A Verified Protocol to Implement Multi-way Synchronisation and Interleaving in CSP

Marcel Vinicius Medeiros Oliveira[1,*], Ivan Soares De Medeiros Júnior[1], and Jim Woodcock[2]

[1] Departamento de Informática e Matemática Aplicada, UFRN, Brazil
[2] Department of Computer Science, University of York, England

Abstract. The complexity of concurrent systems can turn their development into a very complex and error-prone task. The use of formal methods like CSP considerably simplifies this task. Development, however, usually aims at reaching an executable program: a translation into a programming language is still needed and can be challenging. In previous work, we presented a tool, csp2hc, that translates a subset of CSP into Handel-C source code, which can itself be converted to produce files to program FPGAs. This subset restricts parallel composition: multi-synchronisation and interleaving on shared channels are not allowed. In this paper, we present an extension to csp2hc that removes these restrictions. We provide a performance analysis of our code.

Keywords: concurrency, multi-synchronisation, compilation, protocols.

1 Introduction

Concurrent applications are normally complicated since they consist of many components running in parallel. This usually yields to a complex and error-prone development [11]. In order to minimize these problems, formal methods like CSP [11] have been proposed. They are usually process algebras designed for describing and reasoning about synchronisation between processes. Furthermore, phenomena that are exclusive to the concurrent world, like deadlock and livelock, can be much more easily understood and controlled using such formalisms. The tools available for these languages increased their success. For CSP, the model-checker FDR2 [3] provides an automatic check of finite state specifications for correctness and properties like deadlock and divergence freedom. It accepts a machine-processable version of CSP, called CSP_M [11], which combines an ASCII representation with a functional language.

Using CSP, we can describe concurrent systems at various levels of abstraction: specifications, design, and implementation. This allows a stepwise development in a single framework. Nevertheless, a translation into a practical programming language is still needed. In order to minimize this gap, it is better to target languages that directly support the CSP style of concurrency through

* Partially supported by INES and CNPq: grants 573964/2008-4 and 560014/2010-4.

channels, such as occam-2 [5] and Handel-C[1], or packages that add these features such as JCSP [13] for Java, CCSP [6] for C, and C++CSP [2] for C++.

This translation is usually non-trivial and rather problematic. In [9], we presented a methodology for developing verified concurrent applications in which developers: (i) specify the system's concurrent behaviour in CSP and *verify its correctness and further properties* using tools like FDR2; (ii) gradually refine it verifying the correctness of the transformation (again, using tools like FDR2); and finally, (iii) automatically translate the CSP_M implementation into Handel-C code, which can itself be compiled into a Hardware Description Language (HDL) to program Field-Programmable Gate Arrays (FPGAs).

The tool that supports the translation from CSP_M into Handel-C, csp2hc, accepts a subset of CSP_M that includes `SKIP`, `STOP`, sequential and parallel composition, recursion, prefixing, external and internal choice, alternation, guarded processes, datatypes, constants, functions, and some expressions. The translation of some of these constructs, however, was restricted. For instance, due to subtle differences in Handel-C's concurrency model, the translation of CSP_M parallel composition into Handel-C's `par` construct was only possible if: (i) all channels shared by the processes were in the synchronisation set (*e.g.* `cs` in the definition of sharing parallel composition at Page 48); and (ii) there was no multi-synchronisation (more than two processes synchronising on a given channel). In this paper, we present an approach to remove these restrictions.

Translations of process algebras into programming languages have already been presented. They target different programming languages like occam-2 [5], Ada [1], Java and C. Some of them have no tool support, whilst others have limited tool support. None of them, however, achieved a comprehensive support of CSP parallel composition as we do here. For instance, [4] proposes an automatic translation from CSP# [12] into code. They, however, only consider interleaving: parallel composition and multi-synchronisation are left aside. In [8], we presented a translation strategy from *Circus* [7] to Java. This strategy included the treatment of multi-synchronisation and its basic ideas are used here.

In Section 2, we introduce CSP_M, Handel-C, and the previous version of csp2hc. Section 3 describes the approach to implement CSP model of parallelism in Handel-C. In Section 4, we present a performance analysis of the translation and generated code. Finally, our conclusions and future work are in Section 5.

2 Background

In this section, we describe csp2hc's previous version and the languages involved in the translation focusing on the features used in the context of this paper.

2.1 CSP

CSP is a process algebra that can be used to describe systems composed by interacting components, which are independent self-contained processes with

[1] At `http://www.mentor.com/products/fpga/handel-c/`

```
--!!mainp SYSTEM
--!!int_bits 2
datatype ALPHA = a | b
datatype ID = Lt.ALPHA | unknown
channel enter, leave
channel cash, ticket, change : ID

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

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

```
--!!channel enter in within CUST
--!!channel leave in within CUST
--!!channel cash out within CUST
--!!channel ticket in within CUST
--!!channel change in within CUST
--!!arg id ID within CUST
CUST(id) =
 (enter -> cash!id ->
  (ticket.id -> change.id -> SKIP
  []change.id -> ticket.id -> SKIP));
 leave -> CUST(id)
CUSTOMERS =
   CUST(Lt.a) ||| CUST(Lt.b) ||| CUST(unknown)
PAID_PARKING = (CUSTOMERS
               [| {|cash,ticket,change,enter|} |]
               MACHINE) \ {|cash,ticket,change|}
SYSTEM = CAR [| {| enter,leave |} |] PAID_PARKING
```

Fig. 1. CSP$_M$ Example: a Paid Car Park

interfaces that are used to interact with the environment [11]. Most of the CSP tools, like FDR2 and ProBE, accept a machine-processable CSP, called CSP$_M$.

The two basic CSP$_M$ processes are `STOP` and `SKIP`; the former deadlocks, and the latter does nothing but terminate. The prefixing `a -> P` is initially able to perform only the event `a`; afterwards it behaves like process `P`. A boolean guard may be associated with a process: `g & P` behaves like `P` if the predicate `g` is true; it deadlocks otherwise. The operator `P1;P2` combines `P1` and `P2` in sequence. The external choice `P1[]P2` initially offers events of both processes. The performance of the first event or termination resolves the choice in favour of the process that performs either of them. The environment has no control over the internal choice `P1|~|P2`, in which the choice is resolved internally. The sharing parallel composition `P1[|cs|]P2` synchronises `P1` and `P2` on the events in the synchronisation set `cs`; events that are not listed occur independently. The alphabetised parallel composition `P1[|cs1|cs2|]P2` allows `P1` and `P2` to communicate in the sets `cs1` and `cs2`, respectively; however, they must agree on events in `cs1`$\cap$`cs2`. Processes composed in interleaving `P1|||P2` run independently. The event hiding operator `P`$\backslash$`cs` encapsulates the events that are in `cs`. Finally, `P[[a<-b]]` behaves like `P` except all occurrences of `a` in `P` are replaced by `b`. The CSP$_M$ interruption, untimed timeout, exceptions, linked parallel, and replicated operators are omitted here; they are not accepted by csp2hc.

By way of illustration, Figure 1 presents the specification of a parking spot. It contains special comments called directives (`--!!`), which give extra information to csp2hc, such as: information on whether simple synchronisation channels are input channels or output channels within a process; the types of processes arguments; the main behaviour of the system; the length of integers used in the system; and the moment in which internal choices should be resolved.

The process `PAID_PARKING` describes a parking spot with a pay and display machine that accepts cash, and issues tickets and change. First, we declare a datatype `ALPHA`: variables of type `ALPHA` can assume either value `a` or `b`. The next datatype, `ID`, represents identifications: the constructor `Lt` receives an `ALPHA` value and returns a value of `ID` (for example, `Lt.a`); another possibility is the `unknown ID`. After receiving the cash, the machine issues tickets and gives the change. The process `CUST` models a customer: after entering the parking spot, a customer must interact with the ticket machine: he inserts the `cash` into it, picks the `ticket` and the `change` in any order, and finally, `leave`s the parking spot. Customers have unique identification that guarantees that tickets and changes are only issued to the customer who inserted the cash. The identifications are used to instantiate each customer in process `CUSTOMERS`, which is defined as the interleaving of all customers. The paid parking spot is modelled by `PAID_PARKING` as a parallel composition of all customers and a machine; they synchronise on `cash`, `ticket`, `change`, and `enter`; all but `enter` are encapsulated. Finally, the main behaviour of the system, `SYSTEM`, is the parallel composition between the `CAR` and the parking. Using FDR2, we can verify that the `SYSTEM` is deadlock free and livelock free. Furthermore, using FDR2's refinement check, we can also verify that the `SYSTEM` satisfies the abstract specification that only requires that, after entering, a customer must leave before the next customer enters.

Despite being a simple example, this example was not accepted by the previous version of csp2hc. This is due to the existence of both (i) shared channels among the customers (*i.e* `enter`) that are not in the synchronisation channel set since customers are interleaved, and (ii) multi-synchronisation of a customer, the `CAR` and the `MACHINE` on channels `enter` and `leave`.

2.2 Handel-C

Handel-C is a procedural language, rather like occam, but with a C-like syntax. Its main purpose is the compilation into netlists to configure FPGAs or ASICs (Application-Specific Integrated Circuits). Although targeting hardware, it is a programming language with hardware output rather than a hardware description language. This makes Handel-C different from VHDL. A hardware design using Handel-C is more like programming than hardware engineering; this language is developed for programmers who have no hardware knowledge at all.

Handel-C offers a subset of C that includes common constructs like structures, functions, macros, arrays, pointers, logical operators (and their bitwise counterparts), and control flow constructs like `while` and `for` loops, `if` and `switch`. However, it does not include recursion and processor-oriented features like floating point arithmetic, which is supported through external libraries.

Handel-C extends C by providing constructs for describing parallel behaviour. The parallel construct `par{P; Q;}` executes instructions `P` and `Q` in parallel, which may communicate via channels. Its semantics corresponds to the CSP alphabetised parallel `P [|`α`(P) || `α`(Q)|] Q`, where α(`P`) and α(`Q`) denotes all communications of `P` and `Q`, respectively. The `prialt` statement selects one of the channels that are ready to communicate, and communicates via this channel. The only data type allowed in Handel-C is `int`, which can be declared with a fixed size.

By way of illustration, we present a simple `BUFFER` that receives an integer value through a channel `input` and outputs it through channel `output`. This buffer can be decomposed into a process `IN` that receives an integer value and passes it through channel `middle` to another process `OUT` that finally outputs this value. A possible CLIENT can interact with the BUFFER by sending an integer value via channel input and receiving it back via channel output. The Handel-C code presented below implements this interaction.

```
set clock = external "clock1";
chan int 8 input, output, middle;
void IN(){ int 8 v; while(1) { input?v; middle!v; } }
void OUT(){ int 8 v; while(1) { middle?v; output!v; } }
void BUFFER(){ par{ IN(); OUT(); } }
void CLIENT(){ int 8 v; input!10; output?x; }
void main(){ par { BUFFER(); CLIENT(); } }
```

We define an external clock named clock1, and declare the channels used in the system. The Handel-C function IN implements the process of same name. It declares a local variable v and starts an infinite loop: in each iteration, it receives a value via channel input, assigns it to v, and writes its value on middle. The function OUT is very similar; however, it receives a value via middle and writes it on output. The BUFFER is defined as the parallel composition of IN and OUT. The main function is the parallel composition of the BUFFER with the CLIENT.

2.3 The Translator csp2hc

The automatic translation from CSP_M to Handel-C is straightforward for some CSP_M constructs because Handel-C provides constructs that facilitate the description of parallel behaviour based on CSP concepts. The version of csp2hc presented in [9] mechanised the translation of a subset of CSP_M to Handel-C, which included SKIP, STOP, sequential and parallel composition, recursion, prefixing, external and internal choice, alternation, guarded processes, datatypes, constants, functions, and some expressions. It, however, restricted the use of some of these constructs like, for instance, parallel composition.

The implementation of concurrency in Handel-C differs from the CSP concepts. Handel-C has a degenerate kind of multi-way synchronisation, in which one writer and multiple readers can take part, but no participation control takes place: if just one reader and the writer are ready for communicating the synchronisation happens (the multi-synchronisation is not enforced like in CSP). For this reason, the translation of CSP_M parallel composition into Handel-C's par construct was restricted to cases in which there were no multi-way synchronisation, and shared channels between two processes composed in parallel were in the synchronisation channel set of the composition. This guaranteed that processes only synchronised on multi-shared channels when all parts involved were willing to synchronise on that channel, and that processes did not synchronise on channels that were not in the synchronisation channel set.

The extension of csp2hc to accommodate multi-synchronisation and interleaving on shared channels is not trivial. The former requires the implementation of a centralised protocol in which a controller determines when the synchronisation is allowed to happen and the latter requires the translation of renaming. In the next section, we present the results that made it possible to deal with multi-synchronisation and interleaving on shared channels within csp2hc.

3 Parallelism in csp2hc

The CSP parallel composition cannot be directly translated into Handel-C's parallel constructor, `par`, for two reasons: (1) `par` does not enforce synchronisation between multiple parts (multi-synchronisation); and (2) `par` does not prevent the synchronisation on a channel if processes have access to the channel. In our example, such naïve translation would contain the following Handel-C code.

```
void PAID_PARKING(){ par{ CUSTOMERS(); MACHINE(); } }
void SYSTEM(){ par{ CAR(); PAID_PARKING(); } }
void main(){ SYSTEM(); }
```

This implementation, however, is wrong because it does not prevent customers synchronising on `enter` and does not enforce the multi-synchronisation on `enter` between the `CAR`, the `MACHINE`, and one of the customers. In this section, we describe the approach used in csp2hc to accomplish this behaviour.

Our approach has two restrictions that are automatically verified by csp2hc. The first restriction guarantees communications on synchronised channels by requiring the existence of exactly one writer for every channel that is being shared in parallel compositions. For example, `c?x -> SKIP [|{|c|}|] c?y -> SKIP` is not accepted by the approach. Its translation would result in a code in which both parallel branches are reading on a channel, hence, waiting to some other process to write on it. This would characterise a deadlock in the implementation that does not correspond to the specified behaviour in $\mathsf{CSP_M}$, which does not deadlock and terminates. The second restriction guarantees that every parallel branch is either a reader or a writer to every channel, but not both. By way of illustration, `c!0 -> c?x -> SKIP [| {| c |} |] c?x -> c?y -> SKIP` is not accepted by the approach. This process satisfies the first restriction but not the second restriction because the left branch treats `c` as both an output and an input. In this example, a similar deadlock state is reached in the Handel-C code.

As we discuss in Section 4, the solution follows the expected performance results discussed in [14]. The computational arrangements for allowing any of the synchronising processes to back off (which CSP allows) is even more costly than allowing both parties to back off during channel synchronisation. For this reason, we only use the solutions presented here if there are multi-synchronised channels or if we need to enforce interleaving of channels. Otherwise, the parallel composition is directly translated as presented in [9].

The solution for multi-synchronisation is based on a protocol we presented in [15] that controls the accesses to the channels in a parallel composition and the solution to enforce the interleaving is based on $\mathsf{CSP_M}$ renaming. Both solutions use the concept of parallel branch that we describe in the sequel. Their application directly affects the translation of prefixing, external choice and the arguments of the processes within the system, which are slightly changed.

3.1 Analysis of Parallel Compositions

Our tool starts the branch identification from the main process given in the directive `--!!mainp` (in our example `SYSTEM`) and sets an identification to each

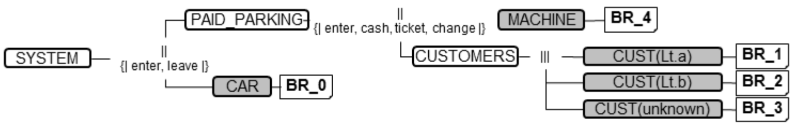

Fig. 2. Example Branches Identification

one of the running parallel branches. The result of this identification process in our example is presented in Figure 2.

The implementation of the concept of branch reuses the solution for datatypes presented in [9] by considering an implicit `datatype BRANCH = B_0 | ... | B_4`. As a result, we have the following extra lines of code.

```
#define BRANCH unsigned int 3
#define BR_0 0
...
#define BR_4 4
```

The translation of a parallel branch considers the current identification of the branch being translated: every process has an extra argument that identifies the branch from which it has been invoked. Our tool translates the left branch first and, before translating the right branch, it updates the current branch identification (`BR_ID`) by incrementing it with the number of sub-branches of the left branch. We have the following translation for the main process.

```
void main(){ BRANCH BR_ID; BR_ID = 0; SYSTEM(BR_ID+0); }
inline void SYSTEM(BRANCH BR_ID){ par{ {CAR(BR_ID+0); }; { PAID_PARKING(BR_ID+1); } } }
```

In the main process, we declare `BR_ID` and initialise it to zero. The `SYSTEM` behaves like a parallel composition between `CAR` and `PAID_PARKING`; they are parametrised by the branch identification. The translation of `CAR` is the first one, hence, the value `BR_ID + 0` is used as argument. Nevertheless, this process itself is a branch; hence, the value `BR_ID + 1` is used as argument to invoke `PAID_PARKING`. The translation of processes `PAID_PARKING` and `CUSTOMERS` though are slightly different as we can see in the code below.

```
inline void PAID_PARKING(BRANCH BR_ID) {
  par{ {CUSTOMERS(BR_ID+0);} ; {MACHINE(BR_ID+3);} } }
inline void CUSTOMERS(BRANCH BR_ID) {
  par{ {CUST(BR_ID+0,ID_Lt_LUT[a]);};
       {par{ {CUST(BR_ID+1,ID_Lt_LUT[b]);}; {CUST(BR_ID+2,unknown);} };} } }
```

In the translation of `PAID_PARKING`, the process `MACHINE` is given the local variable `BR_ID` incremented by three because the left branch, `CUSTOMERS`, has three branches. In the translation of `CUSTOMERS`, the first invocation to `CUSTOMER` does not increment the `BR_ID`; the following invocations, though, do increment it.

The branches identification is used in an analysis of the parallel structure of the system that results on a list of synchronisation for each channel. In our implementation, a synchronisation is a set that contains the identification of all branches that take part in the synchronisation. By way of illustration, in our example, there are three possibilities of synchronisation on `enter`: the `CAR` (BR_0) and the `MACHINE` (BR_4) take part in all of them; the third (and last) element is one of the clients. The list of synchronisations for the channel `enter` is $\langle\{\text{BR\_0}, \text{BR\_4}, \text{BR\_1}\}, \{\text{BR\_0}, \text{BR\_4}, \text{BR\_2}\}, \{\text{BR\_0}, \text{BR\_4}, \text{BR\_3}\}\rangle$. Similar mappings are created for each individual channel.

The branches identification and the synchronisation list play an important role in both solutions presented in this paper: the multi-synchronisation protocol and channel interleaving described in Sections 3.2 and 3.3 that follow. A synchronisation whose cardinality is greater than two characterises a multi-synchronised channel and a synchronisation list with more than one element indicates the need to enforce the interleaving on that channel.

csp2hc's analysis of the parallel structure is based on the channels rather than on the events. For this reason, the translation of some specifications might use the solutions presented in Sections 3.2 and 3.3 unnecessarily. For instance, the customers are composed in interleaving and our strategy uses the solution presented in Section 3.3 to enforce the interleaving on `ticket` because all customers use this channel. Nevertheless, this is not necessary because the synchronisation on `ticket` is parameterised by the customers identification. The translation of such channels uses an array of channels whose size is defined by the cardinality of the channel type. Each element of the array is a different channel that corresponds to a different value. Hence, despite using the same channel, different customers never synchronise (`CUST(unknown)` and `CUST(Lt.a)` work on `ticket[unknown]` and `ticket[Lt_a]`). Although being semantically correct, the use of the protocol adds performance costs (see Section 4) unnecessarily. An optimisation to remove this unneeded use of the protocol is in our research agenda. It requires a static analysis of CSP_M expressions that allows comparing events rather than only channels.

3.2 The Multi-synchronisation Protocol

In [15], we used the *Circus* refinement calculus to develop a protocol that implements an abstract multi-way synchronisation using only pairwise synchronisation: each multi-synchronised channel has a central controller and references to this channel are implemented as a client of this controller. In what follows, we extend the protocol from [15] by allowing both multi-synchronised channels and interruptions (possibly carrying values) to take part in external choices.

Controllers. The controllers are implemented as an infinite loop in which it iteratively runs a two-phase commitment protocol described later in this section. Hence, termination of the controller needs to be guaranteed by external managers. The first one, `PManager`, monitors the main behaviour of the system and communicates its termination to the controllers' manager using `endManager`.

```
inline void PManager() { BRANCH BR_ID; BR_ID = 0; SYSTEM(BR_ID+0); endManager!syncout;}
```

The controllers' manager (`CManager`) receives this communication and propagates it to each controller `MSyncController_`i using channel `end_controller_`i. Each client receives message from the controller on channel `fromSync` and sends

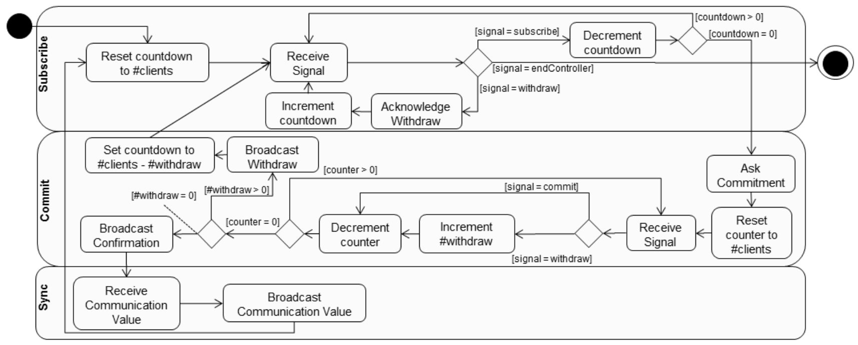

Fig. 3. Controller Activity Diagram

message to the controller on channel `toSync`. A controller has reference pointers to two arrays of channels `fromSync[]` and `toSync[]`. These arrays contain all controller-client communication channels. They are used as argument in each controller's instantiation. We present below the controllers' manager of a system with two multi-synchronised channels `A` and `B`.

```
inline void CManager (){
    chan SYNC end_controller_A; chan SYNC end_controller_B;
    par{ seq{ endSystem?syncin;
              par{ end_controller_A!syncout; end_controller_B!syncout; }; };
         MSyncController_A(&toSync_A[0], &fromSync_A[0] , &end_controller_A);
         MSyncController_B(&toSync_B[0], &fromSync_B[0] , &end_controller_B); }; }
```

The Handel-C `main` function is the parallel composition of both controllers.

```
void main(){ par{ {PManager();} ; {CManager();} } }
```

Handel-C's `prialt` construct is used in the implementation of the controller to offer a choice among various channels. This construct, however, cannot be changed dynamically because Handel-C requires all choices to be statically defined. For this reason, our Handel-C implementation of the protocol provides a different version of the controller for each possible number of multi-synchronisation parts. The behaviours of these versions are almost identical; they only differ in the number of elements in the arrays of channels that are offered in the choices. This is due to the complexity and length. We refrain from presenting the details of the resulting code, which can be found at the project webpage[2]. In what follows, we informally described the protocol workflow.

In Figure 3, we present the controllers' activity diagram. It can be divided into three phases whose composition is presented below: subscription, commitment and synchronisation. Only in some of these phases, the controller allows clients to withdraw from the synchronisation.

[2] Project webpage at `http://www.dimap.ufrn.br/~marcel/research/csp2hc/`

Subscribe The controller waits for the clients to indicate their intention to synchronise on the channel (subscribe). A local `countdown` controls the loop that implements the corresponding tail recursion in the original CSP implementation of the protocol. When all clients have subscribed, the controller moves to the commitment phase. While receiving subscriptions, if the controller receives an indication to terminate, it does so. The controller does not need to broadcast the withdraw because a termination signal will only arrive when the clients have terminated.

Commit The controller asks all clients to commit to the synchronisation and receives answers from all of them. If all clients answer positively, the controller broadcasts a confirmation to all clients and moves to the synchronisation phase.

Sync The controller receives the communication value from the writer and broadcasts this value to all other clients. The controller recurses and goes back to the initial state of the subscription phase.

Withdraw During the subscription phase, if a client withdraws, the controller acknowledges the signal, increments the `countdown` and keeps receiving signal from other clients. If, however, a client withdraws in the commitment phase, the controller broadcasts the withdraw and goes back to the subscription phase. Nevertheless, it expects new signals only from those clients that have withdrawn. Hence, the `countdown` is set to the difference between the total number of clients and the number of clients that have withdrawn.

Clients. At the other end of the protocol, we have the multi-synchronisation clients, which are used in the translation of the processes from the original CSP specification. In this translation, however, communications and choices that involve multi-synchronised channels are replaced by an invocation to a client's execution. The client offers all channels involved in the choice possibly interacting with different controllers. Its execution terminates only when a successful communication takes place. For simple communication, the termination of the client's execution indicates a successful multi-synchronisation. For external choices, however, the termination of the client returns an identification of the communication (either multi-synchronised or not) that happened. The behaviour of the process after this communication depends on this information.

In Figure 4, we present the client's activity diagram. Its behaviour can also be divided into the phases of subscription, commitment and synchronisation. The client's phases are composed as follows.

Subscribe The client sends a subscription to the multi-synchronisation controller. It is possible, though, that a client is involved in many multi-synchronisations. In such cases, this signal is sent to all the corresponding controllers. The client waits to receive a confirmation request from one of the controllers. When such a signal arrives, it moves to the next phase.

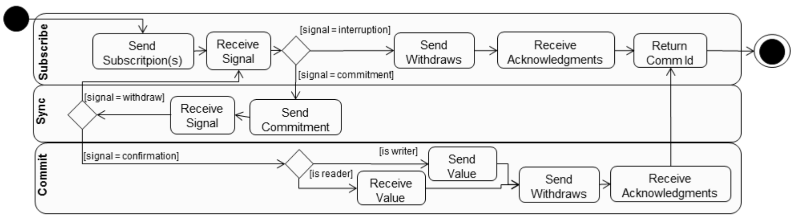

Fig. 4. Clients Activity Diagram

Commit The client commits to communicating with the controller and refrains from communicating on other channels. The client is not allowed to withdraw. It receives from the controller either a confirmation or a withdraw. If the former is received, the client moves to the synchronisation phase.

Sync If the client is the writer, it sends the communication value to the corresponding controller. If, however, the client is a reader, it receives this value from the controller. Finally, it sends a withdraw signal to all other controllers, terminates and returns an indication of a successful communication on the corresponding multi-synchronised channel.

Withdraw During subscription, non-multi-synchronised channels (interruptions) may also happen. In such cases the client sends a withdraw to all controllers that are interacting with it, terminates, and returns a successful communication on the interruption. In the commitment phase, if the controllers sends withdraws, the client returns to the subscription phase. It, however, does not send a new subscription because the controller is already aware of its intention.

3.3 Forced Interleaving

As explained in Section 3, a naïve translation of our example would result in an incorrect implementation because it would not prevent customers to synchronise on `enter`. In this section we present a strategy that transforms the specification in a correct manner to enforce the interleaving on `enter` between the customers.

The strategy is based on the synchronisation information described in Section 3.1 and makes use of CSP renaming. The main idea is to apply the transformation that follows at the source level before the actual compilation. The transformation consists of the following phases: (1) Definition of each branch's renaming for each channel; (2) Creation of renamed copies of the branches; (3) Translation of the extended specification. In what follows, we present a detailed description of each of these phases.

In the **definition of each branch's renaming**, csp2hc defines what renaming must be applied to each individual branch. Formally, for every channel c

and branch *b* in the system, the renaming `[[`*c* `<-` *c_i*`]]` must be applied to *b* if, and only if, the *i-th* element of the synchronisation list contains *b*. For instance, processes `CAR` (`BR_0`) and `MACHINE` (`BR_4`) take part in all synchronisation of channel `enter`; the renaming `[[enter <- enter_0, enter <- enter_1, enter <- enter_2]]` needs to be applied to them. On the other hand, each client takes part in only one synchronisation on `enter`. For example, `CUST(Lt.a)` (`BR_1`) needs to be renamed using `[[enter <- enter_0]]`. The renaming definition of each branch is done in an identical manner for all other channels in the system.

In the **creation of renamed copies of the branches**, the original specification is extended with the declaration of the new channels and the definition of renamed copies of all processes. The copies are needed because a process may be instantiated in different branches requiring different renamings. In our example, we have three renamed copies of `CUST` (one for each instantiation), and one renamed copy of every other process in the system. The new channels are also included in the synchronisation channel sets in which the original channel is present. For conciseness, we present below only the changes related to `enter`. Our example, however, also renames channels `leave`, `cash`, `ticket`, and `change`.

```
CAR_RNO     = CAR [[enter <- enter_0, enter <- enter_1, enter <- enter_2, ...]]
MACHINE_RNO = MACHINE [[enter <- enter_0, enter <- enter_1, enter <- enter_2, ...]]
CUST_RNO(id) = CUST(Lt.a)[[enter <- enter_0, ...]]
CUST_RN1(id) = CUST(Lt.b)[[enter <- enter_1, ...]]
CUST_RN2(id) = CUST(unknown)[[enter <- enter_2, ...]]
CUSTOMERS_RNO =  CUST_RNO(Lt.a) ||| CUST_RN1(Lt.b) ||| CUST_RN2(unknown)
PAID_PARKING_RNO = (CUSTOMERS_RNO
                    [| {|enter,enter_0,enter_1,enter_2,cash,...,ticket,...,change,...|} |]
                    MACHINE_RNO) \ {|cash,...,ticket,...,change,...|}
SYSTEM_RNO = CAR_RNO [| {| enter,enter_0,enter_1,enter_2,leave,... |} |] PAID_PARKING_RNO
```

The extended specification is finally translated resulting in an implementation that correctly implements multi-synchronisation and interleaving.

The **translation of the extended specification** follows the strategy from [9] extended with multi-synchronisation as discussed in Section 3.2. Hence, this translation naturally deals with multi-synchronised channels like `enter_0`. A further extension needed to the original strategy presented in [9] was the translation of renaming explained below.

The translation of functional renaming (channels are renamed once) is rather simple: the original channel is simply replaced by the new channel. For example, `CUST_RNO(id)` is translated as `CUST(id)` but replaces `enter` to by `enter_0`.

The translation of non-functional renaming is slightly more elaborate. In these cases, a channel is renamed to more than one new channel, like in `CAR_RNO`. The result of such translations replaces references to the original channel to an external choice between all new channels. By way of illustration, we present below the specification that corresponds to the translation of `CAR_RNO`.

```
CAR_RNO = (enter_0 -> SKIP [] enter_1 -> SKIP [] enter_2 -> SKIP);
          (leave_0 -> SKIP [] leave_1 -> SKIP [] leave_2 -> SKIP); CAR_RNO
```

It is important to emphasize that, as expected, the external environment is oblivious of the renaming used in our strategy. This is achieved by forbidding channels that are used to communicate with the environment (marked as `buses`

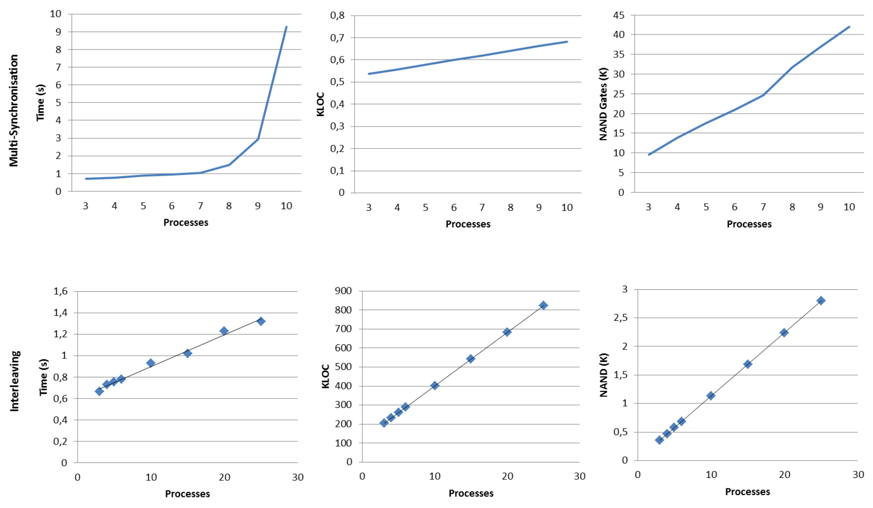

Fig. 5. Experiments Results

using directives) to be interleaved. Hence, the interleaved channels are not visible to the environment and their renaming does not affect the system's interface. The overall resulting code can be found at the project's webpage.

3.4 Formalisation

In [15], we presented a calculational approach to prove the correctness of a protocol for multi-synchronised channels that are not part of an external choice and do not communicate values. The protocol extension presented here accepts both multi-synchronised channels and interruptions (possibly carrying values) in external choices. A relatively simple adaptation of the proof from [15] guarantees the validity of our extension.

Using FDR2, we also verified that a specification with multi-synchronised channels and interruptions being offered in a choice is refined by its corresponding instance of the multi-synchronisation protocol. This verification ensures the correctness of the protocol for a comprehensive instance of the problem with bounded channel types. The same approach was used to ensure the correctness of the translation of interleaved channels.

4 Experiments

In our experiments, we translated simple $\mathsf{CSP_M}$ specifications containing multi-synchronised and interleaved channels and compiled the resulting Handel-C code. The experiments were executed on an Intel i3, 2.53GHz, with 3GB RAM, running Windows 7 (64 bits). We considered the translation time and dimensions of the compiled code like number of lines of code (in thousands - KLOC) and the number of NAND gates (NANDs) and Flip Flops (FFs). The experiments were executed with an increasing number of processes taking part in the multi-synchronisation and interleaving, and we monitored the growth rate of the collected data. These rates were almost identical for the number of NANDs and FFs; hence, we omit below the results on the number of FFs.

Figure 5 presents the results of the experiment. For multi-synchronisation, they presented an exponential growth in the translation time, which enforces csp2hc users to make limited use of this feature. The growth rate of the generated code and its compilation, however, proved to be linear. This indicates the practical usefulness of the protocol on a large scale. Nevertheless, optimisation in the translation process is essential. The results for interleaving presented a linear growth rate and allowed us to consider a much larger number of processes. In these experiments, the growth rates of the generated code and its compilation were linear indicating the scalability of our solution.

5 Conclusions

In [9], we presented a translation from $\mathsf{CSP_M}$ to Handel-C and a tool that automates this translation. They foster a methodology that starts from a $\mathsf{CSP_M}$ specification, which is verified, gradually refined, and automatically translated into Handel-C code. The results presented here provide a further step towards providing a framework that fully supports the development of verified hardware.

Previous versions of csp2hc supported a useful subset of CSP, but imposed restrictions on parallel composition: its translation was allowed only if channels shared between the processes were in the synchronisation set and not multi-synchronised. In this paper, we present translation strategies to both limitations. Although some conditions are still required, we considerably extend the translation strategy of csp2hc by providing means to translate multi-synchronisation and interleaving as those of the example presented in Figure 1.

A relatively simple adaptation of the proof of the protocol we used as a basis presented in [15] guarantees the validity of our extensions. We have also verified that an abstract specification with various multi-synchronised channels and interruptions being offered in a choice is refined by its instance of the multi-synchronisation protocol. The same approach has been used to ensure the correctness of the translation strategy of interleaved channels.

Using csp2hc, we are able to translate some of the classical $\mathsf{CSP_M}$ problems (*e.g.* the dining philosophers) including many of the examples provided with the FDR2 distribution and a complex specification provided by our industrial partner that involves multi-synchronisation and interleaving. There are, however, still optimisations and extensions to be done in csp2hc.

The experiments demonstrated the feasibility of the multi-synchronisation protocol for large networks. The translation, however, presented an exponential growth in time. For this reason, the current translation of multi-synchronisation is feasible only for small networks (up to 11 in our example). An optimisation in the translation process is essential and left as future work. The investigation of the performance of a purely distributed protocol [10] is in our research agenda. Furthermore, in a near future, we will also address an optimisation to remove the unneeded use of the extensions discussed in this paper.

Specifications not accepted by csp2hc need to be manually transformed. This transformation is often possible and can be verified using FDR2. A complete automatic translation from $\mathsf{CSP_M}$ to Handel-C requires the translation of further $\mathsf{CSP_M}$ constructs and expressions, which includes FDR2's functional language.

References

1. Burns, A., Wellings, A.: Concurrency in Ada, 2nd edn. Cambridge University Press (November 1997)
2. Brown, N., Welch, P.: An Introduction to the Kent C++CSP Library. In: Broenink, J.F., Hilderink, G.H. (eds.) Communicating Process Architectures 2003, pp. 139–156 (September 2003)
3. Formal Systems Ltd. FDR: User Manual and Tutorial, version 2.82 (2005)
4. Lin, S.-W., Liu, Y., Hsiung, P.-A., Sun, J., Dong, J.S.: Automatic generation of provably correct embedded systems. In: Aoki, T., Taguchi, K. (eds.) ICFEM 2012. LNCS, vol. 7635, pp. 214–229. Springer, Heidelberg (2012)
5. Hinchey, M.G., Jarvis, S.A.: Concurrent Systems: Formal Development in CSP. McGraw-Hill, Inc., New York (1995)
6. McMillin, B., Arrowsmith, E.: CCSP-A Formal System for Distributed Program Debugging. In: Proceedings of the Software for Multiprocessors and Supercomputers, Theory, Practice, Experience, Moscow, Russia (September 1994)
7. Oliveira, M.V.M.: Formal Derivation of State-Rich Reactive Programs using *Circus*. PhD thesis, Department of Computer Science, University of York (2006)
8. Oliveira, M., Cavalcanti, A.: From*Circus* to JCSP. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 320–340. Springer, Heidelberg (2004)
9. Oliveira, M., Woodcock, J.: Automatic Generation of Verified Concurrent Hardware. In: Butler, M., Hinchey, M., Larrondo-Petrie, M.M. (eds.) ICFEM 2007. LNCS, vol. 4789, pp. 286–306. Springer, Heidelberg (2007)
10. Parrow, J., Sjödin, P.: Designing a multiway synchronization protocol. Computer Communications 19(14), 1151–1160 (1996)
11. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice-Hall Series in Computer Science. Prentice-Hall (1998)
12. Sun, J., Liu, Y., Dong, J.S., Chen, C.: Integrating specification and programs for system modeling and verification. In: Proceedings of the Third IEEE International Symposium on Theoretical Aspects of Software Engineering, pp. 127–135. IEEE Computer Society, Washington, DC (2009)
13. Welch, P.H.: Process oriented design for Java: concurrency for all. In: Arabnia, H.R. (ed.) Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, pp. 51–57. CSREA Press (June 2000)
14. Welch, P.H., Wood, D.C.: Higher Levels of Process Synchronisation. In: Bakkers, A.W.P. (ed.) Proceedings of WoTUG-20: Parallel Programming and Java, pp. 104–129 (1997)
15. Woodcock, J.C.P.: Using *Circus* for Safety-Critical Applications. Electronic Notes Theoretical Computer Science 95, 3–22 (2004)