csp2hc: from CSP_M to Handel-C

Marcel Oliveira and Jim Woodcock

March 23, 2006

1 Introduction

Our tool, *csp2hc*, already mechanises the translation of a considerable subset of CSP_M to Handel-C, which includes the following features.

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

Although they represent a subset of 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 A: 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 2.1, 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 2.2 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 2.3, 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_s`
- `false_s`
- For every type `T` in the specification (including `integer` and `boolean`):
    - `T_set`
    - `T_nil`
    - `T_card`
    - `T_sets_LUT`
- For every simple datatype value `v`
    - `v`
    - `v_s`
- 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`

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 an infinite loop that does nothing.

3. Sequential composition
   (a) **Restrictions:** None
   (b) **Solution:** translates to a sequential composition.

4. Recursion
   (a) Simple Recursion
      i. **Restrictions:**
         A. Only Tail Recursion
      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 2.3.3).

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 = x + 1;
                    PROGRAM_COUNTER = P2;
                    KEEP_LOOPING = true;
                }
                break;
            }

            case P2 :{
                seq{
                    c! P2_local_x;
                    P1_local_x = x - 1;
                    PROGRAM_COUNTER = P1;
                    KEEP_LOOPING = true;
                }
                break;
            }
        }
    }
}
```

5. Prefixing

   (a) **Restrictions:**

      i. Synchronisations of the form

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

      where `csp_expression` is as in Section 10a.

      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.

6. External Choice

   (a) **Restrictions:**

      i. Only for prefixing processes.

      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`

7. Concurrency

   (a) **Restrictions:**

      i. No multi-synchronisation

   (b) Sharing Parallel

      i. **Restrictions:**

         A. Only for processes with no interleaved events in the synchronisation set: for every two processes `P` and `Q` composed in parallel in a channel set `CS` (`P [| CS |] Q`), we have that $\alpha(\mathtt{P}) \cap \alpha(\mathtt{Q}) \subseteq \mathtt{CS}$

      ii. **Solution:** translate to Handel-C parallelism

   (c) Alphabetised Parallel

      i. **Restrictions:**

         A. Only for processes with no interleaved events in the synchronisation set: for every two processes `P` and `Q` synchronising in parallel in the channel sets `CS1` and `CS2` (`P [ CS1 || CS2 ] Q`), we have that the synchronisation sets satisfy the condition $\alpha(\mathtt{P}) \cap \alpha(\mathtt{Q}) \subseteq \mathtt{CS1} \cap \mathtt{CS2}$.

    ii. **Solution:** translate to Handel-C parallelism

  (d) Interleaving

    i. **Restrictions:**

      A. : Only for processes with no events in common: for every two processes `P` and `Q` interleaved (`P ||| Q`), we have that $\alpha(\texttt{P}) \cap \alpha(\texttt{Q}) = \emptyset$

    ii. **Solution:** translate to Handel-C parallelism

8. 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:**

      A. Constants cannot be given as argument to the constructors (tags)

      B. Sets cannot be given as argument to the constructors (tags)

    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. **Restrictions:**
         A. Cannot be used as channels type
         B. Sets cannot be used in a channel usage
      ii. **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.

9. Constants
   (a) **Restrictions:** None
   (b) **Solution:** translate to a Handel-C macro expression
10. Expressions

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

csp_expression ::= logical_expression
| math_expression
| rel_expression
| datatype_member
logical_expression ::= `true` | `false`
| logical_expression `and` logical_expression
| logical_expression `or` logical_expression
| `not` logical_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

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

11. `if _ then _ else _`
    (a) **Restrictions:** None
    (b) **Solution:** Use Handel-C's `if ( _ ) { _ } else { _ }`.

2.1 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 CSP_M, *csp2hc* indicates the error to the user. The verified restrictions are:

- Syntax of expressions (10a)
- Accepted prefixing formats (5(a)i)
- Consistent use of channel projections (5(a)ii)
- No mutually recursive datatypes (8a)
- No constant are given to datatype constructors (8(c)iA)

- No sets are given to datatype constructors (8(c)iB)
- No two branches in an external choice with an on the input variables of the same name (6(a)ii)
- No non-tail recursive processes (4(a)iA)
- Parallel composition on mutually recursive processes only in the main process (4(b)iA)
- No channel sets used in channel declarations (8(f)iA)
- No linked parallelism

However, for timing restrictions only, some restrictions are not being checked by *csp2hc*; in this case, if any of these restrictions is not satisfied by the input CSP_M, *csp2hc* will generate a Handel-C code that does not implement correctly the original CSP_M specification.

- Not all unsupported constructors are being identified by *csp2hc*
- External choice only between prefixing processes (6(a)i)
- Restrictions on the synchronisation channel sets of shared parallel composition (7(b)iA), alphabetised parallel composition (7(c)iA), and interleaving (7(d)iA)
- Sets cannot be used in a channel usage 8(f)iB

2.2 Identified Errors in the Parser/Type Checker

Due to our big number of tests, we have identified a couple of errors in the CSP_M parser/type checker. In what follows, we describe these errors. For some of them, we provide temporary solutions.

- No comments within a CSP_M definition
- The type checker is raising an exception in the invocation of the following parametrised process:

  ```
  P(x) = c -> P(x)
  ```

  Temporary solution: declare a constant `x` in the CSP_M.
- The type checker is raising an exception in the invocation of the following parametrised process:

  ```
  P(x) = c -> P(x+1)
  ```

Temporary solution: declare a function `inc = \ x @ x + 1` in the CSP specification, and invoke `P(inc(x))` instead.

- Intervals (`Interval = {1..3}`) are not being accepted by the type checker
- Datatypes constructors are accepting constants and they shouldn't
  `datatype Alpha = a | b | Z.ZERO`, where `ZERO = 0`
- No constructor `Set`
- Sets of elements of a datatype are raising a
  `CspSemanticException: referenced name 'a' was not declared`

  ```
  datatype Letter = a | b
  SET_LETTER_1 = {a, b}
  ```

In the near future we will receive a new version of the CSP_M parser/type checker that will include the corrections of these errors.

2.3 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*.

2.3.1 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`.

2.3.2 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.

2.3.3 Main Process

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

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

For instance, in the following input:

```
--!! main 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`.

2.3.4 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.

2.3.5 Type of Empty Sets

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

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

For instance, in the following input:

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

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

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

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 4b)
3. Prefixing (Requirement 5)
4. External choice (Requirement 6)
5. Sharing parallel (Requirement 7b)
6. Simple datatype (Requirement 8b)
7. `Int` (Requirement 8d)
8. `Bool` (Requirement 8e)
9. Constants (Requirement 9)

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

1. Support to multi-synchronisation
2. Sets can be given to datatype constructors
3. Boolean guards can take part in the external choice
4. Set expressions
    (a) Sets as channel types
    (b) Sets in communications

   - (c) Sets as arguments
   - (d) Set comprehension
   - (e) Integer ranges
   - (f) `diff`
   - (g) `card`
   - (h) `member`
   - (i) `set`
   - (j) `pick`

5. Declaration of functions
6. Declaration of `nametype`
7. `let _ within _`
8. Constrained inputs
9. Pattern matching
10. Sequences expressions
    - (a) Sequences Display
    - (b) Sequences concatenation
    - (c) `Seq`
11. Tuples expressions

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

Solutions

1. Support to multi-synchronisation
   - (a) **Restrictions:** None. Removes restriction 7(a)i
   - (b) **Solution:** implement multi-synchronisation using a protocol that uses a centralised controller. By way of illustration, let us consider the following three process:

     ```
     P = c!0 -> c?v_1 -> SKIP
     Q = c?v_1 -> c!1 -> SKIP
     R = c?v_1 -> d -> c?v_2 -> SKIP
     ```

     We can statically identify that `c` is a multi-synchronised channel in the parallel composition `P [| {| c, d|} |] (Q [| {| c, d|} |] R)`, and that it involves three processes. The three processes would be translated as follows:

```
inline void P (){
    seq{ offer_c[0]!syncout;
        seq{ offer_c_val! 0;
            seq{ integer dumb_int; c? dumb_int;
                seq{ offer_c[0]!syncout;
                    seq{ integer v_1;
                            c? v_1; }; }; }; }; }
}
inline void Q (){
    seq{ offer_c[1]!syncout;
        seq{ integer v_1; c? v_1;
            seq{ offer_c[1]!syncout;
                seq{ offer_c_val! 1;
                    seq{ integer dumb_int;
                            c? dumb_int;
                    }; }; }; }; }
}
inline void R (){
    seq{ offer_c[2]!syncout;
        seq{ integer v_1; c? v_1;
            seq{ d!syncout;
                seq{ offer_c[2]!syncout;
                    seq{ integer v_2; c? v_2;
                    }; }; }; }; }
}
```

where `dumb_int` is a fresh variable name.
The code for the controller would be as follows:

```
inline void c_MS_controller() {
    boolean stopped_0, stopped_1, stopped_2, stopped;
    par { stopped_0 = false; stopped_1 = false; stopped_2 = false; };
    stopped = stopped_0 || stopped_1 || stopped_2;
    while (!stopped) {
        seq{ par { stopped_0 = c_MS_controller_wait_for(0);
                   stopped_1 = c_MS_controller_wait_for(1);
                   stopped_2 = c_MS_controller_wait_for(2); };
             stopped = stopped_0 || stopped_1 || stopped_2;
             if (!stopped) {
                 seq { integer v; offer_c_val? v; c! v; }
             }
        }
    }
}
```

```
inline boolean c_MS_controller_wait_for(integer i) {
    boolean stopped; stopped = false;
    prialt{
        case stop_c?syncin :{ stopped = true; };
        break;
        case offer_c[(unsigned)i]?syncin:{ ; };
        break;
    }
    return stopped;
}
```

The parallel composition `P [| {| c, d|} |] (Q [| {| c, d|} |] R)` is then implemented as follows:

```
par{
    seq{ par{ P(); Q(); S(); };
         stop_c!syncout; };
    c_MS_controller();
}
```

2. Sets can be given to datatype constructors
   (a) **Restrictions:** None. It removes restriction 8(c)iB
   (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.
3. Boolean guards can take part in the external choice
   (a) **Restrictions:** None. It relaxes restriction 6(a)i
   (b) **Solution:** Use the following transformation before translation

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

4. Set expressions
   (a) Sets as channel types
      i. **Restrictions:** None. It removes restrictions 8(f)iA
      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 communications
  - i. **Restrictions:** None
  - ii. **Solution:** Sets are the bitwise or of the corresponding singleton sets (in the lookup table) of the elements.
- (c) Sets as arguments
  - i. **Restrictions:** None
  - 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 minimun to the maximum value.
- (f) `diff`
  - i. **Restrictions:** None
  - ii. **Solution:** `macro expr diff(s,t) = (s & (~t));`
- (g) `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]]);
    ```

- (h) `Union`
  - i. **Restrictions:** None
  - ii. **Solution:** Bitwise or of all the elements, which are themselves sets.
- (i) `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);
```

(j) `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]
```

(k) `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`

5. Declaration of functions
   (a) **Restrictions:** None
   (b) **Solution:** Declare as macro expressions
6. Declaration of `nametype`
   (a) **Restrictions:** None
   (b) **Solution:** Use Handel-C's `typedef`.
7. `let _ within _`
   (a) **Restrictions:** None
   (b) **Solution:** Declare one global macro for each of the local variables in the order they appear
8. 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 )
```

9. 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

10. 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]};
```

11. 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 4)

5. Prefixing (Requirement 5)
6. External choice (Requirement 6)
7. Concurrency (Requirement 7)
8. Datatypes (Requirement 8)
9. Constants (Requirement 9)

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. Remove restriction on the synchronisation channel set of sharing parallel composition (Restriction 7(b)iA) and interleaving (Restriction 7(d)iA)
6. Indexed sequence
7. Indexed parallelism
8. Replicated choices
9. `module`
10. 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 10c, 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 |}
    ```

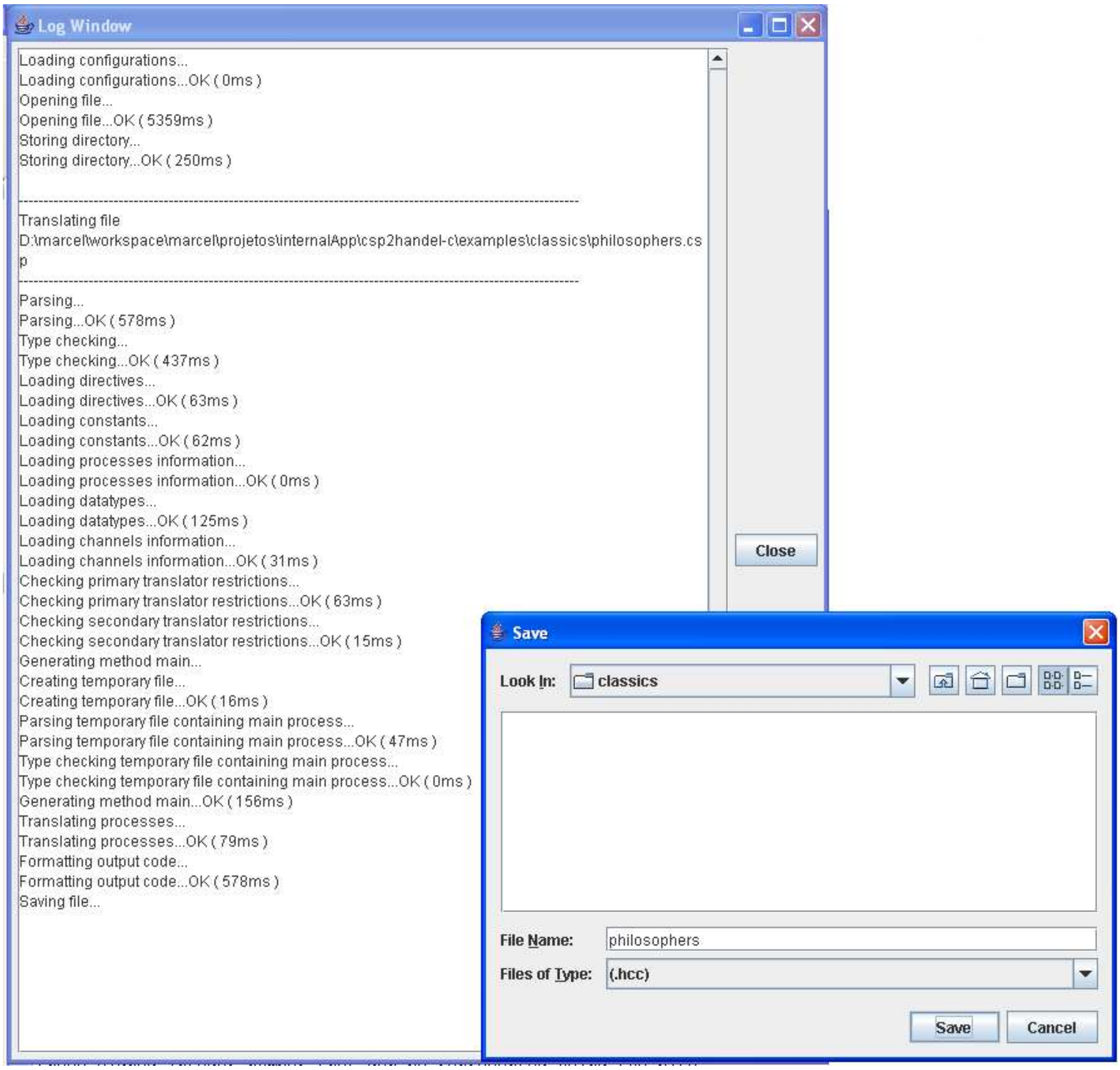
    Since we use `PRIALT` to implement choice, `b` will be given priority, which in the end, means that we are given priority to the τ event.

5. Remove restriction on the synchronisation channel set of sharing parallel composition (Restriction 7(b)iA) and interleaving (Restriction 7(d)iA)
    (a) **Restrictions:** to be analysed
    (b) **Solution:** to be analysed
6. 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 $mapping\_d : d \to \mathbb{N}$.
7. Indexed parallelism
    (a) **Restrictions:** Only closed range numerical indexes for indexed parallel
    (b) **Solution:** The same as 6
8. Replicated choices
    (a) **Restrictions:** Only closed range numerical indexes for replicated choices
    (b) **Solution:** translate the expansion of the replicated choice
9. `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.
10. 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 1: 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



```
Loading configurations...
Loading configurations...OK ( 0ms )
Opening file...
Opening file...OK ( 5359ms )
Storing directory...
Storing directory...OK ( 250ms )

----------------------------------------------------------------------------------------------------
Translating file
D:\marcel\workspace\marcel\projetos\internalApp\csp2handel-c\examples\classics\philosophers.cs
p
----------------------------------------------------------------------------------------------------
Parsing...
Parsing...OK ( 578ms )
Type checking...
Type checking...OK ( 437ms )
Loading directives...
Loading directives...OK ( 63ms )
Loading constants...
Loading constants...OK ( 62ms )
Loading processes information...
Loading processes information...OK ( 0ms )
Loading datatypes...
Loading datatypes...OK ( 125ms )
Loading channels information...
Loading channels information...OK ( 31ms )
Checking primary translator restrictions...
Checking primary translator restrictions...OK ( 63ms )
Checking secondary translator restrictions...
Checking secondary translator restrictions...OK ( 15ms )
Generating method main...
Creating temporary file...
Creating temporary file...OK ( 16ms )
Parsing temporary file containing main process...
Parsing temporary file containing main process...OK ( 47ms )
Type checking temporary file containing main process...
Type checking temporary file containing main process...OK ( 0ms )
Generating method main...OK ( 156ms )
Translating processes...
Translating processes...OK ( 79ms )
Formatting output code...
Formatting output code...OK ( 578ms )
Saving file...
```

Look In: classics

File Name: philosophers

Files of Type: (.hcc)

Figure 1: *csp2hc* Graphic Interface

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. Lambda terms
4. Interrupts
5. `CHAOS`
6. `/\`
7. Untimed time out
8. `external`
9. 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.

A Transcripts

Dining Philosophers

```
-------------------------
-- DINING PHILOSOPHERS --
-------------------------

--!! main TABLE_P

--!! int_bits 4

-- To get around the type checker
i = 0
j = 0

-- Maximum number of Philosophers
MAX_PHIL = 5

---------------
-- FUNCTIONS --
---------------

-- Decrements i by one modulo j
dec_mod = \ i,j @ (i - 1) % j

-- Increments i by one modulo j
inc_mod = \ i,j @ (i + 1) % j

-- Returns the maximum between i and j
max = \ i,j @ if i > j then i else j

-- Returns the minimum between i and j
min = \ i,j @ if i < j then i else j

--------------
-- CHANNELS --
--------------

channel thinks, sits, eats, getsup:Int
-- FOR FDR CHECK, COMMENT LINE ABOVE AND UNCOMMENT LINE BELOW
--channel thinks, sits, eats, getsup:{0..MAX_PHIL-1}
```

```
channel picksup, putsdown:Int.Int
-- FOR FDR CHECK, COMMENT LINE ABOVE AND UNCOMMENT LINE BELOW
--channel picksup, putsdown:{0..MAX_PHIL-1}.{0..MAX_PHIL-1}

-----------
-- FORKS --
-----------
--!! channel picksup in within FORK
--!! channel putsdown in within FORK
--!! arg i integer within FORK
FORK(i) =
    picksup.i.i -> putsdown.i.i -> FORK(i)
    []
    picksup.dec_mod(i,MAX_PHIL).i ->
        putsdown.dec_mod(i,MAX_PHIL).i -> FORK(i)

------------------
-- PHILOSOPHERS --
------------------
--!! channel picksup out within PHIL
--!! channel putsdown out within PHIL
--!! channel thinks out within PHIL
--!! channel sits out within PHIL
--!! channel eats out within PHIL
--!! channel getsup out within PHIL
--!! arg i integer within PHIL
PHIL(i) =
    thinks.i -> sits.i ->
        picksup.i.min(i,inc_mod(i,MAX_PHIL)) ->
            picksup.i.max(i,inc_mod(i,MAX_PHIL)) -> eats.i ->
                putsdown.i.inc_mod(i,MAX_PHIL) ->
                    putsdown.i.i -> getsup.i -> PHIL(i)

PHIL_FORK_SYNC = {| picksup, putsdown |}
```

```
-----------------
-- ENVIRONMENT --
-----------------

--!! channel thinks in within THINKS
--!! arg i integer within THINKS
THINKS(i) = thinks.i -> THINKS(i)
THINKING = ( THINKS(0)
             |||
             (THINKS(1) ||| (THINKS(2) ||| (THINKS(3) ||| THINKS(4)))) )

--!! channel sits in within SITS
--!! arg i integer within SITS
SITS(i) = sits.i -> SITS(i)
SITTING = (SITS(0) ||| (SITS(1) ||| (SITS(2) ||| (SITS(3) ||| SITS(4)))))

--!! channel eats in within EATS
--!! arg i integer within EATS
EATS(i) = eats.i -> EATS(i)
EATTING = (EATS(0) ||| (EATS(1) ||| (EATS(2) ||| (EATS(3) ||| EATS(4)))))

--!! channel getsup in within GETSUP
--!! arg i integer within GETSUP
GETSUP(i) = getsup.i -> GETSUP(i)
GETTINGUP = ( GETSUP(0)
              |||
              (GETSUP(1) ||| (GETSUP(2) ||| (GETSUP(3) ||| GETSUP(4)))) )

ENVIRONMENT_SYNC = {| thinks, sits, eats, getsup |}
ENVIRONMENT = THINKING ||| (SITTING ||| (EATTING ||| GETTINGUP))

-----------
-- TABLE --
-----------

-- ALL PHILOSOPHERS
ALL_PHIL = PHIL(0) ||| (PHIL(1) ||| (PHIL(2) ||| (PHIL(3) ||| PHIL(4))))

-- ALL FORKS
ALL_FORK = FORK(0) ||| (FORK(1) ||| (FORK(2) ||| (FORK(3) ||| FORK(4))))

-- TABLE
TABLE = ALL_PHIL [| PHIL_FORK_SYNC |] ALL_FORK
```

```
---------------------
-- TABLE PROTOTYPE --
---------------------
TABLE_P = ENVIRONMENT [| ENVIRONMENT_SYNC |] TABLE
```

Level Crossing

```
--------------------
-- LEVEL CROSSING --
--------------------

--!! main (CARS ||| TRAINS ||| GATE) [| {|train,car,gate|} |] CONTROL_CAR
---- FOR FDR CHECK, UNCOMMENT LINE BELOW
--MAIN = (CARS ||| TRAINS ||| GATE) [| {|train,car,gate|} |] CONTROL_CAR

---------------
-- DATATYPES --
---------------

datatype LETTER = A | B | C
datatype CAR_REG = consulate | official | L.LETTER
datatype TRAIN_COMPANY = GNER | Virgin
datatype UD = raise | lower
datatype ELA = enter | leave | approach

--------------
-- CHANNELS --
--------------

channel car:ELA.CAR_REG
channel train:ELA.TRAIN_COMPANY
channel gate:UD

----------
-- CARS --
----------
CARS = car.approach!L.A -> car.enter!L.A -> car.leave!L.A ->
            car.approach!L.B -> car.enter!L.B -> car.leave!L.B ->
                car.approach!L.C -> car.enter!L.C -> car.leave!L.C ->
                    car.approach!consulate -> car.enter!consulate ->
                        car.leave!consulate ->
                            car.approach!official -> car.enter!official ->
                                car.leave!official -> CARS
```

```
------------
-- TRAINS --
------------
TRAINS = train.approach!GNER -> train.enter!GNER ->
            train.leave!GNER -> train.approach!Virgin ->
                train.enter!Virgin -> train.leave!Virgin -> TRAINS

----------
-- GATE --
----------
--!! channel gate out within GATE
GATE =  gate.lower -> gate.raise -> GATE

-----------------------------
-- CONTROLLER - GATE IS UP --
-----------------------------
--!! channel gate in within CONTROL_CAR
CONTROL_CAR = car.approach?id -> car.enter?id ->
                car.leave?id -> CONTROL_TRAIN

--!! channel gate in within CONTROL_TRAIN
CONTROL_TRAIN = train.approach?id -> gate.lower ->
                    train.enter?id -> train.leave?id ->
                        gate.raise -> CONTROL_CAR
```