csp2hc: from $\mathsf{CSP_M}$ to Handel-C
Current Status

Marcel Oliveira and Jim Woodcock

January 23, 2009

1 Introduction

Our tool, csp2hc, already mechanises the translation of a considerable subset of $\mathsf{CSP_M}$ to Handel-C, which includes the following features.

1. `SKIP`
2. `STOP`
3. Sequential composition
4. `if _ then _ else _`
5. Guarded Processes
6. Recursion
7. Prefixing
8. External Choice
9. Internal Choice
10. Concurrency
11. Datatypes
12. Constants
13. Expressions

Although they represent a subset of $\mathsf{CSP_M}$, using these constructors, we are already able to automatically translate many of the classical CSP examples in the literature, like the examples presented in Appendix **??**: the dining philosophers and the level crossing. More importantly, the phase controller of the CMOS can already be automatically translated using csp2hc. Currently, we are working on the features needed to achieve

the automatic translation of the heap model of the CMOS. Ultimately, we aim at the automatic translation of the whole CMOS.

In what follows we describe the details of our efforts. Section 2 discussed the aspects involving the current status of csp2hc. These include the conventions and assumptions made by csp2hc on the source CSP$_M$ specification, a list of the CSP$_M$ constructors that are supported by csp2hc, the current restrictions on these constructors, and an outline of the solutions implemented by the translator. *The vast majority of these restrictions are intended to be removed in the next stages of the project.* The solutions for these are presented as needed in Section 3.

In Section 2.2, we list which restrictions on the CSP$_M$ are already being automatically checked by csp2hc and which of them are not being automatically checked by the tool.

Our tool uses a CSP$_M$ parser/type checker that has been implemented by our collaborators in UFPE/Brazil. Our efforts created a heavy load of tests that have identified a couple of errors in this tool. These errors are listed in Section 2.3 and they have already been reported to the developers of the CSP$_M$ parser, who have committed themselves to fix these bugs.

Our translator needs some extra information from the user as, for instance, the number of bits that are used to represent integers. These are given by the user to csp2hc in the form of directives (comments in the source CSP$_M$ specification with a special format). In Section 2.4, we describe the directives that are used by csp2hc.

Our main objective is to fully automate the translation of the CMOS. With this purpose, we have created three milestones: the phase controller, the heap model, and the CMOS. In Section 3, we describe the efforts in the translation of these milestones: we describe the CSP$_M$ constructors whose translation are already mechanised, and the ones whose translation have not yet been mechanised. For those whose translation have not yet been mechanised, we provide possible solutions for their implementation in Handel-C.

Finally, Section 4 present csp2hc and discuss how it can be used.

2 Current Status

csp2hc uses a a CSP$_M$ parser/type checker that has been implemented by our collaborators in UFPE/Brazil. This parser, with exception of a few constructors in which it presented some problems, already accepts all the CSP$_M$ constructors needed for the full translation of the CMOS.

The input to csp2hc is a CSP$_M$ specification that has already been checked in FDR. Furthermore, csp2hc considers that none of the Handel-C keywords are present in the source CSP$_M$. Besides, some further keywords are used by csp2hc and cannot be part of the input CSP$_M$ specification as well. They are:

- `clock1`
- `SYNC`
- `syncout`

- `syncin`
- `integer`
- `integer_offset`
- For every possible integer value `i`:
  - `i` ≥ 0: `integer_i_s`
  - `i` < 0: `integer_neg_i_s`
- `boolean`
- `true`
- `false`
- `true_set`
- `false_set`
- `main`
- `mainp`
- `CHANNEL`
- `INEXISTENT_CHANNEL`
- `BRANCH`
- Starting from the main process, let n be the number of parallel branches. For every i, such that $0 \leq i \leq n$: `BRANCH_`i
- For every type T in the specification (including `integer`, `boolean`, `BRANCH`, and `CHANNEL`):
  - T`_set`
  - T`_nil`
  - T`_card`
  - T`__set_LUT`
- `bus_in`
- `bus_out`
- `OutPort`
- For every channel C that is declared as a bus in a directive, we have:
  - `in_bus_`C

  - out_bus_C
  - bus_C_value
  - C_n, for every n that is between (inclusive) 1 and the length of the type of the channel ($1 \leq n \leq$`length(type(<BUS_NAME>))`)
  - InBus_C
  - OutBus_C
- For every simple datatype value v, v_set
- For every complex datatype value `C.v1.v2`, where `C` is the constructor declared as `C.T1.T2`
  - C_T1_T2_LUT
  - C_v1_v2
  - C_v1_v2_s
  - If any of the values `vi` is a negative integer, then we have `neg_vi` instead.
- MUTUAL_REC
- PROGRAM_COUNTER
- KEEP_LOOPING
- FORCED_SYNC_LUT
- FORCED_SYNC_LUT_SEMA
- CURRENT_SYNC_STATUS_LUT
- CURRENT_SYNC_STATUS_LUT_SEMA
- DONE_EVENTS_n, where n is a unique natural number used to avoid variable clashes
- HAS_JOINED_n, where n is a unique natural number used to avoid variable clashes
- Auxiliary macro expression names and arguments
  - IS_AVAILABLE
  - IS_EMPTY_SET
  - SET_UNION
  - SET_DIFF
  - SET_INTER
  - SET_S
  - SET_T
  - MAY_JOIN

- LOCK_VARIABLE
- TRY_JOIN_SEMAPHOR
- JOIN_SEMAPHOR
- LEAVE_SEMAPHOR
- BRANCH_B
- BRANCH_ID
- CHANNEL_C
- BRANCH_SET_S
- NO_WRITER
- HAS_WRITER_LUT
- MULTI_SYNC_READY
- MULTI_SYNC_LUT
- READERS_READY
- IS_SYNC_READY
- BRANCH_MEMBER
- IS_ALREADY
- HAS_JOINED
- SLEEP
- TRY_CONFIRM_READERS
- ARE_READERS_READY
- CONFIRM_READERS
- CONFIRMED
- confirmed_end
- CONFIRM_FINISHED
- WITHDRAW_IN_CONFIRMATION
- WITHDRAW_IN_COMM
- WITHDRAW_IN_FINISHED
- WITHDRAW_SIGNALS

2.1 Translation

In what follows, we list the CSP_M constructors that are supported by csp2hc. For each one of them, we present the restrictions on these constructors, and an outline of the solutions implemented by the translator.

1. SKIP

(a) **Restrictions:** None

(b) **Solution:** translates to nothing.

2. `STOP`

   (a) **Restrictions:** None

   (b) **Solution:** translates to
   ```
   chan SYNC INEXISTENT_CHANNEL;
   INEXISTENT_CHANNEL?syncin
   ```

3. Sequential composition

   (a) **Restrictions:** None

   (b) **Solution:** translates to a sequential composition.

4. Guarded processes

   (a) **Restrictions:** None

   (b) **Solution:** `g & P` translates as `if g then P else STOP`.

5. Recursion

   (a) Simple Recursion

      i. **Restrictions:**

         A. Only Tail Recursion

         B. No parallel composition in the tail recursion.
         `Q = c -> (Q [| {| c1 |} |] Q)`

      ii. **Solution:** tail recursions are translated to a loop that iterates while a `KEEP_LOOPING` variable is `true`. In each iteration, the loop initially sets `KEEP_LOOPING` to `false`; the tail recursion sets this variable to `true`. Possible process arguments are declared as local copies, which are initialised before the beginning of the loop with the given value and are updated before the end of each iteration.
      For instance, process `P(x) = c1 -> P(x+1)` is translated as follows.

```
inline void P1 (integer x){
    boolean KEEP_LOOPING;
    integer P1_local_x;
    P1_local_x = x;
    KEEP_LOOPING = true;
    while(KEEP_LOOPING){
        KEEP_LOOPING = false;
        seq{
            seq{
                c! P1_local_x;
                P1_local_x = x + 1;
```

```
                KEEP_LOOPING = true;
            }
        }
    }
}
```

(b) Mutual Recursion

i. **Restrictions:**

A. Parallel composition (and interleaving) only in the main process given in the directive (as described in Section 2.4.3).

ii. **Solution:** the solution is to transform the whole model into an action system like model. First, we declare all the processes parameters as global variables. Then, we declare a single method parametrised by a process counter that will represent the whole system. Its body is a loop on a variable `KEEP_LOOPING`. In each iteration, we check the value of the program counter and behave accordingly. Possible process arguments are declared as global copies, which are initialised before each invocation of the mutual recursion.
For instance, let us consider the following specification:

```
P1(x) = c!x -> P2(x+1)
P2(x) = c!x -> P1(x-1)
```

It is translated as follows.

```
integer P1_local_x, P2_local_x;
inline void MUTUAL_REC(int 1 PROGRAM_COUNTER){
    KEEP_LOOPING = true;
    while(KEEP_LOOPING){
        KEEP_LOOPING = false;
        switch(PROGRAM_COUNTER){
            case P1 :{
                seq{
                    c! P1_local_x;
                    P2_local_x = P1_local_x + 1;
                    PROGRAM_COUNTER = P2;
                    KEEP_LOOPING = true;
                }
                break;
            }

            case P2 :{
                seq{
                    c! P2_local_x;
                    P1_local_x = P2_local_x - 1;
```

```
                    PROGRAM_COUNTER = P1;
                    KEEP_LOOPING = true;
                }
                break;
            }
        }
    }
}
```

6. Prefixing

   (a) **Restrictions:**

      i. Synchronisations of the form

      $$\text{channel\_name}[.\text{csp\_expression}]^*[?\text{var\_name} \mid !\text{csp\_expression}]^{0..1}$$

      where csp_expression is as in Section 14a.

      ii. Projections are used consistently. For instance, if a channel is used as `c.e`, it cannot be used as `c!e` elsewhere in the specification.

   (b) **Solution:**

      i. Communications are translated to Handel-C communications

      ii. Simple synchronisations are translated to communications of dumb values. A directive indicates if the channel is an input or output

      iii. Synchronisations `c.e` are translated to an access to the `e`-th element of an array `c` of channels. For each type `T` in the system, we declare a constant `T_card` that contains the number of element of elements in that type. This constant is used in the declaration of the array. Besides, signed integers are cast into unsigned integers.

7. Input/Output Buses

   (a) As explained in Section 2.4.6, a channel may be declared as a bus. This indicates that the channel is used as a means from the environment to communicate with the system.

   (b) **Solution:**

      - For every bus, we declare a global constant and a global Handel-C interface. For example, suppose we have declared integers of 7 bits. For a input bus `read` and an output bus `write` (both declared as type `integer` in the CSP_Mspecification), we have the following declaration.

```
integer in_bus_read;
interface bus_in(integer bus_read_value) InBus_read() with{
    data={
        "read_7","read_6","read_5","read_4","read_3","read_2","read_1"}
```

```
};
integer out_bus_write;
interface bus_out() OutBus_write(integer OutPort=out_bus_write) with{
    data={
        "write_7","write_6","write_5","write_4","write_3","write_2","write_1"}
};
```

- Then, we slightly change the use of these channels in the $\mathsf{CSP_M}$ specification. For instance, the translation of an input communication on thus bus `read?n` becomes the following code.

```
in_bus_read=InBus_read.bus_read_value;
```

- Furthermore, the translation of an output communication on the bus `write!value` becomes the following code.

```
out_bus_write=value;
```

- The changes applied to the translation to deal with buses has no effects on the translation of single recursion and mutual recursion. The translation of forced interleaving and multi-synchronised events also doe not need to be changed. It, however, could be optimised to avoid unnecessary use of semaphores on buses.

(c) **Restrictions:**

i. There can be no projection on buses (i.e `read.0`);
ii. A bus can only be either input or output, not both;
iii. A bus cannot be used in channel sets;
iv. A bus cannot be offered as a choice in an external choice;
v. A bus must be used within the system;
vi. A bus must have have exactly one type; that is, the declaration of the channel in the $\mathsf{CSP_M}$ specification declares exactly one type.

8. External Choice

(a) **Restrictions:**

i. Only for prefixing processes.
   A. No Process call
   B. No `STOP`
   C. No `SKIP`
   D. No `P;Q`
   E. No `P \ cs`
   F. No `g & P`
   G. No `P ||| Q`
   H. No `P [| CS |] Q`

I. No `P [ CS1 || CS2 ] Q`

ii. No two branches in an external choice with an on the input variables of the same name.

(b) **Solution:** translate to Handel-C `prialt`

9. Internal choice

(a) **Restrictions:** None

(b) **Solution:** Using another directive, the user can choose the time he wants internal choices to be carried out. For instance, if we give the directive `--!! int_choice at compiletime` to csp2hc, the internal choice `P |~| Q` will be translated to `P();`. If, however, `runtime` is used, the following translation is given as result.

```
random(random_var);
if((random_var%2)==0){ P(); } else{ Q(); }
```

The global variable `random_var` is an integer; it is given to the macro procedure `random` that updates its value to a random value. Next, if this new value is an even number, the process behaves like `P`; it behaves like `Q`, otherwise. This directive is optional: the default value is `compiletime`.

10. Concurrency

(a) **Restrictions:**

i. No multi-synchronised channel is offered in an external choice

ii. For every parallel composition, let *cs* be the channels on which both process really synchronise.

A. For every channel *c* in *cs*, exactly one of the parallel branches must be the writer.
`c?x -> SKIP [| {| c |} |] c?y -> SKIP` and
`c!0 -> SKIP [| {| c |} |] c!0 -> SKIP` are not deadlock free in handel-c but they are in CSP. These cases are removed with this restriction.

B. No branch can be a reader and a writer to *c*
`c!0 -> c?x -> SKIP [| {| c |} |] c?x -> c?y -> SKIP` is not deadlock free in handel-c but it is in CSP. The restriction above would not consider this case; however, this second restriction removes it.

iii. Synchronisation channel sets must be explicit

A. No constants

B. No set functions

C. No productions

D. No set comprehension

(b) **Solution:** The solution can be split into two parts: one for the simple cases and one for the more complicated cases. First, let us determine what we consider a complicated case. These cases are the ones that include multi-way synchronisation (more than two processes taking part in a communication) and forced interleaved events, which are those events that are in the alphabets of both processes, but not in the synchronisation channel set. For a sharing parallel composition `P [| CS |] Q`, this is $(\alpha(\texttt{P}) \cap \alpha(\texttt{Q})) \setminus \texttt{CS}$; for an alphabetised parallel composition `P [ CS1 || CS2 ] Q`, this is $\alpha(\texttt{P}) \cap \alpha(\texttt{Q}) \setminus \texttt{CS1} \cap \texttt{CS2}$; and finally, for an interleaving `P ||| Q`, this is $\alpha(\texttt{P}) \cap \alpha(\texttt{Q})$.

- **Simple cases**

  If there are no forced interleaved events and no multi-way synchronisation, we simply translate these into Handel-C parallelism

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

- **Further cases**

  If, however, we do have either case, we must implement a multiple-access semaphore protocol that controls the access by the branches in a parallel composition to the channels. In order to use this protocol the translation of prefixing, external choice and the arguments of the processes are changed. For instance, in the translation of the CSP_M code in:

```
--!! int_bits 2

datatype ALPHA = a | b
datatype ID = Letter.ALPHA | unknown

-- Abstract parking spot
channel enter, leave

--!! channel enter in within PARKING_SPOT
--!! channel leave in within PARKING_SPOT
PARKING_SPOT = enter -> leave -> PARKING_SPOT

-- Concrete parking spot
channel cash, ticket, change : ID

--!! channel cash in within MACHINE
--!! channel ticket out within MACHINE
--!! channel change out within MACHINE
MACHINE = cash?id -> cash?id -> ticket.id ->
            change.id -> MACHINE

--!! channel enter in within CUSTOMER
--!! channel leave in within CUSTOMER
--!! channel cash out within CUSTOMER
--!! channel ticket in within CUSTOMER
```

```
--!! channel change in within CUSTOMER
--!! arg id ID within CUSTOMER
CUSTOMER(id) =
    (enter -> cash!id ->
        (ticket.id -> change.id -> SKIP
        [] change.id -> ticket.id -> SKIP));
    leave -> CUSTOMER(id)

PAID_PARKING_SPOT =
    (CUSTOMER(Letter.a)
    [| {| cash, ticket, change |} |]
    MACHINE) \ {| cash, ticket, change |}

--!! channel enter out within CAR
--!! channel leave out within CAR
CAR = enter -> leave -> CAR

--!! mainp CAR [| {| enter,leave |} |] PAID_PARKING_SPOT
```

First, we need to identify each one of the branches: we start this identification from the `MAIN` process given by the directive (in this case `CAR [| {| enter,leave |} |] PAID_PARKING_SPOT`); we give an identification to each one of the leaves in the tree presented in Figure 1 (from left to right). For instance, the left most leaf, process `CAR`, is the branch `BRANCH_0`. The implementation of the concept of branch reuses the solution for datatypes we have already presented. Implicitly, csp2hc considers that we have the following datatype in the specification.

```
datatype BRANCH = BRANCH_0 | BRANCH_1 | BRANCH_2 | BRANCH_3 | BRANCH_4
```

In our example, we have the following extra lines of code.

```
#define BRANCH unsigned int 3
#define BRANCH_0 0
#define BRANCH_1 1
...
#define BRANCH_card 5
#define BRANCH_set unsigned int BRANCH_card
#define BRANCH_4_set 0b10000
...
#define BRANCH_set_nil 0b00000
static BRANCH_set BRANCH__set_LUT[BRANCH_card] = { ... };
```

During the translation of a parallel branch, csp2hc knows which branch is being translated and uses its identification to apply the protocol. In every parallel composition, csp2hc translates the left branch first and, before translating the right branch, it updates the current branch identification,

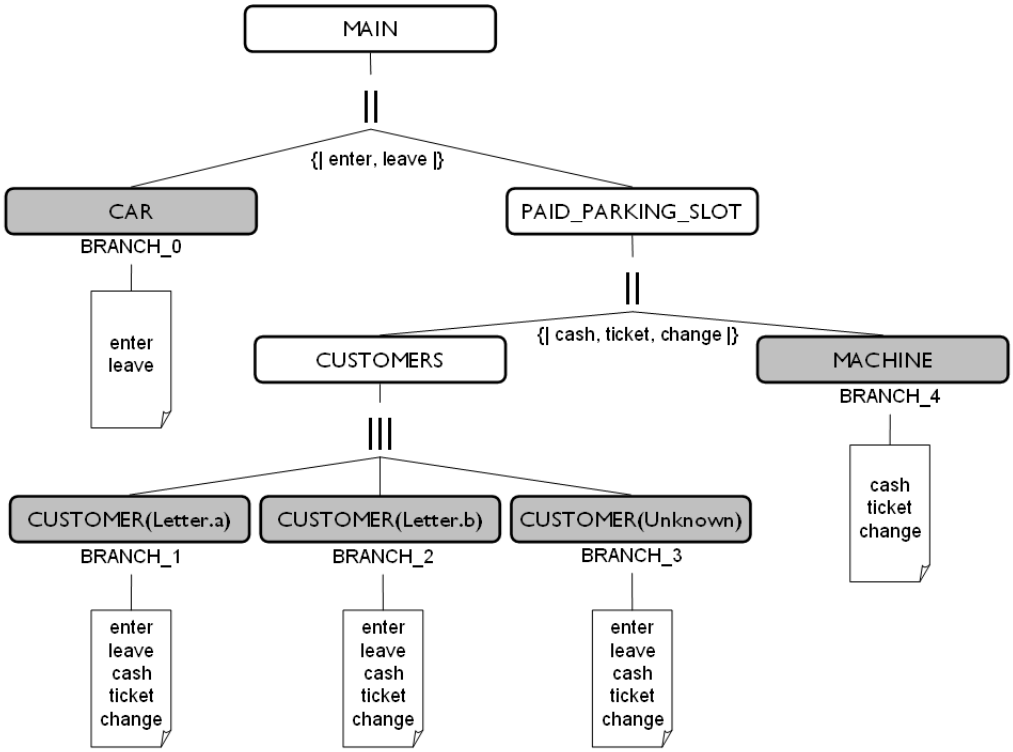

Figure 1: Identifying the branches

which is stored in a local variable BRANCH_ID, by incrementing it with the number of branches identified during the translation of the left branch. Furthermore, every process has an extra argument: it identifies the branch in the parallel composition from which it has been invoked. In our example, we have the following translation for the main process.

```
void main(){
    BRANCH BRANCH_ID;
    BRANCH_ID = 0;
    par{
        {
            CAR(BRANCH_ID+0);
        };
        {
            PAID_PARKING_SPOT(BRANCH_ID+1);
        }
    }
}
```

We declare the local variable that identifies the current branch in the translation and initialise it to zero. The system behaves like a parallel composition between processes CAR and PAID_PARKING_SPOT; they are parametrised by the branch identification. Because before the translation of CAR we have not yet translated anything, we use BRANCH_ID + 0 as argument. Nevertheless, this process is a branch; hence, we use BRANCH_ID + 1 as argument to invoke PAID_PARKING_SPOT. The translation of processes PAID_PARKING_SPOT and CUSTOMERS though are slightly different as we can see in the code below.

```
inline void PAID_PARKING_SPOT(BRANCH BRANCH_ID){
    par{
        {
            CUSTOMERS(BRANCH_ID+0);
        };
        {
            MACHINE(BRANCH_ID+3);
        }
    }
}
inline void CUSTOMERS(BRANCH BRANCH_ID){
    par{
        {
            CUSTOMER(BRANCH_ID+0,ID_Letter_LUT[a]);
        };
        {
```

```
            par{
                {
                    CUSTOMER(BRANCH_ID+1,ID_Letter_LUT[b]);
                };
                {
                    CUSTOMER(BRANCH_ID+2,unknown);
                }
            };
        }
    }
}
```

In the translation of the first one, the local variable `BRANCH_ID` is incremented by three before being given as argument to `MACHINE` because during the translation of `CUSTOMERS`, csp2hc identifies three branches. In the translation of `CUSTOMERS`, the first invocation to `CUSTOMER` does not increment the local variable `BRANCH_ID`; the following invocations, though, do increment it. Now that we have correctly identified each one of the branches that take part in the main behaviour of the system, we need to control the accesses to the channels.
First, we have some constants that are used to express the possible actions on channels.

```
#define ACTION unsigned int 1
#define A_READ 0
#define A_WRITE 1
#define MAX_MULTI_SYNC_SET_card 3
```

The first one, `ACTION` is the type of arguments that represent the action of a branch on a channel. These arguments can assume two values: `A_READ` or `A_WRITE`. Finally, we have a constant that represent the maximum number of possibilities of synchronisation on any channel. In our case, we have 3 because for every channel we have three possibilities of synchronisation to happen.
Next, we use a global static constant and a global variable. The former, `FORCED_SYNC_LUT`, is a two-dimensional array that stores the information about the forced interleaved events, and the later, `CURRENT_SYNC_STATUS_LUT`, is a one dimensional array that stores, for each channel, the identification of the branches that are being allowed to access it. In what follows we describe how we define both of them.
The static constant `FORCED_SYNC_LUT` is an array of arrays of sets of branches (`BRANCH_set`). The first dimension corresponds to the channels used in the system, and the second dimension corresponds to the parallel branches.

```
        static BRANCH_set FORCED_SYNC_LUT[CHANNEL_card][BRANCH_card] = {
```

For every constant channel_c that represents the channel c used in the system and for every constant BRANCH_n that represents the n-th parallel branch, FORCED_SYNC_LUT[channel_c][BRANCH_n] is the set of branches with which the branch n cannot synchronise on c. By way of illustration, we present below the element of the FORCED_SYNC_LUT generated from our example that indicates this information for the channel change (FORCED_SYNC_LUT[channel_

```
static BRANCH_set FORCED_SYNC_LUT[CHANNEL_card][BRANCH_card] ={
    // -----------
    // chan_change
    // -----------
    {

        // --------
        // BRANCH_0
        // --------
        BRANCH_set_nil,
        // --------
        // BRANCH_1
        // --------
        SET_UNION(BRANCH_2_set,SET_UNION(BRANCH_3_set,BRANCH_set_nil)),
        // --------
        // BRANCH_2
        // --------
        SET_UNION(BRANCH_1_set,SET_UNION(BRANCH_3_set,BRANCH_set_nil)),
        // --------
        // BRANCH_3
        // --------
        SET_UNION(BRANCH_1_set,SET_UNION(BRANCH_2_set,BRANCH_set_nil)),
        // --------
        // BRANCH_4
        // --------
        BRANCH_set_nil}
    ,
```

In the code above, the first and last elements are the empty set of branches BRANCH_set_nil. This indicates that branches 0 and 4, which correspond to the processes CAR and MACHINE, respectively (see Figure 1), may synchronise with any other branch on channel change. Nevertheless, branches 1, 2, and 3, which represent each individual CUSTOMER, have mutually exclusive access to this channel. For generalisation, we implement the set {BRANCH_2, BRANCH_3} as the union of the singletons BRANCH_2_set and BRANCH_3_set with the empty set of branches.

In our example, the processes CUSTOMER cannot synchronise in any event in their alphabets; and this is the only enforcement needed. For this reason, since the alphabet of CUSTOMER contains all the channels in the system, the elements that correspond to the other channels in FORCED_SYNC_LUT are identical to the one presented above.

```
    // chan_enter
    { ... } ,
    // chan_ticket
    { ... } ,
    // chan_leave
    { ... } ,
    // chan_cash
    { ... }
};
```

The dimension of the array of sets of branches CURRENT_SYNC_STATUS_LUT is the number of channels used in the system. For every constant channel_c, the element CURRENT_SYNC_STATUS_LUT[channel_c] is the set of branches that are currently allowed to access the channel c. Since no branch is initially allowed to access any channel, every element of this variable is initially set to zero.

```
BRANCH_set CURRENT_SYNC_STATUS_LUT[CHANNEL_card] ={

    // -----------
    // chan_change
    // -----------
    BRANCH_set_nil,
    // ----------
    // chan_enter
    // ----------
    BRANCH_set_nil,
    // -----------
    // chan_ticket
    // -----------
    BRANCH_set_nil,
    // ----------
    // chan_leave
    // ----------
    BRANCH_set_nil,
    // ---------
    // chan_cash
    // ---------
    BRANCH_set_nil};
```

The array HAS_WRITER_LUT[CHANNEL_card] is an array of boolean: for each channel, it registers if there exists any writer with access to it. Its dimension is also the number of channels used in the system. Since no branch is initially allowed to access any channel, every element of this variable is initially set to false.

```
boolean HAS_WRITER_LUT[CHANNEL_card] ={

    // -----------
    // chan_change
    // -----------
    false,
    // ----------
    // chan_enter
    // ----------
    false,
    // -----------
    // chan_ticket
    // -----------
    false,
    // ----------
    // chan_leave
    // ----------
    false,
    // ---------
    // chan_cash
    // ---------
    false};
```

A last static constant, MULTI_SYNC_LUT is an array of arrays of sets of branches (BRANCH_set). The first dimension corresponds to the channels used in the system, and the second dimension corresponds to the maximum number of parallel branches in the system.

```
static BRANCH_set MULTI_SYNC_LUT[CHANNEL_card][MAX_MULTI_SYNC_SET_card] ={
```

For every constant channel_c that represents the channel c used in the system, MULTI_SYNC_LUT[channel_c][i], is an array of sets of branches: each element of this array corresponds to a possible synchronisation on c. By way of illustration, we present below the element of the MULTI_SYNC_LUT generated from our example that indicates this information for the channel change (MULTI_SYNC_LUT[channel_change]).

```
    // -----------
    // chan_change
    // -----------
```

```
{
    SET_UNION(BRANCH_2_set,SET_UNION(BRANCH_4_set,BRANCH_set_nil)),
    SET_UNION(BRANCH_3_set,SET_UNION(BRANCH_4_set,BRANCH_set_nil)),
    SET_UNION(BRANCH_1_set,SET_UNION(BRANCH_4_set,BRANCH_set_nil))}
,
```

In the code above, we indicate that there are three possible synchronisations on channel `c`: branches `2` and `4` (`CUSTOMER(Letter.b)` and `MACHINE`), branches `3` and `4` (`CUSTOMER(Unknown)` and `MACHINE`), or branches `1` and `4` (`CUSTOMER(Letter.a)` and `MACHINE`). In this particular example, we do not have multi-synchronisation; however, this would be indicated here by having any set of branches with cardinality higher than 3. The behaviour of the protocol, however, does not change because of this.
In the same way, we have the definition of `MULTI_SYNC_LUT` for the other channels.

```
// ----------
// chan_enter
// ----------
{
    SET_UNION(BRANCH_0_set,SET_UNION(BRANCH_1_set,BRANCH_set_nil)),
    SET_UNION(BRANCH_0_set,SET_UNION(BRANCH_3_set,BRANCH_set_nil)),
    SET_UNION(BRANCH_0_set,SET_UNION(BRANCH_2_set,BRANCH_set_nil))}
,
// -----------
// chan_ticket
// -----------
{
    SET_UNION(BRANCH_2_set,SET_UNION(BRANCH_4_set,BRANCH_set_nil)),
    SET_UNION(BRANCH_3_set,SET_UNION(BRANCH_4_set,BRANCH_set_nil)),
    SET_UNION(BRANCH_1_set,SET_UNION(BRANCH_4_set,BRANCH_set_nil))}
,
// ----------
// chan_leave
// ----------
{
    SET_UNION(BRANCH_0_set,SET_UNION(BRANCH_1_set,BRANCH_set_nil)),
    SET_UNION(BRANCH_0_set,SET_UNION(BRANCH_3_set,BRANCH_set_nil)),
    SET_UNION(BRANCH_0_set,SET_UNION(BRANCH_2_set,BRANCH_set_nil))}
,
// ---------
// chan_cash
// ---------
{
    SET_UNION(BRANCH_2_set,SET_UNION(BRANCH_4_set,BRANCH_set_nil)),
```

```
        SET_UNION(BRANCH_3_set,SET_UNION(BRANCH_4_set,BRANCH_set_nil)),
        SET_UNION(BRANCH_1_set,SET_UNION(BRANCH_4_set,BRANCH_set_nil))}
};
```

The look-up tables `FORCED_SYNC_LUT` and `CURRENT_SYNC_STATUS_LUT` are global variables and can be accessed by any part of the code. Nevertheless, concurrent access to them may lead to unexpected behaviours. Fortunately, Handel-C provides semaphores; we define one semaphore for each one of these look-up tables.

```
sema CURRENT_SYNC_STATUS_LUT_SEMA;
sema HAS_WRITER_LUT_SEMA;
```

Finally, we have an array of Handel-C boolean signals. They behave like wires: changes to their value last one clock cycle only; their value return to the original one in the next clock cycle. In our case, we set false as the original value. The idea is to use these signals as a means of communication between the readers and the writer if the reader withdraws on a multi-synchronisation (by setting the element in the array of signals that corresponds to the channel that is being withdrawn to true.

```
signal <boolean> WITHDRAW_SIGNALS[CHANNEL_card] ={

    // -----------
    // chan_change
    // -----------
    false,
    // ----------
    // chan_enter
    // ----------
    false,
    // -----------
    // chan_ticket
    // -----------
    false,
    // ----------
    // chan_leave
    // ----------
    false,
    // ---------
    // chan_cash
    // ---------
    false};
```

The Protocol The basic idea if that, before any communication on a channel, a process (branch) must join the corresponding semaphore, and finally, after synchronising on the channel, releases their corresponding semaphores. For instance, a branch BRANCH_ID, reads from channel c, as follows.

```
JOIN_SEMAPHOR(BRANCH_ID,chan_c,A_READ);
c? y;
LEAVE_SEMAPHOR(BRANCH_ID,chan_c,A_READ);
```

If the branch is the writer, the only difference, is that, before actually writing to the channel, it confirms with the readers that they are actually willing to read, which means that they don't have withdrawn (at this point the branch actually waits until all readers have confirmed.

```
JOIN_SEMAPHOR(BRANCH_ID,chan_c,A_WRITE);
CONFIRM_READERS(BRANCH_ID,chan_c);
c!v;
LEAVE_SEMAPHOR(BRANCH_ID,chan_c,A_READ);
```

In what follows, we explain in details the auxiliary functions and macros that are used in the implementation of the protocol. We will take a bottom-up approach: starting from the simplest macros and functions we will build the final functions JOIN_SEMAPHOR, CONFIRM_READERS, and LEAVE_SEMAPHOR.

Let us start with the macro expressions. The firest one indicates if a branch can have access to a channel regarding the forced interleaved events.

```
macro expr IS_AVAILABLE(BRANCH_B,CHANNEL_C) =
    (IS_EMPTY_SET(SET_INTER(FORCED_SYNC_LUT[CHANNEL_C][BRANCH_B],
                            CURRENT_SYNC_STATUS_LUT[CHANNEL_C])));
```

For us, a branch b can have access to a channel, if there is no branch with which b may not synchronise on c having access to c. Hence, we check if the intersection between the set of branches with which b may not synchronise on c and the set of current channels having access to c is empty.

The next macro expression checks if there is no writer having access to the channel. This can be simply done by checking the element in the array HAS_WRITER that corresponds to the given channel as follows.

```
macro expr NO_WRITER(CHANNEL_C) = (!HAS_WRITER_LUT[CHANNEL_C]);
```

The macro MULTI_SYNC_READY returns true if the given set of branches corresponds to an acceptable synchronisation (including multi-synchronisation) on the given channel. It does so by checking every possible synchronisation for the given channel as follows.

```
macro expr MULTI_SYNC_READY(CHANNEL_C,BRANCH_SET_S) =
    (MULTI_SYNC_LUT[CHANNEL_C][0]==BRANCH_SET_S)
    ||
    (MULTI_SYNC_LUT[CHANNEL_C][1]==BRANCH_SET_S)
```

```
    ||
    (MULTI_SYNC_LUT[CHANNEL_C][2]==BRANCH_SET_S);
```

The next macro indicates if, by including a given branch, the synchronisation on a given channel is ready. This is checked by giving a union of the current branches that have access to the giving channel with the singleton that contains the given set as argument to MULTI_SYNC_READY.

```
macro expr MULTI_SYNC_READY_WITH_ME(WRITER_BRANCH_B,CHANNEL_C) =
    MULTI_SYNC_READY(CHANNEL_C,
                     SET_UNION(CURRENT_SYNC_STATUS_LUT[CHANNEL_C],
                               BRANCH__set_LUT[WRITER_BRANCH_B]));
```

As we did for the synchronisation regarding forced interleaving events, we have a macro expression that indicates if a given branch may have access to the a given channel to make a given action regarding multi-synchronisation.

```
macro expr IS_SYNC_READY(BRANCH_B,CHANNEL_C,ACTION_A) =
    ( (ACTION_A==A_READ)
      ||(ACTION_A==A_WRITE
         && MULTI_SYNC_READY_WITH_ME(BRANCH_B,CHANNEL_C)));
```

In our protocol, there are only two situations in which this is true: the branch wants to read from the channel or the branch wants to write and all readers are ready. This enforces the writer to be the last to have access to a given channel. Of course, after the writer has joined the channel, it might be possible that readers withdraw the synchronisation. This, however, is also dealt by our protocol as we will discuss latter. For the moment, let us continue with the description of the auxiliary macro expressions.
The set membership for branches is implement by the following macro expression.

```
macro expr BRANCH_MEMBER(BRANCH_B,BRANCH_SET_S) =
    (SET_UNION(BRANCH__set_LUT[BRANCH_B],BRANCH_SET_S)
     ==
     BRANCH_SET_S);
```

It is used by the next macro expression, which determines if a given branch already has access to a given channel.

```
macro expr IS_ALREADY(BRANCH_B,CHANNEL_C) =
    (BRANCH_MEMBER(BRANCH_B,CURRENT_SYNC_STATUS_LUT[CHANNEL_C]));
```

The macro expression MAY_JOIN indicates if a given branch may have access to a given channel regarding both forced interleaved events and multi-synchronisation.

```
macro expr MAY_JOIN(BRANCH_B,CHANNEL_C,ACTION_A) =
    ( (IS_ALREADY(BRANCH_B,CHANNEL_C))
       ||
      (IS_AVAILABLE(BRANCH_B,CHANNEL_C)
       && IS_SYNC_READY(BRANCH_B,CHANNEL_C,ACTION_A)));
```

A branch BRANCH_B may only have access to a channel CHANNEL_C to perform an action ACTION_A either if it already has access to the channel, or if it may have access to the channel regarding both separate concern: forced interleaved events (IS_AVAILABLE) and multi-synchronisation (IS_SYNC_READY). Handel-C semaphores can only be accessed via two constructs. The first one, trysema(s), checks if a semaphore s is already taken. If it is not taken, it takes this semaphore and returns one; it returns zero otherwise. The second one, releasesema(s) releases the semaphore s. The following procedure tries to obtain a semaphore until it succeeds. Since semaphores cannot be passed directly to functions, we pass them by reference.

```
inline void LOCK_VARIABLE(sema *sema_S){
    while(trysema(*sema_S)==0) delay;
}
```

The next function, TRY_JOIN_SEMAPHOR(BRANCH BRANCH_B,CHANNEL CHANNEL_C, ACTION ACTION_A), tries to give ACTION_A access to branch BRANCH_B to channel CHANNEL_C. If the branch is entitled to join the channel, it locks the variables i n order to update them. Then, it verifies if the situation has not changed during variable locks by checking again. If everything worked fine, it adds itself to the array that stores the synchronisation current status. Furthermore, if it is the writer, it also updates the writers array. In this case, the return of this function is true. If, however, access was not granted, it returns false. Before returning, however, the function releases the semaphores.

```
inline boolean TRY_JOIN_SEMAPHOR (BRANCH BRANCH_B,
                                  CHANNEL CHANNEL_C,
                                  ACTION ACTION_A){
    boolean HAS_JOINED;
    if(MAY_JOIN(BRANCH_B,CHANNEL_C,ACTION_A)){
        LOCK_VARIABLE(&CURRENT_SYNC_STATUS_LUT_SEMA);
        LOCK_VARIABLE(&HAS_WRITER_LUT_SEMA);
        if(MAY_JOIN(BRANCH_B,CHANNEL_C,ACTION_A)){
            CURRENT_SYNC_STATUS_LUT[CHANNEL_C] =
                SET_UNION(CURRENT_SYNC_STATUS_LUT[CHANNEL_C],
                          BRANCH__set_LUT[BRANCH_B]);
            if(ACTION_A==A_WRITE){
                HAS_WRITER_LUT[CHANNEL_C] = true;
            }
            HAS_JOINED = true;
        }
        else{
            HAS_JOINED = false;
        }
```

```
        releasesema(CURRENT_SYNC_STATUS_LUT_SEMA);
        releasesema(HAS_WRITER_LUT_SEMA);
    }
    else{
        HAS_JOINED = false;
    }
    return HAS_JOINED;
}
```

The procedure JOIN_SEMAPHOR is used by a branch to obtain access to a channel. It continually invokes the function TRY_JOIN_SEMAPHOR while it returns false.

```
// ------------------------------------------------------------
// Indicates that a given branch has access to a given channel
// ------------------------------------------------------------
inline void JOIN_SEMAPHOR(BRANCH BRANCH_B,CHANNEL CHANNEL_C,
                          ACTION ACTION_A){
    boolean HAS_JOINED;
    HAS_JOINED = TRY_JOIN_SEMAPHOR(BRANCH_B,CHANNEL_C,ACTION_A);
    while(!HAS_JOINED){
        HAS_JOINED = TRY_JOIN_SEMAPHOR(BRANCH_B,CHANNEL_C,ACTION_A);
        delay;
    }
}
```

The next function updates the synchronisation status look-up table removing the branch from the set that stores the branches that currently have access to that channel. Furthermore, if the branch is the writer, it also removed from the writers look-up table by setting the corresponding value to false.
As usual, it locks the semaphores of both look-up tables, changes them accordingly, and releases their semaphores.

```
inline void LEAVE_SEMAPHOR(BRANCH BRANCH_B,CHANNEL CHANNEL_C,
                           ACTION ACTION_A){
    LOCK_VARIABLE(&CURRENT_SYNC_STATUS_LUT_SEMA);
    LOCK_VARIABLE(&HAS_WRITER_LUT_SEMA);
    CURRENT_SYNC_STATUS_LUT[CHANNEL_C] =
        SET_DIFF(CURRENT_SYNC_STATUS_LUT[CHANNEL_C],
                 BRANCH__set_LUT[BRANCH_B]);
    if(ACTION_A==A_WRITE){
        HAS_WRITER_LUT[CHANNEL_C] = false;
    }
```

```
    releasesema(CURRENT_SYNC_STATUS_LUT_SEMA);
    releasesema(HAS_WRITER_LUT_SEMA);
}
```

The function `TRY_CONFIRM_READERS(BRANCH BRANCH_B,CHANNEL CHANNEL_C)` tries to confirm that the synchronisation on channel `CHANNEL_C` will be ready if branch `BRANCH_B` joins the synchronisation. As usual, it locks the used look-up table.

```
inline boolean TRY_CONFIRM_READERS(BRANCH BRANCH_B,CHANNEL CHANNEL_C){
    boolean ARE_READERS_READY;
    LOCK_VARIABLE(&CURRENT_SYNC_STATUS_LUT_SEMA);
    ARE_READERS_READY = MULTI_SYNC_READY_WITH_ME(BRANCH_B,CHANNEL_C);
    releasesema(CURRENT_SYNC_STATUS_LUT_SEMA);
    return ARE_READERS_READY;
}
```

After confirming that the readers are ready, the writer is able to write to a channel. In some cases, however, a reader might withdraw the synchronisation. This will make the writer wait for the reader that withdrew. When this reader is willing again to synchronise, the look-up table will be changed and the writer is be able to write in the next clock cycle. However, he must wait for the reader that returned to the synchronisation. For that, we make use of a procedure `SLEEP` that does nothing but takes `n` clock cycles to complete.

```
inline void SLEEP(unsigned int 4 SLEEP_TIME){
    while(SLEEP_TIME > 0){
        delay;
        SLEEP_TIME--;
    }
}
```

This procedure is used in the last auxiliary procedure, which only terminates when it is confirmed that the synchronisation on a given channel is ready (provided a given branch joins the synchronisation), and that there was no withdraw between reading the corresponding look-up table and the end of the procedure. This procedure uses a couple of local variables: `CONFIRMED` indicates that the procedure should terminate (initially, it is set to `false` and the procedure body is a loop on this variable); `CONFIRM_FINISHED` indicates that the procedure has checked if the synchronisation is ready; `WITHDRAW_IN_CONFIRMATION` indicates that during the confirmation of the synchronisation status, some branch withdrew the synchronisation; similarly, we have variables to indicate that withdraws have happened during the internal communication of the procedure (`WITHDRAW_IN_COMM`) or during the procedure was setting `CONFIRM_FINISHED` to `true`. All variables are initially set to `false` on each iteration of the

loop.

```
inline void CONFIRM_READERS(BRANCH BRANCH_B,CHANNEL CHANNEL_C){
    boolean CONFIRMED;
    boolean CONFIRM_FINISHED;
    boolean WITHDRAW_IN_CONFIRMATION;
    boolean WITHDRAW_IN_COMM;
    boolean WITHDRAW_IN_FINISHED;
    chan boolean confirmed_end;

    CONFIRMED = false;
    while(!CONFIRMED){
        CONFIRM_FINISHED = false;
        WITHDRAW_IN_CONFIRMATION = false;
        WITHDRAW_IN_COMM = false;
        WITHDRAW_IN_FINISHED = false;
```

The main idea here, is that in each clock cycle within the procedure, the withdraw signal for the given channel is checked. These variables are used to store if any signal happened. The main behaviour of the procedure is a parallel composition: one process repeatedly tries to confirm that the synchronisation is ready. When it manages to do so, it communicates with the other process via channel `confirmed_end`. In other to check the withdraw signal during this communication, whcih takes one clock cycle, it sends the current signal value via the channel.

```
        par{
            {
                CONFIRMED = TRY_CONFIRM_READERS(BRANCH_B,CHANNEL_C);
                while(!CONFIRMED){
                    CONFIRMED = TRY_CONFIRM_READERS(BRANCH_B,CHANNEL_C);
                    delay;
                };
                SLEEP(10);
                confirmed_end!WITHDRAW_SIGNALS[CHANNEL_C];
            };
```

The other process, repeatedly waits for the indication from the first process that the confirmation has happened. At each iteration, if the communication does not happen, the process simply checks the withdraw signal. If, however, the communication does happen, we stop the loop by setting `CONFIRM_FINISHED` to `true`. Since this assignment takes one clock cycle, we also need to check the withdraw signal during this clock cycle.

```
            {
                while(!CONFIRM_FINISHED){
                    prialt{
                        case confirmed_end?WITHDRAW_IN_COMM:{
```

```
                        par{
                            CONFIRM_FINISHED = true;
                            WITHDRAW_IN_FINISHED =
                                WITHDRAW_SIGNALS[CHANNEL_C];
                        }
                    };
                    break;
                    default:{
                        if(WITHDRAW_SIGNALS[CHANNEL_C]){
                            WITHDRAW_IN_CONFIRMATION = true;
                        };
                        delay;
                    };
                    break;
                }
            }
        }
    };
```

Finally, the procedure has actually terminated (`CONFIRMED = true`) only if there was no withdraw during confirmation (`!WITHDRAW_IN_CONFIRMATION`), no withdraw during the communication on `confirmed_end` (`!WITHDRAW_IN_COMM`), no withdraw whilst the procedure was setting the variable `CONFIRM_FINISHED` to `true` (`!WITHDRAW_IN_FINISHED`), and during the return of the function (by checking the signal again).

```
    CONFIRMED = !WITHDRAW_IN_CONFIRMATION
                && !WITHDRAW_IN_COMM
                && !WITHDRAW_IN_FINISHED
                && !WITHDRAW_SIGNALS[CHANNEL_C];
}
}
```

Every process must use the procedures `JOIN_SEMAPHOR`, `LEAVE_SEMAPHOR`, and `CONFIRM_READERS` (if it is the writer) to access a channel.
By way of illustration , we present below the new translation of the process `MACHINE`, which is similar to the translation we presented before, but adds `JOIN_SEMAPHOR` and `LEAVE_SEMAPHOR` before and after, respectively, every access to a channel `c`.

```
inline void MACHINE(BRANCH BRANCH_ID){
    boolean KEEP_LOOPING;
    KEEP_LOOPING = true;
    while(KEEP_LOOPING){
        KEEP_LOOPING = false;
        seq{
            seq{
```

```
                ID id;
                JOIN_SEMAPHOR(BRANCH_ID+0,chan_cash,A_READ);
                cash? id;
                LEAVE_SEMAPHOR(BRANCH_ID+0,chan_cash,A_READ);
                seq{
                    JOIN_SEMAPHOR(BRANCH_ID+0,chan_ticket,A_WRITE);
                    ticket[id]!syncout;
                    LEAVE_SEMAPHOR(BRANCH_ID+0,chan_ticket,A_WRITE);
                    seq{
                        JOIN_SEMAPHOR(BRANCH_ID+0,chan_change,A_WRITE);
                        change[id]!syncout;
                        LEAVE_SEMAPHOR(BRANCH_ID+0,chan_change,A_WRITE);
                        KEEP_LOOPING = true;
                    };
                };
            }
        }
    }
}
```

The confirmation of the status of the synchronisation before writing to a channel (`CONFIRM_READERS`) is only needed for multi-synchronised channels. In this particular example, we do not have multi-synchronisation; hence, no channel is multi-synchronised. That is why we do not have this confirmation before writing to channels `ticket` and `change`. If this were the case, we would have the following code instead (supposing a multi-synchronisation on `ticket`, but not on `change`).

```
...
JOIN_SEMAPHOR(BRANCH_ID+0,chan_ticket,A_WRITE);
CONFIRM_READERS(BRANCH_ID+0,chan_ticket);
ticket[id]!syncout;
LEAVE_SEMAPHOR(BRANCH_ID+0,chan_ticket,A_WRITE);
seq{
    JOIN_SEMAPHOR(BRANCH_ID+0,chan_change,A_WRITE);
    change[id]!syncout;
    LEAVE_SEMAPHOR(BRANCH_ID+0,chan_change,A_WRITE);
...
```

The changes in the translation of process `CAR` and the part of the translation of process `CUSTOMER` are very similar. Nevertheless, the translation of the external choice is considerably changed. The reason is that before offering any channel in an `prialt` statement, the branches need to request access to them. Our protocol gradually requests access to each one of the channels and offers only those channels who accesses have been granted.

Afterwards, the branches also need to indicate that they no longer need to access the channels.
Our solution uses two auxiliary local variables. The first one, DONE_EVENTS_0, is a channel set that stores any events that have already taken place in the external choice; hence, if this set is empty, no events have happened and we must still offer an external choice. The second one, HAS_JOINED_0 is an array of boolean whose length is the number of channels used in the system. For every channel c, HAS_JOINED_0[chan_c] indicates if this branch has been allowed to access the channel c, or not. In order to avoid variable name clashes in nested external choices, we index these variable with an unique identifier (_0 in this example).

```
CHANNEL_set DONE_EVENTS_0;
boolean HAS_JOINED_0[CHANNEL_card];
```

Initially, no events have happened and no permission has been given to access any channel. Hence, the set of done events is empty and every element in the array HAS_JOINED_0 is false.

```
DONE_EVENTS_0 = CHANNEL_set_nil;
HAS_JOINED_0[chan_cash] = false;
HAS_JOINED_0[chan_leave] = false;
HAS_JOINED_0[chan_ticket] = false;
HAS_JOINED_0[chan_enter] = false;
HAS_JOINED_0[chan_change] = false;
```

In our example we have the choice ticket.id -> ... [] change.id -> .... This external choice is implemented as follows. For each channel offered in the external choice, ticket and change, the branch requests access to that channel and then it behaves like a loop that stops if any event happens or the access to that channel has been granted. In the end of each iteration, the branch request access to the channel again. The first channel in the choice is ticket: the branch request access to it and then starts a loop that stops only if another event happens during the loop or if the access to ticket is granted to this branch.

```
HAS_JOINED_0[chan_ticket] = TRY_JOIN_SEMAPHOR(BRANCH_ID,chan_ticket);
while(IS_EMPTY_SET(DONE_EVENTS_0) && !HAS_JOINED_0[chan_ticket]){
    // begin body of iteration
    ...
    // end body of iteration
    HAS_JOINED_0[chan_ticket] = TRY_JOIN_SEMAPHOR(BRANCH_ID,chan_ticket);
}
// begin loop for chan_change
...
// end loop for chan_change
```

In the body of the iteration, the branch sequentially requests access to every other channel in the choice; this creates a (possibly nested) alternation whose body is a `prialt` statement that offers only those channels which the branch has been allowed to access. The next channel in our example is `change`; hence, the branch need to request access to it.

```
// begin body of iteration
HAS_JOINED_0[chan_change] = TRY_JOIN_SEMAPHOR(BRANCH_ID,chan_change);
```

Next, since we do not have other channels being offered, we have the alternation. If the requested access to the channel `change` has not been granted, this branch has access to no channel yet and the `prialt` statement only delays (`default` case). Nevertheless, if the access to `change` was granted, this branch has access only to this channel which is offered in the `prialt` statement. If any concurrent process is willing to communicate on this channel, the communication happens; this updates the set of events that happened causing the loop to stop and indicates to the protocol that this branch is no longer accessing the channel. If, however, there is no process willing to communicate on `change`, the `default` case, a `delay`, takes place.

```
if(!HAS_JOINED_0[chan_change]){
    prialt{
        default:delay;
        break;
    }
}
else{
    prialt{
        case change[CUSTOMER_local_id]?syncin :{
            DONE_EVENTS_0 = SET_UNION(DONE_EVENTS_0,chan_change_set);
            LEAVE_SEMAPHOR(BRANCH_ID+0,chan_change,A_READ);
            seq{
                JOIN_SEMAPHOR(BRANCH_ID+0,chan_ticket,A_READ);
                ticket[CUSTOMER_local_id]?syncin;
                LEAVE_SEMAPHOR(BRANCH_ID+0,chan_ticket,A_READ);
            }
        };
        break;
        default:delay;
        break;
    }
}
HAS_JOINED_0[chan_ticket] =
```

```
        TRY_JOIN_SEMAPHOR(BRANCH_ID+0,chan_ticket,A_READ);
    // end body of iteration
```

This loop may end either because an event happened or because the access to the channel `change` has been granted.
The initialisation and the condition of the next loop, which relates to the channel `change` is very similar to the previous loop.

```
// begin loop for chan_change
HAS_JOINED_0[chan_change] = TRY_JOIN_SEMAPHOR(BRANCH_ID,chan_change);
while(IS_EMPTY_SET(DONE_EVENTS_0) && !HAS_JOINED_0[chan_change]){
```

Nevertheless, there are no other channels in the choice; hence, there is no alternation but only a choice that offers the only channel to which the access has been granted so far, `ticket`, and the `default` delay.

```
prialt{
    case ticket[CUSTOMER_local_id]?syncin :{
        DONE_EVENTS_0 = SET_UNION(DONE_EVENTS_0,chan_ticket_set);
        LEAVE_SEMAPHOR(BRANCH_ID+0,chan_ticket,A_READ);
        seq{
            JOIN_SEMAPHOR(BRANCH_ID+0,chan_change,A_READ);
            change[CUSTOMER_local_id]?syncin;
            LEAVE_SEMAPHOR(BRANCH_ID+0,chan_change,A_READ);
        }
    };
    break;
    default:delay;
    break;
}
HAS_JOINED_0[chan_change] =
    TRY_JOIN_SEMAPHOR(BRANCH_ID+0,chan_change,A_READ);
// end loop for chan_change
```

Once again, this loop may end either because an event happened or because the access to the channel `change` has been granted. If the previous loops did not stop their execution because an event happened, then an offer to all channel in the choice must be made; however, it must only be made if no events in the choice happened before. After the choice has been made, the branch must release all the channels in the choice.

```
if(IS_EMPTY_SET(DONE_EVENTS_0)){
    prialt{
        case ticket[CUSTOMER_local_id]?syncin :{
            DONE_EVENTS_0 = SET_UNION(DONE_EVENTS_0,chan_ticket_set);
            LEAVE_SEMAPHOR(BRANCH_ID+0,chan_ticket,A_READ);
            LEAVE_SEMAPHOR(BRANCH_ID+0,chan_change,A_READ);
```

```
            seq{
                JOIN_SEMAPHOR(BRANCH_ID+0,chan_change,A_READ);
                change[CUSTOMER_local_id]?syncin;
                LEAVE_SEMAPHOR(BRANCH_ID+0,chan_change,A_READ);
            }
        };
        break;
        case change[CUSTOMER_local_id]?syncin :{
            DONE_EVENTS_0 = SET_UNION(DONE_EVENTS_0,chan_change_set);
            LEAVE_SEMAPHOR(BRANCH_ID+0,chan_ticket,A_READ);
            LEAVE_SEMAPHOR(BRANCH_ID+0,chan_change,A_READ);
            seq{
                JOIN_SEMAPHOR(BRANCH_ID+0,chan_ticket,A_READ);
                ticket[CUSTOMER_local_id]?syncin;
                LEAVE_SEMAPHOR(BRANCH_ID+0,chan_ticket,A_READ);
            }
        };
        break;
    }
}
```

This concludes the translation of the external choice in our example. There is, however, a small further change that must be made in the presence of multi-synchronisation. Immediately after the choice has been made, any with draws on multi-synchronised channels must be indicated. This is done by setting the corresponding withdraw channel to `true`. For example, if the channel `change` were multi-synchronised, the choice for `ticket` in the code above would be as follows.

```
        ...
        case ticket[CUSTOMER_local_id]?syncin :{
            WITHDRAW_SIGNALS[chan_change] = true;
            DONE_EVENTS_0 = SET_UNION(DONE_EVENTS_0,chan_ticket_set);
            LEAVE_SEMAPHOR(BRANCH_ID+0,chan_ticket,A_READ);
            LEAVE_SEMAPHOR(BRANCH_ID+0,chan_change,A_READ);
```

In cases where more the one signal must be sent, this is done in parallel. Any changes to this protocol might reflect the following translation.

- Events
- External choice
- Parallel Compositon
- Mutual Recursion
- Tail Recursion

One further possible solution would be to implement multi-synchronisation using a protocol that uses a centralised controller like the one presented in my PhD thesis. However, during tests, we found out that this would increment a lot the size of the code. We have made a prototype of the simplest version of the protocol in which we do not have multi-synchronised events on external choices. For this simplest case, I collected some data that might support the case of not translating external choice I mentioned in the previous e-mail.
If we had multi-synchronisation support direct in the Handel-C language, the compilation of a simple program that has three processes (one that writes on a channel c and two that read from it) would lead to:

– `336 NAND Gates (32 Flip-Flops) Execution time` (1 clock-cycle)

By implementing Jim's protocol for a multi-synchronisation between the same three processes we would have:

– `5541 NAND Gates (339 Flip-Flops) Execution time` (81 clock-cycle)

Our protocol for a multi-synchronisation between the same three processes we have:

– `12915 NAND Gates (573 Flip-Flops) Execution time` (46 clock-cycle)

We must emphasise that our protocol does take into account external choice on multi-synchronised channels, and that is why we have a larger result. We, however, are much faster than the old protocol.

11. Hiding

(a) **Restrictions:** In order to ignore the hiding $P \setminus cs$ the following conditions must be satisfied:

i. For every channel $c \in cs$ that is the alphabet of P we must guarantee that there exists a communication on c between two parallel branches in P. For this, we consider the hiding only of those channels that are in both cs and in the alphabet of P and build a stack that stores the hidden channels that must be considered in nested hiding (in the following, call these CCS).

A. For every parallel composition $P[|\ sync\ |]Q$, we have that:
$S_1 = (Visible(P) \cap Written(P)) \cap sync \cap (Visible(Q) \cap Read(Q))$
$S_2 = (Visible(Q) \cap Written(Q)) \cap sync \cap (Visible(P) \cap Read(P))$
If $CCS \subseteq S_1 \cup S_2$ the all right (ignore the hiding on CCS in the remaining analysis); otherwise, we have an error.

B. For every parallel composition $P[|\ sync\ |]Q$, increment a parallelism counter PC. If we find an event on c: if $c \in CCS \wedge PC \leq 0$ then we have an error.

C. In order to avoid the insertion of communication that should not happen in the specification, for every composition $P \llbracket sync \rrbracket Q$:
$Hidden(P) \cap sync \cap Alpha(Q)$
$Hidden(Q) \cap sync \cap Alpha(P)$

(b) **Solution:** Simply ignore

12. Datatypes

(a) **Restrictions:** No mutually recursive datatypes

(b) Simple datatypes

i. **Restrictions:** None

ii. **Solution:** the type is declared as an `unsigned int i`, where `i` is the number of bits need to represent the cardinality of the type. Each element of the datatype corresponds to an integer value (starting from 0). We also declare the cardinality of the type. By way of illustration, `datatype Alpha = a | b` is translated as follows.

```
#define Alpha unsigned int 1
#define b 0
#define a 1
#define Alpha_card 2
```

(c) Complex datatypes

i. **Restrictions:** None

ii. **Solution:** constructors are seen as functions. For each element in the domain of the constructor there exists a corresponding value in the enumeration that corresponds to the datatype. For each possible constructor in the system, create a lookup table that, given the values of the domain of the constructor, returns the corresponding value in the enumeration of the datatype. For instance, let us consider the following datatype:

```
datatype Char = Letter.Alpha | Number.Int
```

The translation of this datatype is presented below:

```
#define Char unsigned int 3
#define Number_neg_1 0
#define Number_neg_2 1
#define Number_1 2
#define Number_0 3
#define Letter_a 4
#define Letter_b 5
#define Char_card 6 static Char
Char_Number_LUT[integer_card] ={
    Number_0 ,Number_1 ,Number_neg_2 ,Number_neg_1};
static Char Char_Letter_LUT[Alpha_card] ={
    Letter_b ,Letter_a};
```

(d) `Int`

i. **Restrictions:** None

ii. **Solution:** Declare a constant `integer` that represents the integer within the Handel-C code as an `int` of `n` bits, where `n` is given as a directive and represents the number of bits in the representation of integers within the system.

(e) `Bool`

i. **Restrictions:** None

ii. **Solution:** Declare the constants `true` as 1 and `false` as 0, and declare `boolean` as `unsigned int 1`

(f) Sets

i. Set expressions can be used in constant and function definitions, and as processes arguments.

ii. Set expressions can also be used in channel declaration and datatype declarations.

iii. **Restrictions:**

A. Sets cannot be used in a channel communication.

B. Sets of sets are not accepted neither as a type nor as a expression.

iv. **Solution:** we use a bit presentation for sets. For every type `T` in the system we declare a constant `T_set unsigned int T_card`; furthermore, for every element `e`, we declare a singleton set `e_s`; finally, we declare the empty set `T_nil_s`. For instance, for datatype `Alpha = a | b`:

```
#define Alpha_set unsigned int Alpha_card
#define a_s 0b10
#define b_s 0b01
#define Alpha_nil_s 0b00
```

We also declare a lookup table `M_sets_LUT` that that returns singleton sets for every possible value in the system. When reading elements of a set, each element `e` is translated to `M[e]`; we make the bitwise logical or of the translation of every element.

13. Constants

(a) **Restrictions:** None

(b) **Solution:** translate to a Handel-C macro expression

14. Expressions

(a) **Restrictions:** just in the following syntax

```
csp_expression      ::= logical_expression
                      | math_expression
                      | rel_expression
                      | set_expression
                      | datatype_member
logical_expression  ::= true | false
                      | logical_expression and logical_expression
                      | logical_expression or logical_expression
                      | not logical_expression
                      | if logical_expression then csp_expression else csp_expression
math_expression     ::= [0 .. 9]^+1
                      | - math_expression
                      | math_expression + math_expression
                      | math_expression - math_expression
                      | math_expression * math_expression
                      | math_expression / math_expression
                      | math_expression % math_expression
rel_expression      ::= math_expression == math_expression
                      | math_expression != math_expression
                      | math_expression > math_expression
                      | math_expression >= math_expression
                      | math_expression < math_expression
                      | math_expression <= math_expression
set_expression      ::= set_display
                      | union(set_expression , set_expression)
                      | inter(set_expression , set_expression)
                      | diff(set_expression , set_expression)
set_display         ::= {set_members}
set_members         ::= set_member
                      | set_member , set_members
set_member          ::= math_expression
                      | logical_expression
                      | constant
                      | datatype_member
```

(b) **Solution:** translate to the corresponding Handel-C expression

15. `if _ then _ else _`

(a) **Restrictions:** None

(b) **Solution:** Use Handel-C's `if ( _ ) { _ } else { _ }`.

2.2 Restrictions Verification

csp2hc is able to automatically verify most of the restrictions currently imposed on the accepted constructors. This means that if any of these restrictions is not satisfied by the input $\mathsf{CSP_M}$, csp2hc indicates the error to the user. The verified restrictions are:

- No non-tail recursive processes (5(a)iA)
- No parallel composition in a tail recursion (5(a)iB)
- Parallel composition on mutually recursive processes only in the main process (5(b)iA)
- Accepted prefixing formats (6(a)i)
- Consistent use of channel projections (6(a)ii)
- There can be no projection on buses (7(c)i)
- A bus can only be either input or output, not both (7(c)ii)
- A bus cannot be used in channel sets (7(c)iii)
- A bus cannot be offered as a choice in an external choice (7(c)iv)
- A bus must be used within the system (7(c)v)
- A bus must have have exactly one type; that is, the declaration of the channel in the $\mathsf{CSP_M}$ specification declares exactly one type (7(c)vi)
- External choice only between prefixing processes (8(a)i)
- No two branches in an external choice with an input variables of the same name (8(a)ii)
- No multi-synchronised channel is offered in an external choice 10(a)i
- Numbers of readers and writers on a synchronisation 10(a)ii
- Explicit synchronisation channel sets 10(a)iii
- Conditions for ignoring the hiding 11a
- No mutually recursive datatypes (12a)
- No sets are communicated via a channel communication 12(f)iiiA
- Sets of sets are not accepted neither as a type nor as a expression 12(f)iiiB
- Unsupported $\mathsf{CSP_M}$ constructors
- No keyword present in the $\mathsf{CSP_M}$ specification
- Syntax of expressions (14a)

2.3 Identified Errors in the Parser/Type Checker

1. Não permite comentários durante definição

```
channel c:Int

P1 =
    -- Se for isto
    if true then (c!0 -> SKIP)
    -- senao
    else (c!1 -> SKIP)
```

2.4 Directives

In order to be able to translate the source CSP$_M$ code, our translator needs some extra information from the user. These are called directives, and are input in the form of comments in the source CSP$_M$ specification with a special format. This format is a line commented as follows:

```
--!! DIRECTIVE
```

In what follows we discuss the current directives used by csp2hc.

2.4.1 Input and Output Channels

This directive is used to give to csp2hc the indication that a channel, which is not explicitly used as an input or as an output, is either an input or an output.

- Format: `--!! channel` channel_name [`in` , `out`][1] `within` process_name
- Mandatory: for every channel c used as an synchronisation event anywhere in the system

For instance, in the following input:

```
--!! channel c in within P
P = c -> SKIP
```

The directive indicates that `c` is an input channel in process `P`.

2.4.2 Argument type

This directive is used to give to csp2hc the type of each of the arguments of a parametrised process.

- Format: `--!! arg` variable_name handelc_type `within` process_name
- Mandatory: for every process argument

For instance, in the following input:

```
--!! arg x integer within P
--!! arg y Alpha within P
P(x,y) = c.x!y -> SKIP
```

The directive indicates that the types of arguments `x` and `y` of `P` are Handel-C's `integer` and `Alpha`, respectively.

2.4.3 Main Process

This directive is used to give to csp2hc the main process, which represents the main behaviour of the system.

- Format: `--!! mainp` csp_process_expression
- Optional: Default is `SKIP`

For instance, in the following input:

```
--!! mainp P [| {| c |} |] Q

--!! channel c in within P
P = c -> SKIP

--!! channel c in within Q
Q = c -> SKIP
```

The directive indicates that system behaves like the parallel composition of `P` and `Q`.

Multiple lines are accepted. The example above could have been written as follows.

```
--!! mainp P
--!!       [| {| c |} |]
--!!       Q
```

2.4.4 Number of bits for integers

This directive is used to give to csp2hc the number of bits used to represented integer number in the system.

- Format: `--!! int_bits` $[1\,..\,9]^{+1}[0\,..\,9]^{*}$
- Optional: Default is 1

For instance, in the following input:

```
--!! int_bits 2
```

The directive indicates that integers numbers in the system are those signed integer number that can be represented using two bits, thus `-2`, `-1`, `0`, and `1` are valid integer numbers within this system.

Bit 1	Bit 0	**Signed Number**	**Unsigned Number**
0	0	0	0
0	1	1	1
1	0	-2	**2**
1	1	-1	**3**

Table 1: Unsigned and Signed Integers

Warning The number of bits declared by this directive must be sufficient to include the evaluation of all integer expressions within the specification. Otherwise, this inconsistency generates the following problem in the generated code. As previously described, if we declare integers to be of 2 bits, we are considering `-2`, `-1`, `0`, and `1` as the possible values for integers in the specifications. If, however, in some point of the specification the values 2 and 3 are used, the Handel-C compiler accepts the generated code, but it interprets these values as `-2` and `-1`, respectively. So, the specification of a parallel composition of the events `c.-2` and `c.2` does not synchronise, but the generated Handel-C code will synchronise. Although the Handel-C compiler should not accept such behaviour, the table 1 gives an insight why such behaviour happens: when unsigned, the numbers 2 and 3 have the same bitwise representation as the signed numbers `-2` and `-1`, respectively.

2.4.5 Type of Sets

Type inferencing is not implemented. For this reason, csp2hc needs a directive that indicates the type of empty sets. Currently, this is given for each process.

- Format: `--!! set_of` handelc_type `within` [process_name | datatype_name | constant_name]
- Mandatory: for every set used in the system

For instance, in the following input:

```
--!! set_of integer within P
P = c!{1} -> SKIP
```

The directive indicates that set within `P` is a set of `integer`s.

2.4.6 System Input/Output (Buses)

The input and output of the overall system is made via Handel-C buses. For that, we must declare that the channel is a bus. This is made using the directive `bus`.

- Format: `--!! bus` channel_name
- Optional

For instance, in the following input:

```
--!! bus read
```

The directive indicates that channel `read` is a bus. The translator automatically inferences the channel type from the $\mathsf{CSP_M}$ specification.

3 Translating CMOS

The final objective of the project is to automatically translate the whole of the CMOS specification; however, due to the complexity of the CMOS, we have drawn three milestones for the project: the phase controller, the heap model, and the CMOS. In what follows, we indicate for each of these milestones, the constructors whose translation are already mechanised, and the ones whose translation have not yet been mechanised. For those whose translation have not yet been mechanised, we provide possible solutions for their implementation in Handel-C.

3.1 Phase Controller

The constructors needed for the translation of the phase controller are:

1. Mutual Recursion (Requirement 5b)
2. Prefixing (Requirement 6)
3. External choice (Requirement 8)
4. Simple datatypes (Requirement 12b)

The current version of csp2hc is already able to automatically translate the phase controller.

3.2 Heap Model

The current version of csp2hc is already able to automatically translate part of the constructors used in the heap model. These constructors are:

1. `STOP` (Requirement 2)
2. Mutual recursion (Requirement 5b)
3. Prefixing (Requirement 6)
4. External choice (Requirement 8)
5. Simple datatype (Requirement 12b)
6. `Int` (Requirement 12d)
7. `Bool` (Requirement 12e)

8. Constants (Requirement 13)

For timing restrictions only, the translation of the following constructors are not yet mechanised.

1. Sets can be given to datatype constructors
2. Boolean guards can take part in the external choice
3. Set expressions
   (a) Sets as channel types
   (b) Sets in communications
   (c) Sets as arguments
   (d) Set comprehension
   (e) Integer ranges
   (f) `card`
   (g) `member`
   (h) `set`
   (i) `pick`
4. Declaration of `nametype`
5. `let _ within _`
6. Constrained inputs
7. Pattern matching
8. Sequences expressions
   (a) Sequences Display
   (b) Sequences concatenation
   (c) `Seq`
9. Tuples expressions

In what follows we discuss the solutions that will be implemented for each of these constructors.

Solutions

1. Sets can be given to datatype constructors
   (a) **Restrictions:** None. It removes restriction **??**
   (b) **Solution:** Since this specification has already been checked by FDR, we can use the maximal type of the set instead of the set itself. For this, we need to infer the type of the set; the types of empty sets need to be given via a directive.
2. Boolean guards can take part in the external choice
   (a) **Restrictions:** None. It relaxes restriction 8(a)i
   (b) **Solution:** Use the following transformation before translation

```
(g & P) [] Q = if g then (P [] Q) else Q
```

3. Set expressions
   (a) Sets as channel types
      i. **Restrictions:** None. It removes restrictions **??**
      ii. **Solution:**Since this specification has already been checked by FDR, we can use the maximal type of the set instead of the set itself. For this, we need to infer the type of the set; the types of empty sets need to be given via a directive.
   (b) Sets in synchronisation channel set.
      i. **Restrictions:** None.
      ii. **Solution:** Sets are the bitwise or of the corresponding singleton sets (in the lookup table) of the elements.
   (c) Sets in communications
      i. **Restrictions:** None. It removes restrictions 12(f)iiiA
      ii. **Solution:** Sets are the bitwise or of the corresponding singleton sets (in the lookup table) of the elements.
   (d) Set comprehension
      i. **Restrictions:** None
      ii. **Solution:** Create a library in Handel-C that allows the translation of such constructions
   (e) Integer ranges
      i. **Restrictions:** None
      ii. **Solution:** Create a bitwise or of every integer from the minimum to the maximum value.
   (f) `card`

i. **Restrictions:** None

ii. **Solution:** Create a lookup table containing 0 and 1.

```
static integer card_LUT[2] = {0 , 1};
```

Declare the following macro, where `n` is the maximum integer value, based on the given directive.

```
macro expr card(b) = (LUT[b[0]] + LUT[b[1]] + ... + LUT[b[n]]);
```

(g) `Union`

i. **Restrictions:** None

ii. **Solution:** Bitwise or of all the elements, which are themselves sets.

(h) `member`

i. **Restrictions:** None

ii. **Solution:** First, for every type, there will be a lookup table that returns the singleton set that contain each one of the elements in that type. For instance, for the booleans we have:

```
static boolean_set singleton_boolean_sets_LUT[boolean_card] ={
    true_s, false_s
};
```

For the integers (i.e. 2 bits integers), the lookup table will look like this:

```
static integer_set singleton_integer_sets_LUT[integer_card] ={
    integer_0_s, integer_1_s,
    integer_neg_2_s, integer_neg_1_s
};
```

Then, the set membership will be given by the following macro expression:

```
macro expr member(e,s) =
    ((singleton_integer_sets_LUT[(unsigned)e] | s) == s);
```

(i) `set`

i. **Restrictions:** None

ii. **Solution:** returns the bitwise or of all the elements. For instance, `set([-1,0,1,2])` is

```
singleton_integer_sets_LUT[(unsigned)-1] |
singleton_integer_sets_LUT[(unsigned)0] |
singleton_integer_sets_LUT[(unsigned)1] |
singleton_integer_sets_LUT[(unsigned)2]
```

(j) `pick`

i. **Restrictions:** None

ii. **Solution:** we illustrate our solution with a set of a type with cardinality eight. These are the possible singleton sets `b`, and the binary representation of the element `x` of the singleton set.

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	x_2	x_1	x_0
1	0	0	0	0	0	0	0	1	1	1
0	1	0	0	0	0	0	0	1	1	0
0	0	1	0	0	0	0	0	1	0	1
0	0	0	1	0	0	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	0	0	1	0	0	0

From this table, we notice that we actually have a pattern in which, `pick(s) = x[2] @ x[1] @ x[0]`, where $x_i = \bigvee\{b_j \mid x_i = 1$ in binary representation of $j\}$. In our example, we have that:

```
x[2] = b[7] || b[6] || b[5] || b[4]
x[1] = b[7] || b[6] || b[3] || b[2]
x[0] = b[7] || b[5] || b[3] || b[1]
```

Thus, `pick(0b01000000) = 0b110 = 6`

4. Declaration of functions

   (a) **Restrictions:** None

   (b) **Solution:** Declare as macro expressions

5. Declaration of `nametype`

   (a) **Restrictions:** None

   (b) **Solution:** Use Handel-C's `typedef`.

6. `let _ within _`

   (a) **Restrictions:** None

   (b) **Solution:** Declare one global macro for each of the local variables in the order they appear

7. Constrained inputs

   (a) **Restrictions:** None

   (b) **Solution:** two solutions have already been considered. They, however, need further improvements in order to achieve a more general solution.

i. Provided we have **only one-to-one communications**, once we find a constrained input in the program tree, we have to go back up to the first parallel composition and do the following transformation from there.

```
(c!e -> Q [] i_l -> R)
[| {| c |} |]
(c?x:S -> P(x) [] i_r -> T)
=
( (try!e -> (c!e -> Q [] i_l -> R))
  [| {| c , try |} |]
  (try?x -> if member(x,S) then c?x -> P(x) [] i_r -> T
            else i_r -> T) ) \ {| try |}
```

Notice that we consider `try` to be a fresh channel name and `y` to be a fresh variable name within the protocol. Otherwise we index their names with the first integers `n` and `m`, such that `try_n` and `y_m` are fresh names within the system. **However**, if this parallel composition is in parallel with another action, as the one below

```
c?x -> V [] i_m -> W
```

It may the case where the whole parallel composition in CSP allows `c` to happen; if `member(e,S)`. However, in Handel-C this may not happen because of the clock. It may the case in which some of the interruption events `i_x` happens in the clock cycle inserted by the `try` event; this will cause the event `c` not to be allowed to happen.

ii. Provided **the expression `e` is in terms only of global variables (parameters are NOT global)**, then we already have the value available in both sides; only the communication is needed.

```
(c!e -> Q [] i_l -> R)
[| {| c |} |]
(c?x:S -> P(x) [] i_r -> T)
=
(c!e -> Q [] i_l -> R)
[| {| c |} |]
( if member(e,S) then c?x -> P(x) [] i_r -> T
  else i_r -> T )
```

8. Pattern matching

   (a) **Restrictions:** The whole solution is equivalent to writing a functional language compiler. Just the pattern matching used in the CMOS will be accepted.

   (b) **Solution:**

      i. For `c?_`, replace `_` by a fresh new name

ii. For `let (S,_,M,V) = E within P`, translate it as
`let S = E.1, M = E.3, V = E.4 within P`

iii. Remaining must be refined

9. Sequences

(a) Sequences Display

i. **Restrictions:** None

ii. **Solution:** Sequences can be represented as arrays with a high water mark, using `struct`. A directive must be given in order to establish the maximum size of the array. So, for instance, if the maximum size of the sequences is said to be six, the sequence `[1,2,3]` is represented as `[_,_,_,(1),2,3]`, with the high water mark set to 3 (we denote the high water mark by putting the element on which the mark is between parenthesis. Furthermore, we write `_` when the value can be any value).

(b) Sequences concatenation

i. **Restrictions:** None

ii. **Solution:** simply introduce the elements of the left-hand side sequence to the right hand side array as follows.

```
[1,2] ^ [4,5,6]                          [Representation of sequences]
= [_,_,_,_,(1),2] ^ [_,_,_,_,(4),5,6]    [Calculation]
= [_,_,_,_,(1),_] ^ [_,_,_,(2),4,5,6]    [Calculation]
= [_,_,_,_,_,_,_] ^ [_,_,(1),2,4,5,6]    [Base case]
= [_,_,(1),2,4,5,6]
```

(c) `Seq`

i. **Restrictions:** None

ii. **Solution:** as we have a maximum number of elements for the sequences, the sets of sequences is not infinite as in the CSP. In order to reuse the already existing translation strategies, we declare a datatype whose elements are all the possible sequences and translate this datatype. For instance, suppose we have three for the maximum length of the sequences. In this case, `Seq(Bool)` will be translated as the following datatype:

```
datatype SEQ_Bool ==
    SEQ_Bool_empty
    | SEQ_Bool_true | SEQ_Bool_false
    | SEQ_Bool_true_true | SEQ_Bool_true_false
    | SEQ_Bool_false_true | SEQ_Bool_false_false
```

A lookup table has also to be provided in order to construct the elements of this type.

```
static SEQ_Bool SEQ_Bool_LUT[7] =
    {[_,_] ,
     [_,true], [_,false],
     [true,true], [true,false], [false,true], [false,false]};
```

10. Tuples
    (a) Tuples Display
        i. **Restrictions:** None
        ii. **Solution:** Tuples can be translated using `struct`.

3.3 CMOS

The current version of csp2hc is already able to automatically translate part of the constructors used in the CMOS. These constructors are:

1. `SKIP` (Requirement 1)
2. `STOP` (Requirement 2)
3. Sequential Composition (Requirement 3)
4. Recursion (Requirement 5)
5. Prefixing (Requirement 6)
6. External choice (Requirement 8)
7. Concurrency (Requirement 10)
8. Datatypes (Requirement 12)
9. Constants (Requirement 13)

For timing restrictions only, besides those discussed in Section 3.2 the translation of the following constructors are not yet mechanised.

1. Channel sets
2. Renaming
    (a) Non-relational Renaming
    (b) Relational Renaming
3. `include`
4. `chase`

5. Indexed sequence
6. Indexed parallelism
7. Replicated choices
8. `module`
9. Sequence comprehension

In what follows we discuss the solutions that will be implemented for each of these constructors.

1. Channel sets
   (a) **Restrictions:** None
   (b) **Solution:** as for requirement 8c, we will translate a datatype that contains all the possible channels within the system. For instance, for a system containing the following channels

   ```
   channel b
   channel c:Int
   ```

   and assuming two bits integers, we have the following datatype.

   ```
   CHANNELS == CHANNEL_b
               | CHANNEL_c_neg_2 | CHANNEL_c_neg_1
               | CHANNEL_c_0 | CHANNEL_c_1
   ```

2. Renaming
   (a) Non-relational Renaming
      i. **Restrictions:** None
      ii. **Solution:** By replacing as the following example.
      `P = Q[a <- b]`

      - Get the body of `Q`
      - Replace `a` by `b` (let's call this `NewQ` in this example)
      - Translate `P = NewQ`

   (b) Relational Renaming
      i. **Restrictions:** to be analysed
      ii. **Solution:** to be analysed
3. `include`

(a) **Restrictions:** None

(b) **Solution:** Append files before parsing.

4. `chase`

   (a) **Restrictions:**

   (b) **Solution:** Provided **we have no use of the internal choice operator**, the only τ events are generated by hiding. In this case, every time we find a hiding `P \ cs` in the tree, we analyse `P`: every external choice in `P` must be rewritten such that any initial channel in the choice that is hidden must come first in the choice. For instance:

   ```
   ((a -> P) [] (b -> Q)) \ {| b |}
   ```

   must be rewritten as

   ```
   ((b -> Q) [] (a -> P)) \ {| b |}
   ```

   Since we use `PRIALT` to implement choice, `b` will be given priority, which in the end, means that we are given priority to the τ event.

5. Indexed sequence

   (a) **Restrictions:** Only closed range numerical indexes for indexed sequences

   (b) **Solution:** For every basic `datatype d` within the model there will be a function $mapping\_d : d \to \mathbb{N}$.

6. Indexed parallelism

   (a) **Restrictions:** Only closed range numerical indexes for indexed parallel

   (b) **Solution:** The same as 5

7. Replicated choices

   (a) **Restrictions:** Only closed range numerical indexes for replicated choices

   (b) **Solution:** translate the expansion of the replicated choice

8. `module`

   (a) **Restrictions:** Only closed range numerical indexes for replicated choices

   (b) **Solution:** flatten the whole specification before translation, renaming the module components in order to avoid name clashes.

9. Sequence comprehension

   (a) **Restrictions:** None

(b) **Solution:** Create a library in Handel-C that allows the translation of such constructions

Once the translation of all these constructors are implemented the translation of the CMOS into Handel-C will be fully automated. Besides, the vast majority of the CSP constructs will have been translated and, as such, a large number of CSP_M specifications will be automatically translated into Handel-C.

4 Using csp2hc

In order to execute csp2hc, simply execute the file `csp2hc.bat`. The JVM used must be of version `1.5.0_06` or higher. The interface is very simple and presented in Figure 2: it is composed, basically, by a log window, in which all the stages of the translation are logged. In order to use the translator, simply open the CSP file (`.csp`) you want to translate and, if the translation is successful, save the result in a Handel-C (`.hcc`) file; the user is then given the choice of translating another file. If, however, the translation is not successful, an error message is given, and the reason for the error is shown in the log window. The user is given the choice of correcting the file and trying to translate it again. In order to finish the execution of the tool, simply click `close` in the log window.

In the future, the interface could be incremented. For instance, it could have three tabs: the translation log, the source CSP file, and the target Handel-C file. This would allow the user to edit the files without the need of an extra text editor.

5 Further requirements

In this project, we have concentrated on the features and requirements of the CMOS; however, some constructs are not used in the CMOS and were not considered. They are listed below.

1. Two processes writing to the same channel on the same clock cycle
2. The same channels is involved in a choice, it must be in the same direction.
3. Interrupts
4. Untimed time out
5. `external`
6. Nested blocks of comment markers

In order to achieve a full automatic translation from CSP_M to Handel-C these are some of the constructs that still need to be taking into account.

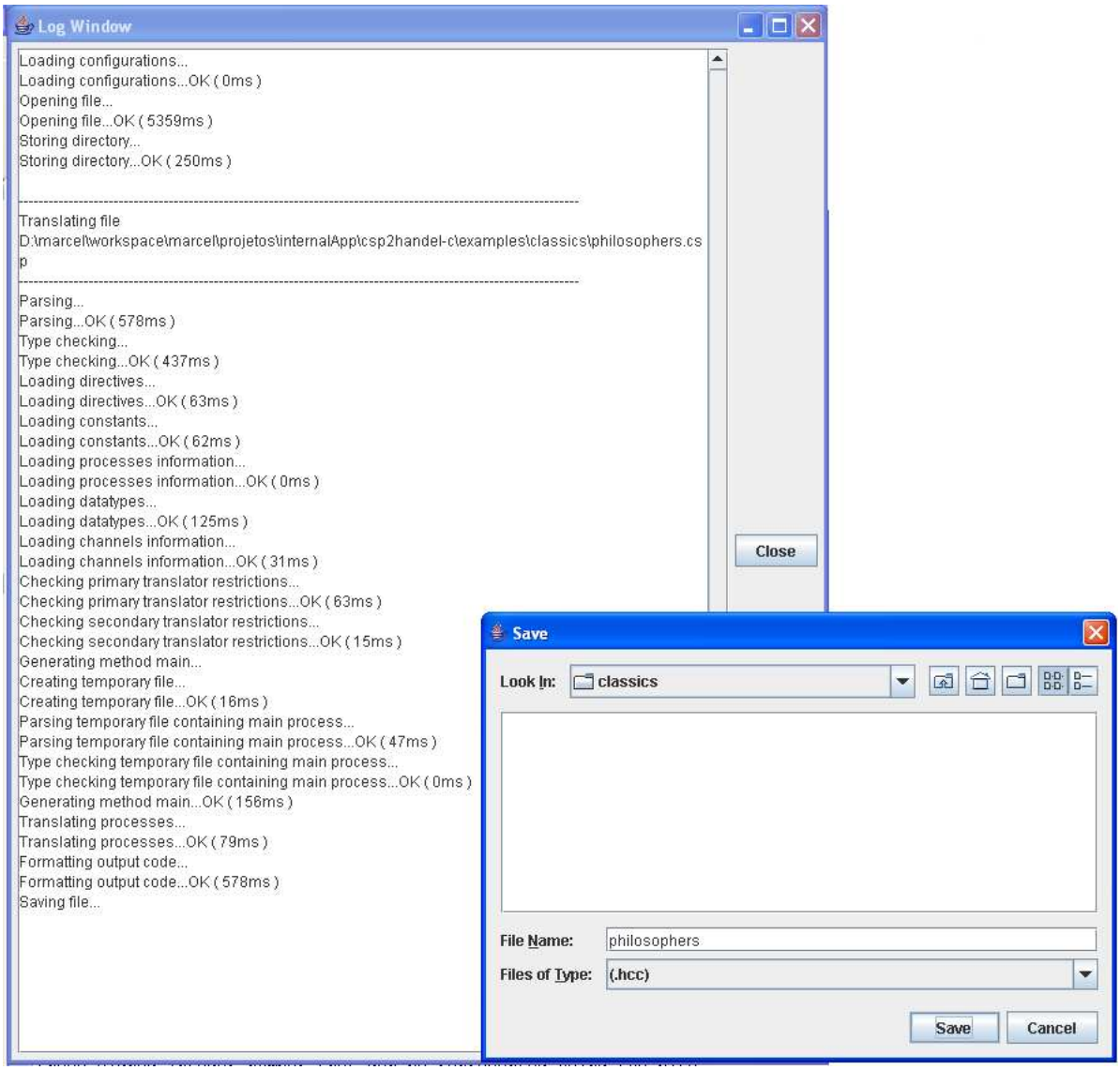

Figure 2: csp2hc Graphic Interface