

A correct compiler for the Handel-C language

Juan I. Perna and Jim Woodcock

Computer Science Department
The University of York
jjperna@cs.york.ac.uk

February 27, 2009

Handel-C by example: a two-places buffer

```
void main(void) {  
    chan int 8 link,in,out;  
    unsigned int state[2];  
  
    par {  
        while (1) {  
            in ? state[0];  
            link ! state[0];  
        }  
  
        while (1) {  
            link ? state[1];  
            out ! state[1];  
        }  
    }  
}
```

General features

- Shared variables
- Parallelism (hardware-like)
- Communications
- Prioritised choice
- No hardware-level control

Simple time model

- Synchronous
- Assignment/delay: one clock cycle
- Other constructs: combinatorial

Handel-C compilation

Compilation approach

Based on template-based approach proposed by Ian Page

Main features

- Simple interface: start/finish wires
- Compositional

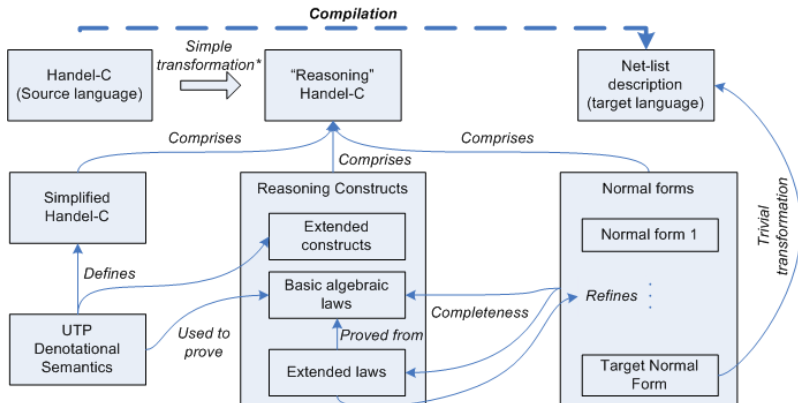
Correctness

- Validated: extensively used (Celoxica)
- Formal model (denotational semantics)
 - Correctness of the wiring schema
- Ad-hoc implementation

Our research project

Goal: Correct Handel-C synthesis

- Algebraic compiler



Outline

- Motivation + Introduction
- **Input language**
 - Simplifying Handel-C
 - UTP-semantics for Handel-C
 - Laws about Handel-C (proved from the semantics)
- Extended Handel-C
- Normal forms

Restricted Handel-C

Declarations

- Standard types: **int/ bool**
- Channels: **chan**

Core features of the language

- one-cycle delay (**delay**)
- assignment ($v \stackrel{:=}{\text{HC}} e$)
- sequential composition ($c_1 \circ c_2$)
- parallel composition ($c_1 \parallel_{\text{HC}} c_2$)
- selection (**if** b **then** c_1 **else** c_2)
- while loops (**while** b **do** c)
- communication ($ch?x$ and $ch!e$)

Restricted Handel-C (cont)

- priorities (**priAlt** {⟨guards⟩})

⟨guards⟩ ::= ⟨guard⟩ ; ⟨guards⟩

| ⟨guard⟩

| **default: P ; break**

⟨guard⟩ ::= **case ch?x: P ; break**

| **case ch!e: P ; break**

What is left out?

- Pointers, complex datatypes, arrays
- RAM/ROM commands
- Functions (and function calls)

UTP: from alphabetised relations to reactive processes

Alphabetised predicate calculus

- *Programs are predicates*
- Correctness: $S \sqsubseteq P$ iff $[P \Rightarrow S]$
- **Semantics for recursion:** Recovery from divergence!

Design theory

- ok/ok' \rightarrow the process has *started/terminated*
- Initial-final state relationship: $(P \vdash Q) =_{df} (ok \wedge P \Rightarrow ok' \wedge Q)$
- **H1-H4:** from predictability to feasibility
- Constructs: sequential and parallel composition, assignment, recursion, non determinism
 - Algebraic laws already proven for all operators!

A bit of the state-of-the-art

Existing semantics

- Branching sequences of state-transformers
 - Too operational → too complicated for proofs
- Hardware-semantics
 - Based on state machines
 - (Deep) embedding in UTP reactive designs
- Typed-assertion traces
 - Based on the reactive-designs theory
 - Slotted *Circus*
 - Too much expressive power for Handel-C?

Only trivial laws/properties have been proved from them

Our UTP semantics for Handel-C

Based on the theory of designs

Key idea: observations are required at clock edges only

Extensions to the design theory

- New observation: clock cycle counter 'c'
 - Controlled by the action **tick** $=_{df} c := c + 1$
- Keep track of variable's history
 - $(x := 2; x := 5) = (x := 5) \neq (x_{c,c+1} := \langle 2, 5 \rangle) = (x_{HC} := 2 \text{ ; } x_{HC} := 5)$
- Redefinition of parallel by merge
 - Removes restriction of equal duration of parallel branches
- Input/output/bus observations for communication
 - Semantics in terms of fixed points

Our UTP semantics for Handel-C (cont)

C-based constructs: shallow embedding

- $X \stackrel{:=}{\text{HC}} e =_{df} X := e; \mathbf{tick}$
- $\mathbf{delay} =_{df} \mathit{II}; \mathbf{tick}$
- $P \circledast Q =_{df} P; Q$
- $P \parallel_{\text{HC}} Q =_{df} P \parallel_{\hat{M}} Q$
- $\mathbf{if} \ b \ \mathbf{then} \ P \ \mathbf{else} \ Q =_{df} P \triangleleft b \triangleright Q$
- $\mathbf{while} \ b \ \mathbf{do} \ P =_{df} \mu X \bullet P \triangleleft b \triangleright \mathit{II}$

CSP-based constructs: 'deeper' embedding

$$\llbracket ch?m \rrbracket =_{df} \mu X \bullet ch?_c := true;$$
$$((m.in_c := \overleftarrow{ch}'_c, \mathbf{tick}) \triangleleft \overrightarrow{ch}'_c \triangleright (m.in_c := m_{c-1}; \mathbf{tick}; X))$$

Basic laws proved from the semantics

Sequential composition, parallel composition and assignment...

$$\mathbf{L1} \quad P \parallel_{\text{HC}} Q = Q \parallel_{\text{HC}} P \quad \parallel_{\text{HC}}\text{-comm}$$

$$\mathbf{L2} \quad (P \parallel_{\text{HC}} Q) \parallel_{\text{HC}} R = P \parallel_{\text{HC}} (Q \parallel_{\text{HC}} R) \quad \parallel_{\text{HC}}\text{-assoc}$$

$$\mathbf{L3} \quad \mathbb{I} \parallel_{\text{HC}} P = P = P \parallel_{\text{HC}} \mathbb{I} \quad \parallel_{\text{HC}}\text{-skip}$$

$$\mathbf{L4} \quad x \stackrel{:=}{\text{HC}} e \circ (P \parallel_{\text{HC}} Q) = (x \stackrel{:=}{\text{HC}} e \circ P) \parallel_{\text{HC}} (x \stackrel{:=}{\text{HC}} e \circ Q)$$

$$\mathbf{L5} \quad x \stackrel{:=}{\text{HC}} e \circ (P \parallel_{\text{HC}} Q) = (x \stackrel{:=}{\text{HC}} e \circ P) \parallel_{\text{HC}} (\mathbf{delay} \circ Q)$$

$$\mathbf{L6} \quad x, y \stackrel{:=}{\text{HC}} e_1, e_2 \circ (P \parallel_{\text{HC}} Q) = (x \stackrel{:=}{\text{HC}} e_1 \circ P) \parallel_{\text{HC}} (y \stackrel{:=}{\text{HC}} e_2 \circ Q)$$

$$\mathbf{L7} \quad (\mathbf{if } b \mathbf{ then } P \mathbf{ else } Q) \parallel_{\text{HC}} S = \mathbf{if } b \mathbf{ then } P \parallel_{\text{HC}} S \mathbf{ else } Q \parallel_{\text{HC}} S$$

Basic laws proved from the semantics

Some of the laws we got 'for free' from UTP

$$\mathbf{L8} \quad P \circledast (Q \circledast S) = (P \circledast Q) \circledast S \quad \circledast\text{-assoc}$$

$$\mathbf{L9} \quad P \circledast \mathbb{I} = P = \mathbb{I} \circledast P \quad \circledast\text{-skip}$$

$$\mathbf{L10} \quad \text{if } \textit{true} \text{ then } P \text{ else } Q = P$$

$$\mathbf{L11} \quad \text{if } b \text{ then } P \text{ else } Q = \text{if } \neg b \text{ then } Q \text{ else } P$$

$$\mathbf{L12} \quad (\text{if } b \text{ then } P \text{ else } Q) \circledast S = \text{if } b \text{ then } P \circledast S \text{ else } Q \circledast S$$

The semantics in action

A simple program...

$$\begin{aligned}
 & x_{\text{HC}} := 8 \circledast ((x_{\text{HC}} := x + 1) \parallel_{\text{HC}} (y_{\text{HC}} := 1 \circledast x_{\text{HC}} := x + y + 1)) \\
 = & \text{ [L8, L4]} \\
 & x_{\text{HC}} := 8 \circledast (x, y_{\text{HC}} := x + 1, 1) \circledast (I \parallel_{\text{HC}} x_{\text{HC}} := x + y + 1) \\
 = & \text{ [L5]} \\
 & x_{\text{HC}} := 8 \circledast (x, y_{\text{HC}} := x + 1, 1) \circledast x_{\text{HC}} := x + y + 1
 \end{aligned}$$

main function + semantics + predicate calculus

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

Outline

- Motivation + Introduction
- Input language
- **Extended Handel-C**
 - Extensions for reasoning
 - Extended semantics for the reasoning language
- Normal forms

Miracle, Abort and so on...

Constructs to be included in the reasoning language

- Miracle: \top
- Abort: \perp
- One clock cycle skip: \mathbb{I}_1
- 'Combinational' assignment: $x := e$
- One clock cycle assignment: $(x := e)_1 =_{df} x := e; \mathbb{I}_1$
- Assumption: $(b)^\top =_{df} \mathbb{I} \triangleleft b \triangleright \top$
- Assertion: $(b)_\perp =_{df} \mathbb{I} \triangleleft b \triangleright \perp$
- Variable scope: **var** x and **end** x

Miracle, Abort and so on...

Constructs to be included in the reasoning language (cont)

- Non-deterministic choice (demonic): $P \sqcap Q$
- Guarded command: $b \rightarrow P =_{df} P \triangleleft b \triangleright \mathbb{I}$
- Choice: $b \rightarrow P \sqcap c \rightarrow Q =_{df} (b \rightarrow P) \parallel_{\hat{M}} (c \rightarrow Q)$
- Iteration: $b * P =_{df} \mu X \bullet (P; X) \triangleleft b \triangleright \mathbb{I}$
- Guarded iteration:
 $*(b \rightarrow P \sqcap c \rightarrow Q) =_{df} (b \vee c) * (b \rightarrow P \sqcap c \rightarrow Q)$
- Refinement: $P \sqsubseteq Q$ (Q is at least as good as P)

UTP semantics for the extended input language

Key ideas

- Still within the Design theory + Handel-C extensions
- Use UTP constructs as much as possible

Denotational model for the 'basic' extended constructs

- $\mathbb{I}_1 \xrightarrow{UTP} \mathbf{tick} (=_{df} c := c + 1)$
- $(x := e)_1 \xrightarrow{UTP} x_c := e; c := c + 1$

Problems yet to be sorted out

- How to define **var** , **end** in the semantics?
 - Re-define parallel composition?
- Proofs of some basic axioms are still missing

Outline

- Motivation + Introduction
- Input language
- Extended Handel-C
- **Normal forms**
 - First normal form: state-based encoding
 - Second normal form: removing parallel by merge

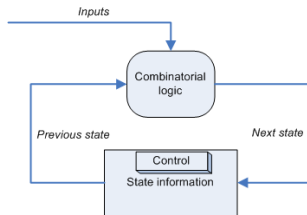
Encoding control information in the state

Key ideas

- Encode control information as state
- One-clock cycle long descriptions
 - Simple model (good!)
 - Can we express everything in this way?

Mealey-machine encoding

- Inherent time control
 - Control change = next clock cycle
- Next state depends on
 - Current state (including control)
 - Control flow
 - Inputs (communication/priAlt)



Our normal form

$$c : [a, (b \rightarrow P), f] =_{df} \mathbf{var} \ c; a^T; b * P; f_{\perp}; \mathbf{end} \ c$$

- c is control variable/s of the machine
- a and f are the initial/final conditions over c
- $b \rightarrow P$ describes the actions in the normal form
 - $b_1 \rightarrow P_1 \square c_1 \rightarrow P_2 = (b_1 \vee c_1) \rightarrow (b_1 \rightarrow P_1 \square c_1 \rightarrow P_2)$
- $f \wedge b = \mathit{false}$

Key feature

- Inherently parallel

Normal form encoding of some of the constructs

Basic constructs

- **delay** = $c : [s, (s \rightarrow (v, c := v, f)_1), f]$
- $x \stackrel{:=}{\text{HC}} e = c : [s, (s \rightarrow (x, c := e, f)_1), f]$

Sequential and parallel composition

- $c : [s, (b_1 \rightarrow P_1), f_1] \circ c : [f_1, (b_2 \rightarrow P_2), f] \sqsubseteq$
 $c : [s, (b_1 \rightarrow P_1 \square b_2 \rightarrow P_2), f]$
- $c_1 : [s, (b_1 \rightarrow P_1), f] \parallel_{\hat{M}} c_2 : [s, (b_2 \rightarrow P_2), f] \sqsubseteq$
 $c_1, c_2 : [(s, s), \left(\begin{array}{l} b_1 \rightarrow P_1 \square (\neg b_1 \wedge b_2) \rightarrow (c_1 := f)_1 \square \\ b_2 \rightarrow P_2 \square (b_1 \wedge \neg b_2) \rightarrow (c_2 := f)_1 \end{array} \right), (f, f)]$

Encoding channels in the normal form

Channels are special elements in Handel-C

- Can only be manipulated by input/output

Input/Output are complicated!

- Behaviour in terms of underlying wires (hidden from the user)
- Inherently parallel (context dependency)
- Multi-synchronisation

Normal form/implementation issues

- Channels cannot be directly represented in hardware
- How to capture input/output in a normal form?

First approach: data-based approach

Turning channel into wires: key ideas

- Introduce special variables (wires) in the model
- Replace input/output/priAlt with lower level equivalents

```
void main() {  
  int chan ch;  
  int var;  
  
  par {  
    while (1) {  
      ch ! var;  
    }  
  }  
  ...  
}
```

⇒

```
void main() {  
  bool ch.out, ch.in;  
  int ch.val, var;  
  
  par {  
    while (1) {  
      [[ch ! var]];  
    }  
  }  
  ...  
}
```

Encoding input/output/priAlt

The Input construct implementation (from the semantics!)

$\mu X \bullet ch?(id) := true;$

$$\left((x := ch; ch?(id) := false)_1 \triangleleft \bigvee_{i \in ch!} ch!(i) \triangleright \mathbf{delay}; X \right)$$

Problems of this approach

- How to prove correctness of the algebraic law?
 - It is EQUAL to the semantic definition!
- Transformation has to be carried over the whole program
 - Not very 'algebraic'
- External wire dependency breaks compositionality

Additional reasoning constructs

Alternative approach: operational view

- Request: **in-req**(ch) and **out-req**(ch)
- Reader/writer presence: **rd**(ch) and **wr**(ch) x
- Unconditional input/output: **in**(ch) and **out**(ch, e)

Key results

Equivalence with Input and Output Handel-C constructs

- $c?x = \mu X \bullet \mathbf{in\text{-}req}(c); ((x := \mathbf{in}(c))_1 \triangleleft \mathbf{wr}(c) \triangleright \mathbf{delay}; X)$
- $c!e = \mu X \bullet \mathbf{out\text{-}req}(c); \mathbf{out}(c, e); (\mathbf{delay} \triangleleft \mathbf{rd}(c) \triangleright \mathbf{delay}; X)$

- Semantics in terms of underlying wiring

The problem with the While construct...

A 'false' while terminates **immediately**

The problem in our normal form:

Each guarded step takes **one clock cycle** (definition)

Our solution

- Allow the generation of a 'vacuous' clock cycle
- Replicate the first actions in the next construct in the slot

Can we always find a 'next construct'?

- No, but we can force the user to put one!
 - New code may be slightly less efficient (how frequently?)
- Aim of this work: correctness (pay the price for it?)

Putting it all together: a simple example

A sequence of assignments

$$(x_{\text{HC}} := 3 \triangleleft b \triangleright 10 \parallel y_{\text{HC}} := 98) \circ x_{\text{HC}} := 4$$

Generates...

$$c : [c = 0, \left(\begin{array}{l} c = 0 \rightarrow (x, y, c := 3 \triangleleft b \triangleright 10, 98, 1)_1 \square \\ c = 1 \rightarrow (x, y, c := 4, y, 2)_1 \end{array} \right), c = 2]$$

Very nice but...

the parallelism is not implementable at the hardware level

A new parallel operator

Key ideas

- No more than one update to any variable per clock cycle
 - Handel-C's property!
- Each guarded step in the normal form takes one clock cycle
 - We have defined it like that!

No need for a parallel by merge operator!

Some problems with this idea...

- Semantics: Processes' alphabets still not disjoint
- Operational: Parallel reading should be allowed
- Model perspective: the clock is still shared

A new parallel operator (cont)

Semi-disjoint parallel operator ($\textcircled{\wedge}$)

- Relaxes alphabet-disjoint condition from UTP
 - Only one process can update the 'after state'
 - Many processes can refer to the 'before state'
- Allows ordering of parallel processes in the design theory

Clock-shared parallel ($\textcircled{\parallel}$)

- Multi-cycle, synchronous, hardware-like parallelism
- Defined in terms of $\textcircled{\wedge}$
 - Actions within each clock cycle are semi-disjoint
- Only merges clock cycles
 - Restriction from the semantics
 - Each process synchronises by updating its clock

From the first to the second normal form

First normal form (recap)

$$c : [a, (b \rightarrow P_1 \parallel_{\hat{M}} c \rightarrow P_2), f]$$

If we focus on $b \rightarrow P_1$:

$$b \rightarrow (x, y := v_1, v_2)_1$$

= {UTP assignment normal form rules}

$$(x, y := v_1, v_2)_1 \triangleleft b \triangleright (x, y := x, y)_1$$

⊆ {UTP assignment normal form rules then reasoning language rules}

$$\mathbf{var} \ 1.x; (y, 1.x := v_2, v_1 \triangleleft b \triangleright y, x)_1 \parallel_{\hat{C}} (x := 1.x')_1; \mathbf{end} \ 1.x$$

$(y, i.x := v(x, y) \triangleleft b \triangleright y, f(x, v))_1$ does not modify shared x !

From the first to the second normal form (cont)

Applying the previous ideas to the first normal form...

```

var c; sT;
  (
    (var 1.x; (y, 1.x := v2, v1 ◁ b ▷ y, x)1 ||Ĉ (x := 1.x')1; end 1.x ||Ĥ
    (var 2.x; (z, 2.x := v3, v4 ◁ b ▷ z, x)1 ||Ĉ (x := 2.x')1; end 2.x
  );
f⊥; end c
    
```

After some additional 'massaging':

```

var V‡; sT;
  (
    (y, 1.x := v2, v1 ◁ b ▷ y, x)1 ||Ĉ
    (z, 2.x := v3, v4 ◁ b ▷ z, x)1 ||Ĉ
  ); f⊥; end V
  x := MERGE(x, 1.x', 2.x')
    
```

[‡] V = c, 1.x, 2.x

The same example in second normal form

A sequence of assignments

$$(x \stackrel{:=}{\text{HC}} 3 \triangleleft b \triangleright 10 \parallel y \stackrel{:=}{\text{HC}} 98) \circ x \stackrel{:=}{\text{HC}} 4$$

Generates...

var $c, 0.c, 1.c, 0.x, 1.x, 0.y, 1.y; (c = 0)_{\perp};$

$$\left(\begin{array}{l} (0.x, 0.y, 0.c := (3 \triangleleft b \triangleright 10), 98, 1 \triangleleft c = 0 \triangleright x, y, c)_1 \parallel \hat{c} \\ (1.x, 1.y, 1.c := 4, y, 2 \triangleleft c = 0 \triangleright x, y, c)_1 \parallel \hat{c} \\ c := \text{MERGE}(c, 0.c', 1.c') \parallel \hat{c} \\ x := \text{MERGE}(x, 0.x', 1.x') \parallel \hat{c} \\ y := \text{MERGE}(y, 0.y', 1.y') \end{array} \right);$$

$(c = 2)_{\perp};$ **end** $c, 0.c, 1.c, 0.x, 1.x, 0.y, 1.y$

From our normal form to hardware...

Key ideas

- Control state as flip-flops (synthesised)
 - Controls the clock-based timing
- Variables in RAM modules
 - Allocation of addresses is not defined
- 'Right-hand sides' as combinatorial logic
- Assumptions and assertions are ignored

```
var c, ..., 0.x, 1.x, ; (c = 0)⊥;  
  ((1.x := 4 < c = 0 > x)1 ||Ĉ ... ||Ĉ c := MERGE(c, 0.c', 1.c'));  
(c = 2)⊥; end c, ..., 0.x, 1.x
```

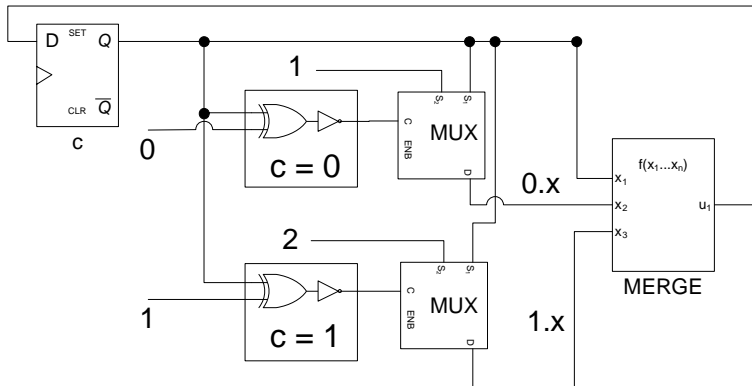
From our normal form to hardware...

Key ideas

- 'i.var' variables as wires
 - i.var' as terminal end for i.var
- MERGE implemented as combinatorial logic
 - n-way MUX
 - Control by chained XORs (difference in values)
- Control loop 'disappears'

```
var c, ..., 0.x, 1.x, ; (c = 0)⊥ ;  
  ((1.x := 4 < c = 0 > x)1 ||Ĉ ... ||Ĉ c := MERGE(c, 0.c', 1.c')) ;  
(c = 2)⊥ ; end c, ..., 0.x, 1.x
```

The control loop disappears?



A bit of the state-of-the-art (again!)

Iyoda's Verilog synthesis

- Similar in the sense it is UTP-based
- No shared variables, parallelism is different
- No recursion
- Time model is different

He Jifeng & J. Bowen's Occam compilation (state based)

- Different time model
- Incomplete (communication informally addressed)
- Not fully consistent with algebraic laws (Hoare & Roscoe)
- No mechanisation

Concluding remarks

Results so far

- Verified part of the commercial solution (wiring)
- UTP denotational semantics for Handel-C
- Normal forms (parallel, priAlt and recursion missing)

Missing results

- Handel-C semantics:
 - Definition of dynamic scope
 - Prove basic axioms about local variables
- Reasoning language:
 - Re-prove some of normal form reduction rules
- Mechanisation (partly done!)

Concluding remarks

Expected contributions

- Denotational semantics for Handel-C
- Denotational model for the compilation in UTP
- Algebraic treatment of
 - Parallelism + shared variables
 - Priorities (non associative)
 - Different kind of communication primitives
 - Multi-synchronisation
- Mechanisation (proof of concept)

Interesting extension lines

- Wire-based encoding
- Hardware optimisations