CSP / FDR2 to Handel-C translation

Susan Stepney

University of York Technical Report YCS-2003-357

June 2003

Contents

1. Introduction

1.1 Background

This report is adapted from a study performed for AWE in 2001 by the author while at Logica. It investigates the feasibility of using the CSP specification language and FDR2 model-checking tool, and Handel-C programming language, in combination, with FDR2 as a front-end specification and proof tool, then automatically translating the formal designs into executable Handel-C. Such an approach could provide a development path from an abstract specification to a correct executable implementation running on an FPGA.

This report contains the following

- Identification of a subset of CSP that maps closely to Handel-C. That subset is large enough to accommodate a large subset of CSP specifications.
- A sketch of a mapping from the identified CSP subset to Handel-C, suitable to be performed automatically. As proof-of-concept, the identified mappings are hand-applied to two example CSP specifications to produce Handel-C implementations.

1.2 The languages of CSP, FDR2 and Handel-C

1.2.1 CSP and FDR2

CSP (Communicating Sequential Processes) was first described in [Hoare78], then in more detail in [Hoare85]. The version that forms the basis of the FDR2 tool is that described in [Roscoe97], which differs only in a minor technical way, on the treatment of *alphabets* (essentially, the events that can occur).[1] Throughout the rest of this report, "CSP" refers to the version in [Roscoe97].

The FDR2 tool model-checks a machine-processable subset of CSP. (When it is important to distinguish the concrete form of the language supported by FDR2 from full CSP, it is referred to as "CSP_M".) FDR2 mechanically checks (proves) whether a proposed implementation is a correct refinement of a more abstract specification.

CSP_M has a properly defined syntax, including ASCII versions of CSP's large collection of unusually-shaped operators. [Roscoe97] provides the link between the CSP and CSP_M syntaxes.

[1] [Hoare85] requires every process P to have a pre-defined alphabet αP, with the parallel operator synchronising on the intersection of alphabets: $P \| Q$ synchronises on $\alpha P \cap \alpha Q$. [Roscoe97] and FDR define the alphabet of interest explicitly on the parallel operator: $P\ {}_X\|_Y\ Q$ synchronises on $X \cap Y$, and $P \|_A Q$ synchronises on the events in A.

A CSP_M program comprises definitions of processes and other objects, a declarative style rather like that of a functional language program.

CSP_M supports multiway input and output channels (one event can simultaneously comprise inputs and outputs, at multiple ends). For example, **`a.2?x!y`** is a communication over channel **`a`** of three components: the first is the value **`2`**, the second is an input and will bind to **`x`**, and the third is the output **`y`**. It is equivalent to **`a!2?x!y`**

1.2.2 Handel-C

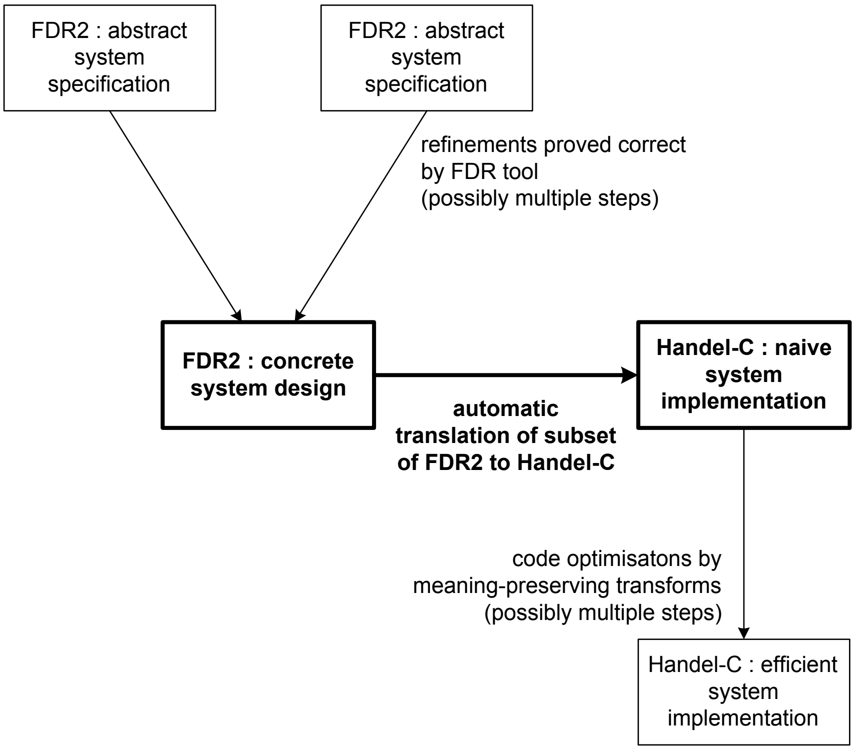
Handel-C is an implementation language, targeting FPGAs, with a semantics based on CSP. It is a procedural style language, rather like occam, but with a C-like concrete syntax. It has a parallel construct PAR, channel communication, and an ALT construct.

The language supports single input or single output point-to-point channels (one communication event comprises an output from one end and an input at the other end).

2. Overview and scope

2.1 Development process structure

The envisioned full development process, from abstract CSP safety specifications down to their efficient Handel-C implementation, has the following components and transformations.


FDR2 : abstract system specification
FDR2 : abstract system specification
refinements proved correct by FDR tool (possibly multiple steps)
FDR2 : concrete system design
automatic translation of subset of FDR2 to Handel-C
Handel-C : naive system implementation
code optimisatons by meaning-preserving transforms (possibly multiple steps)
Handel-C : efficient system implementation

- **Abstract system specification(s)** – The high level CSP specification(s) of the system, which state the desired properties of the system – probably relatively abstractly as safety properties.
- **Concrete system design** – The concrete CSP design specification, close to an executable implementation, is proved a correct refinement of the abstract specification by using the model checking facilities of the FDR2 tool.
    - There may need to be multiple refinement steps to move from specification to design, each proved using the tool
    - If the specification has gone through at least one proof step, assumptions can be made about it that are stronger than merely “syntactically correct” – what are these assumptions? are they helpful for translation?
    - The design specification will need to be restricted to a subset of the CSP_M language, to be translatable to Handel-C.

- **Naive implementation** – The concrete CSP design specification is translated automatically into an abstract Handel-C implementation. This is runnable, but may not be very efficient.
- **Efficient implementation** – The naive Handel-C implementation is optimised, by meaning-preserving transforms, into a suitable efficient Handel-C implementation.
    - There is a trade-off between doing "optimisations" in FDR2 with existing tool support, or as meaning-preserving Handel-C transformations, requiring some tool support to be built.

2.2 Scope of this study

This study focuses on the **translation step** from the CSP concrete design specfciation to the Handel-C naive implementation. It considers the following

- the constraints on the CSP_M language to make it translatable
- the feasibility of automatic translation – any need for "translation directives"

3. Translator from CSP to Handel-C

The approach taken is to identify a subset of CSP_M that can be translated directly to Handel-C. The FDR2 tool should be used to refine the specification into this subset language.

The translations described below follow the structure of the CSP_M syntax given in [FDR2, Appendix A]. The syntax of CSP_M is described only informally in [FDR2]. CSP_M specification fragments are written in **`bold typewriter font`**. The BNF syntax of Handel-C is given in [Handel-C]. Handel-C code fragments are written in `typewriter font`.

The semantics of each language is described only sketchily. The translations suggested below have been tested only to a limited degree: a more in-depth exploration is needed before a translator can be built.

3.1 Recursion

The main point to note is that CSP definitions tend to be recursive. They are often process definitions of the form “process P does this, then that, then behaves like P”. Handel-C does not support recursion.

3.1.1 Tail recursion

Fortunately, most of the recursive definitions in actual use are tail recursive, and so can easily be turned into loops.

For example, consider a typical CSP process definition like:

```
P = in ? x -> out ! sq -> P
```

If recursion were available in Handel-C, this could be naively translated as

```
void P() {
  in ? x;
  out ! sq;
  P();
}
```

This is not possible. However, because the definition is tail recursive, it can be translated as

```
void P() {
  while (1) {
    in ? x;
    out ! sq
  }
}
```

3.1.2 Parameterised recursion

Recursive processes can be parameterised. For example (ignoring overflow for now)

```
COUNT(n) = in -> COUNT(n+1)
```

If recursion were available in Handel-C, this could be naively translated as

```
void COUNT(int n) {
  in ? null;
  COUNT(n+1);
}
```

Using tail recursion and a local variable, it can be translated as (ignoring where the initial value comes from for now):

```
void COUNT() {
  int n = 0;
  while (1) {
    in ? null;
    n = n+1;
  }
}
```

3.1.3 Exponential recursion

Not all recursive processes are tail recursive. For example

```
P(n) = if n == 0 then a -> STOP else P(n-1) ||| P(n-1)
```

If recursion were available in Handel-C, this could be naively translated as

```
void P(int n) {
   if (n == 0) {
     a ? null;
     STOP();
   } else {
     par { P(n-1); P(n-1); }
   }
}
```

This could in principle be translated by unfolding the recursion and maybe using a replicated `par`. But such a step is probably too large to do in a translation. Better would be to do the unfolding within FDR2 and prove it correct, and then do a simple translation.

3.1.4 Summary

Translation of tail recursive processes is straightforward and can be translated as the equivalent loop. Non-tail recursive processes should be refined within FDR2, and then translated.

3.2 Channels

3.2.1 Channel communications

3.2.1.1 Simple channels

Channels are key in both CSP and Handel-C. The mapping is straightforward for single item communications

```
chan ? var     --->   chan ? var
chan ! expr    --->   chan ! expr
```

3.2.1.2 Structured channels

Where the CSP channels have dotted types, a Handel-C `struct` can be used to capture the data structure passed across the channel. A data structure is defined. The channel is declared to have this type. Local variables of this type might also be needed, to assign values to the struct before outputting it.

For example,

```
channel pipe1  : BYTE.BYTE.OPCODE.BYTE.REG
... pipe1!x.y.a.i.d ...
... pipe1?v.w.add.i.d ...

  --->

struct pipe1_DATA {
  BYTE byte1;
  BYTE byte2;
  enum OPCODE opcode1;
  BYTE byte3;
  REG reg1;
} ;
chan struct pipe1_DATA pipe1 ;

struct pipe1_DATA pipe11;
pipe11.byte1 = opcode_set1.byte1 ;
pipe11.byte2 = opcode_set1.byte2 ;
pipe11.opcode1 = opcode_set1.opcode1 ;
pipe11.byte3 = byte1 ;
pipe11.reg1 = reg2 ;
pipe1!pipe11 ;

pipe1?pipe11 ;
```

3.2.1.3 Constrained input

CSP also has constrained input, `c?x:a`. Before translation to Handel-C, the CSP specification must be refined to one with only unconstrained inputs.

3.2.1.4 Combined input and output

CSP also allows mixed input and output, `c?x!y`. Before translation to Handel-C, the CSP specification must be refined to one where each communication is purely input or purely output.

3.2.2 Hiding in CSP

CSP declares all the channels used globally. The channels actually used by a process are determined by examining the process. If a compound process composes two subprocesses on a channel, then hides it, the compound process does not have that channel visible. Consider

```
CHANNEL in, mid, out : INTEGER
LEFT = in ? x -> mid ! x -> LEFT
RIGHT = mid ? x -> out ! x -> RIGHT
BUFFER = ( LEFT [| {|mid|} |] RIGHT ) \ {| mid |}
```

The three channels are globally declared. Examining the processes, we see that `LEFT` has `in` and `mid` visible; `RIGHT` has `mid` and `out` visible, and `BUFFER` had `in` and `out` visible.

3.2.2.1 Parameterised and local Handel-C channels

If we want to replicate this behaviour in Handel-C, the channels visible to a process could be passed as parameters, and the internal ones could be hidden by being declared locally.

```
chan int in, out ;
void LEFT(chan int in, chan int mid) {
  int x ;
  while (1) { in ? x ; mid ! x ; }
}
void RIGHT(chan int mid, chan int out) {
  int x ;
  while (1) { mid ? x ; out ! x ; }
}
void BUFFER(chan int in, chan int out) {
  chan int mid ;
  par { LEFT(in, mid) ; RIGHT(mid, out) ; }
}
```

Although this seems to give good correspondence with the CSP, it is actually quite hard to work out from the CSP specification which channels are truly global and which are intended to be local. So we do not follow this approach.

3.2.2.2 Global Handel-C channels

Alternatively the channels could be left global, and the hiding ignored. (This is the approach adopted.)

```
chan int in, mid, out ;
void LEFT() {
  int x ;
  while (1) { in ? x ; mid ! x ; }
}
void RIGHT() {
  int x ;
  while (1) { mid ? x ; out ! x ; }
}
```

```
void BUFFER() {
  par { LEFT() ; RIGHT() ; }
}
```

3.2.3 Input and output channels

In CSP the visible channels are all those that have not been hidden. In Handel-C they are those that have only one end, and are explicitly declared to be input or output channels. A translation directive is needed to mark the channels intended for input or output.

```
--!! channel in Name
channel Name : Type

   --->

chanin Type Name ;
```

```
--!! channel out Name
channel Name : Type

   --->

chanout Type Name ;
```

3.2.4 Indexed array of channels

Consider the channels used in the Dining Philosopher example (later).

```
channel pickupleft:{0..4}.Bool
```

This appears to be communicating a structured value (comprising an integer and a boolean) on the channel **`pickupleft`**, and with many processes communicating separate values on this one channel.

```
struct pickupleft_struct {
  Int s1 ;
  Bool s2 ;
}
chan pickupleft_struct pickupleft;
```

But this is actually being used as an *array* of channels, indexed by the first value, with an array of processes, also indexed, each communicating on separate elements of this array. This instead translates to Handel-C as

```
chan Bool pickupleft[5];
```

A translation directive is needed to say when the first parameter of a channel should be interpreted as an index, and the range of that index.

```
--!! channel array pickupleft index 5
```

3.2.5 Pure event channels

In CSP an event is the channel and any values communicated across it. There need be no actual communicated value, the processes can merely synchronise on a bare event.

In Handel-C a communication always involves a value, and a direction.

The translation provides a dummy SYNC value for use in these cases. A translation directive can be used to provide the (arbitrary) direction, if required.

For example

```
channel c

--!! sync in c
P = c -> P
--!! sync out c
Q = c -> Q

R = P |[ c ]| Q

   --->

typedef unsigned int 1 SYNC ;
const SYNC syncout = 0 ;
SYNC syncin ;

chan SYNC c ;

void P() { while (1) { c?syncin ; } }
void Q() { while (1) { c!syncout ; } }
void R() { par { P() ; Q() ; } }
```

3.3 Expressions

3.3.1 Identifiers

A CSP_M identifier starts with a letter, followed by any number of letters, digits and underscore characters, followed by any number of prime characters. Identifiers with trailing underscores are reserved for machine-generated code, and so will not occur in user scripts. [Note: this assumes the “trailing underscores” occur before the primes, and that addition of a prime does not affect the rule on trailing underscores.]

```
csp_id ::= letter [ { letter | digit | _ } ( letter | digit ) ] { ʻ }
```

Identifiers in Handel-C may start and end with an underscore.

```
hc_id ::= ( letter | _ ) [ { letter | digit | _ } ]
```

Letters, digits and underscores are translated unchanged, and final primes are translated as final underscores. Any resulting name that conflicts with a Handel-C keyword gets an initial underscore.

```
letter | digit | _    --->   letter | digit | _

                ‘    --->   _

          <keyword>   --->   _<keyword>
```

3.3.2 Numbers

CSP_M integer literals are written only in decimal format, so can be translated directly.

```
number   --->   number
```

All the CSP_M arithmetic operators have the same syntax, associativity and meaning as C arithmetic integer operations, and so translate directly to the Handel-C equivalents.

3.3.3 Sequences

CSP_M has a sequence data type. This includes sequence literals, sequence comprehensions, and some elementary operations for manipulating sequences.

It would be possible to build in some support for sequence literals and the operations as a library. Sequence comprehensions would be much more difficult. So, at least initially, we require that before translation to Handel-C, the CSP specification must be refined into one with no occurrences of sequences.

3.3.4 Sets

CSP_M has a set data type. This includes set literals, set comprehensions, and some operations for manipulating sets.

It would be possible to build in some support for set literals and the operations as a library. Set comprehensions would be much more difficult. So, at least initially, we require that before translation to Handel-C, the CSP specification must be refined into one with no occurrences of sets.

3.3.5 Booleans

CSP_M's boolean corresponds to Handel-C's `int 1`, with **`false`** represented as 0, and **`true`** as 1.

```
false   --->   0
true    --->   1
```

The CSP_M boolean operators translate directly to their Handel-C counterparts:

```
and    --->   &&
or     --->   ||
not    --->   !
==     --->   ==
!=     --->   !=
<     --->   <                 and similarly for other comparisons
```

```
  if b then e1 else e2   --->  if ( b ) { e1 } else { e2 }
```

3.3.6 Tuples

CSP_M has a tuple data type.

It would be possible to build in some support for tuples as `structs`. But, at least initially, we require that before translation to Handel-C, the CSP specification must be refined into one with no occurrences of tuples.

3.3.7 Local definitions

CSP_M supports local definitions with a **`let ... within`** clause.

In Handel-C, functions cannot be defined within other functions. It would be possible to provide some support for local declarations, by making them global with a suitable unique renaming. But, at least initially, we require that before translation to Handel-C, the CSP specification must be refined into one with no occurrences of local definitions.

3.3.8 Lambda terms

CSP_M supports lambda terms (anonymous functions).

In Handel-C, functions cannot be anonymous. It would be possible to provide some support for lambda terms, by converting them to functions and providing a suitable unique name. But, at least initially, we require that before translation to Handel-C, the CSP specification must be refined into one with no occurrences of lambda terms.

3.4 Pattern Matching

CSP_M supports pattern matching in defining its functions like a full functional programming language.

Handel-C is a declarative language, with no support for pattern matching. Supporting pattern matching in a translator is a task equivalent to writing a functional language compiler. So we require that before translation to Handel-C, the CSP specification must be refined into one with no occurrences of pattern matching.

3.4.1 Functions

Once CSP_M function definitions have been refined to have no occurrence of pattern matching, they can be translated into the obvious Handel-C equivalent function. Translation directives communicate the parameter types and return type. The function may need to be inlined if it is used in multiple processes (indicated by a translation directive).

For example

```
--!! function inline int dec(int x)
dec(x) = (x - 1) % 5

  --->

inline int dec(int x) { return( (x - 1) % 5); }
```

Parameterless functions are translated as constants.

3.4.2 Processes

Process definitions, with or without parameters, are translated as procedures, that is, `void` functions (see later). Translation directives communicate the parameter types.

3.5 Types

3.5.1 Simple types

CSP_M has two simple types, **`Int`** and **`Bool`**.

`Int` is the set of 32 bit signed integers. This corresponds to Handel-C's `int 32`, or, equivalently, `long`. The translation to the explicit `int 32` is to be preferred, as it emphasises the possibility of reducing the variable's width. To make the translation more transparent, we define the Handel-C name `Int`.

```
typedef unsigned int 32 Int ;

Int   --->  Int
```

`Bool` is the set of Boolean values **`true`** and **`false`**. This corresponds to Handel-C's `int 1`, with **`false`** represented as 0, and **`true`** as 1. To make the translation more transparent, we define the Handel-C name `Bool`.

```
typedef unsigned int 1 Bool ;

Bool   --->  Bool
```

CSP_M can use various expressions in type expressions. The only one not already required to be refined away is the dot operation, which denotes a composite type. This can be translated as a Handel-C `struct`.

```
T1.T2. ... Tn

  --->

struct <typename> {
  T1 s1 ;
  T2 s2 ;
  ...
  Tn sn ;
}

<typename>
```

3.5.2 Named types

CSP_M allows types to be named. A similar effect can be achieved in Handel-C using a `typedef`

```
nametype Name = Type   --->  typedef Type Name ;
```

For example (remember in this example that CSP integer ranges are uniformly translated to `Int`s)

```
nametype Values = {0..199}   --->  typedef Int Values ;

nametype Ranges = Values . Values

   --->

struct Ranges_struct {
  Values s1 ;
  Values s2 ;
}
typedef Ranges_struct Ranges ;
```

3.5.3 Data types

CSP_M's simple datatypes can be translated to Handel-C `enum`s.

```
datatype Name = t1 | t2 | ... | tn

   --->

enum Name { t1 , t2 , ... , tn } ;
```

For example

```
datatype SimpleColour = Red | Green | Blue

   --->

enum SimpleColour { Red , Green , Blue } ;
```

CSP_M datatypes may also have values associated with the tags. This would correspond to a `union` type in C, but Handel-C does not support the `union` construct. So we require that before translation to Handel-C, the CSP specification must be refined into one with only simple datatypes.

3.5.4 Subtypes

CSP_M Integer variables restricted to subranges and subtypes could possibly, with some calculation, be used to specify narrower Handel-C types. But the resulting mixed width Handel-C variables would probably require considerable casting to be usable together, and so this approach is not recommended. Rather, some form of assertion mechanism, checking that the Handel-C variables stay within their CSP_M ranges, might be tried.

3.5.5 Closure operations

The main use for CSP_M closure operations is in writing the communication sets for parallel operators. Since these sets are refined away (usually to the full event set) before translation into Handel-C, no specific translation of them is required.

3.6 Processes

3.6.1 Primitive processes

3.6.1.1 SKIP

The CSP process **`SKIP`** does nothing except terminate successfully.

The obvious way to implement this in Handel-C is for the process to do nothing except return. `delay` is the Handel-C statement that takes one clock cycle to do nothing. (For technical Handel-C reasons we want the process to take at least one clock cycle.)

```
void SKIP() { delay; }
```

```
SKIP   --->   SKIP();
```

3.6.1.2 STOP

The CSP process **`STOP`** does nothing. It never communicates, and never terminates. It is deadlocked. Although the entire process should not deadlock, it is possible that branches of it may, whilst other branches continue executing.

The obvious way to implement this in Handel-C is for the process to do nothing, not even return.

```
void STOP() { while (1) { delay; } }
```

```
STOP   --->   STOP();
```

In a true parallel implementation, the deadlocked branch sits in a loop doing nothing. In a simulated environment, there is a danger that the entire process will get caught in this busy loop, and livelock. We assume the Handel-C simulator does not behave this way.

3.6.1.3 CHAOS

The CSP process **`CHAOS(a)`** non-deterministically either stops, or engages in any event from the set **`a`** then behaves like **`CHAOS(a)`**. So it can engage in any event in **`a`**, any number of times, and just stop at any time. It never terminates.

If we choose to allow the translation into Handel-C to include refinement, there is a particularly simple implementation of **`CHAOS`**, that, independently of **`a`**, resolves all the non-determinism to **`STOP`**.

```
void CHAOS() { STOP(); }
```

```
CHAOS   --->   CHAOS();
```

3.6.2 Sequential composition

3.6.2.1 Prefix

Channels are the basis of events – the values communicated over them form the rest of the event.

3.6.2.2 Simple sequential

With sequential composition, **P ; Q**, the two processes proceed in sequence, first **P** until it terminates, then **Q**.

Handel-C provides two syntaxes for sequential composition: the conventional C semi-colon, and the `seq` keyword. To allow for nested block structure, and to highlight the connection between the CSP and the Handel-C, the `seq` keyword is preferred.

```
P ; Q ... ; R   --->   seq { P(); Q(); ...; R(); }
```

Multi-statement Handel-C translations of complex processes should be enclosed in the usual braces. Such braces are elided here, for clarity.

3.6.2.3 Replicated sequential

Because Handel-C replicators require a numerical index, the CSP specification must be refined to one using only “closed range” numerical replicators (using say an array of events).

With numerical replicated sequential, **; n:{i..j} @ P(n)**, the processes indexed by **n** proceed sequentially.

```
; n:{i..j} @ P(n)   --->   seq (n=i; n<=j; ++n) { P(n); }
```

3.6.3 Choice

3.6.3.1 External choice

With external environmental choice, **P [] Q**, the process offers the environment the choice of the first event of **P** and the first event of **Q**. If the first event chosen is that from **P**, then **P [] Q** behaves like **P**; if the first event chosen is that from **Q**, then **P [] Q** behaves like **Q**. If the first events from **P** and **Q** are the same, then **P [] Q** nondeterministically behaves like **P** or like **Q** .

Handel-C’s `PRIALT` construct implements simple external choice on channels (input or output channels, but no Boolean guards). The channels must all be different. The first channel in the list whose other end is ready to communicate is chosen. If no channel is ready to communicate and there is no `default` branch, the entire `PRIALT` statement waits until a channel becomes ready.

Before translation to Handel-C, the CSP specification must be refined into one where each process in an external choice is written as a prefix process (that is, must be written as **`a -> P`**), and with no occurrences of repeated channels in the external choices.

In particular, there must be no Boolean guards. These can be transformed away in CSP as

```
(ba & a -> P) [] (b -> Q) ... [] (c -> R)

   --->

if ba then (a -> P) [] (b -> Q) ... [] (c -> R)
      else (b -> Q) ... [] (c -> R)
```

The suitably reduced CSP construct has the translation

```
(a -> P) [] (b -> Q) ... [] (c -> R)

   --->

prialt {
  case a : P(); break;
  case b : Q(); break;
  ...
  case c : R(); break;
}
```

3.6.3.2 Internal choice

With internal nondeterministic choice, **`P |~| Q`**, the system non-deterministically behaves like either **`P`** or **`Q`**.

If we choose to allow the translation into Handel-C to include refinement, there is a particularly simple implementation of nondeterministic choice that resolves the determinism in favour of **`P`**, or maybe of the "easiest" process to translate (say **`STOP`** or **`SKIP`** if they were present).

```
P |~| Q ... |~| R   --->   P();
```

This suggests that a CSP specification containing non-deterministic choice may be at too high a level of abstraction to be realistically translated into an implementation. The translation could proceed as above, but with a warning.

3.6.3.3 Replicated choice

Handel-C has no replicated `ALT`. Before translation to Handel-C, the CSP specification must be refined into one with all occurrences of replicated choice unwound to an explicit list of choices.

3.6.3.4 Untimed timeout

The untimed time-out process, `P [> Q`, begins by offering the option of `P` before it opts to behave like `Q`.

```
P [> Q  ==  (P |~| STOP) [] Q
```

If we resolve the nondeterminism of the internal choice always to pick the `STOP` branch (equivalent to always timing out before there is a chance to engage in `P`), and then use the identity `STOP [] Q == Q`, this can be translated as `Q`.

```
P [> Q   --->   Q();
```

This suggests that an CSP specification containing untimed timeout may be at too high a level of abstraction to be realistically translated into an implementation. The translation could proceed as above, but with a warning.

3.6.3.5 Conditional choice

The conditional choice process, `if b then P else Q`, behaves like `P` if `b` is true, otherwise it behaves like `Q`.

```
if b then P else Q   --->   if (b) then { P(); } else { Q(); }
```

CSP_M also has a shorthand form

```
b & P  ==  if b then P else STOP
```

which can be used to guard external choice in CSP specifications. However, such a guard cannot be reduced to the prefix form required before translation into Handel-C (Handel-C does not allow Boolean guards in `PRIALT`s).

3.6.4 Parallel

In CSP, processes can communicate by synchronising on events. Multiple processes in parallel can synchronise on a shared event. Processes can be combined in parallel in several ways that specify whether they do, or do not, synchronise on a potentially shared events.

In Handel-C, this synchronisation communication is implemented by channels. The flexibility of the CSP events is greatly reduced. A Handel-C channel is a *unidirectional point-to-point* (two ended) link from one process to one other process. To use a channel there must be precisely two processes, running in parallel, one inputting and one outputting. The following are illegal:

1. two processes in parallel, each inputting or each outputting on the same channel
2. one process both inputting and outputting on a channel

These constraints disallow more complex cases. For example, consider trying to implement a CSP specification of three processes in parallel synchronising on an event. Since in Handel-C every synchronisation event looks like either an input or an output, at least two of these three processes must be either inputting or outputting. But this is illegal in Handel-C.

There is one parallel construct in Handel-C, the `par` statement. Processes execute in parallel, and synchronise on any shared channels. Hence processes within a `par` block must obey the rules for shared channel usage.

3.6.4.1 Sharing parallel

With sharing, or synchronised, parallel, **`P [| a |] Q`**, the two processes proceed in parallel, synchronising on the events in **`a`**. So if both **`P`** and **`Q`** share an event in **`a`**, they synchronise. If both **`P`** and **`Q`** share an event not in **`a`**, they do not synchronise with each other; they may synchronise with another parallel process, or with the environment.

Provided that **`a`** is big enough to contain all the events common to **`P`** and **`Q`**, synchronised parallel translates directly to the Handel-C `par` construct. Such a set of events that *is* big enough is Σ (CSP) or **`Events`** (CSP_M), the set of all possible events (the entire alphabet). Before translation to Handel-C, the CSP specification must be refined into one with all occurrences of synchronised parallel referencing only the set of all events.

```
P [| Events |] Q ... [| Events |] R

   --->

par { P(); Q(); ...; R(); }
```

3.6.4.2 Alphabetised parallel

With alphabetised parallel, **`P [ a || b ] Q`**, the two processes run in parallel, **`P`** engaging only in events in **`a`**, **`Q`** engaging only in events in **`b`**, and synchronising on the events in the intersection of **`a`** and **`b`**.

Provided **`P`** and **`Q`** never communicate outside **`a`** and **`b`** respectively, alphabetised parallel reduces to synchronised parallel of the intersection:

```
P [ a || b ] Q  ==  P [| a ∩ b |] Q
```

Before translation to Handel-C, the CSP specification must be refined into one with all occurrences of (replicated) alphabetised parallel replaced by (replicated) sharing parallel.

3.6.4.3 Interleaving parallel

With interleaving parallel, **`P ||| Q`**, the two processes run independently of each other, with no synchronised communication. Any event in the parallel composition arises in only one of **`P`** or **`Q`**. So if both **`P`** and **`Q`** can engage in an event in **`a`**, only one of them does, synchronising with another parallel process, or with the environment.

So interleaving is equivalent a synchronised parallel composition synchronised on the empty set.

```
P ||| Q  ==  P [| {| |} |] Q
```

Since in Handel-C, all channels are unidirectional and point to point, an interleaving composition where there are potentially shared events is illegal. An interleaving where there are no potentially shared events reduces to a synchronised parallel (because in this case the synchronisation set is already empty).

Before translation to Handel-C, the CSP specification must be refined into one with all occurrences of (replicated) interleaving parallel replaced by (replicated) sharing parallel.

3.6.4.4 Replicated sharing parallel

The only replicated parallel we consider here is replicated sharing parallel, since the other replicated parallels must be refined to this before translation. In addition, because Handel-C replicators require a numerical index, the CSP specification must be refined to one using only "closed range" numerical replicators (using say an array of events).

With numerical replicated synchronised parallel, `[| a |] n : {i..j} @ P(n)`, the processes indexed by `n` proceed in parallel, synchronising on the events in `a`.

Before translation to Handel-C, the CSP specification must be refined into one with all occurrences of replicated synchronised parallel referencing only the set of all events.

```
[| Events |] n : {i..j} @ P(n)

   --->

par (n=i; n<=j; ++n) { P(n); }
```

3.6.5 Channel names and visibility

3.6.5.1 Hiding

In CSP, channels can be hidden, to make them unavailable to participate in further events. In Handel-C, channels are unidirectional and point to point. Any channel engaged in a communication is unavailable for further communication, and so is implicitly hidden.

Once the CSP specification has been refined to one in which all channels are used in this Handel-C manner, there is no need to translate the hidings explicitly.

3.6.5.2 Renaming

In CSP a process's channels can be renamed.

The translation of channels into Handel-C does not explicitly link them with a process. Processes are not parameterised by thier channels. Hence there is no simple way to translate renaming.

Before translation to Handel-C, the CSP specification must be refined into one with all occurrences of renaming removed.

3.6.5.3 Linked parallel

With linked parallel **`P [a1 <-> b1, ..., an <-> bn] Q`**, the two processes are in parallel. Channel **`ai`** of **`P`** and channel **`bi`** of **`Q`** are joined together, and hidden. This provides a more sophisticated form of "chaining". It can be defined in terms of renaming and hiding.

```
P [a1 <-> b1, ..., an <-> bn] Q  ==
  ( P [[a1 <- temp1, ..., an <- tempn]]
    |[ {| temp1, ..., tempn |} |]
    Q [[b1 <- temp1, ..., bn <- tempn]]
  )
    \ {| temp1, ..., tempn |}
```

Before translation to Handel-C, the CSP specification must be refined into one with all occurrences of linked parallel replaced by sharing parallel.

3.6.6 Interrupt

The interrupt process, **`P /\ Q`**, begins by behaving like **`P`** and then behaves like **`Q`**.

There is no support for interrupts in Handel-C. Before translation to Handel-C, the CSP specification must be refined into one without any interrupted processes.

3.7 Others

3.7.1 Operator precedence

In a translation, the module that parses the CSP_M and the module that outputs the Handel-C must be aware of their respective languages' precedences. The translation module operates at the syntax tree level (manipulating already-parsed forms), and so it does not need to worry about precedences. Any necessary bracketing will be the concern of the Handel-C output.

3.7.2 External

The **`external`** keyword is provided by CSP_M to allow the use of functions defined by other tools. Handel-C does not use such functions, and so no translation support is required.

3.7.3 Transparent

The **transparent** keyword is provided by CSP_M to allow the use of certain semantically neutral optimisations. The functions introduced have the semantics of the identity function. Hence the translator should translate any transparent names to the identity function (that is, remove them) wherever they occur.

```
transparent Name
T1 Name T2

   --->

T1 T2
```

For example

```
transparent diamond, normalise
squidge(P) = normalise(diamond(P))

   --->

squidge(P) = P
```

3.7.4 Assert

The **assert** keyword is provided by CSP_M to state properties to be proved by the model checker. There is no need to translate these **assert** statements.

3.7.5 Print

The **print** keyword is provided by CSP_M to enable certain expressions to be evaluated and examined with the tool. There is no need to translate these **print** statements.

3.7.6 Comments

CSP_M end of line comments start with **--** and continue to the end of the line. Block comments start with **{-** and continue to the matching **-}** (that is, they can be nested).

Handel-C end of line comments start with `//` and continue to the end of the line. Block comments start with `/*` and continue to the first `*/` (that is, they may not be nested).

So CSP_M comments can be translated directly to Handel-C comments (to help document the generated code), provided any nested closing block comment markers are stripped.

3.8 Summary

Quite a large subset of CSP_M can be mapped to Handel-C. Of the remaining language, much of it can readily be refined into a translatable form.

4. Translator from Handel-C to CSP

The translations mentioned in the previous section can often be applied in reverse, to map Handel-C into CSP_M, to allow it to be analysed.

Handel-C that has been written in the style of the target of the translator can be translated back into CSP_M. There are some features of Handel C that need care.

- Procedures need to be written as infinite `while` loops, to be translated into tail recursive processes.
- No translation for signals has been determined.
- Hardware specific features (`ram`, `rom`, etc) should be ignored.

More work is needed to determine the appropriate sub-language and style for translation.

5. Dining Philosophers Example

5.1 Original specification

This example is taken from
http://www.cs.bris.ac.uk/Teaching/Resources/COMS40204/Exercise/Files/dphil_butler.fdr2

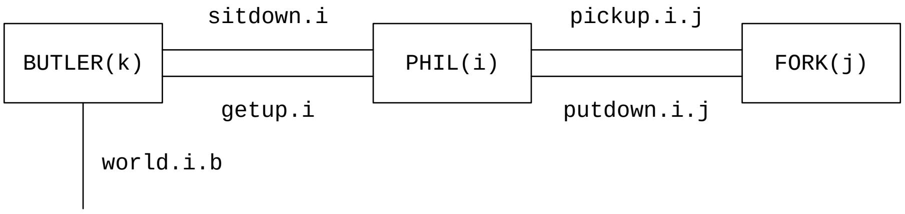
```
-- The dining philosophers with a butler.
--
-- Simon Gay, Royal Holloway, January 1999
--
channel pickup:{0..4}.{0..4}
channel putdown:{0..4}.{0..4}
channel sitdown:{0..4}
channel getup:{0..4}

inc(x) = (x + 1) % 5
dec(x) = (x - 1) % 5

PHIL(i) =  sitdown.i -> pickup.i.inc(i) -> pickup.i.i ->
           putdown.i.inc(i) -> putdown.i.i -> getup.i -> PHIL(i)

FORK(i) = pickup.i.i -> putdown.i.i -> FORK(i)
        [] pickup.dec(i).i -> putdown.dec(i).i -> FORK(i)

PHILS = || i:{0..4} @ [{|pickup.i.i, pickup.i.inc(i),
                         putdown.i.i, putdown.i.inc(i),
                         sitdown.i, getup.i|}]
                      PHIL(i)

FORKS = || i:{0..4} @ [{|pickup.i.i, putdown.i.i,
                         pickup.dec(i).i, putdown.dec(i).i|}]
                      FORK(i)

DINNER = PHILS [ {|pickup,putdown,sitdown,getup|} ||
                 {|pickup,putdown|} ]  FORKS

BUTLER(i) = if i == 0
            then sitdown?x -> BUTLER(1)
            else if i == 4
                 then getup?y -> BUTLER(3)
                 else (  sitdown?x -> BUTLER(i+1)
                      [] getup?y -> BUTLER(i-1) )

NEWDINNER = DINNER [ {|pickup,putdown,sitdown,getup|} ||
                     {|sitdown,getup|} ] BUTLER(0)
```

5.2 Introducing external channels

This has all the events “visible”. In a Handel-C implementation, all channels are point-to-point, and the only visible communications are those that take place with the outside world. So we modify this specification by adding a new channel, **`world`**, that the **`BUTLER`** process uses to communicate with the world whenever a **`PHIL`** sits down or gets up. All the other channels are hidden in the **`NEWDINNER`** process.

```
-- (0) explicit external channel

channel pickup:{0..4}.{0..4}
channel putdown:{0..4}.{0..4}
channel sitdown:{0..4}
channel getup:{0..4}
channel world_sit:{0..4}
channel world_up:{0..4}

inc(x) = (x + 1) % 5
dec(x) = (x - 1) % 5

PHIL(i) =  sitdown.i -> pickup.i.inc(i) -> pickup.i.i ->
           putdown.i.inc(i) -> putdown.i.i -> getup.i -> PHIL(i)

FORK(i) = pickup.i.i -> putdown.i.i -> FORK(i)
        [] pickup.dec(i).i -> putdown.dec(i).i -> FORK(i)

PHILS = || i:{0..4} @ [{|pickup.i.i, pickup.i.inc(i),
                         putdown.i.i, putdown.i.inc(i),
                         sitdown.i, getup.i|}]
                    PHIL(i)

FORKS = || i:{0..4} @ [{|pickup.i.i, putdown.i.i,
                         pickup.dec(i).i, putdown.dec(i).i|}]
                    FORK(i)

DINNER = PHILS [ {|pickup,putdown,sitdown,getup|} ||
                 {|pickup,putdown|} ]  FORKS

BUTLER(i) = if i == 0
            then sitdown?x -> world_sit!x -> BUTLER(1)
            else if i == 4
                 then getup?y -> world_up!y -> BUTLER(3)
                 else (  sitdown?x -> world_sit!x -> BUTLER(i+1)
                      [] getup?y -> world_up!y -> BUTLER(i-1) )

NEWDINNER = (DINNER [ {|pickup,putdown,sitdown,getup|} ||
                      {|sitdown,getup|} ] BUTLER(0) )
        \ {|pickup,putdown,sitdown,getup |}
```

A diagram of the processes and channels helps shows how these are arranged.

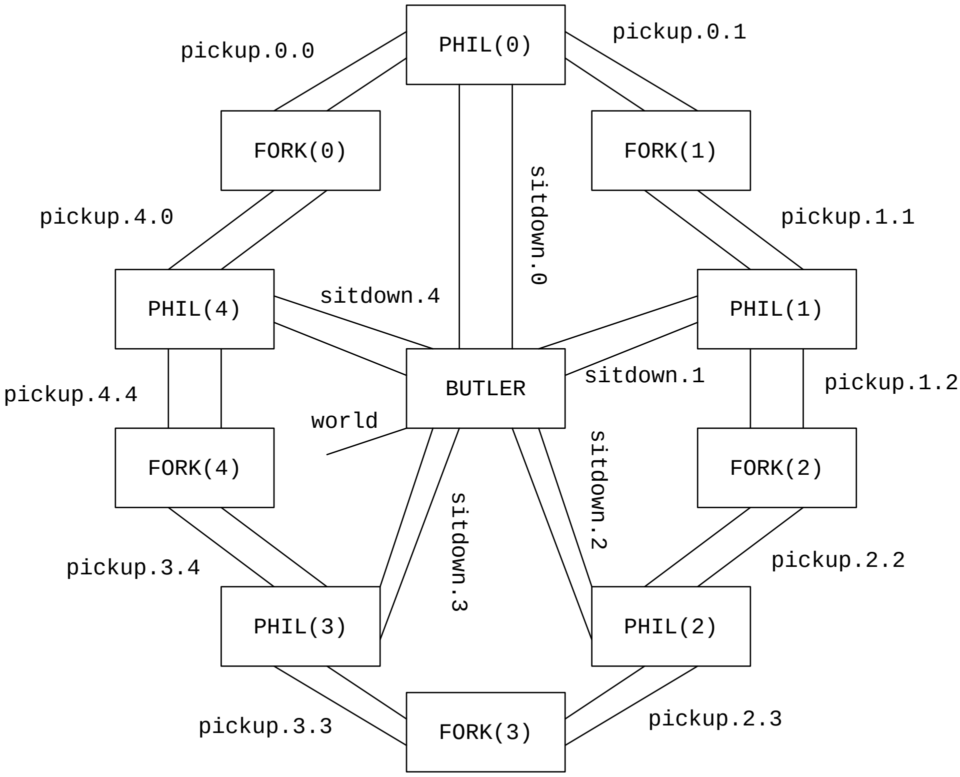


The **pickup** channels and **putdown** channels communicate two values: the philosopher doing the picking up, and the fork being picked up. Many values possible from the type are not actually communicated, such as pickup.3.0. A naïve implementation as a 2-D array of channels would result in 5×5 = 25 channels, rather than the 10 that are ever used.

The **sitdown** channels and **getup** channels communicate one value: the philosopher doing the sitting down and getting up.

The **world_sit** channel and the **world_up** channel communicate one value: the philosopher involved in sitting down and getting up respectively.

Expanding the arrays, we see the processes have the following communication structure. (The **pickup** channels have similar **putdown** channels; the **sitdown** channels have similar **getup** channels.)



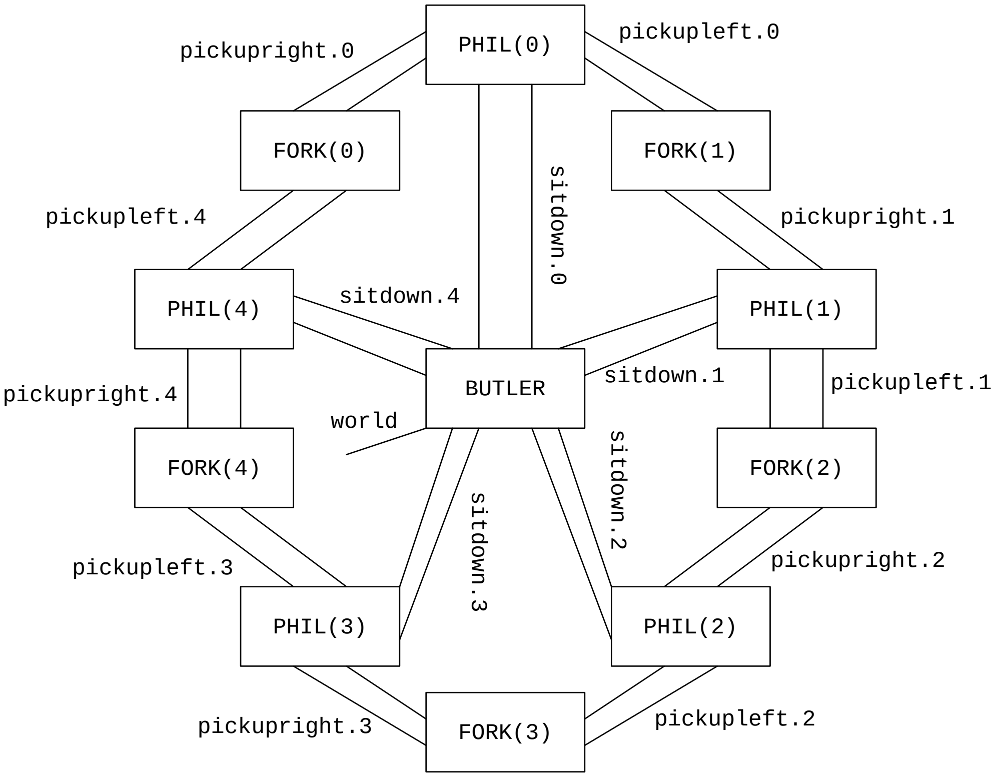
5.3 Left and right pickup channels

Let us simplify the example to use the 10 **pickup** channels explicitly. These channels are labelled with the philosopher, and indicate whether the fork on the left (higher index) or on the right (same index) is being picked up. (Note that from the fork's point of view, the **pickupleft** channel communicates that it has been picked up by the left hand of the philosopher, who is on the fork's right.)

```
-- The dining philosophers with a butler.
--
-- (1) simpler channel structure

channel pickupleft:{0..4}
channel pickupright:{0..4}
channel putdownleft:{0..4}
channel putdownright:{0..4}
channel world_sit:{0..4}
channel world_up:{0..4}

PHIL_1(i) =  sitdown.i -> pickupleft.i -> pickupright.i ->
           putdownleft.i -> putdownright.i -> getup.i -> PHIL_1(i)

FORK_1(i) = pickupright.i -> putdownright.i -> FORK_1(i)
        [] pickupleft.dec(i) -> putdownleft.dec(i) -> FORK_1(i)

PHILS_1 = || i:{0..4} @ [{|pickupleft.i, pickupright.i,
                           putdownleft.i, putdownright.i,
                           sitdown.i, getup.i|}]
                      PHIL_1(i)
```

```
FORKS_1 = || i:{0..4} @ [{|pickupright.i, putdownright.i,
                          pickupleft.dec(i), putdownleft.dec(i)|}]
                    FORK_1(i)

DINNER_1 = PHILS_1 [ {|pickupleft,pickupright,
                     putdownleft,putdownright,sitdown,getup|}
                 || {|pickupleft,pickupright,
                      putdownleft,putdownright|} ]
         FORKS_1

BUTLER_1(i) = if i == 0
            then sitdown?x -> world_sit!x -> BUTLER_1(1)
            else if i == 4
                 then getup?y -> world_up!y -> BUTLER_1(3)
                 else (  sitdown?x -> world_sit!x -> BUTLER_1(i+1)
                      [] getup?y -> world_up!y -> BUTLER_1(i-1) )

NEWDINNER_1 = (DINNER_1 [ {|pickupleft,pickupright,
                         putdownleft,putdownright,sitdown,getup|}
                     || {|sitdown,getup|} ]
            BUTLER_1(0) )
        \ {|pickupleft,pickupright,
                         putdownleft,putdownright,sitdown,getup|}
```

Now a diagram of the processes and channels looks like:



5.4 Proving first refinement

This modified specification is proved a refinement of the original. (See later for FDR2 screen shot.)

```
assert NEWDINNER [FD= NEWDINNER_1
```

5.5 Unidirectional channels

Now we need to make the channels unidirectional, one end outputting, one end inputting.

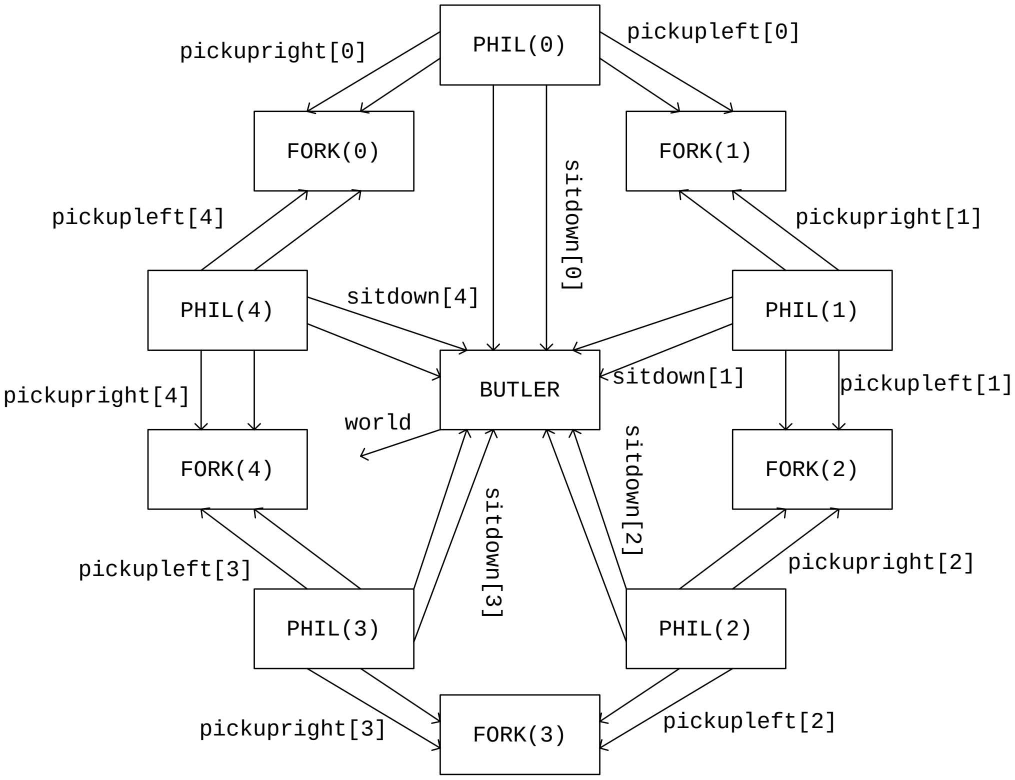
The **sitdown** and **getup** channels are inputs at the butler process (the input value is the index of the relevant philosopher), so it seems sensible to make them outputs at the philosopher process. Also we have the identity **sitdown.i = sitdown!i**. However, we also want these channels to be an array, with **i** as the array index. We also note that the value transmitted is not used by the butler process. So we keep **i** as the index in the philosopher process, and make it into an index in the butler process, using a replicated external choice. We also add a dummy boolean value (which can be implemented in Handel-C as a one-bit channel) to be communicated, from philosopher to butler.

We similarly add a dummy value to the pickup and putdown channels, and arbitrarily choose the communication direction to be from philosopher to fork.

```
-- The dining philosophers with a butler.
--
-- (2) point-to-point channel structure

channel pickupleft_2:{0..4}.Bool
channel pickupright_2:{0..4}.Bool
channel putdownleft_2:{0..4}.Bool
channel putdownright_2:{0..4}.Bool
channel sitdown_2:{0..4}.Bool
channel getup_2:{0..4}.Bool
channel world_sit:{0..4}
channel world_up:{0..4}

PHIL_2(i) =  sitdown_2.i!true ->
             pickupleft_2.i!true ->  pickupright_2.i!true ->
             putdownleft_2.i!true -> putdownright_2.i!true ->
             getup_2.i!true -> PHIL_2(i)

FORK_2(i) = pickupright_2.i?b -> putdownright_2.i?b -> FORK_2(i)
        [] pickupleft_2.dec(i)?b -> putdownleft_2.dec(i)?b -> FORK_2(i)

PHILS_2 = || i:{0..4} @ [{|pickupleft_2.i, pickupright_2.i,
                          putdownleft_2.i, putdownright_2.i,
                          sitdown_2.i, getup_2.i|}]
                     PHIL_2(i)

FORKS_2 = || i:{0..4} @ [{|pickupright_2.i, putdownright_2.i,
                          pickupleft_2.dec(i), putdownleft_2.dec(i)|}]
                     FORK_2(i)

DINNER_2 = PHILS_2 [ {|pickupleft_2, pickupright_2,
                       putdownleft_2, putdownright_2,
                       sitdown_2, getup_2|}
                 ||    {|pickupleft_2,pickupright_2,
                       putdownleft_2,putdownright_2|} ]
           FORKS_2
```

```
BUTLER_2(i) = if i == 0
            then [] x:{0..4} @ sitdown_2.x?b ->
                               world_sit!x -> BUTLER_2(1)
            else if i == 4
                 then [] y:{0..4} @ getup_2.y?b ->
                                    world_up!y -> BUTLER_2(3)
                 else (  [] x:{0..4} @ sitdown_2.x?b ->
                                       world_sit!x ->
                                       BUTLER_2(i+1)
                      [] [] y:{0..4} @ getup_2.y?b ->
                                       world_up!y ->
                                       BUTLER_2(i-1) )

NEWDINNER_2 = (DINNER_2 [ {|pickupleft_2,pickupright_2,
                       putdownleft_2,putdownright_2,sitdown_2,getup_2|}
                    || {|sitdown_2,getup_2|} ]
            BUTLER_2(0) )
            \ {|pickupleft_2,pickupright_2,
                       putdownleft_2,putdownright_2,sitdown_2,getup_2|}
```

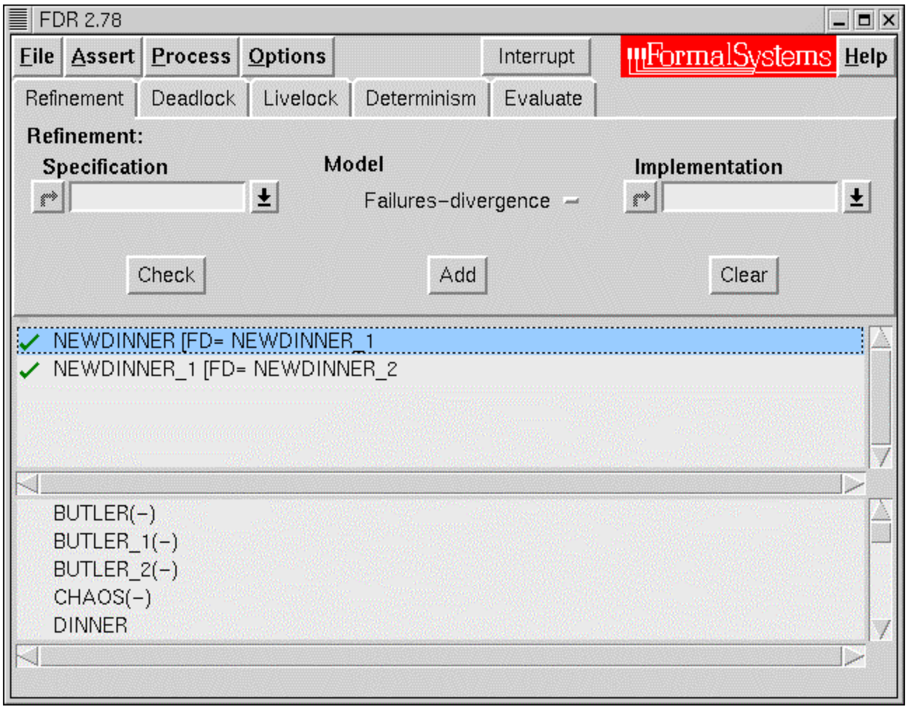
Now a diagram of the processes and point-to-point channels looks like:



5.6 Proving second refinement

This modified specification is proved a refinement of the original.

```
assert NEWDINNER_1 [FD= NEWDINNER_2
```



Refinement:

Specification

Model

Failures-divergence

Implementation

✓ NEWDINNER [FD= NEWDINNER_1

✓ NEWDINNER_1 [FD= NEWDINNER_2

BUTLER(-)
BUTLER_1(-)
BUTLER_2(-)
CHAOS(-)
DINNER

5.7 Conversion to Handel-C

The CSP specification has been refined to a point where it is suitable for translation into Handel-C, on addition of certain translation directives.

The index on the philosopher and fork process represents replication, of process and channels. The index on the butler process represents internal state.

The initial **`{0..4}`** type is taken as an array index.

```
//--!! channel pickupleft array 5  (etc)
chan Bool pickupleft[5] ;
chan Bool pickupright[5] ;
chan Bool putdownleft[5] ;
chan Bool putdownright[5] ;
chan Bool sitdown[5] ;
chan Bool getup[5] ;

//--!! channel out world_sit
chanout int world_sit ;
//--!! channel out world_up
chanout int world_up ;
```

We inline the decrement function to ensure that the parallel philosophers get one each

```
//--!! function inline int dec(int x)
inline int dec(int x) { return( (x - 1) % 5); }
```

The parameter to the philosopher process is used internally as a channel index. The tail recursive process is translated to a `while` loop.

```
void PHIL(int i) {
  while (1) {
    sitdown[0 @ i]!true ;
    pickupleft[0 @ i]!1 ;
    pickupright[0 @ i]!1 ;
    putdownleft[0 @ i]!1 ;
    putdownright[0 @ i]!1 ;
    getup[0 @ i]!true ;
  }
}
```

The parameter to the fork process is used internally as a channel index. The tail recursive process is translated to a `while` loop. The external choice is translated to a `prialt`. The dummy variable `b` has to be declared in the Handel-C.

```
void FORK(int i) {
    unsigned int 1 b ;
    while (1) {
      prialt {
        case pickupright[0 @ i]?b :
          putdownright[0 @ i]?b ;
          break ;
        case pickupleft[0 @ dec(i)]?b :
          putdownleft[0 @ dec(i)]?b ;
          break;
      }
    }
}
```

The replicated parallel is translated to a replicated `par`. The list of synchronised channels disappears.

```
void PHILS() {
  par (i = 0; i < 5,; ++i) {
    PHIL(i) ;
  }
}

void FORKS() {
  par (i = 0; i < 5,; ++i) {
    FORK(i) ;
  }
}

void DINNER() {
  par {
    PHILS ;
    FORKS ;
  }
}
```

The parameter to the butler process is used internally as state, so an internal variable is declared, and initialised to the actual parameter. The tail recursive process is translate to a `while` loop. The replicated external choice is expanded out (there being no replicated `prialt` in Handel-C) and translated to a `prialt`. The dummy variable `b` has to be declared in the Handel-C.

```
 void BUTLER(int iparam) {
   int i = iparam ;
   unsigned int b;

   while (1) {
     if (i == 0) {
       s.s2 = 1;
       prialt {
         case sitdown[0]?b : world_sit!0 ; i = 1 ; break ;
         case sitdown[1]?b : world_sit!1 ; i = 1 ; break ;
         case sitdown[2]?b : world_sit!2 ; i = 1 ; break ;
         case sitdown[3]?b : world_sit!3 ; i = 1 ; break ;
         case sitdown[4]?b : world_sit!4 ; i = 1 ; break ;
       }
     }
     else if (i == 4) {
       s.s2 = 0;
       prialt (y = 0; y < 5,; ++y) {
         case getup[0]?b : world_up!0 ; i = 3 ; break ;
         case getup[1]?b : world_up!1 ; i = 3 ; break ;
         case getup[2]?b : world_up!2 ; i = 3 ; break ;
         case getup[3]?b : world_up!3 ; i = 3 ; break ;
         case getup[4]?b : world_up!4 ; i = 3 ; break ;
       }
     }
     else prialt {
           case sitdown[0]?b : world_sit!0 ; i = i+1 ; break ;
           case sitdown[1]?b : world_sit!1 ; i = i+1 ; break ;
           case sitdown[2]?b : world_sit!2 ; i = i+1 ; break ;
           case sitdown[3]?b : world_sit!3 ; i = i+1 ; break ;
           case sitdown[4]?b : world_sit!4 ; i = i+1 ; break ;
           case getup[0]?b : world_up!0 ; i = i-1 ; break ;
           case getup[1]?b : world_up!1 ; i = i-1 ; break ;
           case getup[2]?b : world_up!2 ; i = i-1 ; break ;
           case getup[3]?b : world_up!3 ; i = i-1 ; break ;
           case getup[4]?b : world_up!4 ; i = i-1 ; break ;
     }
   }
}

 void NEWDINNER() {
   par {
     DINNER ;
     BUTLER(0) ;
   }
 }
```

6. Pipeline processor example

This example is provided by AWE, augmented with some diagrams showing the channel structure.

6.1 Original specification

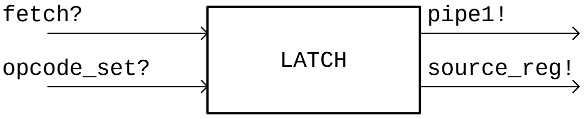
```
-- AEP Analysis by W. Ifill, AWE
-- v1.6

nametype BYTE   = {0..1} -- 7
datatype OPCODE = ldi | nop | hlt | add
nametype REG    = {0..1} -- 8
datatype ACCESS = start | finish
datatype MODE   = read  | write

channel read_reg    : REG.ACCESS
channel write_reg   : REG.ACCESS
channel reg         : REG.ACCESS

channel fetch  : REG.OPCODE.BYTE.REG
channel pipe1  : BYTE.BYTE.OPCODE.BYTE.REG
channel pipe2  : REG.BYTE
channel fail
channel unknown_reg

channel req_read     : REG
channel req_write    : REG
channel data_read    : BYTE
channel data_write   : BYTE
channel source_reg   : REG.OPCODE
channel dest_reg     : REG
channel opcode_set   : BYTE.BYTE.OPCODE
```

STAGE 0 of pipeline is the fetching of **`s.o.i.d`** (Source_register, Opcode, Immediate_data, Destination_register) *from program memory. No specification for this stage as yet.*

STAGE 1 of the pipeline: Fetch gets the contents of the next instruction from the instruction reg. Access to the register bank is signalled as started (like register bank enable). A read is made (data on reg databus). Two consecutive regs are read. The reg. bank is disabled (finish). The fetch data and reg data is communicated to the next pipeline stage.

```
LATCH  = fetch?s.o.i.d ->
           source_reg!s.o -> opcode_set?x.y.a
            -> pipe1!x.y.a.i.d -> LATCH
```

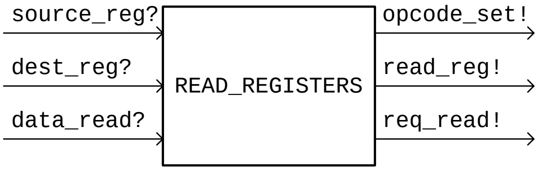
Channel structure:



To prevent a data hazard we define **READ_REGISTERS**. *The incoming opcode will be changed to a nop to prevent simultaneous access of the same register location (loc).*

```
READ_REGISTERS = source_reg?s.o -> dest_reg?d ->
                  if ( s == d ) then
                    (opcode_set!0.0.nop -> READ_REGISTERS)
                  else
                    (read_reg!s.start  ->
                      req_read!s -> data_read?x ->
                      req_read!((s+1)%2) -> data_read?y ->
                        read_reg!s.finish ->
                          opcode_set!x.y.o   -> READ_REGISTERS)
```

Channel structure:

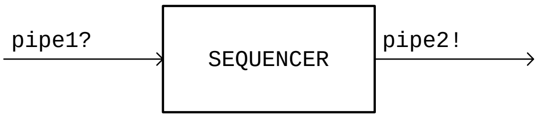


STAGE 2: These are the actions of the individual commands.

- *ADD: retrieved register values are added.*
- *LDI: new value passes for loading in destination register.*
- *NOP: to prevent the setting of the destination register on NOP the destination register is set to r0.*
- *HLT: stops processor.*

```
SEQUENCER = ADD [] LDI [] NOP [] HLT

ADD =  pipe1?v.w.add.i.d -> pipe2!d.((v+w)%2) -> SEQUENCER
LDI =  pipe1?v.w.ldi.i.d -> pipe2!d.i         -> SEQUENCER
NOP =  pipe1?v.w.nop.i.d -> pipe2!0.i         -> SEQUENCER
HLT =  pipe1?v.w.hlt.i.d -> STOP
```

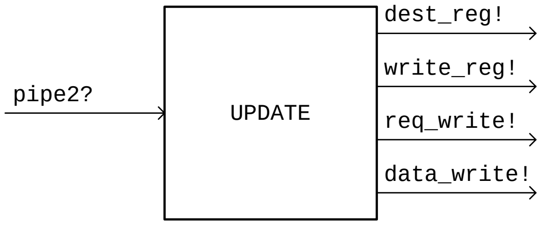
Channel structure:



STAGE 3: The update performs a write if the two zero'ed registers are not requested. **UPDATE** *works with* **READ_REGISTERS** *to prevent a data hazard as does* **LATCH**. **LATCH** *and* **READ_REGISTERS** *together latch a safe instruction.*

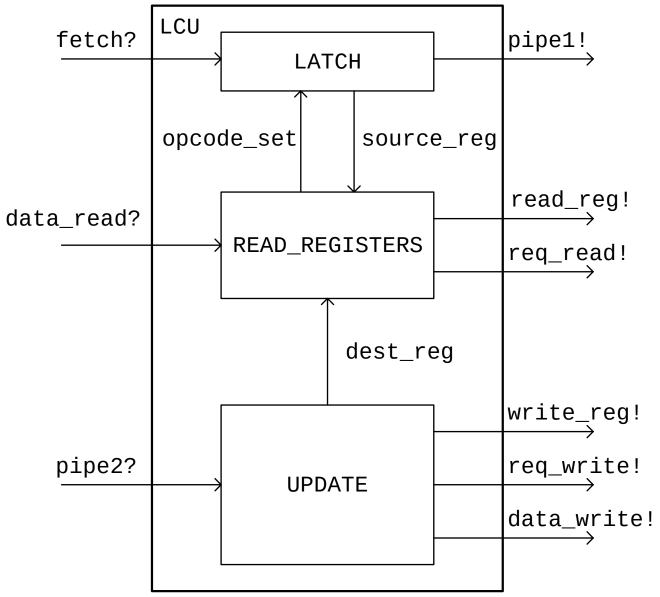
```
UPDATE  =  pipe2?d.i     ->
             if (d != 0)
             then
                (dest_reg!d ->
                   write_reg!d.start  -> req_write!d  ->
                       data_write!i ->
                            write_reg!d.finish -> UPDATE)
             else
                 UPDATE
```

Channel structure:



```
LCU = (LATCH [|{| source_reg, opcode_set |}|] READ_REGISTERS)
             [|{| dest_reg  |}|] UPDATE
      \    {|dest_reg, source_reg, opcode_set|}
```

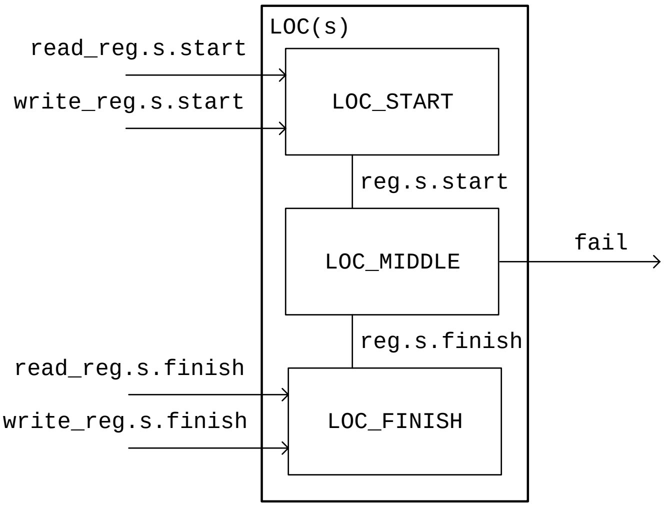
Channel structure:



THE REGISTER BANK: If access to the same register (loc) is required simultaneously then the second request should have been delay. If not fail will be signalled. To speed up analysis only two reg from reg bank are used: 0 and 1.

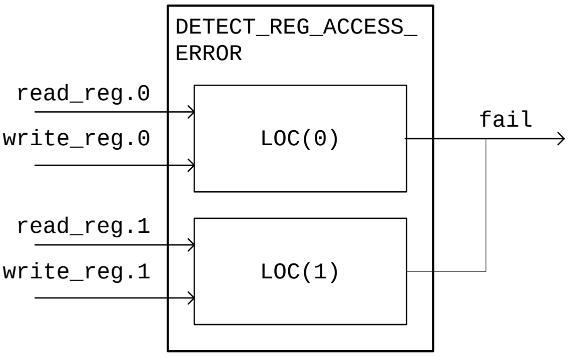
```
LOC_START(s) =
      read_reg.s.start -> reg.s.start -> LOC_START(s)
   [] write_reg.s.start -> reg.s.start -> LOC_START(s)

LOC_FINISH(s) =
      read_reg.s.finish -> reg.s.finish -> LOC_FINISH(s)
   [] write_reg.s.finish -> reg.s.finish -> LOC_FINISH(s)

LOC_MIDDLE(s) =
      reg.s.start ->
       (   reg.s.finish -> LOC_MIDDLE(s)
        [] reg.s.start -> fail -> STOP)

LOC(s) = ((LOC_START(s) [|{|reg|}|] LOC_MIDDLE(s))
         [|{|reg|}|] LOC_FINISH(s))
       \ {|reg|}
```

Channel structure: (Note: the **read_reg** *and* **write_reg** *channels are shown as inputs, because that fits the way they are used later. Also, these channels, and the* **reg** *channel, are used by parallel processes.)*



```
DETECT_REG_ACCESS_ERROR = LOC(0) ||| LOC(1)
```

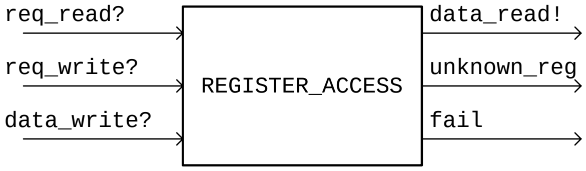
Channel structure: (Note: the **fail** *channel is used by two parallel processes.)*



READING AN INDIVIDUAL REGISTER IN THE REGISTER BANK. This process outputs/inputs values of registers. To speed up analysis only two reg from reg bank are used: 0 and 1.

```
REGISTER_ACCESS(s0,s1,s2,s3,s4,s5,s6,s7) =

(req_read?s ->
    if s == 0
    then data_read!s0 -> REGISTER_ACCESS(s0,s1,s2,s3,s4,s5,s6,s7)
    else if s == 1
    then data_read!s1 -> REGISTER_ACCESS(s0,s1,s2,s3,s4,s5,s6,s7)
    else  unknown_reg -> fail -> STOP)
[]
(req_write?s ->
    if s == 0
    then data_write?s0 -> REGISTER_ACCESS(s0,s1,s2,s3,s4,s5,s6,s7)
    else if s == 1
    then data_write?s1 -> REGISTER_ACCESS(s0,s1,s2,s3,s4,s5,s6,s7)
    else unknown_reg -> fail -> STOP)
```

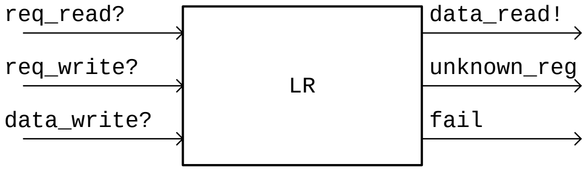
Channel structure:



INTERFACING WITH REGISTER BANK

```
LR(s0,s1,s2,s3,s4,s5,s6,s7) =
     REGISTER_ACCESS(s0,s1,s2,s3,s4,s5,s6,s7)
```

Channel structure:



PUTTING THE PIPELINE TOGETHER: when latch does a read it reads a value from the **REGISTER_ACCESS** *process. Initial values are fixed (No write as yet. Similar structure but read?...)*

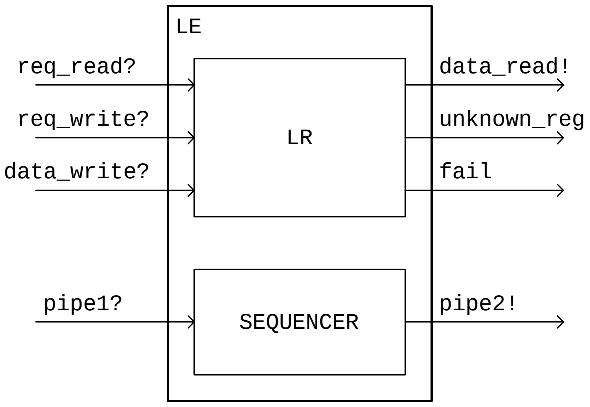
pipeline stages interfaces.

```
P1 = {| pipe1 |}

P2 = {| pipe2, req_read, req_write, data_read, data_write |}
```

Build up pipe line: interfacing stage 1 and 2. The output of one stage is the input of the next

```
LE(s0,s1,s2,s3,s4,s5,s6,s7) =
    LR(s0,s1,s2,s3,s4,s5,s6,s7) ||| SEQUENCER
```

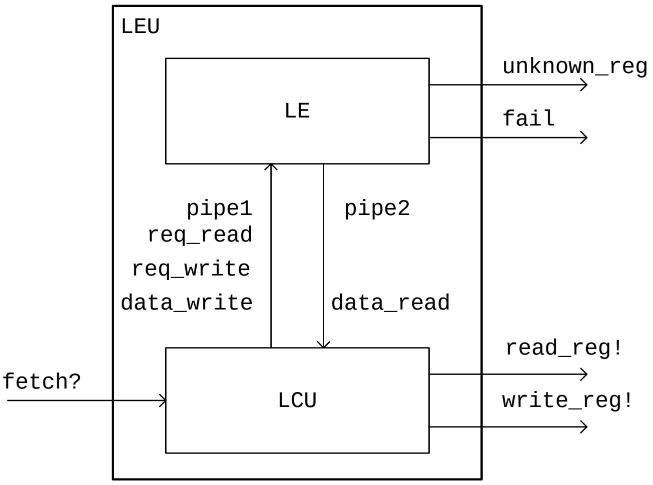
Channel structure:



Interfacing stages 2 and 3.

```
LEU(s0,s1,s2,s3,s4,s5,s6,s7) =
     LE(s0,s1,s2,s3,s4,s5,s6,s7) [|P2|] LCU
```

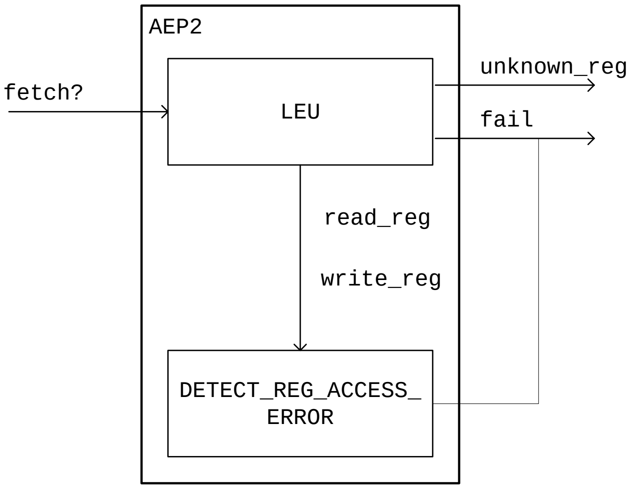
Channel structure: (Note: this assumes **`pipe2`** *is in the communication set. Note: the "internal" channels are not hidden, so are visible to the FDR2 analysis.)*



Reg bank access action is synchronised with the pipeline. Faults in the pipeline may force the reg bank to **`STOP`**

```
AEP2(s0,s1,s2,s3,s4,s5,s6,s7) =
     LEU(s0,s1,s2,s3,s4,s5,s6,s7)
   [|{| read_reg, write_reg |}|]
     DETECT_REG_ACCESS_ERROR
```

Channel structure: (Note: the **`fail`** *channel is used by two parallel processes. Note: the "internal" channels are not hidden, so are visible to the FDR2 analysis.)*



6.2 Hiding internal channels

For the purposes of refining to Handel-C, we take this final specification, and hide all the interior channels, to find our starting point.

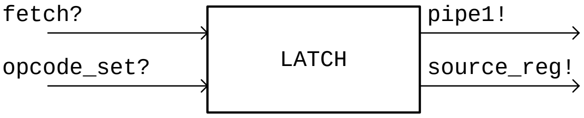
```
P3_0 = {| pipe1, pipe2, req_read, req_write,
          data_read, data_write,
          read_reg, write_reg |}

AEP_0(s0,s1,s2,s3,s4,s5,s6,s7) =
     AEP2(s0,s1,s2,s3,s4,s5,s6,s7) \ P3_0
```

6.3 Refining shared channels

Now we need to remove the shared channels **read_reg**, **write_reg**, **reg**, and **fail**. The first three cases can be achieved by having separate **start** and **finish** channels in each case, as these are used by separate processes. The last case can be achieved by using a multiplexor.

```
-- AEP Analysis
-- separate start and finish channels, multiplexed fail channel

nametype BYTE   = {0..1} -- 7
datatype OPCODE = ldi | nop | hlt | add
nametype REG    = {0..1} -- 8

channel read_reg_start    : REG
channel write_reg_start   : REG
channel reg_start         : REG
channel read_reg_finish    : REG
channel write_reg_finish   : REG
channel reg_finish         : REG

channel fetch  : REG.OPCODE.BYTE.REG
channel pipe1  : BYTE.BYTE.OPCODE.BYTE.REG
channel pipe2  : REG.BYTE
channel fail_loc  : REG
channel fail_reg
channel unknown_reg
channel fail

channel req_read     : REG
channel req_write    : REG
channel data_read    : BYTE
channel data_write   : BYTE
channel source_reg   : REG.OPCODE
channel dest_reg     : REG
channel opcode_set   : BYTE.BYTE.OPCODE

LATCH  = fetch?s.o.i.d ->
           source_reg!s.o -> opcode_set?x.y.a
            -> pipe1!x.y.a.i.d -> LATCH
```

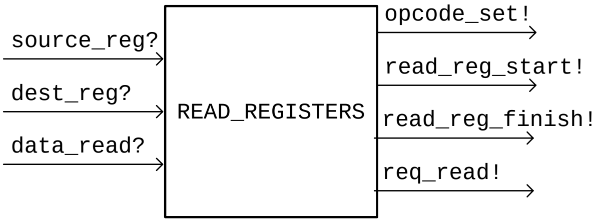
Channel structure:



We change **READ_REGISTER** *to use the separate start and finish channels.*

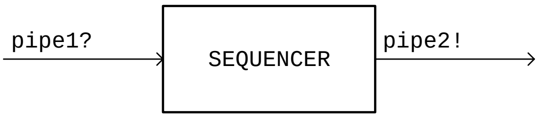
```
READ_REGISTERS = source_reg?s.o -> dest_reg?d ->
                  if ( s == d ) then
                    (opcode_set!0.0.nop -> READ_REGISTERS)
                  else
                    (read_reg_start!s ->
                      req_read!s -> data_read?x ->
                      req_read!((s+1)%2) -> data_read?y ->
                        read_reg_finish!s ->
                          opcode_set!x.y.o   -> READ_REGISTERS)
```

Channel structure:



```
SEQUENCER = ADD [] LDI [] NOP [] HLT

ADD =  pipe1?v.w.add.i.d -> pipe2!d.((v+w)%2) -> SEQUENCER
LDI =  pipe1?v.w.ldi.i.d -> pipe2!d.i         -> SEQUENCER
NOP =  pipe1?v.w.nop.i.d -> pipe2!0.i         -> SEQUENCER
HLT =  pipe1?v.w.hlt.i.d -> STOP
```

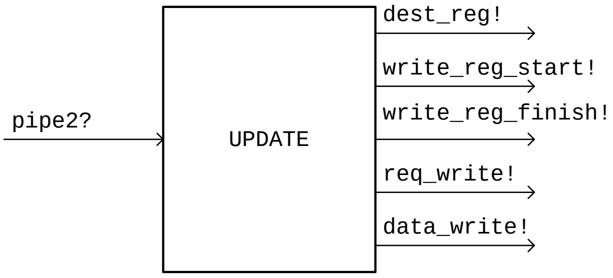
Channel structure:



We change **UPDATE** *to use the separate start and finish channels.*

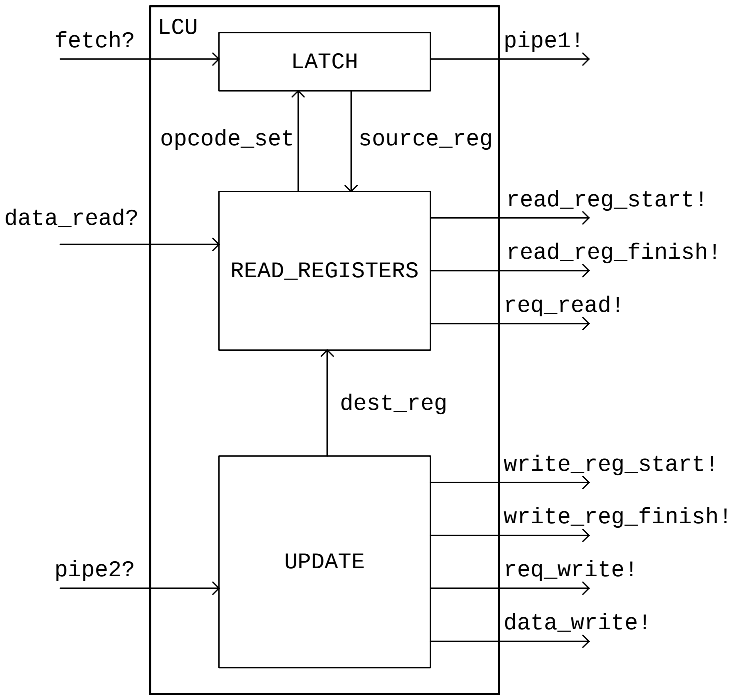
```
UPDATE  =  pipe2?d.i     ->
             if (d != 0)
             then
                (dest_reg!d ->
                   write_reg_start!d  -> req_write!d  ->
                       data_write!i ->
                            write_reg_finish!d -> UPDATE)
             else
                 UPDATE
```

Channel structure:



```
LCU = (LATCH [|{| source_reg, opcode_set |}|] READ_REGISTERS)
             [|{| dest_reg  |}|] UPDATE
      \    {|dest_reg, source_reg, opcode_set|}
```

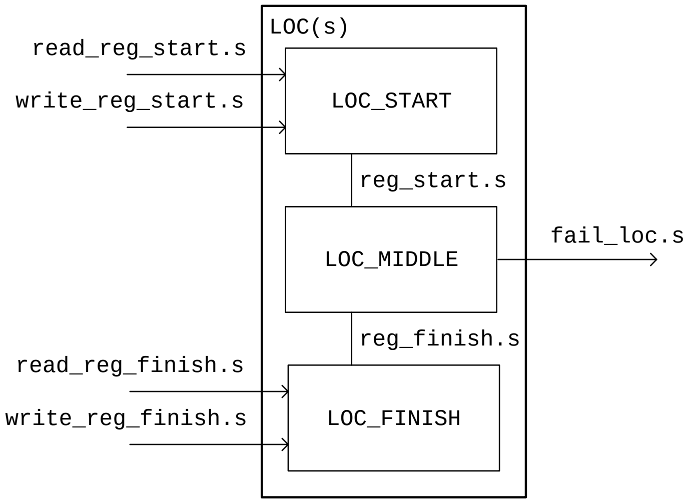
Channel structure:



We change **`LOC`** *to use the separate start and finish channels, and the indexed* **`fail_loc`** *channels.*

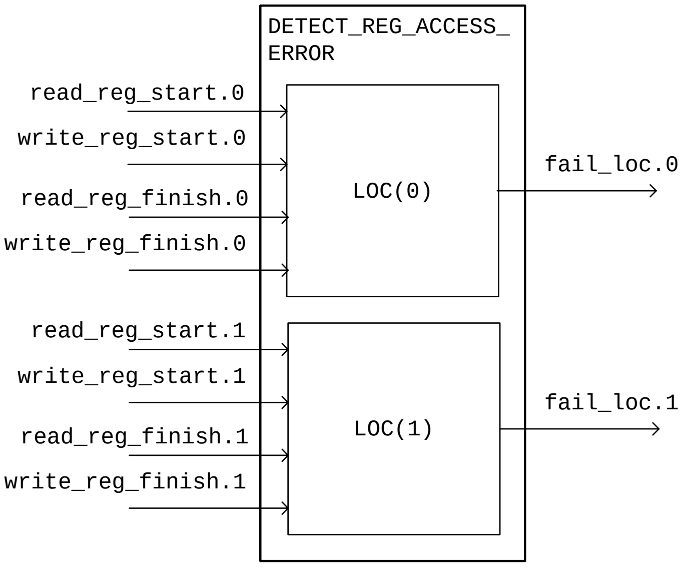
```
LOC_START(s) =
      read_reg_start.s -> reg_start.s -> LOC_START(s)
   [] write_reg_start.s -> reg_start.s -> LOC_START(s)

LOC_FINISH(s) =
      read_reg_finish.s -> reg_finish.s -> LOC_FINISH(s)
   [] write_reg_finish.s -> reg_finish.s -> LOC_FINISH(s)

LOC_MIDDLE(s) =
      reg_start.s ->
       (   reg_finish.s -> LOC_MIDDLE(s)
        [] reg_start.s -> fail_loc.s -> STOP)

LOC(s) = ((LOC_START(s) [|{|reg_start|}|] LOC_MIDDLE(s))
         [|{|reg_finish|}|] LOC_FINISH(s))
       \ {|reg_start,reg_finish|}
```

Channel structure: (Note: the **read_reg** *and* **write_reg** *channels are shown as inputs, because that fits the way they are used later.)*



```
DETECT_REG_ACCESS_ERROR = LOC(0) ||| LOC(1)
```

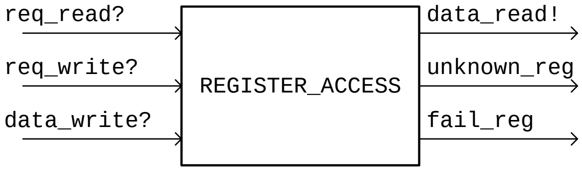
Channel structure:



We change **REGISTER_ACCESS** *to use the separate* **fail_reg** *channel.*

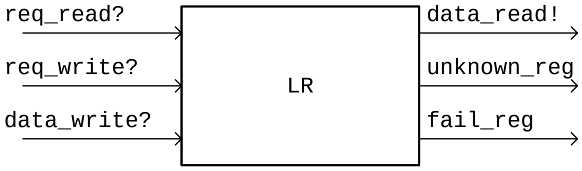
```
REGISTER_ACCESS(s0,s1,s2,s3,s4,s5,s6,s7) =

(req_read?s ->
    if s == 0
    then data_read!s0 -> REGISTER_ACCESS(s0,s1,s2,s3,s4,s5,s6,s7)
    else if s == 1
    then data_read!s1 -> REGISTER_ACCESS(s0,s1,s2,s3,s4,s5,s6,s7)
    else  unknown_reg -> fail_reg -> STOP)
[]
(req_write?s ->
    if s == 0
    then data_write?s0 -> REGISTER_ACCESS(s0,s1,s2,s3,s4,s5,s6,s7)
    else if s == 1
    then data_write?s1 -> REGISTER_ACCESS(s0,s1,s2,s3,s4,s5,s6,s7)
    else unknown_reg -> fail_reg -> STOP)
```

Channel structure:



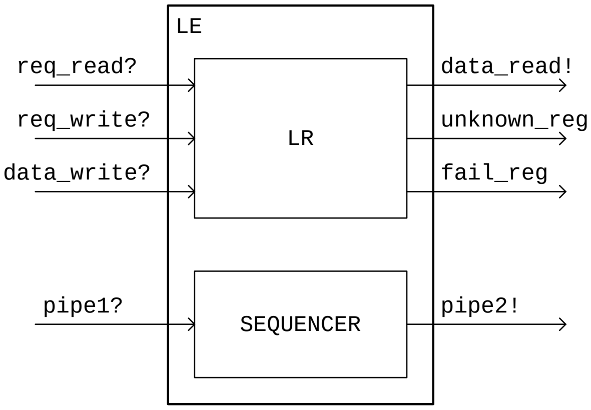
```
LR(s0,s1,s2,s3,s4,s5,s6,s7) =
     REGISTER_ACCESS(s0,s1,s2,s3,s4,s5,s6,s7)
```

Channel structure:



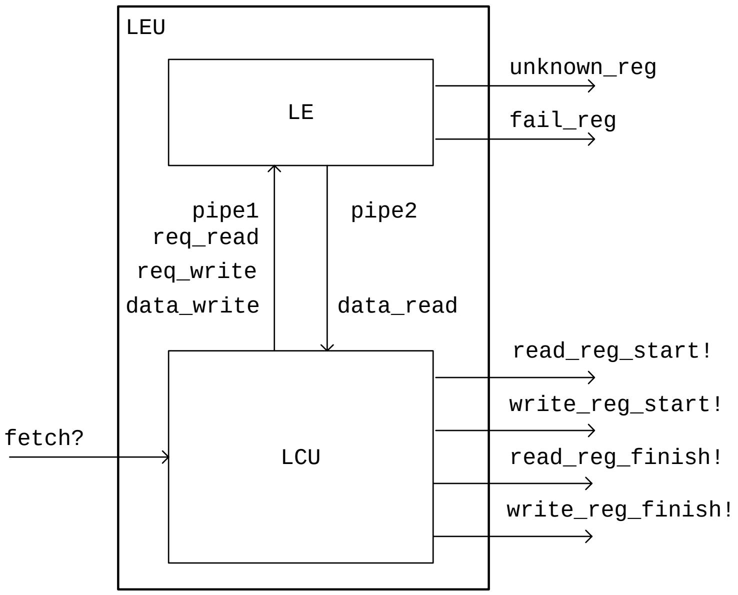
```
P1 = {| pipe1 |}

P2 = {| pipe2, req_read, req_write, data_read, data_write |}

LE(s0,s1,s2,s3,s4,s5,s6,s7) =
    LR(s0,s1,s2,s3,s4,s5,s6,s7) ||| SEQUENCER
```

Channel structure:



```
LEU(s0,s1,s2,s3,s4,s5,s6,s7) =
     LE(s0,s1,s2,s3,s4,s5,s6,s7) [|P2|] LCU
```

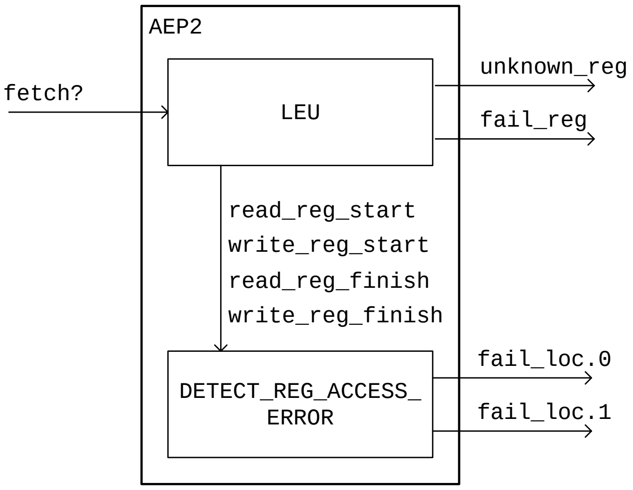
Channel structure:



We change **AEP2** *to use the separate start and finish channels.*

```
AEP2(s0,s1,s2,s3,s4,s5,s6,s7) =
     LEU(s0,s1,s2,s3,s4,s5,s6,s7)
   [|{| read_reg_start, write_reg_start,
         read_reg_finish, write_reg_finish |}|]
     DETECT_REG_ACCESS_ERROR
```

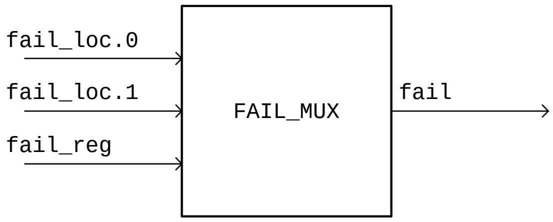
Channel structure:



We add a multiplexing process for the fail channels.

```
FAIL_MUX =
    [] s:{0..1} @ fail_loc?s -> fail -> FAIL_MUX
    [] fail_reg -> fail -> FAIL_MUX
```

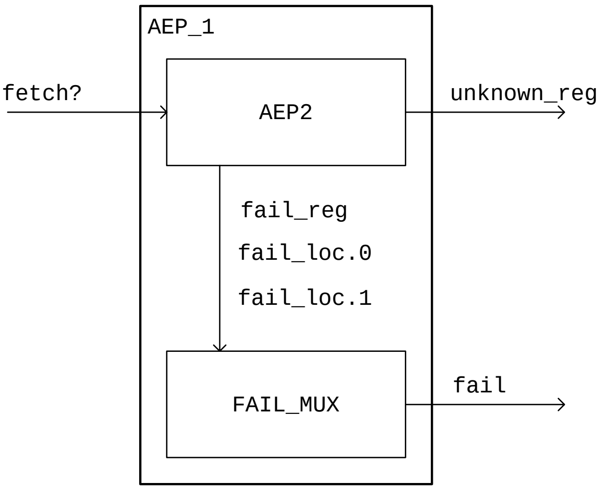
Channel structure:



We put this in parallel with AEP, and hide the internal comms.

```
P3_1 = {| pipe1, pipe2, req_read, req_write,
         data_read, data_write,
         read_reg_start, write_reg_start,
         read_reg_finish, write_reg_finish,
         fail_loc, fail_reg |}

AEP_1(s0,s1,s2,s3,s4,s5,s6,s7) =
        ( AEP2_1(s0,s1,s2,s3,s4,s5,s6,s7)
        [| {| fail_loc, fail_reg |} |] FAIL_MUX )
    \  P3_1
```

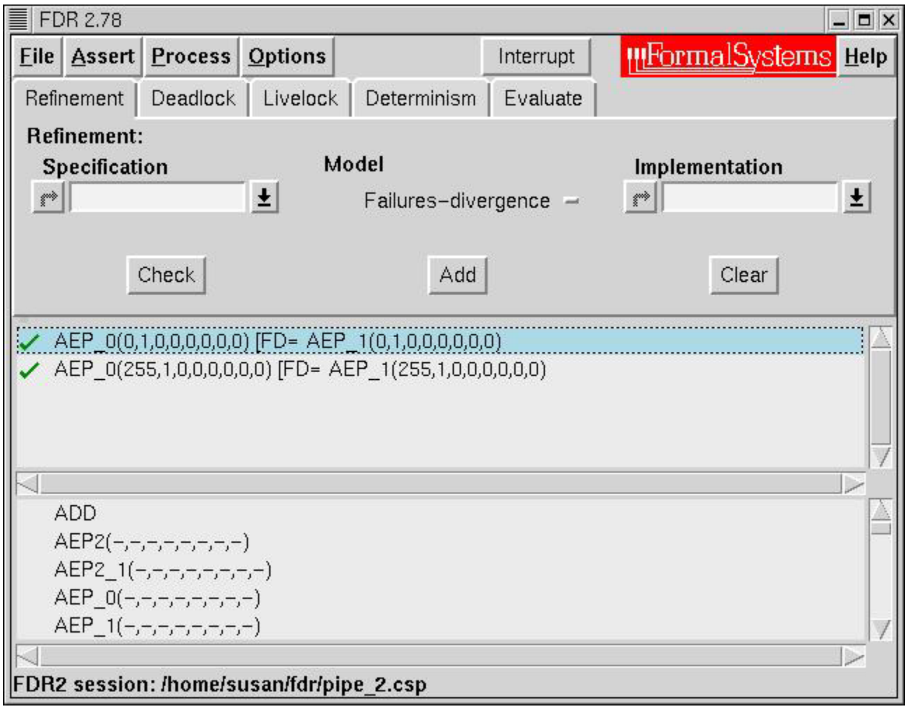
Channel structure:



6.4 Proving refinement

We show that this new version is a refinement of the original.

```
assert AEP_0(0,1,0,0,0,0,0,0)
  [FD=  AEP_1(0,1,0,0,0,0,0,0)

assert AEP_0(255,1,0,0,0,0,0,0)
  [FD=  AEP_1(255,1,0,0,0,0,0,0)
```



Model: Failures–divergence

✓ AEP_0(0,1,0,0,0,0,0,0) [FD= AEP_1(0,1,0,0,0,0,0,0)
✓ AEP_0(255,1,0,0,0,0,0,0) [FD= AEP_1(255,1,0,0,0,0,0,0)

ADD
AEP2(–,–,–,–,–,–,–,–)
AEP2_1(–,–,–,–,–,–,–,–)
AEP_0(–,–,–,–,–,–,–,–)
AEP_1(–,–,–,–,–,–,–,–)

FDR2 session: /home/susan/fdr/pipe_2.csp

6.5 Conversion to Handel-C

Now we convert the final specification into Handel-C, including addition of certain translation directives.

The **nametype**s translate to typedefs. (The specification as it stands defined a **BYTE** and a **REG** to be a bit, but we assume a full byte here.)

```
typedef unsigned int 8 BYTE ;
typedef unsigned int 8 REG ;
```

The datatype **OPCODE** translates to an enum.

```
enum OPCODE { ldi , nop , hlt , add } ;
```

The channels that talk to the registers use **REG** as an index (here we use the actual maximum value of **REG**), and communicate that the event has occurred.

```
//--!! channel array read_reg_start index 8  (etc)
chan SYNC read_reg_start[8] ;
chan SYNC write_reg_start[8] ;
chan SYNC reg_start[8] ;
chan SYNC read_reg_finish[8] ;
chan SYNC write_reg_finish[8] ;
chan SYNC reg_finish[8] ;

chan SYNC fail_loc[8] ;
```

The **fail_reg** channel also communicates that the event has occurred.

```
chan SYNC fail_reg ;
```

Other channels communicate a simple **REG**, or **BYTE**.

```
chan REG req_read ;
chan REG req_write ;
chan REG dest_reg ;

chan BYTE data_read ;
chan BYTE data_write ;
```

Other channels communicate structured types.

```
struct pipe1_DATA {
  BYTE byte1;
  BYTE byte2;
  enum OPCODE opcode1;
  BYTE byte3;
  REG reg1;
} ;
chan struct pipe1_DATA pipe1 ;

struct pipe2_DATA {
  REG reg1;
  BYTE byte1;
} ;
chan struct pipe2_DATA pipe2 ;

struct source_reg_DATA {
  REG reg1;
  enum OPCODE opcode1;
} ;
chan struct source_reg_DATA source_reg ;

struct opcode_set_DATA {
  BYTE byte1;
  BYTE byte2;
  enum OPCODE opcode1;
} ;
chan struct opcode_set_DATA opcode_set ;
```

The input channel communicates a structured type. This is not possible in Handel-C, so a sequence of inputs is used.

```
//--!! channel in fetch
struct fetch_DATA {
  REG reg1;
  enum OPCODE opcode1;
  BYTE byte1;
  REG reg2;
} ;
chanin REG fetch_reg1 ;
chanin enum OPCODE fetch_opcode1 ;
chanin BYTE fetch_byte1 ;
chanin REG fetch_reg2 ;
```

The output channels communicate that the event has occurred.

```
//--!! channel out unknown_reg
chanout SYNC unknown_reg ;
//--!! channel out fail
chanout SYNC fail ;
```

LATCH is a tail recursive process. Structures need to be declared for the various inputs and outputs.

```
void LATCH() {
  REG reg1;
  enum OPCODE opcode1;
  BYTE byte1;
  REG reg2;
  struct source_reg_DATA source_reg1;
  struct opcode_set_DATA opcode_set1;
  struct pipe1_DATA pipe11;

  while (1) {
    fetch_reg1?reg1 ;
    fetch_opcode1?opcode1 ;
    fetch_byte1?byte1 ;
    fetch_reg2?reg2 ;

    source_reg1.reg1 = reg1 ;
    source_reg1.opcode1 = opcode1 ;
    source_reg!source_reg1 ;

    opcode_set?opcode_set1 ;

    pipe11.byte1 = opcode_set1.byte1 ;
    pipe11.byte2 = opcode_set1.byte2 ;
    pipe11.opcode1 = opcode_set1.opcode1 ;
    pipe11.byte3 = byte1 ;
    pipe11.reg1 = reg2 ;
    pipe1!pipe11 ;
  }
}
```

READ_REGISTERS is a tail recursive process. Structures need to be declared for the various inputs and outputs.

```
void READ_REGISTERS() {
  struct source_reg_DATA source_reg1;
  REG s, d ;
  BYTE x, y ;
  struct opcode_set_DATA opcode_set1;

  while(1) {
    source_reg?source_reg1 ;
    s = source_reg1.reg1 ;

    dest_reg?d;
    if ( s == d ) {

      opcode_set1.byte1 = 0 ;
      opcode_set1.byte1 = 0 ;
      opcode_set1.opcode1 = nop ;
      opcode_set!opcode_set1 ;

    } else {

      read_reg_start[0 @ s]!syncout ;
      req_read!s ;
      data_read?x ;
      req_read!(s+1)%2 ;
      data_read?y ;
      read_reg_finish[0 @ s]!syncout ;

      opcode_set1.byte1 = x ;
      opcode_set1.byte1 = y ;
      opcode_set1.opcode1 = source_reg1.opcode1 ;
      opcode_set!opcode_set1 ;
    }
  }
}
```

SEQUENCER is a tail recursive process (one branch is **STOP**, but the process would have the same semantics if it were **STOP ; SEQUENCER**). It is an ALT on CSP events, but it is not an ALT on Handel-C channels, it is a case statement on one of the values communicated on a single channel.

```
void SEQUENCER() {
  struct pipe1_DATA pipe11;
  struct pipe2_DATA pipe21;

  while(1) {
    pipe1?pipe11 ;

    switch ( pipe11.opcode1 ) {
      case add:
        pipe21.reg1 = pipe11.reg1 ;
        pipe21.byte1 = (pipe11.byte1+pipe11.byte2)%2 ;
        pipe2!pipe21 ;
        break ;
      case ldi:
        pipe21.reg1 = pipe11.reg1 ;
        pipe21.byte1 = pipe11.byte3 ;
        pipe2!pipe21 ;
        break ;
      case nop:
        pipe21.reg1 = 0 ;
        pipe21.byte1 = pipe11.byte3 ;
        pipe2!pipe21 ;
        break ;
      case hlt:
        STOP() ;
        break ;
      default:
        STOP() ;
        break ;
    }
  }
}
```

UPDATE is a tail recursive process.

```
void UPDATE() {
  struct pipe2_DATA pipe21;
  REG d ;

  while (1) {
    pipe2?pipe21 ;
    d = pipe21.reg1 ;
    if (d != 0) {
      dest_reg!d ;
      write_reg_start[0 @ d]!syncout ;
      req_write!d  ;
      data_write!pipe21.byte1 ;
      write_reg_finish[0 @ d]!syncout ;
    } else {
      SKIP() ;
    }
  }
}
```

LCU is a parallel composition of three processes.

```
void LCU() {
  par {
    LATCH() ;
    READ_REGISTERS() ;
    UPDATE() ;
  }
}
```

LOC_START, **LOC_MIDDLE**, and **LOC_FINISH** are all tail recursive processes parameterised by a channel index.

```
void LOC_START(unsigned int s) {
  SYNC syncin ;
  while (1) {
    prialt {
      case read_reg_start[0 @ s]?syncin :
        reg_start[0 @ s]!syncout ;
        break ;
      case write_reg_start[0 @ s]?syncin :
        reg_start[0 @ s]!syncout ;
        break ;
    }
  }
}

void LOC_FINISH(unsigned int s) {
  SYNC syncin ;
  while (1) {
    prialt {
      case read_reg_finish[0 @ s]?syncin :
        reg_finish[0 @ s]!syncout ;
        break ;
      case write_reg_finish[0 @ s]?syncin :
        reg_finish[0 @ s]!syncout ;
        break ;
    }
  }
}

void LOC_MIDDLE(unsigned int s) {
  SYNC syncin ;
  while (1) {
    read_reg_start[0 @ s]?syncin ;
    prialt {
      case reg_finish[0 @ s]?syncin :
        break ;
      case reg_start[0 @ s]?syncin :
        fail_loc[0 @ s]!syncout ;
        STOP() ;
        break ;
    }
  }
}
```

LOC is a parallel composition of these three processes.

```
void LOC(unsigned int s) {
  par {
    LOC_START(s) ;
    LOC_MIDDLE(s) ;
    LOC_FINISH(s) ;
  }
}
```

DETECT_REG_ACCESS_ERROR is a parallel composition of two particular **LOC** processes.

```
void DETECT_REG_ACCESS_ERROR() {
  par {
    LOC(0) ;
    LOC(1) ;
  }
}
```

REGISTER_ACCESS is a tail recursive process containing an ALT.

```
void REGISTER_ACCESS(BYTE s0, BYTE s1) {
  REG s ;
  while (1) {
    prialt {
      case req_read?s :
        if (s == 0) {
          data_read!s0 ;
        } else if (s == 1) {
          data_read!s1 ;
        } else {
          unknown_reg!syncout ;
          fail_reg!syncout ;
          STOP() ;
       }
        break ;
      case req_write?s :
        if (s == 0) {
          data_write?s0 ;
        } else if (s == 1) {
          data_write?s1 ;
        } else {
          unknown_reg!syncout ;
          fail_reg!syncout ;
          STOP() ;
        }
        break ;
    }
  }
}
```

LR is a trivial process.

```
void LR(BYTE s0, BYTE s1) {
  REGISTER_ACCESS(s0, s1) ;
}
```

LE is a parallel composition of two processes.

```
void LE(BYTE s0, BYTE s1) {
  par {
    LR(s0, s1) ;
    SEQUENCER() ;
  }
}
```

LEU is a parallel composition of two processes.

```
void LEU(BYTE s0, BYTE s1) {
  par {
    LE(s0, s1) ;
    LCU() ;
  }
}
```

AEP2 is a parallel composition of two processes.

```
void AEP2(BYTE s0, BYTE s1) {
  par {
    LEU(s0, s1) ;
    DETECT_REG_ACCESS_ERROR() ;
  }
}
```

FAIL_MUX is a tail recursive process containing an ALT.

```
//--!! sync in fail_reg
void FAIL_MUX() {
  SYNC syncin ;
  while (1) {
    prialt {
      case fail_loc[0]?syncin :
        fail!syncout ;
        break ;
      case fail_loc[1]?syncin :
        fail!syncout ;
        break ;
      case fail_reg?syncin :
        fail!syncout ;
        break ;
    }
  }
}
```

AEP_1 is a parallel composition of two processes.

```
void AEP_1(BYTE s0, BYTE s1) {
  par {
    AEP2(s0, s1) ;
    FAIL_MUX() ;
  }
}
```

7. References

[FDR2] *Failures-Divergence Refinement*: FDR2 User Manual, version 2.67. Formal Systems (Europe) Ltd. 3 May 2000.

[HandelC] *Handel-C*: Language Reference Manual, version 3.0 beta 2. Embedded Solutions Ltd.

[Hoare78] C. A. R. Hoare. "Communicating Sequential Processes." CACM **21**(8), 666–77. 1978.

[Hoare85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall. 1985.

[Roscoe94] A. W. Roscoe. "Model checking CSP". In *A Classical Mind, essays in honour of C.A.R. Hoare*. Prentice-Hall. 1994.

[Roscoe97] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall. 1997.