c)\}) \wedge$$
$$smPred(c, \langle init_c\ link_c \rangle) \Rightarrow wireSAT(init_c \cup \{\pi_s(c)\})$$

This result is proving consideration (a) from the previous section: the activation signal is propagated from the parent/previous circuit into the start signal of the current one. In fact, even though the above theorem is just stating that it happens that the right *start*/*finish* signals get the high value in the appropriate clock cycle, evidence gathered during the proof process showed that the *high* value actually gets propagated between them, proving consideration (a) to its full extent. It is also showing that consideration (b) holds: the start *signal* is given to each circuit at the appropriate time (provided that the time models of the hardware compilation are correct regarding the Handel-C's semantics).

We also proved that the epilogue set of combinatoric actions satisfies $wireSAT$:

Lemma 4.8

$$smPred(c, \lceil init_c\ seq_c\ link_c \rceil) \Rightarrow wireSAT(link_c) \wedge$$
$$smPred(c, \langle init_c\ link_c \rangle) \Rightarrow wireSAT(link_c)$$

Provided that (a) and (b) hold, the verification of (c) can be reduced to proving that the assertions in the behavioural part of the semantic predicate are satisfied. The rationale behind this affirmation is that the base cases for the $smPred$ trivially satisfy (c) while compound circuits can be regarded as placeholders linking *start*/*finish* wires of different components by means of combinatoric actions. The assertions introduced in order to verify (a) and (b) are, in this context, checking that those actions are propagating the right value among the different involved components.

In order to verify the behavioural sequences from the semantic predicate we first need to extend the concept of wire-satisfiability to sequences. We do so by defining the function $wireSAT_{seq}$ as follows:

Definition 4.9

$$wireSAT_{seq}(\bot) = F \wedge wireSAT_{seq}([]) = T$$
$$wireSAT_{seq}(\texttt{Cb}\ a\ s_1) = wireSAT(a) \wedge wireSAT_{seq}(s_1)$$
$$wireSAT_{seq}(\texttt{Ck}\ a\ s_1) = wireSAT_{seq}(s_1)$$

Before being able to use the $wireSAT_{seq}$ function to prove that the wiring is correct in the behavioural sequences we need to prove two lemmas regarding $\boxplus$ and $\uplus$ preserving the wire-satisfiability property for sequences if it holds true for their arguments. The case of concatenation is straightforward by induction over the composed sequences:

Lemma 4.10

$$wireSAT_{seq}(s_1 \boxplus s_2) \Leftrightarrow wireSAT_{seq}(s_1) \wedge wireSAT_{seq}(s_2)$$

To prove the equivalent result for the parallel-merge operator is a very complicated task given the already mentioned lack of an induction principle capable of handling two sequences as a pair. As we are still within the context of the semantic predicate (and hence, within the subclass of clock-bounded sequences) we can prove the easier goal:

Lemma 4.11

$$wireSAT_{seq}(s_1) \wedge wireSAT_{seq}(s_2) \wedge parMerge(s_1, s_2, res) \Rightarrow wireSAT_{seq}(res)$$

and then use the equivalence between $parMerge$ and $\uplus$ to deduce the equivalent result for $\uplus$. With these two results, we are able to prove the correctness of the wiring for the behavioural sequences produced by $smPred$:

Theorem 4.12

$$smPred(c, [init_c\ seq_c\ link_c]) \Rightarrow wireSAT_{seq}(seq_c)$$

With the three main theorems of this section, we show that the wiring is correct (regarding our wire-satisfiability criteria) for any given construct in the core language.

5 Conclusions and future work

The main contributions of this work are an improved semantic model for the hardware components synthesised from Handel-C programs and the mechanical verification of the wiring schema used to handle the control flow among those components.

This work presents a more abstract semantic domain than the one used in previous works and allows a better description of the parallelism exhibited by the synthesised hardware. In particular, we have defined our semantic domain in terms of a deep embedding of sequences of state-transformers in Higher Order Logic. We have also established a partial order relationship over the domain and proved the existence of fix-point solutions to our inductive approximations for recursive constructs.

The synthesis process we are formalising is based on the assumption that no hardware component will be activated unless a precise signal has been given to it through its interface. We have captured this notion by embedding Handel-C's syntactic constructs in HOL and providing a semantic function that maps each construct in the language to its representation in our semantic model. Moreover, the way in which Handel-C's synchronous nature is encoded in the model introduces explicit information about the value held by the wires used to link different components. We have taken advantage of this feature to formally verify the correctness of

the wiring schema used in the compilation.

Even though we have proved that each of the hardware components propagate the control token in the right way, we still need to prove that the hardware generated by the compilation rules is *correct* (i.e., semantically equivalent to its original Handel-C code). This correctness proof will also allow us to discharge the only assumption of this work: that the timing model of the generated hardware is consistent with the one for Handel-C. Towards this end, our next step is to prove the existence of an equivalence relationship using the semantic models for Handel-C [2] and the semantics for the generated hardware presented in this paper.

References

[1] A. Butterfield. Denotational semantics for prialt-free Handel-C. Technical report, The University of Dublin, Trinity College, December 2001.

[2] A. Butterfield. A denotational semantics for Handel-C. In *Formal Methods and Hybrid Real-Time Systems*, pages 45–66, 2007.

[3] R. Floyd. Assigning meaning to programs. In *Mathematical Aspects of Computer Science*, number 19, pages 19–32. American Mathematical Society, 1967.

[4] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 26(1):100–106, 1983.

[5] C.A.R. Hoare and H. Jifeng. *Unifying Theories of Programming*. Prentice-Hall, 1998.

[6] Celoxica Ltd. *DK3: Handel-C Language Reference Manual*, 2002.

[7] Celoxica Ltd. *The Technology behind DK1*, August 2002. Application Note AN 18.

[8] T. F. Melham. A Package for Inductive Relation Definitions in HOL. In *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications*, pages 350–357. IEEE Computer Society Press, 1992.

[9] I. Page and W. Luk. Compiling Occam into field-programmable gate arrays. In *Oxford Workshop on Field Programmable Logic and Applications*, pages 271–283. 1991.

[10] J. Perna and J. Woodcock. A denotational semantics for Handel-C hardware compilation. In Michael Butler, Michael G. Hinchey, and María M. Larrondo-Petrie, editors, *ICFEM*, volume 4789 of *Lecture Notes in Computer Science*, pages 266–285. Springer, 2007.

[11] J. Perna and J. Woodcock. Proving wire-wise correctness for Handel-C compilation in HOL. Technical Report YCS-2008-429, Computer Science Department, The University of York, December 2007.

[12] D. Scott and C. Strachey. Towards a mathematical semantics for computer languages. In J. Fox, editor, *Proceedings Symposium on Computers and Automata*, pages 19–46. Pol. Inst. of Brooklyn Press, New York, 1971.