HAL
open science

On the Semantics of Communicating Hardware Processes and their Translation into LOTOS for the Verification of Asynchronous Circuits with CADP

Hubert Garavel, Gwen Salaun, Wendelin Serwe

To cite this version:

Hubert Garavel, Gwen Salaun, Wendelin Serwe. On the Semantics of Communicating Hardware Processes and their Translation into LOTOS for the Verification of Asynchronous Circuits with CADP. Science of Computer Programming, 2009. inria-00381642

HAL Id: inria-00381642

https://inria.hal.science/inria-00381642

Submitted on 6 May 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

On the Semantics of Communicating Hardware Processes and their Translation into LOTOS for the Verification of Asynchronous Circuits with CADP

Hubert Garavel, Gwen Salaün, Wendelin Serwe *

INRIA Centre de Recherche Grenoble – Rhône-Alpes, VASY project team, France

Abstract

Hardware process calculi, such as CHP (*Communicating Hardware Processes*), BALSA, or HASTE (formerly TANGRAM), are a natural approach for the description of asynchronous hardware architectures. These calculi are extensions of standard process calculi with particular synchronisation features implemented using handshake protocols. In this article, we first give a structural operational semantics for value-passing CHP. Compared to the existing semantics of CHP defined by translation into Petri nets, our semantics is general enough to handle value-passing CHP with communication channels open to the environment, and is also independent of any particular (2- or 4-phase) handshake protocol used for circuit implementation. We then describe the translation of CHP into the process calculus LOTOS (ISO standard 8807), in order to allow asynchronous hardware architectures expressed in CHP to be verified using the CADP verification toolbox for LOTOS. A translator from CHP to LOTOS has been implemented and successfully used for the compositional verification of two industrial case studies, namely an asynchronous implementation of the DES (*Data Encryption Standard*) and an asynchronous interconnect of a NoC (*Network on Chip*).

Key words: Asynchronous circuit, asynchronous logic, asynchrony, CHP, formal method, GALS architecture, handshake protocol, hardware architecture, hardware design, LOTOS, modelling, network on chip, process calculus, specification, structured operational semantics, translation, validation, verification

1. Introduction

In the currently predominating synchronous approach to hardware design, a global clock is used to synchronise all parts of a circuit. Unfortunately, the global clock requires significant chip space and power. Asynchronous architectures [1] attempt at avoiding the issues arising from a global clock. There are two main kinds of asynchronous architectures:

- GALS (*Globally Asynchronous, Locally Synchronous*) architectures replace the global clock with several local clocks, by splitting a design into asynchronous parts, each with a synchronous clock domain, and
- asynchronous circuits totally remove the notion of clock.

* Corresponding Author. Address: INRIA, VASY project team, 655, avenue de l'Europe, F-38334 St. Ismier Cedex, France
Email address: `Wendelin.Serwe@inria.fr` (Wendelin Serwe).
URL: `http://www.inrialpes.fr/vasy/people/Wendelin.Serwe` (Wendelin Serwe).

In these architectures, the different parts of a circuit evolve concurrently at different speeds, with no constraints on communication delays. Such asynchronous designs reduce power consumption, enhance modularity, and increase performance [2]. However, they raise problems that do not exist in the synchronous approach, e.g. proving the absence of deadlocks in a circuit. Also, there is no widely established asynchronous design methodology with mature tool support available up to now.

To master the design of asynchronous circuits, adequate description languages are necessary. For this purpose, several process calculi (or process algebras) dedicated to the description of asynchronous hardware have been proposed, such as CHP (*Communicating Hardware Processes*) [3], BALSA [4], and HASTE [5] (formerly TANGRAM [6]). These *hardware* process calculi are based on similar principles as *standard* process calculi [7,8] such as ACP, CCS, CSP, LOTOS, μCRL, etc. Especially, they provide operators such as non-deterministic choice, sequential, and parallel composition. However, compared to standard process calculi, they offer extensions to express the low-level aspects of hardware communications, such as the implementation of synchronisation by handshake protocols. In particular, communication in CHP, BALSA, or HASTE is not necessarily atomic as it is in standard process calculi, and may combine message-passing with shared memory communication. For instance, the *probe* operation [9] of CHP allows to check whether the communication partner is ready for a communication or not, but without performing the communication actually. CHP, BALSA, and HASTE are supported by synthesis tools that can generate the implementation of a circuit from its process algebraic description. For instance, the TAST tool [10] can generate netlists from CHP descriptions.

In this article, we focus on CHP, which is used in the geographical area of the authors by major actors, such as STMicroelectronics, France Telecom R&D, and the CEA/Leti laboratory. Although there exist synthesis tools for CHP, the verification of CHP descriptions still lacks tool support. Continuing previous work [11] towards verification of CHP, our goal is to enable the application of the CADP toolbox [12] developed for standard process calculi, such as the international standard LOTOS [13], to the particular case of CHP. Therefore, we study the translation from CHP to LOTOS, and consequently the formal semantics of CHP.

So far, the semantics of CHP has only been partially formalised by a translation into Petri nets; this formalisation was based on a subset of CHP supporting only pure synchronisation and closed systems [14]. A formal semantics for full CHP is mandatory to ensure a safe development and verification of CHP designs. In this article, we address this problem by giving a formal SOS (*Structural Operational Semantics*) [7, chapter 3] semantics for value-passing CHP with communication channels open to the environment.

We present in a second step a translation algorithm from full CHP into LOTOS. This algorithm is based on a structural induction on the CHP description. To minimise the state space corresponding to the generated LOTOS code, our translation implements an optimisation based on *code specialisation*. This optimisation technique avoids as much as possible the generation of LOTOS gates and processes by distinguishing several translation strategies for CHP channels depending on the use of probes in the CHP specification. A CHP to LOTOS translator has been implemented and successfully used for the compositional verification of two industrial case studies.

The remainder of the article is organised as follows. Section 2 presents the main features of the CHP hardware process algebra, putting emphasis on the probe operation, an original feature of CHP. Section 3 defines an SOS semantics for CHP. Section 4 presents translation rules from CHP into LOTOS. Section 5 briefly describes the CHP to LOTOS translator implementing these translation rules, and reports about its application to two industrial case studies, namely an asynchronous implementation of the DES (*Data Encryption Standard*) [15] and an asynchronous communication interconnect of a NoC (*Network on Chip*) [16]. Finally, Section 6 compares our approach with related work, and Section 7 gives some concluding remarks.

2. Main Features of CHP

In this section, we focus on the behavioural part of CHP [10], and omit additional structures, such as modules and component libraries, which are intended to designers convenience but have no impact on the operational semantics. We do not detail the low-level aspects of concrete syntax, but present a high-level view of CHP, focusing on abstract syntax and semantics features.

2.1. *Syntax*

A CHP *description* is a tuple $\langle \mathcal{C},\ \mathcal{X},\ \hat{B}_0 \parallel \cdots \parallel \hat{B}_n \rangle$ consisting of a finite set of *channels* $\mathcal{C} = \{c_1, \ldots, c_m\}$ for handshake communication, a finite set of typed *variables* $\mathcal{X} = \{x_1, \ldots, x_l\}$, and a finite set of concurrent *processes* $\hat{B}_i$ communicating by the channels $\mathcal{C}$. Let $\mathcal{C}_i$ be the set of channels used by $\hat{B}_i$. Without loss of generality, we suppose that all identifiers (channels and variables) are distinct.

Each variable is local to a single process, and there is no shared variable between processes. Let $\mathcal{X}_i$ be the set of local variables of process $\hat{B}_i$, the set $\mathcal{X}$ being the disjoint union of the n sets $\mathcal{X}_0, \ldots, \mathcal{X}_n$. CHP provides a set of predefined data types (booleans, natural numbers, fixed-length bit vectors, and one-dimensional arrays of bit-vectors) together with a set of predefined side-effect-free *operations*, written $f_1, \ldots, f_k$ (logic, arithmetic, as well as shift and rotation operations); the list of predefined types and operations is detailed in [10]. Variables are typed; the type of a variable x is written as $type(x)$. $\mathcal{V}$ stands for the set of value expressions built using the predefined data types and operations.

A channel c is either binary (if it connects two processes) or unary (if it connects a process and the environment); in the latter case, c is also called a *port*; the predicate $port(c)$ is *true* iff c is a port. Channels are typed, using the same set of predefined data types already used for variables; the type of a channel c is written as $type(c)$. Channels are unidirectional, i.e. a process $\hat{B}_i$ must use a given channel c only for *emissions* or only for *receptions*; in such a case, we say that $\hat{B}_i$ is an *emitter* (respectively, a *receiver* on c). Furthermore, a process is either always *active* or always *passive* for a channel c; all communications on c must be initiated by the process active for c. This distinction between active and passive is also present in other hardware process calculi such as HASTE and BALSA. Note that these two process attributes (emitter/receiver and active/passive) are statically defined and orthogonal, e.g. passive emission is possible and useful in applications as discussed in [16, Section 4.2]. In the examples of this article, however, emission is generally assumed active. Binary channels can only connect matching processes, i.e. for each binary channel, there is one emitter and one receiver, as well as one active and one passive process, both notions being orthogonal. For each process $\hat{B}_i$, we define a function H_i that maps channels to elements of the set $\{neutral, active, passive\}$; for a channel c, $H_i(c) = active$ (respectively, *passive*) iff process $\hat{B}_i$ is active (respectively, passive) for channel c; otherwise, i.e. if $\hat{B}_i$ never uses c, $H_i(c) = neutral$.

A CHP process B is described using communication actions, assignments, collateral and sequential compositions, and nondeterministic guarded commands. B corresponds to any piece of behaviour, whereas we note $\hat{B}_i$ a process definition belonging to the top level CHP description "$\hat{B}_0 \parallel \cdots \parallel \hat{B}_n$". In the following abstract syntax, lower case identifiers stand for terminals and upper case identifiers stand for non terminals.

$B ::= \mathbf{nil}$	*deadlock*[1]
$\mid\ \mathbf{skip}$	*null action*
$\mid\ c\mathbf{!}V$	*emission on channel c*
$\mid\ c\mathbf{?}x$	*reception on channel c*
$\mid\ x\mathbf{:=}V$	*assignment*
$\mid\ B_1\mathbf{;}\ B_2$	*sequential composition*
$\mid\ B_1\mathbf{,}\ B_2$	*collateral composition*
$\mid\ \mathbf{@}[V_0 \Rightarrow B_0\mathbf{;}\ T_0\ \ldots\ V_n \Rightarrow B_n\mathbf{;}\ T_n]$	*guarded commands*
$T ::= \mathbf{break} \mid \mathbf{loop}$	*terminations*
$V ::= x \mid f(V_1, \ldots, V_n)$	*value expression*
$\mid\ c\mathbf{\#}V \mid c\mathbf{\#}$	*probe on a passive channel*

Collateral composition has higher priority than sequential composition. Brackets can be used to express the desired behaviour, e.g. "$B_1\mathbf{,}\ (B_2\mathbf{;}B_3)$".

[1] The deadlocking process "**nil**" is not present in the version of CHP implemented in TAST [10], but is required in Section 3 for a proper definition of the operational semantics.

The precise semantics of these constructs will be explained in Section 3. Most of them have a meaning as is usual in process calculi. Here we only elaborate on the non-standard constructs.

2.2. *Informal Semantics of Parallel Composition in CHP*

The collateral composition "**,**" and the parallel composition of processes "$\|$", which is used at the top level in "$\hat{B}_0 \| \cdots \| \hat{B}_n$", correspond to two different notions of concurrency. The collateral composition "**,**" specifies concurrent execution without any communication, whereas the parallel composition "$\|$" specifies concurrent execution with handshake communications between processes. In a process "B_1**,**B_2", if B_1 modifies a variable x, B_2 must neither access the value of x nor modify x, and the sets of channels used by B_1 and B_2 must be disjoint (which also prohibits two interleaved emissions or receptions on a same channel). In $\hat{B}_1 \| \hat{B}_2$, there are also no shared variables between $\hat{B}_1$ and $\hat{B}_2$.

2.3. *Informal Semantics of Guarded Choice in CHP*

A guarded command "B = **@**[$V_0 \Rightarrow B_0$**;** T_0 ... $V_n \Rightarrow B_n$**;** T_n]" expresses the choice between the behaviours B_0, ..., B_n. A B_i can only be chosen if the boolean value V_i, called *guard*, is valid. The "**break**" keyword indicates that the execution of B terminates; the "**loop**" keyword indicates that B must be executed once more, thus allowing loops to be specified in CHP. The choice is *internal*, i.e. other concurrent behaviours do not have control nor influence on the choice. The version of CHP implemented in TAST [10] also allows deterministic guarded commands which are a particular case of nondeterministic guarded commands with mutually exclusive guards; without loss of generality we consider only nondeterministic guarded commands in this article.

2.4. *Informal Semantics of Handshake Communication in CHP*

Communication between concurrent processes in CHP is implemented by means of hardware handshake protocols. As mentioned in [14], there exists several implementation protocols, such as the *2-phase protocol* (based on transition signalling) and the *4-phase protocol* (based on level signalling). Depending on the chosen protocol, each CHP channel c will be implemented physically as a set of wires x_c needed to carry data transmitted on c, and two control wires c_{req} and c_{gr} implementing an access protocol to x_c. Common to both protocols is that a communication on a channel c is initiated by the process active for c, which has to wait until the communication is completed by the passive process.

The 2-phase protocol for communication on a channel c between two processes B_1 (active) and B_2 (passive) consists of the following two phases:

(i) *Initiation*: B_1 sends a *request* to B_2 by performing an electrical transition ("zero-to-one" or "one-to-zero") on c_{req}.

(ii) *Completion*: B_2 sends an acknowledgement (or *grant*) to B_1 by performing an electrical transition on c_{gr}; the emitted value is assigned to the variable of the receiver using the wires x_c.

A 4-phase protocol consists of the following four phases:

(i) Initiation: electrical transition "zero-to-one" on c_{req}

(ii) Completion: electrical transition "zero-to-one" on c_{gr}

(iii) "one-to-zero" for c_{req}

(iv) "one-to-zero" for c_{gr}

2.5. *Informal Semantics of the Probe Operation in CHP*

The probe operation of CHP was introduced in [9] and was found to be useful in the design of asynchronous hardware. The notation "$c\#$" allows a passive process (either the emitter or the receiver) to check whether its active partner has already initiated a communication on c. The notation "$c\#V$", which can only be used

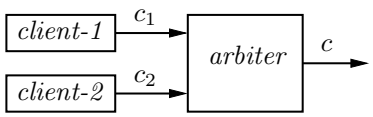

Fig. 1. Architecture of the asynchronous arbiter

by a receiver, checks whether the emitter has initiated the emission of the particular value V. Thus, the probe allows a process to query information about the current internal state (i.e. whether the communication has been initiated or not) of a concurrent process without completing the communication actually.

The probe operation allows to express external choice in CHP: this can be done by including in the guards appropriate probe operations to guarantee that the chosen branch can be executed. The arbiter examples given in Section 2.6 illustrate this use of the probe operation. The probe operation also allows to implement multiple reads, by executing "$c\,\#V$" several times, which avoids the hardware cost of an additional variable to store V.

Similar operators are also present in other hardware process calculi. For instance, HASTE [5] provides a "`probe`(c)" operation that can be used only in guards and is similar to "$c\,\#$" in CHP. BALSA provides a particular form of reception, called *input enclosure* [4], which allows the receiver to perform several commands before acknowledging the reception, whereas the emitter witnesses an atomic communication. Hence, in hardware process calculi, a rendezvous communication might be decomposed into different steps, corresponding to the handshake protocol used for implementing the rendezvous.

Notice that the abstract syntax allows to use the probe operator "$\#$" in any expression, thus also in emissions and assignments. Although we are not aware of any application using behaviours such as "$c_1\,!\,c_2\,\#$", we do not want to exclude them a priori, as they might be useful in some situations. For instance, the process "$c\,!\,c\,\#$" transmits a boolean value to the active receiver, indicating whether the passive emitter is ready for the emission before (transmission of the value *false*) or after (transmission of the value *true*) the initialisation of the communication by the receiver.

2.6. *Running Examples: Asynchronous Arbiters Without and With Priorities*

Throughout this article we consider the example of an asynchronous arbiter presented in [14], which we generalise in two ways: we use value-passing communications instead of pure synchronisations, and we model an arbiter open to its environment by keeping the shared resource outside of the arbiter itself.

Arbiters are commonplace in digital systems, whenever a restricted number of resources (such as memories, ports, buses, etc.) have to be allocated to different client processes. Figure 1 depicts the situation in which two clients compete for accessing a common resource. Each client transmits a request for the resource to the arbiter via an individual channel (c_1 or c_2). A third channel c allows the arbiter to emit the number of the selected client (1 or 2) to the environment, i.e. the resource. The arbiter chooses nondeterministically between the clients with pending requests. The corresponding CHP description is

$$\langle \{c, c_1, c_2\},\ \{\},\ \textit{client-1} \parallel \textit{client-2} \parallel \textit{arbiter} \rangle$$

where all three channels have an active emitter and a passive receiver, the set of variables is empty, and the three processes are described as follows:

$$\begin{pmatrix} \text{arbiter} \\ \text{without} \\ \text{priorities} \end{pmatrix} \qquad \begin{array}{ll} \textit{client-1:} & \mathbf{@}[\textit{true} \Rightarrow c_1\mathbf{!;\ loop}] \\ \textit{client-2:} & \mathbf{@}[\textit{true} \Rightarrow c_2\mathbf{!;\ loop}] \\ \textit{arbiter:} & \mathbf{@}[c_1\,\# \Rightarrow (c\mathbf{!}1, c_1\mathbf{?});\ \mathbf{loop} \quad c_2\,\# \Rightarrow (c\mathbf{!}2, c_2\mathbf{?});\ \mathbf{loop}] \end{array}$$

This example shows how the probe operation allows to model external choice. The arbiter uses probe operations to check whether a client has a pending request for the resource; since the arbiter selects a client only if a request for this client is pending, the clients can influence the choice made by the arbiter. Without the probe operations, the arbiter might select *client-1* even if *client-1* has no pending request; in this case, any request of *client-2* (even if pending) is granted only after waiting for and handling a request of *client-1*.

To illustrate the general case of probes used in expressions, we extend this arbiter in order to distinguish high and low priority requests. In this new system, the behaviour of both clients differs, since *client-1* can emit requests with different priorities. In this case, a boolean parameter is sent along channel c_1 to indicate whether the request has a high (parameter set to *true*) or normal (parameter set to *false*) priority. Consequently, *client-1* has precedence when submitting a priority request. The corresponding CHP description is

$$\langle \{c, c_1, c_2\},\ \{x\},\ \textit{client-1} \parallel \textit{client-2} \parallel \textit{arbiter} \rangle$$

where the boolean variable x — taking values in the set $\{\textit{true}, \textit{false}\}$ — is local to the arbiter, and where the three processes are described as follows:

$$\begin{pmatrix} \text{arbiter} \\ \text{with} \\ \text{priorities} \end{pmatrix} \qquad \begin{array}{ll} \textit{client-1:} & \mathbf{@}[\textit{true} \Rightarrow c_1\mathbf{!}\textit{true}\,\mathbf{;\ loop} \quad \textit{true} \Rightarrow c_1\mathbf{!}\textit{false}\,\mathbf{;\ loop}] \\ \textit{client-2:} & \mathbf{@}[\textit{true} \Rightarrow c_2\mathbf{!;\ loop}] \\ \textit{arbiter:} & \mathbf{@}[c_1\,\# \Rightarrow (c\mathbf{!}1, c_1\mathbf{?}x)\mathbf{;\ loop} \quad c_2\,\#\ \textit{and}\ (\textit{not}(c_1\,\#\textit{true})) \Rightarrow (c\mathbf{!}2, c_2\mathbf{?})\mathbf{;\ loop}] \end{array}$$

In this example, requests from *client-1* can always be served, while the arbiter accepts requests from *client-2* only if *client-1* is not trying to access the resource with a high priority.

3. A Structural Operational Semantics for CHP

In this section, we propose an SOS semantics for CHP with value-passing communications. This semantics allows probe operations in any expression, extending a previous version [17, Section 3], which allowed probe operations only in guards.

This semantics is defined without expanding communications according to a particular handshake protocol. This approach is general in the sense that it gives to any CHP description $\langle \mathcal{C},\ \mathcal{X},\ \hat{B}_0 \parallel \cdots \parallel \hat{B}_n \rangle$ a unique behavioural semantics by means of an LTS (*Labelled Transition System*). In this LTS, a state consists of two parts: data and behaviour. A transition corresponds either to an observable action (communication on a channel)[2] or an internal action, noted τ.[3] As stated by the semantics of CHP given in [14], internal actions are generated whenever a process assigns one of its local variables. Our definitions follow the usual interleaving semantics of process calculi, i.e. at every instant, at most one (observable or internal) action can take place.

We first present the notion of environment that gives the semantics of the data part of CHP. Then, we define the behavioural semantics of CHP in two steps, starting with the semantics of a single process $\hat{B}_i$ taken in isolation, followed by the semantics of a set of communicating processes "$\hat{B}_0 \parallel \cdots \parallel \hat{B}_n$". Finally, we relate our semantics to the approach of [14].

3.1. *Environments*

A key semantic difficulty in CHP is the treatment of the probe operation, since this operation exploits the fact that communication is not atomic at the lower level of implementation. Inspired by the actual hardware implementation of CHP, we associate to each channel c a shared variable noted x_c that is modified only by the process active for c and might be read by the process passive for c. For a channel c with an active emitter, the type of x_c is $\textit{type}(c)$. The active emitter assigns the emitted value to x_c when initiating the communication, and resets x_c (to the undefined value, noted "$\perp$") when completing the communication. A variable x_c is equal to $\perp$ iff all initiated communications on c have been completed. For a channel with an active receiver, the type of x_c is the singleton $\{\textit{ready}\}$, meaning that the active receiver has initiated the communication. Formally, we define the extended set of variables as $\mathcal{X}^* = \mathcal{X} \cup \{x_c \mid c \in \mathcal{C}\}$, and define $\mathcal{X}_i^*$ as the set of the local variables of $\hat{B}_i$ and all the variables x_c such that channel c is used by $\hat{B}_i$. Notice that the additional variables x_c ensure, for each channel c with an active emitter, that a value sent by the

[2] CHP has no hiding or restriction operator; thus, all inputs and outputs are observable.

[3] This is generally not the case in standard process calculi, in which an assignment to a local variable does not create an internal action, all actions being created by input/output communications.

active process on c can be read or probed as often as desired by the passive process before completion of the communication.

We define an environment E on $\mathcal{X}^*$ as a partial mapping from $\mathcal{X}^*$ to ground values (i.e. constants), and write environments as sets of associations "$x \mapsto v$" of a ground value v to a variable x. Environment updates are described by the operator $\oslash$, which is defined as follows:

$$(\forall x \in \mathcal{X}^*)\ \big(E_1 \oslash E_2\big)(x) = \begin{cases} E_1(x) & \text{if } E_2(x) = \bot \\ E_2(x) & \text{otherwise} \end{cases}$$

The environment obtained by resetting a variable x to $\bot$ in an environment E is defined by the function $reset(E, x)$:

$$(\forall x' \in \mathcal{X}^*)\ \big(reset(E, x)\big)(x') = \begin{cases} \bot & \text{if } x' = x \\ E(x') & \text{otherwise} \end{cases}$$

The semantics of a value expression V is defined by an evaluation function $eval(E, V)$, which behaves as usual, but takes the probe operation into account. $eval(E, V)$ can be undefined, since CHP (contrary to LOTOS) does not force all variables to be initialised. In the sequel, we suppose that all variables are properly initialised.

$$\begin{aligned} &eval(E, x) = E(x) \\ &eval\big(E, f(V_1, \ldots, V_n)\big) = f\big(eval(E, V_1), \ldots, eval(E, V_n)\big) \\ &eval(E, c\,\#) = true \iff E(x_c) \neq \bot \\ &eval(E, c\,\#V) = true \iff E(x_c) = eval(E, V) \end{aligned}$$

3.2. *Behavioural Semantics for a Single Process*

Our semantics associates to each process $\hat{B}_i$ an LTS $\big\langle \mathcal{S}_i, \mathcal{L}_i, \rightarrow_b, \langle E_i, \hat{B}_i \rangle \big\rangle$, where:

- The set of states $\mathcal{S}_i$ contains pairs $\langle E, B \rangle$ of a process B and an environment E on $\mathcal{X}_i^*$.
- The set of labels $\mathcal{L}_i$ contains emissions, receptions, τ actions (representing internal transitions, such as assignments to local variables), and a particular label $\surd$ representing successful termination.
- The transition relation "$\rightarrow_b$" is defined below by SOS rules derived from those used for BPA$_\varepsilon$ (*Basic Process Algebra with empty process* ε) in [7, chapter 3], extended with values and environments. As for BPA$_\varepsilon$, we write $\langle E, B \rangle \surd$ as a shorthand notation for $\langle E, B \rangle \xrightarrow{\surd}_b \langle E, \mathbf{nil} \rangle$.
- The initial state is $\langle E_i, \hat{B}_i \rangle$, where the initial environment E_i assigns the undefined value $\bot$ to all variables of $\mathcal{X}_i^*$.

This LTS is constructed using a function H that maps each channel to an element of the set $\{neutral, active, passive\}$; for each $\hat{B}_i$, the actual value of H is the function H_i defined in Section 2.1.

***Rules for* nil.** There are no rules for **nil** because **nil** is in a deadlock state and cannot evolve.

***Rules for* skip.** Similar to the *empty process* ε of BPA$_\varepsilon$, the process **skip** can always be executed and terminates successfully.

$$\overline{\langle E, \mathbf{skip} \rangle \surd}$$

The difference between a deadlock and a process that terminates successfully is subtle. In both cases, no more transition is possible, but a successful termination is always preceded by an observable transition with special label $\surd$. The same approach to distinguish deadlock states and terminating states using special transitions is also used in other process algebras, such as BPA$_\varepsilon$ or LOTOS.

Rules for Assignments. An assignment can always be executed and modifies the environment by updating the value of the variable to be assigned.

$$\frac{}{\langle E,\ x\,\mathtt{:=}V\rangle \xrightarrow{\tau}_b \langle E \oslash \{x \mapsto eval(E,V)\},\ \mathbf{skip}\rangle}$$

Rules for Emissions. A passive emission can always be executed.

$$\frac{H(c) = passive}{\langle E,\ c\mathbf{!}V\rangle \xrightarrow{c\mathbf{!}eval(E,V)}_b \langle E,\ \mathbf{skip}\rangle}$$

An active emission on a channel c involves two successive transitions: the first (internal) one assigns a value to the shared variable x_c, which is initially set to $\perp$, and the second one completes the communication by resetting x_c. This two step approach is general enough to represent 2-phase and 4-phase protocols.

$$\frac{H(c) = active \qquad eval(E,x_c) = \perp}{\langle E,\ c\mathbf{!}V\rangle \xrightarrow{\tau}_b \langle E \oslash \{x_c \mapsto eval(E,V)\},\ c\mathbf{!}V\rangle} \qquad \frac{H(c) = active \qquad eval(E,x_c) \neq \perp}{\langle E,\ c\mathbf{!}V\rangle \xrightarrow{c\mathbf{!}eval(E,V)}_b \langle reset(E,x_c),\ \mathbf{skip}\rangle}$$

Rules for Receptions. These rules are dual of those for emissions.

$$\frac{H(c) = passive}{\langle E,\ c\mathbf{?}x\rangle \xrightarrow{c\mathbf{?}eval(E,x_c)}_b \langle E \oslash \{x \mapsto eval(E,x_c)\},\ \mathbf{skip}\rangle}$$

$$\frac{H(c) = active \qquad eval(E,x_c) = \perp}{\langle E,\ c\mathbf{?}x\rangle \xrightarrow{\tau}_b \langle E \oslash \{x_c \mapsto ready\},\ c\mathbf{?}x\rangle} \qquad \frac{H(c) = active \qquad eval(E,x_c) \neq \perp \qquad V \in type(c)}{\langle E,\ c\mathbf{?}x\rangle \xrightarrow{c\mathbf{?}V}_b \langle reset(E,x_c) \oslash \{x \mapsto V\},\ \mathbf{skip}\rangle}$$

Contrary to the first rule, which uses the value of x_c as the received value, the third rule enumerates all possible ground values V that might be received on channel c.

Rules for Sequential Composition. The first rule applies as long as behaviour B_1 has not terminated. The second rule indicates that when B_1 can terminate successfully, the execution of B_2 starts.

$$\frac{L \neq \surd \qquad \langle E,\ B_1\rangle \xrightarrow{L}_b \langle E',\ B_1'\rangle}{\langle E,\ B_1\mathbf{;}B_2\rangle \xrightarrow{L}_b \langle E',\ B_1'\mathbf{;}B_2\rangle} \qquad \frac{\langle E, B_1\rangle\surd \qquad \langle E, B_2\rangle \xrightarrow{L}_b \langle E', B_2'\rangle}{\langle E,\ B_1\mathbf{;}B_2\rangle \xrightarrow{L}_b \langle E', B_2'\rangle}$$

Rules for Collateral Composition. The two first rules correspond to the independent evolution of behaviours B_1 and B_2, respectively. The third rule expresses that when both behaviours terminate, the collateral composition terminates successfully.

$$\frac{L \neq \surd \qquad \langle E,\ B_1\rangle \xrightarrow{L}_b \langle E',\ B_1'\rangle}{\langle E,\ B_1\mathbf{,}B_2\rangle \xrightarrow{L}_b \langle E',\ B_1'\mathbf{,}B_2\rangle} \qquad \frac{L \neq \surd \qquad \langle E,\ B_2\rangle \xrightarrow{L}_b \langle E',\ B_2'\rangle}{\langle E,\ B_1\mathbf{,}B_2\rangle \xrightarrow{L}_b \langle E',\ B_1\mathbf{,}B_2'\rangle} \qquad \frac{\langle E, B_1\rangle\surd \qquad \langle E, B_2\rangle\surd}{\langle E,\ B_1\mathbf{,}B_2\rangle\surd}$$

Rules for Guarded Commands. The rules for guarded commands express that a branch whose guard is *true* can be selected. As in [14], the selection of a branch is modelled by an internal transition, reflecting that choice is internal in CHP. If the chosen branch ends with "**break**", the guarded command terminates when the branch terminates. If it ends with "**loop**" the guarded command will be restarted once more after executing the branch.

$$\frac{(\exists i \in \{1,\ldots,n\}) \qquad eval(E,V_i) = true \qquad T_i = \mathbf{break}}{\langle E,\ \mathbf{@}[V_0 \Rightarrow B_0\mathbf{;}\ T_0\ \ldots\ V_n \Rightarrow B_n\mathbf{;}\ T_n]\rangle \xrightarrow{\tau}_b \langle E,\ B_i\rangle}$$

$$\frac{(\exists i \in \{1,\ldots,n\}) \qquad eval(E,V_i) = true \qquad T_i = \mathbf{loop}}{\langle E,\ \mathbf{@}[V_0 \Rightarrow B_0\mathbf{;}\ T_0\ \ldots\ V_n \Rightarrow B_n\mathbf{;}\ T_n]\rangle \xrightarrow{\tau}_b \langle E',\ B_i\mathbf{;}\ \mathbf{@}[V_0 \Rightarrow B_0\mathbf{;}\ T_0\ \ldots\ V_n \Rightarrow B_n\mathbf{;}\ T_n]\rangle}$$

The semantics of a CHP description $\langle \mathcal{C},\ \mathcal{X},\ \hat{B}_0 \parallel \cdots \parallel \hat{B}_n \rangle$ is defined by the parallel composition of the $(n+1)$ "local" LTSs $\langle \mathcal{S}_i, \mathcal{L}_i, \rightarrow_b, \langle E_i, \hat{B}_i \rangle \rangle$ produced from the individual processes $\hat{B}_i$ and the corresponding functions H_i according to the rules of Section 3.2. Formally, this composition is a "global" LTS $\langle \mathcal{S},\ \mathcal{L},\ \rightarrow, \langle E_0, \hat{B}_0, \ldots, \hat{B}_n \rangle \rangle$, where:

- The set of states $\mathcal{S}$ contains tuples $\langle E, B_0, \ldots, B_n \rangle$ consisting of $(n+1)$ processes $B_0, \ldots, B_n$ and a global environment E on $\mathcal{X}^* = \bigcup_{i=0}^{n} \mathcal{X}_i^*$. E is the union of the local environments E_i on $\mathcal{X}_i^*$ of the processes $\hat{B}_i$. The sets $\mathcal{X}_i^*$ are disjoint for the sets $\mathcal{X}_i$ (local variables of $\hat{B}_i$), but for each binary channel c connecting $\hat{B}_i$ and $\hat{B}_j$ $(i \neq j)$, the variable x_c occurs in $\mathcal{X}_i^*$ and $\mathcal{X}_j^*$; this is not a problem, since x_c is only modified by the process active for c.
- The set of labels $\mathcal{L}$ is the union of the sets of labels $\mathcal{L}_i$ minus those labels that correspond to receptions on binary channels. We represent synchronised communications (i.e. an emission and a reception) using the same symbol "**!**" as for emissions (following the convention used in the LTS generated from LOTOS using CADP [12]).
- The transition relation "$\rightarrow$" is defined by the three SOS rules Internal, Com, and Env below.
- The initial state is $\langle E_0, \hat{B}_0, \ldots, \hat{B}_n \rangle$, where E_0 is the empty initial environment, i.e. $(\forall x \in \mathcal{X}^*)\ \ E_0(x) = \bot$.

Let *internal*(L) be the predicate that is *true* iff L is either a τ or corresponds to a communication on a port (i.e. a unary channel open to the environment):

$$(\forall L \in \mathcal{L}) \quad \mathit{internal}(L) \iff \Big(L = \tau \ \vee\ \big((\exists c \in \mathcal{C})\ (\exists V \in \mathcal{V})\ (L = c!V \ \vee\ L = c?V) \ \wedge\ \mathit{port}(c) \big) \Big)$$

The first SOS rule describes the evolution of the i-th process B_i independently of the others. It models either an assignment to a variable, or the communication on a port c that is open to the environment and does not need to be synchronised with another process B_j $(i \neq j)$.

$$\frac{(\exists i \in \{0, \ldots, n\}) \qquad \langle E, B_i \rangle \xrightarrow{L}_b \langle E', B_i' \rangle \qquad \mathit{internal}(L)}{\langle E, B_0, \ldots, B_i, \ldots, B_n \rangle \xrightarrow{L} \langle E', B_0, \ldots, B_i', \ldots, B_n \rangle} \qquad (\mathsf{Internal})$$

The next rule describes the communication between an emitter process B_i and a receiver process B_j exchanging values over channel c.

$$\frac{(\exists i \in \{0, \ldots, n\})\ \ \langle E, B_i \rangle \xrightarrow{c!V}_b \langle E', B_i' \rangle \qquad (\exists j \in \{0, \ldots, n\})\ \ \langle E, B_j \rangle \xrightarrow{c?V}_b \langle E'', B_j' \rangle}{\langle E, B_0, \ldots, B_i, \ldots, B_j, \ldots, B_n \rangle \xrightarrow{c!V} \langle \mathit{reset}(E'', x_c), B_0, \ldots, B_i', \ldots, B_j', \ldots, B_n \rangle} \qquad (\mathsf{Com})$$

Note that i and j in rule (Com) are different and uniquely defined, since communications are binary (one emitter and one receiver for each channel). Note also that, if E' and E differ in rule (Com), the only possible modification (resetting x_c) is applied to E'' in the right hand side of the conclusion of rule (Com).

The rules presented so far are sufficient to define the semantics of (closed) systems without passive ports, i.e. unary channels for which no process $\hat{B}_i$ is active. The following rule completes the semantics by modelling the environment as an active process that communicates with each passive port c:

$$\frac{(\exists i \in \{0, \ldots, n\})\ H_i(c) = \mathit{passive} \qquad (\forall j \in \{0, \ldots, n\})\ \neg H_j(c) = \mathit{active} \qquad \mathit{eval}(E, x_c) = \bot \qquad V \in \mathit{type}(x_c)}{\langle E, B_0, \ldots, B_n \rangle \xrightarrow{\tau} \langle E \oslash \{x_c \mapsto V\}, B_0, \ldots, B_n \rangle} \qquad (\mathsf{Env})$$

This rule is similar to those defining the semantics of asynchronous processes communicating via shared memory, as for instance concurrent constraint programming [18] or concurrent declarative programming [19, Table 5.3, page 142].

Example 1 This example shows the necessity of rule (Env). Consider the following two processes B_1 and B_2, which are derived from the arbiter of Section 2.6:

$$B_1 = \mathbf{@}[c_1\,\# \Rightarrow (c\mathbf{!}1, c_1\mathbf{?})\mathbf{;\ loop}]$$
$$B_2 = \mathbf{@}[c_2\,\# \Rightarrow (c\mathbf{!}2, c_2\mathbf{?})\mathbf{;\ loop}]$$

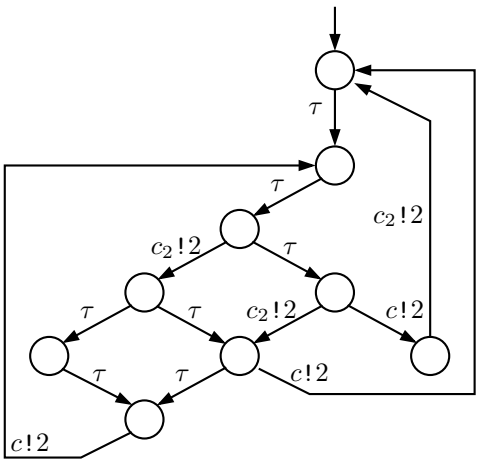

Fig. 2. LTS for "$B_2 \parallel$ client-2" of Example 1

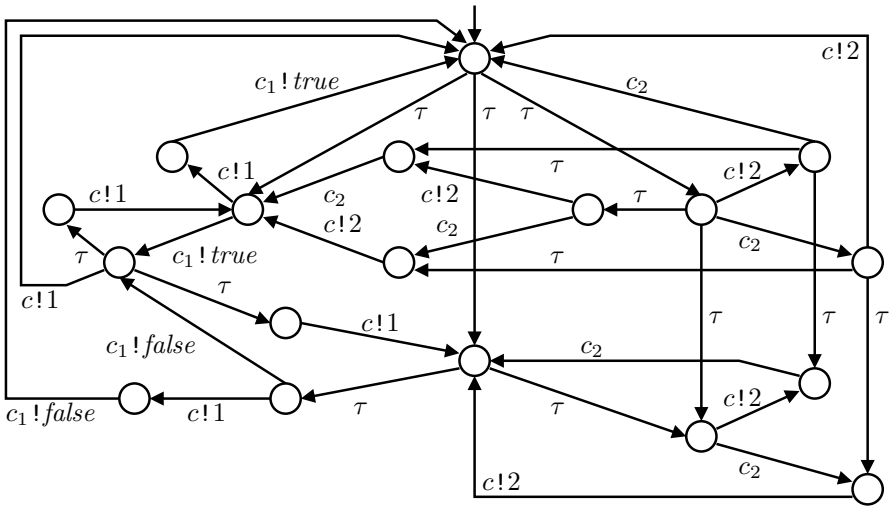

Fig. 3. LTS for the arbiter with priorities — minimised with respect to branching bisimulation

Here, c_1 and c_2 are passive ports open to the environment. Without rule (Env), both B_1 and B_2 would be equivalent to the deadlock process **nil**. Thus, one would expect that in the behaviour "$B_1 \parallel$ client-2", B_1 could be replaced by B_2 without modifying the semantics. However, "$B_1 \parallel$ client-2" is equivalent to **nil**, but "$B_2 \parallel$ client-2" is not — the corresponding LTS is shown in Figure 2. Rule (Env) solves this issue by giving a different semantics to B_1 and B_2.

Example 2 For the arbiter with priorities of Section 2.6, the LTS generated according to the operational semantics has 51 states and 112 transitions. After minimisation with respect to branching bisimulation [20], the resulting LTS (Figure 3) has 18 states, 34 transitions, and 6 different labels corresponding to internal τ actions and communications on channels c, c_1, and c_2.

3.4. *Comparison with the Existing Petri Net Translation*

So far, the only existing semantics for CHP proceeds by a translation of CHP into Petri nets [14]. This formalisation only handles a subset of CHP that, compared to full CHP presented in Section 2, is restricted in two ways: it allows only pure synchronisations (instead of value-passing communications), allows probe operations only in guards, and forbids ports open to the environment. By handling full CHP, our semantics

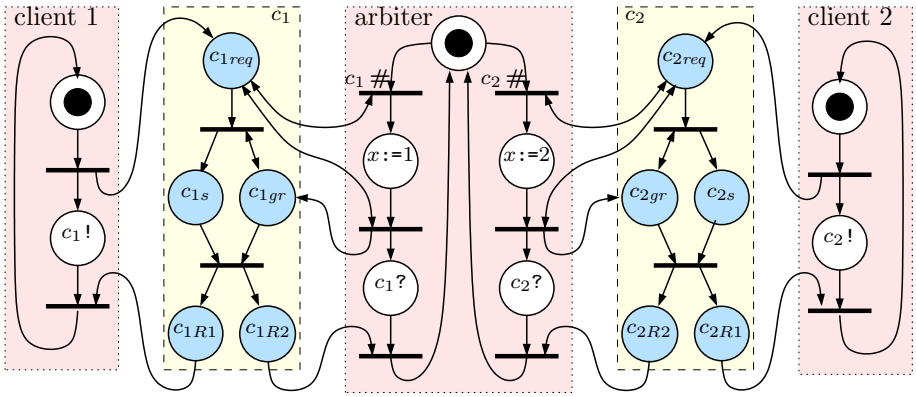

Fig. 4. Petri net for the arbiter in Example 3

allows to describe circuits with inputs and outputs properly. In this section, our goal is not to compare formally our semantics with the Petri nets encoding, but to explain the principles of such a comparison, and point out the differences between both semantics.

This section is split into three parts. First, we introduce the basics of the Petri net encoding. Second, we sketch how LTSs can be obtained from these Petri nets. Last, we stress differences between LTSs generated from both approaches.

Translation of CHP to Petri Nets. Similar to our SOS semantics, [14] defines the translation of a CHP description $\langle \mathcal{C},\ \mathcal{X},\ \hat{B}_1 \parallel \cdots \parallel \hat{B}_n \rangle$ into Petri nets in two successive steps:

- In a first step, the Petri nets corresponding to the processes $\hat{B}_i$ are constructed separately following the patterns sketched in [14]. Petri net places may be labelled with assignments, emissions, and receptions. Petri net transitions may be labelled with the guards of CHP guarded commands.
- In a second step, the separate Petri nets are merged into one global Petri net. To model synchronisation on channels, [14] gives two different translations, depending on the handshake protocol (2- or 4-phase) used for the implementation. In both cases, channels are modelled by additional places and transitions that encode the chosen handshake protocol. Notice that for each channel c the places labelled "c!" and "c?" are kept separate, i.e. there is no transition merging (contrary to [21] for instance).

Example 3 Consider an adaptation of the simple arbiter of Section 2.6 in order to meet the restrictions of [14] (no ports open to the environment). The corresponding CHP description is $\langle \{c_1, c_2\},\ \{x\},$ client-1 $\parallel$ client-2 $\parallel$ arbiter$\rangle$, where channels c_1 and c_2 have an active emitter and a passive receiver, where variable x is local to the arbiter, and where the three processes are defined as follows:

client-1: $@[\mathit{true} \Rightarrow c_1\,!\,;\ \mathbf{loop}]$

client-2: $@[\mathit{true} \Rightarrow c_2\,!\,;\ \mathbf{loop}]$

arbiter: $@[c_1\,\#1 \Rightarrow x{:=}1;\ c_1?;\ \mathbf{loop} \quad c_2\,\#2 \Rightarrow x{:=}2;\ c_2?;\ \mathbf{loop}]$

The corresponding Petri net for a 4-phase protocol is shown in Figure 4. Places are represented by circles, and transitions by thick lines. Whenever a place is both an input and an output place of some transition, we use a double-headed arrow (as for places labelled c_{1req}, c_{1gr}, c_{2req}, and c_{2gr}). The Petri nets corresponding to the three processes are framed in dotted boxes. The places modelling the channels c_1 and c_2 are framed in dashed boxes.

Deriving an LTS from a CHP Petri Net. As for ordinary Petri nets, a Petri net model of [14] has tokens in the places; the Petri nets generated from CHP descriptions are one-safe, i.e. each place contains at most one token. A transition can be fired when all its input places contain a token and its guard (if any) is true; after firing, the input places lose their token and the output places get a token. The Petri nets of [14] have also a different behaviour than ordinary Petri nets, to take into account the actions attached to places. For instance, if two places with action (e.g. "c_1!" and "c_2!" in Figure 4) have a token, then interleaved

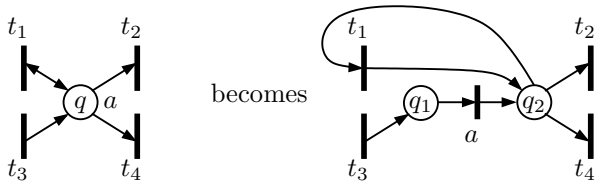

Fig. 5. Duplication of Petri net places

transitions have to be created for these actions. From [14] and following discussions with the first author of [14], we conjecture that these LTSs can be obtained by the following two steps:

– First, the Petri net needs to be transformed into a (more standard) Petri net model, in which actions are attached to transitions. As shown in Figure 5, each place q labelled with an action a (i.e. emission, reception, or assignment) is replaced by two places q_1 and q_2, linked with a new transition labelled with action a. Place q is replaced by q_1 (respectively q_2) in the sets of output (respectively input) places of all transitions arriving in (respectively, leaving) q. In the case a transition t has q both as an input and an output place (i.e. t is linked to q with a double-headed arrow such as t_1 in Figure 5), q is replaced by q_2 in the sets of input and output places of t.
– Then, the LTS is obtained by applying to the modified Petri net a marking graph construction algorithm extended with the evaluation of expressions and guards. The transitions of this LTS are labelled with the emissions and receptions attached to Petri net transitions. If a Petri net transition is not labelled with an emission or a reception, the corresponding LTS transition is labelled with τ.

Comparison of both Approaches. We can try to compare the LTS_{Sos} obtained by our SOS semantics and LTS_{PN} obtained after translation of CHP into Petri nets according to [14]. Given that [14] does not deal with value-passing communications and open systems, comparison is only possible for closed systems with pure synchronisations. Comparison of LTS_{Sos} and LTS_{PN} is not immediate, for several reasons.

First, the places and transitions added to the Petri nets for the communication channels introduce τ-transitions in LTS_{PN} that have no counterpart in LTS_{Sos}; thus, LTS_{Sos} and LTS_{PN} are not strongly equivalent [4]. Second, the sets of labels of LTS_{Sos} and LTS_{PN} are different: on the one hand, LTS_{PN} contains both "$c!$" and "$c?$" as labels, since the places labelled "$c!$" and "$c?$" are kept separate in the Petri net model of [14]; on the other hand, for closed systems LTS_{Sos} does not contain labels of the form "$c?$"; thus, comparing LTS_{Sos} and LTS_{PN} would require to rename into τ all labels of LTS_{PN} corresponding to communications by active processes, and to replace all remaining "?" by "!".

4. Principles of a Translation from CHP into LOTOS

In order to check the correctness of asynchronous circuit designs, our approach is to translate CHP into LOTOS so that tools for LOTOS (namely, the CADP verification toolbox [12]) can be applied.

4.1. *Overview of LOTOS*

LOTOS (Language Of Temporal Ordering Specification) is a specification language for distributed open systems, standardised by ISO [13]. A LOTOS specification is composed of two parts: a process part based on CCS [23] and CSP [24], and a data part based on the algebraic data type language ACTONE [25]. We present hereafter only the subset of LOTOS needed for expressing CHP; the data part being straightforward, we focus on the process part. For a complete description of LOTOS, we refer the reader to existing tutorials, such as [26].

[4] Strong equivalence (or strong bisimulation) [22] treats τ-transitions like visible transitions.

The following grammar uses the same conventions as for CHP, i.e. lower case identifiers stand for terminals and upper case identifiers stand for non terminals; x is a variable, s a sort, f a function, and g a gate for rendezvous communication.

$V ::= x \mid f(V_1, \ldots, V_n)$	*value expression*
$O ::= \mathbf{!}V$	*emission*
$\mid \mathbf{?}x : s$	*reception*
$B ::= \mathbf{stop}$	*deadlock*
$\mid \mathbf{exit}$	*successful termination*
$\mid \mathbf{let}\ x : s \texttt{=} V\ \mathbf{in}\ B_0$	*variable definition*
$\mid \tau\mathbf{;}\ B_0$	*internal action*
$\mid g\ O_1\ \ldots\ O_n\mathbf{;}\ B_0$	*rendezvous on gate g with offers O_1, ..., O_n*
$\mid B_1 \texttt{>>}\ \mathbf{accept}\ x_0 : s_0, \ldots, x_n : s_n\ \mathbf{in}\ B_2$	*sequential composition*
$\mid \texttt{[}V\texttt{]->}\ B_0$	*guarded behaviour*
$\mid B_1\ \texttt{[]}\ B_2$	*nondeterministic choice*
$\mid \mathbf{choice}\ x_0 : s_0, \ldots, x_n : s_n\ \texttt{[]}\ B$	*choice over values*
$\mid B_1\ \texttt{|[}g_1, \ldots, g_n\texttt{]|}\ B_2$	*parallel composition*
$\mid B_1\ \texttt{|||}\ B_2$	*parallel interleaving*
$\mid P\texttt{[}g_1, \ldots, g_m\texttt{]}(V_1, \ldots, V_n)$	*process call*
$F ::= \mathbf{noexit} \mid \mathbf{exit}(s_1, \ldots, s_n)$	*functionality*

The behaviour of a process B in LOTOS is described using rendezvous communications, sequential and parallel compositions, guarded behaviours, nondeterministic choices, choices over values, process calls, etc. Rendez vous communications in LOTOS allow to exchange several values — called *offers* — at the same time. The behaviour "**choice** $x_0 : s_0, \ldots, x_n : s_n$ **[]** B" assigns to each variable x_i a nondeterministically chosen value in the domain of sort s_i, and then behaves as B. The so-called *functionality* of a LOTOS process P is "**noexit**" if P never terminates, or "**exit**$(s_1, \ldots, s_n)$" if P may terminate and return a possibly empty set of values of sorts $s_1, \ldots, s_n$.

The definition of a LOTOS process P involves a list of formal gates $g_1, \ldots, g_m$, a list of formal parameters $x_1, \ldots, x_n$ of sort $s_1, \ldots, s_n$, a functionality F, a behaviour B, and possibly other process definitions $d_1, \ldots, d_p$:

process $P\texttt{[}g_1, \ldots, g_m\texttt{]}(x_1 : s_0, \ldots, x_n : s_n) : F$ **:=**
  B
where
  $d_1, \ldots, d_p$
endproc

4.2. *Overview of the CHP to LOTOS Translation*

We highlight first the main features of the translation of CHP into LOTOS:

- Each CHP type (booleans, natural numbers, bit vectors, etc.) is translated into a LOTOS sort (i.e. algebraic data types).
- Each CHP operation is translated into a LOTOS operation, which is implemented either using LOTOS algebraic equations or directly by C code (as CADP allows to import hand-written C code).
- Each CHP channel c is translated into a LOTOS gate with the same name c together with, in case c is probed, a LOTOS process to handle variable x_c.
- Each CHP variable x is translated into one or more LOTOS variables (i.e. *value identifiers* in the LOTOS standard terminology) with the same name and the same type as x. Several LOTOS variables might be required, since LOTOS processes follow the functional paradigm in which a variable can be assigned only once (whereas CHP variables can be assigned several times).

– Sequential composition "**;**" in CHP is symmetric, whereas LOTOS has two different operators for sequential composition: an asymmetric action prefix "**;**" and a symmetric sequential composition "**>>**". Variables assigned on the left hand side of a CHP "**;**" can be used on the right hand side, whereas variables assigned on the left hand side of a LOTOS "**>>**" must be explicitly listed (in an "**accept** ... **in**" clause) to be used on the right hand side. Furthermore, "**>>**" creates an internal τ-transition, contrary to the "**;**" operators of both CHP and LOTOS. There are two options when translating CHP to LOTOS. A simple approach is to generate LOTOS code containing only "**>>**", but this adds unnecessary τ-transitions in the corresponding LTS, thus contributing to state space explosion. A better approach is to generate "**;**" whenever possible and "**>>**" only when needed. In this article, we adopt the second approach, which produces better LOTOS code (in the sense that the corresponding LTS has less states and transitions) at the expense of a more involved translation.
– CHP has a neutral element (**skip**) for its sequential composition, whereas LOTOS lacks neutral elements for both "**;**" (which is asymmetric) and "**>>**" (which creates a τ-transition).
– CHP has a loop operator, whereas LOTOS does not; thus, CHP loops must be translated into recursive LOTOS processes.
– CHP expressions may contain probes, i.e. accesses to shared variables x_c, whereas LOTOS forbids shared variables. As a consequence, the translation of the probe operation requires the generation of additional transitions modelling accesses to communication channels.
– Evaluation of an expression V is an atomic operation in CHP, whereas LOTOS requires a sequence of as many transitions as there are probe operations in V. To preserve atomicity, we will introduce a semaphore-based locking mechanism discussed in Section 4.5.

The remainder of this section is organised as follows. First, we introduce data-flow analysis and other auxiliary definitions. Then, we present preliminary simplifications of CHP descriptions prior to the translation. The translation function *c2l* itself is defined in two steps: translation of a single CHP process and translation of several CHP processes composed in parallel. Then, we illustrate the translation on the arbiter example. Finally, we discuss the possible relations between our LOTOS translation with the SOS semantics of Section 3; to prepare this discussion, we will enhance the definition of the translation function *c2l* with remarks about preservation of branching equivalence during the translation.

4.3. *Auxiliary Definitions*

Data-flow Analysis. We introduce the following data-flow sets inspired from [27, Section 3]. Let $def(B)$ be the set of all variables modified (in assignments or receptions) in *at least one* branch of process B:

$$
\begin{array}{ll}
def(\mathbf{nil}) = \varnothing & def(x{:=}V) = \{x\} \\
def(\mathbf{skip}) = \varnothing & def(B_1;B_2) = def(B_1) \cup def(B_2) \\
def(c!V) = \varnothing & def(B_1,B_2) = def(B_1) \cup def(B_2) \\
def(c?x) = \{x\} & def(@[V_0 \Rightarrow B_0;\, T_0 \ \ldots\ V_n \Rightarrow B_n;\, T_n]) = \bigcup_{i=0}^{n} def(B_i)
\end{array}
$$

Let $def_n(B)$ be the set of all variables modified in *all* branches of process B:

$$
\begin{array}{ll}
def_n(\mathbf{nil}) = \varnothing & def_n(x{:=}V) = \{x\} \\
def_n(\mathbf{skip}) = \varnothing & def_n(B_1;B_2) = def_n(B_1) \cup def_n(B_2) \\
def_n(c!V) = \varnothing & def_n(B_1,B_2) = def_n(B_1) \cup def_n(B_2) \\
def_n(c?x) = \{x\} & def_n(@[V_0 \Rightarrow B_0;\, T_0 \ \ldots\ V_n \Rightarrow B_n;\, T_n]) = \bigcap_{i=0}^{n} def_n(B_i)
\end{array}
$$

We have $def(B) = def_n(B)$ for any behaviour B that does not contain a choice, i.e. a guarded command with more than one branch.

Let $use_v(V)$ be the set of all variables read in value expression V:

$$
\begin{array}{ll}
use_v(x) = \{x\} & use_v(c\#) = \varnothing \\
use_v\big(f(V_1, \ldots, V_n)\big) = \bigcup_{i=1}^{n} use_v(V_i) & use_v(c\#V) = use_v(V)
\end{array}
$$

Let $use(B)$ be the set of all variables used (in expressions of assignments, emissions, or guards) in a process B before any modification of these variables in B:

$$
\begin{array}{ll}
use(\mathbf{nil}) = \varnothing & use(x\,\mathtt{:=}V) = use_v(V) \\
use(\mathbf{skip}) = \varnothing & use(B_1\mathbf{;}B_2) = use(B_1) \cup \big(use(B_2) \setminus def_n(B_1)\big) \\
use(c\mathbf{!}V) = use_v(V) & use(B_1\mathbf{,}B_2) = use(B_1) \cup use(B_2) \\
use(c\mathbf{?}x) = \varnothing & use(\mathbf{@}[V_0 \Rightarrow B_0\mathbf{;}\ T_0\ \ldots\ V_n \Rightarrow B_n\mathbf{;}\ T_n]) = \bigcup_{i=0}^{n}\big(use_v(V_i) \cup use(B_i)\big)
\end{array}
$$

Functionalities. The LOTOS functionality corresponding to a CHP process B is given by the function $func(B, D, U)$, where D and U are two sets (in fact, alphabetically ordered lists) of variables, specifying the context in which B is executed, precisely, the set of variables defined before B is executed, and used after B is executed, respectively.

$$
\begin{array}{l}
func(\mathbf{nil}, D, U) = \mathbf{noexit} \\
func(\mathbf{skip}, D, U) = \mathbf{exit}(D \cap U) \\
func(c\mathbf{!}V, D, U) = \mathbf{exit}(D \cap U) \\
func(c\mathbf{?}x, D, U) = \mathbf{exit}\big((D \cup \{x\}) \cap U\big) \\
func(x\,\mathtt{:=}V, D, U) = \mathbf{exit}\big((D \cup \{x\}) \cap U\big) \\
func(B_1\mathbf{;}B_2, D, U) = \begin{cases} \mathbf{noexit} & \text{if } func(B_1, D, U) = \mathbf{noexit} \ \vee\ func(B_2, D, U) = \mathbf{noexit} \\ \mathbf{exit}\big((D \cup def(B_1) \cup def(B_2)) \cap U\big) & \text{otherwise} \end{cases} \\
func(B_1\mathbf{,}B_2, D, U) = \begin{cases} \mathbf{noexit} & \text{if } func(B_1, D, U) = \mathbf{noexit} \ \vee\ func(B_2, D, U) = \mathbf{noexit} \\ \mathbf{exit}\big((D \cup def(B_1) \cup def(B_2)) \cap U\big) & \text{otherwise} \end{cases} \\
func(\mathbf{@}[V_0 \Rightarrow B_0\mathbf{;}\ T_0\ \ldots\ V_n \Rightarrow B_n\mathbf{;}\ T_n], D, U) = \\
\qquad \begin{cases} \mathbf{noexit} & \text{if } (\forall i \in \{0, \ldots, n\})\ (func(B_i, D, U) = \mathbf{noexit} \ \vee\ T_i = \mathbf{loop}) \\ \mathbf{exit}\Big(\big(D \cup \bigcup_{i=0}^{n} def(B_i)\big) \cap U\Big) & \text{otherwise} \end{cases}
\end{array}
$$

Let $inf(B)$ be the predicate that is *true* iff $func(B, \varnothing, \varnothing) = \mathbf{noexit}$.

Channels. Let $chan_v(V)$ be the set of channels occurring in value expression V; by definition, these channels are probed in V:

$$
\begin{array}{l}
chan_v(x) = \varnothing \\
chan_v\big(f(V_1, \ldots, V_n)\big) = \bigcup_{i=1}^{n} chan_v(V_i) \\
chan_v(c\,\#) = \{c\} \\
chan_v(c\,\#V) = \{c\} \cup chan_v(V)
\end{array}
$$

Let $chan(B)$ be the set of channels occurring in behaviour B:

$$
\begin{array}{ll}
chan(\mathbf{nil}) = \varnothing & chan(x\,\mathtt{:=}V) = chan_v(V) \\
chan(\mathbf{skip}) = \varnothing & chan(B_1\mathbf{;}B_2) = chan(B_1) \cup chan(B_2) \\
chan(c\mathbf{!}V) = \{c\} \cup chan_v(V) & chan(B_1\mathbf{,}B_2) = chan(B_1) \cup chan(B_2) \\
chan(c\mathbf{?}x) = \{c\} & chan(\mathbf{@}[V_0 \Rightarrow B_0\mathbf{;}\ T_0\ \ldots\ V_n \Rightarrow B_n\mathbf{;}\ T_n]) = \\
 & \qquad \bigcup_{i=0}^{n}\big(chan_v(V_i) \cup chan(B_i)\big)
\end{array}
$$

Channel Profiles. In order to minimise the state space corresponding to the generated LOTOS code, the translation of a channel c and of the communications on c depends on the *profile* of c, i.e. whether and how c is probed by the passive process for c. The profile of a channel c can take one out of three enumerated values:

– *unprobed*: the channel c is never probed,

– *single*: any probe operation on c appears as a guard, e.g. "$c_1\,\#\Rightarrow\ldots$" (as in the simple arbiter without priorities), and
– *general*: a probe on c is used in an expression, e.g. "$c_2\,\#\ \mathit{and}\ (\mathit{not}(c_1\,\#\mathit{true}))\Rightarrow\ldots$" (as in the arbiter with priorities, where both channels c_1 and c_2 have profile "*general*").

We define an order *unprobed* < *single* < *general* (the smaller the value, the more "efficient" the generated LOTOS will be). We note "$\max(x,y)$" the maximal element of the two profiles x and y;

Channel profiles for a CHP description are computed statically as defined below. Let a *profile environment* be a partial function mapping channels to their profiles; as for environments (see Section 3.1), we write profile environments as sets of associations $c\mapsto\{\mathit{unprobed},\mathit{single},\mathit{general}\}$, where c is a channel. We define

$$(\forall c\in\mathcal{C})\ (E_1\uplus E_2)(c)=\begin{cases}E_1(c) & \text{if } E_2(c) \text{ is undefined}\\ E_2(c) & \text{if } E_1(c) \text{ is undefined}\\ \max\big(E_1(c),E_2(c)\big) & \text{otherwise}\end{cases}$$

Let $\mathit{profile}_v(V)$ be the partial function associating the "*general*" profile to each channel probed in the value expression V:

$$\mathit{profile}_v(V)=\big\{c\mapsto \mathit{general}\mid c\in \mathit{chan}_v(V)\big\}$$

The boolean expressions occurring as guards in the "@[...]" operator need to handled differently from "ordinary" expressions. Indeed, if the outermost operation of a guard V is a probe on channel c, and V does not contain any other occurrence of a probe operation, the profile of c is "*single*". Let $\mathit{profile}_g(V)$ be the partial function computing profile environments for those channels occurring in guard V:

$$\begin{aligned}&\mathit{profile}_g(x)=\varnothing\\ &\mathit{profile}_g\big(f(V_1,\ldots,V_n)\big)=\mathit{profile}_v\big(f(V_1,\ldots,V_n)\big)\\ &\mathit{profile}_g(c\,\#)=\{c\mapsto \mathit{single}\}\\ &\mathit{profile}_g(c\,\#V)=\begin{cases}\{c\mapsto \mathit{single}\} & \text{if } \mathit{profile}_v(V)=\varnothing\\ \{c'\mapsto \mathit{general}\mid c'=c\ \vee\ c'\in \mathit{chan}_v(V)\} & \text{otherwise}\end{cases}\end{aligned}$$

Let $\mathit{profile}_b(B,H)$ be the partial function computing profiles for channels occurring in the behaviour B, where parameter H is a partial function mapping channels to the set $\{\mathit{neutral},\mathit{active},\mathit{passive}\}$ (see Section 2.1). Function $\mathit{profile}_b$ is defined as follows:

$$\begin{aligned}&\mathit{profile}_b(\mathbf{nil},H)=\varnothing\\ &\mathit{profile}_b(\mathbf{skip},H)=\varnothing\\ &\mathit{profile}_b(c\mathbf{!}V,H)=\begin{cases}\{c\mapsto \mathit{unprobed}\}\uplus \mathit{profile}_v(V) & \text{if } H(c)=\mathit{passive}\\ \mathit{profile}_v(V) & \text{otherwise}\end{cases}\\ &\mathit{profile}_b(c\mathbf{?}x,H)=\begin{cases}\{c\mapsto \mathit{unprobed}\} & \text{if } H(c)=\mathit{passive}\\ \varnothing & \text{otherwise}\end{cases}\\ &\mathit{profile}_b(x\mathbf{:=}V,H)=\mathit{profile}_v(V)\\ &\mathit{profile}_b(B_1\mathbf{;}B_2,H)=\mathit{profile}_b(B_1,H)\uplus \mathit{profile}_b(B_2,H)\\ &\mathit{profile}_b(B_1\mathbf{,}B_2,H)=\mathit{profile}_b(B_1,H)\uplus \mathit{profile}_b(B_2,H)\\ &\mathit{profile}_b(\mathbf{@}[V_0\Rightarrow B_0\mathbf{;}\ T_0\ \ldots\ V_n\Rightarrow B_n\mathbf{;}\ T_n],H)=\biguplus_{i=0}^{n}\big(\mathit{profile}_g(V_i)\uplus \mathit{profile}_b(B_i,H)\big)\end{aligned}$$

Finally, for a CHP description $\langle\mathcal{C},\ \mathcal{X},\ \hat{B}_0\parallel\cdots\parallel\hat{B}_n\rangle$, the partial function *profile* mapping a channel to its profile is defined by:

$$\mathit{profile}=\big(\biguplus_{i=0}^{n}\mathit{profile}_b(\hat{B}_i,H_i)\big)\uplus\big\{c\mapsto \mathit{unprobed}\mid c\in\mathcal{C}\ \wedge\ \mathit{port}(c)\big\}$$

Notice that *profile* uses the profile "*unprobed*" as default value for each port; this simplifies the LOTOS code produced by the translation and reduces the size of the corresponding state space.

4.4. *Preliminary Simplifications*

Prior to translation, we simplify all CHP processes by applying the following transformations in sequence:

- All occurrences of **skip** are removed wherever possible, based on the facts that (1) **skip** is neutral element for sequential and collateral composition, (2) any branch "$V \Rightarrow$ **skip; loop**" of a guarded command can be removed, and (3) any $\hat{B}_i$ equal to **skip** can be removed from "$\hat{B}_0 \parallel \cdots \parallel \hat{B}_n$". After these transformations, **skip** may occur only in branches "$V \Rightarrow$ **skip; break**" of guarded commands.
- The abstract syntax tree of all processes $\hat{B}_i$ is reorganised so that the sequential composition operator "**;**" is right bracketed (based on the associativity of CHP sequential composition). After transformation, each sequence "B_1**;** B_2**;** B_3" is bracketed as "B_1**;** (B_2**;** B_3)", even if it was bracketed as "(B_1**;** B_2)**;** B_3" before.
- If the rightmost process B_n of a maximal sequence "B_0**;**...**;**B_n" is of the form "x**:=**V", "c**!**V", or "c**?**x", a final **skip** is added, leading to the sequence "B_0**;**...**;**B_n**; skip**". This transformation simplifies the syntax-driven induction presented in Section 4.5.
- For each process of the form "B_1**;**B_2" such that $inf(B_1)$, B_2 can be removed as it will never be executed. Similarly, in each process of the form "B_1**,**B_2" such that $inf(B_1) = \neg inf(B_2)$, we replace the process B_i ($i \in \{1, 2\}$) such that $inf(B_i)$ is false, by "B_i**; nil**". Also, if $\neg inf(\hat{B}_i)$, then $\hat{B}_i$ is replaced by "$\hat{B}_i$**; nil**". These transformations are needed to obey the static check of functionalities in LOTOS.

After these transformations, all assignments and all communications (emissions and receptions) occur in prefix-form, i.e. they occur only in processes of the form "x**:=**V**;** B", "c**!**V**;** B", or "c**?**x**;** B". Precisely, CHP processes obtained after these transformations have the following syntax:

$$B ::= \textbf{nil} \mid \textbf{skip} \mid A\textbf{;}B \mid B_1\textbf{,}B_2 \mid \textbf{@}[V_0 \Rightarrow B_0\textbf{;}\ T_0\ \ldots\ V_n \Rightarrow B_n\textbf{;}\ T_n]$$
$$A ::= x\textbf{:=}V \mid c\,\textbf{!}V \mid c\,\textbf{?}x \mid B_1\textbf{,}B_2 \mid \textbf{@}[V_0 \Rightarrow B_0\textbf{;}\ T_0\ \ldots\ V_n \Rightarrow B_n\textbf{;}\ T_n]$$

where T and V are defined as in Section 2. In the sequel, we use this new grammar for the definition of the translation functions.

4.5. *Translation of a Single Process*

A CHP process $\hat{B}_i$ is translated into a LOTOS process whose body is obtained using the recursive function $c2l_b(B, H, D, U, \Delta)$, where B is a CHP process to translate, H is a mapping from channels to $\{neutral, active, passive\}$, and D, U, and Δ are sets (in fact, alphabetically ordered lists) of variables necessary to compute the variables transmitted over LOTOS sequential composition operators "**>>**". Intuitively, D is the set of variables that have a defined value before the execution of B, U is the set of variables that may be used after the execution of B, and Δ is an auxiliary set of defined variables suitable for translating collateral compositions. We have the invariant property that $D \subseteq \Delta$.

Translation of Channels. We apply code specialisation to optimise the translation of channels, i.e. the translation of a channel depends on its profile — "*unprobed*", "*single*", or "*general*".

- A CHP communication on a channel c of profile "*unprobed*" is translated directly into a single LOTOS rendezvous on gate c.
- The translation of a channel c of profile "*single*" also requires only one LOTOS gate c. Probe operations are translated into communications on gate c; they are distinguished by an additional offer "**!***Probe*", where "*Probe*" is a special constant belonging to an enumerated type with a single value. This translation is based on the value-matching communication feature of LOTOS, which allows the emitter and the receiver both to synchronise on the same offer "**!***Probe*".
- A channel c of profile "*general*" is translated into a LOTOS process *channel_c* which manages the shared variable x_c introduced in the CHP operational semantics. *Five* LOTOS gates are introduced, namely three gates (c, *c_init*, *c_probe*) dedicated to the channel c, and two gates (*probe_enable* and *probe_disable*) common to all channels of profile "*general*". A CHP communication on c is translated into two LOTOS communications: a binary communication on gate *c_init* between the active process and *channel_c*, followed

by a three-party rendezvous on gate c between the emitter, the receiver, and *channel_c*. A probe is translated into a binary communication on *c_probe* between the passive process and *channel_c*.

Because channels are binary, for a given shared variable x_c the potential conflicts on x_c to be considered are only those between one single writer (the process active for c) and one single reader (the process passive for c). However, the situation is not so simple, because there are several shared variables and because each process may wish to read several variables simultaneously in an atomic step (during which one should prevent these variables from being modified).

Different approaches are possible to ensure atomicity of such a transition sequence, as for instance a locking mechanism, or a single process for all channel variables x_c (so as to allow several channels to be probed in a single transition). We chose the locking approach for its modularity (when considering a subset of processes, only those channels linking the processes are required). Without loss of generality, we present in this article a single lock for all channels; using several locks (where all channels probed in a same expression share the same lock) might even lead to larger but equivalent state spaces (due to the interleaving of the confluent [28] transitions).

The locking mechanism is implemented by the two gates "*probe_enable*" (to acquire the lock) and "*probe_disable*" (to release the lock). A shared variable can be modified (for initialisation — communication on *c_init* — and reset after the communication on c) if no process owns the lock (i.e. if no process is between a synchronisation on *probe_enable* and the next synchronisation on *probe_disable*). Conversely, a shared variable can only be read if some process owns the lock. Stated otherwise, a synchronisation on *probe_enable* places all shared variables in a read access mode, and a synchronisation on *probe_disable* places all shared variables in a write access mode (this is the default initial mode).

The precise definition of *channel_c* can take two forms:

- If channel c links an active emitter and a passive receiver, the process *channel_c* is defined as follows:

```
process channel_c[probe_enable, probe_disable, c, c_init, c_probe] : noexit :=
  c_init ?x : s;  channel_c_comm[probe_enable, probe_disable, c, c_init, c_probe](x)
  []
  probe_enable; (
    c_probe !false !⊥;  probe_disable;  channel_c[probe_enable, probe_disable, c, c_init, c_probe]
    []
    probe_disable;  channel_c[probe_enable, probe_disable, c, c_init, c_probe]  )
where
  process channel_c_comm[probe_enable, probe_disable, c, c_init, c_probe](x : s) : noexit :=
    c !x;  channel_c[probe_enable, probe_disable, c, c_init, c_probe]
    []
    probe_enable; (
      c_probe !true !x; probe_disable; channel_c_comm[probe_enable, probe_disable, c, c_init, c_probe](x)
      []
      probe_disable;  channel_c_comm[probe_enable, probe_disable, c, c_init, c_probe](x)  )
  endproc
endproc
```

- If channel c links a passive emitter to an active receiver, the translation is slightly simpler, as one can remove the parameter x of *channel_c_comm* since the value exchanged on c does not need to be stored:

```
process channel_c[probe_enable, probe_disable, c, c_init, c_probe] : noexit :=
  c_init; channel_c_comm[probe_enable, probe_disable, c, c_init, c_probe]
  []
  probe_enable; (
    c_probe !false; probe_disable; channel_c[probe_enable, probe_disable, c, c_init, c_probe]
    []
    probe_disable; channel_c[probe_enable, probe_disable, c, c_init, c_probe] )
where
  process channel_c_comm[probe_enable, probe_disable, c, c_init, c_probe]: noexit :=
    c ?x : s; channel_c[probe_enable, probe_disable, c, c_init, c_probe]
    []
    probe_enable; (
      c_probe !true; probe_disable; channel_c_comm[probe_enable, probe_disable, c, c_init, c_probe]
      []
      probe_disable; channel_c_comm[probe_enable, probe_disable, c, c_init, c_probe] )
  endproc
endproc
```

Remark. Compared to the CHP operational semantics, the translation into LOTOS introduces additional transitions (those containing "**!***Probe*" offers, synchronisations on the gates *probe_enable*, *probe_disable*, and communications on gates of one of the forms *c_init* and *c_probe*). Anticipating the discussion about the correctness of the translation (see Section 4.10), these transitions should be hidden (i.e. renamed into τ) when comparing the translation into LOTOS with the CHP operational semantics. □

Translation of Value Expressions. The translation of a value expression V is straightforward except for the probe operations that may occur in V. For each channel c probed by V, the shared variable x_c must be read. Thus, each evaluation of a value expression V (in an emission, an assignment, or a guard) must be preceded by a sequence of communications with all channels probed by V. We define

$$\begin{aligned}&\mathit{query_probe}(V_0, \ldots, V_n) = \\ &\qquad \mathit{probe_enable};\ c_1\text{-}\mathit{probe}\ \mathbf{?}\overline{x_{c_1}} : \mathit{bool}\ \mathbf{?}x_{c_1} : s_1;\ \ldots;\ c_m\text{-}\mathit{probe}\ \mathbf{?}\overline{x_{c_m}} : \mathit{bool}\ \mathbf{?}x_{c_m} : s_m;\ \mathit{probe_disable}\end{aligned}$$

where $\{c_1, \ldots, c_m\} = \{c' \mid \mathit{profile}(c') = \mathit{general} \wedge c' \in \bigcup_{i=0}^{n} \mathit{chan}_v(V_i)\}$ is the set of channels of profile "*general*" probed by value expressions V_0, ..., V_n. For each probed channel c, this translation introduces two auxiliary variables, a boolean $\overline{x_c}$ (which is true iff channel c is ready to communicate) and x_c (which contains the value carried by c when $\overline{x_c}$ is true).

Remark. There are only two places at which the translation generates synchronisations on *probe_enable* and *probe_disable*; the other one will be seen below during the translation of guarded commands. The above definition of *query_probe* guarantees that a synchronisation on *probe_enable* is eventually followed by a synchronisation on *probe_disable*, since all rendezvous between *probe_enable* and *probe_disable* are communications with processes *channel_c* that always accept to synchronise on gates *c_probe* (after a synchronisation on *probe_enable*). Thus, because there is only a single lock which is eventually released, the locking mechanism does not introduce deadlocks. □

We define $c2l_v(x)$ as the function translating a CHP value expression into a LOTOS value expression:

$$\begin{aligned}&c2l_v(x) = x \\ &c2l_v\big(f(V_1, \ldots, V_n)\big) = f\big(c2l_v(V_1), \ldots, c2l_v(V_n)\big) \\ &c2l_v(c\,\#) = \overline{x_c} \\ &c2l_v(c\,\#V) = \overline{x_c} \wedge \big(x_c = c2l_v(V)\big)\end{aligned}$$

***Translation of* nil.** **nil** is translated into **stop**.

Remark. The CHP operational semantics and the translation into LOTOS coincide in associating **nil** to a deadlock. □

***Translation of* skip.** Due to the preliminary transformations of Section 4.4, **skip** may only occur in a guarded command as a branch "$V \Rightarrow$ **skip; break**" (this case is handled below as part of guarded commands) or at the end of a sequence. In the latter case:

$$c2l_b(\mathbf{skip}, H, D, U, \Delta) = \mathbf{exit}(\xi_1, \ldots, \xi_n)$$

where $\{x_1, \ldots, x_n\} = \Delta \cap U$, and $(\forall i \in \{1, \ldots, n\})\ \ \xi_i = x_i$ if $x_i \in D$ or $\xi_i =$ "**any** $type(x_i)$" otherwise.

Remark. The translation into LOTOS generates a transition (labelled with "$\mathbf{exit}(\xi_1, \ldots, \xi_n)$") that corresponds to the $\surd$-transition of the CHP operational semantics. □

***Translation of "*$c\,!V; B$*" and "*$c\,?x; B$*" when** $profile(c) = unprobed$**.** Communication on a channel of profile "*unprobed*" is translated directly into a LOTOS rendezvous communication:

$$c2l_b(\text{“}c\,!V; B\text{”}, H, D, U, \Delta) = query_probe(V);\ c\,!c2l_v(V);\ c2l_b(B, H, D, U, \Delta)$$
$$c2l_b(\text{“}c\,?x; B\text{”}, H, D, U, \Delta) = c\,?x : s;\ c2l_b(B, H, D \cup \{x\}, U, \Delta \cup \{x\})$$

Remark. As regards the relationship between this translation and the operational semantics given in Section 3 for CHP emissions and receptions, four (i.e. 2×2) cases should be considered depending on:
- Emission v. reception: the translation into LOTOS generates, for a reception, single rendezvous on c for receptions; for an emission it generates a rendezvous on c that occurs after a sequence of transitions (communications on *c_probe* and synchronisations on *probe_enable* and *probe_disable*, see the definition of *query_probe* above), required to evaluate the value V.
- $H(c) = passive$ v. $H(c) = active$: in the former case, the first rule of the CHP operational semantics for emissions (respectively receptions) produces a single transition (communication on c); in the latter case, two transitions are produced: a first τ-transition (second inference rule on the left, corresponding to an assignment to the variable x_c) followed by a communication on c (third rule on the right).

Thus, the four cases are the following:
- *Passive reception*: both the CHP operational semantics and the translation into LOTOS generate one single rendezvous on c.
- *Active emission*: the τ-transition the CHP operational semantics matches the sequence of transitions (generated by *query_probe*); if these transitions are hidden, branching equivalence is preserved.
- *Passive emission*: the translation into LOTOS introduces a sequence of transitions that, if hidden, preserve branching equivalence (this is a particular case of the more general notion of τ-confluence [28]).
- *Active reception*: the τ-translation generated by the second rule (on the left) of the CHP operation semantics for receptions is confluent; therefore the LOTOS code can be optimised by not generating a τ-transition while still preserving branching equivalence.

□

***Translation of "*$c\,!V; B$*" and "*$c\,?x; B$*" when** $profile(c) = general$**.** The translation of an emission on a channel c depends whether $H(c)$ is "*active*" or "*passive*":
- A passive emission — when $H(c) = passive$ — is translated as follows:

$$c2l_b(\text{“}c\,!V; B\text{”}, H, D, U, \Delta) = query_probe(V);\ c\,!c2l_v(V);\ c2l_b(B, H, D \cup \{x\}, U, \Delta \cup \{x\})$$

– The translation of an active emission — when $H(c) = \mathit{active}$ — requires in addition to initialise the variable x_c before communication. Thus, the translation of an active emission is as follows:

$$c2l_b(\text{“}c\,\mathbf{!}V\mathbf{;}\,B\text{”}, H, D, U, \Delta) = \\ \mathit{query_probe}(V)\mathbf{;}\ \mathit{c_init}\ \mathbf{!}c2l_v(V)\mathbf{;}\ c\ \mathbf{!}c2l_v(V)\mathbf{;}\ c2l_b(B, H, D \cup \{x\}, U, \Delta \cup \{x\})$$

The translation of a reception on a channel c of sort $s = \mathit{type}(c)$ depends whether $H(c)$ is “*active*” or “*passive*”:

– A passive reception — when $H(c) = \mathit{passive}$ — is translated as follows:

$$c2l_b(\text{“}c\,\mathbf{?}x\mathbf{;}\,B\text{”}, H, D, U, \Delta) = c\ \mathbf{?}x : s\mathbf{;}\ c2l_b(B, H, D \cup \{x\}, U, \Delta \cup \{x\})$$

– An active reception — when $H(c) = \mathit{active}$ — additionally requires to initiate the communication on the channel c:

$$c2l_b(\text{“}c\,\mathbf{?}x\mathbf{;}\,B\text{”}, H, D, U, \Delta) = \mathit{c_init}\mathbf{;}\ c\ \mathbf{?}x : s\mathbf{;}\ c2l_b(B, H, D \cup \{x\}, U, \Delta \cup \{x\})$$

Remark. As regards the relationship between this translation and the operational semantics given in Section 3 for CHP emissions and receptions, three out of four cases are similar to the case where $\mathit{profile}(c) = \mathit{unprobed}$ (see above). In the case of an active reception, the second rule (on the left) of the CHP operational semantics generates a τ transition, which is matched by the rendezvous on *c_init*; if *c_init* is hidden, branching equivalence is preserved. □

Translation of “$c\,\mathbf{!}V\mathbf{;}\,B$” and “$c\,\mathbf{?}x\mathbf{;}\,B$” when $\mathit{profile}(c) = \mathit{single}$. The translation depends on the value of $H(c)$:

– If $H(c) = \mathit{passive}$, the translation is straightforward (identical to the “*unprobed*” case):

$$c2l_b(\text{“}c\,\mathbf{!}V\mathbf{;}\,B\text{”}, H, D, U, \Delta) = \mathit{query_probe}(V)\mathbf{;}\ c\ \mathbf{!}c2l_v(V)\mathbf{;}\ c2l_b(B, H, D, U, \Delta)$$
$$c2l_b(\text{“}c\,\mathbf{?}x\mathbf{;}\,B\text{”}, H, D, U, \Delta) = c\ \mathbf{?}x : s\mathbf{;}\ c2l_b(B, H, D \cup \{x\}, U, \Delta \cup \{x\})$$

where $s = \mathit{type}(c)$.

Remark. Branching equivalence is preserved for the same reasons as for the “*unprobed*” case. □

– If $H(c) = \mathit{active}$, the translation is more involved because the active process $\hat{B}_i$ must allow its passive partner to probe channel c an arbitrary number of times. Thus, the active process must allow any actual communication on c to be preceded by a (possible empty) sequence of communications with an additional offer “**!***Probe*” (as mentioned during the translation of channels). This protocol is encapsulated in an auxiliary LOTOS process *channel_c*, the definition of which depends whether c is used for emission or reception.

- An active emission “$c\,\mathbf{!}V\mathbf{;}\,B$” is translated as follows:

$$c2l_b(\text{“}c\,\mathbf{!}V\mathbf{;}\,B\text{”}, H, D, U, \Delta) = \mathit{query_probe}(V)\mathbf{;}\ \mathit{channel_c}\texttt{[}c\texttt{]}\big(c2l_v(V), x_1, \ldots, x_n\big) \\ \texttt{>>}\ \mathbf{accept}\ x_1 : s'_1, \ldots, x_n : s'_n\ \mathbf{in}\ c2l_b(B, H, D', U, \Delta')$$

where $\{c_1, \ldots, c_m\} = \mathit{chan}_v(V)$, $(\forall i \in \{1, \ldots, m\})\ \ s_i = \mathit{type}(x_{c_i})$, $U' = \mathit{use}(B) \cup U$, $D' = D \cap U'$, $\Delta' = \Delta \cap U'$, $\{x_1, \ldots, x_n\} = D'$, and $(\forall i \in \{1, \ldots, n\})\ \ s'_i = \mathit{type}(x_i)$. Process *channel_c* is defined by:

$$\mathbf{process}\ \mathit{channel_c}\texttt{[}c\texttt{]}(x : s, x_1 : s_1, \ldots, x_n : s_n) :\ \mathbf{exit}(s_1, \ldots, s_n)\ \texttt{:=} \\ c\ \mathbf{!}x\mathbf{;}\ \mathbf{exit}(x_1, \ldots, x_n)\ \ \texttt{[]}\ \ c\ \mathbf{!}\mathit{Probe}\ \mathbf{!}x\mathbf{;}\ \mathit{channel_c}\texttt{[}c\texttt{]}(x, x_1, \ldots, x_n) \\ \mathbf{endproc}$$

where $s = \mathit{type}(c)$.

Remark. As for all active emissions, the translation into LOTOS introduces a sequence of communications that, after hiding, matches the τ-transition generated by the CHP operational semantics, thus preserving branching equivalence. □

- An active reception "$c\,\mathbf{?}x\mathbf{;}\,B$" translates as follows:

$$c2l_b(\text{“}c\,\mathbf{?}x\mathbf{;}\,B\text{”}, H, D, U, \Delta) = \mathit{channel_c}[c](x''_1, \ldots, x''_n) \\ \gg \mathbf{accept}\ x'_1 : s'_1, \ldots, x'_m : s'_m\ \mathbf{in}\ c2l_b(B, H, D', U, \Delta')$$

where $U' = \mathit{use}(B) \cup U$, $D' = (D \cup \{x\}) \cap U'$, $\Delta' = (\Delta \cup \{x\}) \cap U'$, $D'' = D' \setminus \{x\}$, $\{x'_1, \ldots, x'_m\} = D'$, $\{x''_1, \ldots, x''_n\} = D''$, $(\forall i \in \{1, \ldots, m\})\ \ s'_i = \mathit{type}(x'_i)$, and $(\forall i \in \{1, \ldots, n\})\ \ s''_i = \mathit{type}(x''_i)$. Process *channel_c* is defined by:

$$\begin{array}{l} \mathbf{process}\ \mathit{channel_c}[c](x''_1 : s''_1, \ldots, x''_n : s''_n) :\ \mathbf{exit}(s'_1, \ldots, s'_m)\ \ \mathbf{:=} \\ \quad c\,\mathbf{?}x : s\mathbf{;}\ \mathbf{exit}(x'_1, \ldots, x'_m)\ \ \mathbf{[]}\ \ c\,\mathbf{!}\mathit{Probe}\mathbf{;}\ \mathit{channel_c}[c](x''_1, \ldots, x''_n) \\ \mathbf{endproc} \end{array}$$

where $s = \mathit{type}(c)$.

Remark. As for the "*unprobed*" case, the translation into LOTOS removes the confluent τ-transition introduced by the CHP operational semantics, thus preserving branching equivalence. □

Notice that, contrary to the "*general*" case above, in the "*single*" case, the process *channel_c* never needs (neither for an active emission nor for an active reception) to evaluate a probe, and, thus, never needs to synchronise on *probe_enable* and *probe_disable*.

Remark. Preservation of branching equivalence albeit the communications containing offers of the form "**!***Probe*" introduced by the translation into LOTOS (and that do not exist in the CHP operational semantics) is discussed during the translation of guards and the translation of guarded commands. □

Translation of "x:=V;B". An assignment to a variable x of sort S is translated as follows:

$$c2l_b(\text{“}x\mathbf{:=}V\mathbf{;}B\text{”}, H, D, U, \Delta) = \mathit{query_probe}(V)\mathbf{;}\ \mathbf{let}\ x : s = c2l_v(V)\ \mathbf{in}\ \tau\mathbf{;}\ c2l_b\big(B, H, D \cup \{x\}, U, \Delta \cup \{x\}\big)$$

Remark. The translation generates the internal transition "τ", also present in the CHP operational semantics (see Section 3) and the Petri net model of [14]. Moreover, as for emissions, if the value V contains probe operations, *query_probe* introduces a confluent sequence of communications that, after hiding, preserves branching equivalence. □

Translation of "A; B". This translation rule applies only if A is a collateral composition or a guarded command (all other cases have been handled before):

$$c2l_b(\text{“}A\mathbf{;}\,B\text{”}, H, D, U, \Delta) = c2l_b(A, H, D, U', D) \gg \mathbf{accept}\ x_1 : s_1, \ldots, x_n : s_n\ \mathbf{in}\ c2l_b(B, H, D', U, \Delta')$$

where $U' = \mathit{use}(B) \cup \big(U \setminus \mathit{def}_n(B)\big)$, $D' = \big(D \cup \mathit{def}(A)\big) \cap U'$, $\{x_1, \ldots, x_n\} = D'$, and $\Delta' = \big(\Delta \cup \mathit{def}(A)\big) \cap U'$.

Remark. The symmetric sequential composition operator of LOTOS "$\gg$" differs from CHP sequential composition by the fact that it creates a τ-transition. In general, this transition is not always confluent (as in the LOTOS behaviour "(c**; stop [] exit**) $\gg$ B", which is equivalent to "c**; stop []** τ**;** B"). However, the preliminary simplifications applied to the CHP description forbid such situations, as the LOTOS behaviour generated for A (which is either a collateral composition and a guarded command) always contains at least one transitions, so that the τ-transition created by "$\gg$" is always prefixed by some other transitions, and thus cannot occur as the first transition of a non-deterministic choice. □

Translation of "B_1,B_2". A collateral composition is translated as follows:

$$c2l_b(\text{“}B_1\mathbf{,}B_2\text{”}, H, D, U, \Delta) = c2l_b(B_1, H, D_1, U, \Delta')\ \ \mathbf{|||}\ \ c2l_b(B_2, H, D_2, U, \Delta')$$

where $D_1 = D \setminus \mathit{def}(B_2)$, $D_2 = D \setminus \mathit{def}(B_1)$, and $\Delta' = \Delta \cup \mathit{def}(B_1) \cup \mathit{def}(B_2)$.

Remark. The translation into LOTOS exploits the similarity of the CHP operator "**,**" with the LOTOS operator " ||| ", the only difference being that the translation into LOTOS introduces a synchronisation on the special termination gate, noted "δ", in order to express the synchronous termination of B_1 and B_2. Because the preliminary simplifications (Section 4.4) always introduce a final "**nil**" wherever a process terminates (i.e. "$\hat{B}_i$" is replaced by "$\hat{B}_i$**; nil**" when $\neg inf(\hat{B}_i)$), such δ-transitions always occur on the left-hand side of a LOTOS "**>>**" sequential composition operator. Therefore, these δ-transitions are captured by the "**>>**" operator, which transforms them into τ-transitions, thus making them confluent (see the remark for sequential compositions). □

Translation of Guards. If a guard V is of the form "$c\#$" or "$c\#V'$", where channel c is of profile "*single*", then V is translated as a communication on the LOTOS gate c with a "**!***Probe*" offer. Otherwise, V is translated directly into a LOTOS guard:

$$c2l_g(V) = \begin{cases} c\ \mathbf{!}Probe\ \mathbf{;} & \text{if } V = c\# \ \wedge\ profile(c) = single \text{ and} \\ & \quad c \text{ links a passive emitter to an active receiver} \\ c\ \mathbf{!}Probe\ \mathbf{?}x : type(c)\ \mathbf{;} & \text{if } V = c\# \ \wedge\ profile(c) = single \text{ and} \\ & \quad c \text{ links an active emitter to a passive receiver} \\ c\ \mathbf{!}Probe\ \mathbf{!}c2l_v(V')\ \mathbf{;} & \text{if } V = c\#V' \ \wedge\ profile(c) = single \\ \texttt{[}c2l_v(V)\texttt{]}\ \texttt{->} & \text{if } cond(V) \end{cases}$$

where x is a new variable (to receive the value of x_c) and the predicate $cond(V)$ is defined by

$$cond(V) \ \text{ iff } \ \neg\big((V = c\# \ \vee\ V = c\#V') \ \wedge\ profile(c) = single\big)$$

Remark. Compared to the CHP operational semantics, the translation into LOTOS adds, in the case where $profile(c) = single$, a communication that — after hiding — corresponds to the τ-transition generated by the first rule for guarded commands in the CHP operational semantics (see Section 3 and the translation of guarded commands below). □

Translation of Guarded Commands. A guarded command $B =$ "**@**$[V_0 \Rightarrow B_0\mathbf{;}\ T_0\ \ldots\ V_n \Rightarrow B_n\mathbf{;}\ T_n]$" is translated into a call to an auxiliary process P_B:

$$c2l_b(\text{“}\mathbf{@}[V_0 \Rightarrow B_0\mathbf{;}\ T_0\ \ldots\ V_n \Rightarrow B_n\mathbf{;}\ T_n]\text{”}, H, D, U, \Delta) =$$
$$\mathbf{choice}\ x_{i_1} : s_{i_1}, \ldots, x_{i_k} : s_{i_k}\ \texttt{[]} \qquad P_B\,\texttt{[}probe_enable, probe_disable, gate(\hat{B}_i, H)\texttt{]}\,(x_1, \ldots, x_m)$$

where

$$\begin{aligned} gate(\hat{B}_i, H) = \ & chan(\hat{B}_i) \cup \\ & \big\{c_init \mid c \in chan(\hat{B}_i) \ \wedge\ profile(c) = general \ \wedge\ H(c) = active\big\} \cup \\ & \big\{c_probe \mid c \in chan(\hat{B}_i) \ \wedge\ profile(c) = general \ \wedge\ H(c) = passive\big\} \end{aligned}$$

and where $D' = \big(D \cup def(B)\big) \cap \big(use(B) \cup (U \setminus def_n(B))\big)$, $\{x_1, \ldots, x_m\} = D'$, $\{i_1, \ldots, i_k\} \subseteq \{1, \ldots, m\}$, i.e. $\{x_{i_1}, \ldots, x_{i_k}\} \subseteq \{x_1, \ldots, x_m\}$, $(\forall j \in \{1, \ldots, m\})\ \ s_j = type(x_j)$, and $(\forall j \in \{i_1, \ldots, i_k\})\ \ x_j \notin D$. The "**choice** $x_{i_1} : s_{i_1}, \ldots, x_{i_k} : s_{i_k}$" is used to assign nondeterministically chosen values to the variables x_{i_1}, ..., x_{i_k}, which might be read when executing P_B, but are not defined before the first call to P_B. This is required, since CHP allows uninitialised variables to be read, whereas LOTOS prohibits this using syntactic restrictions, which are too restrictive in some cases, as for instance the initialisation of a variable in the last iteration of a loop. If $k = 0$, the (empty) "**choice**" statement should be omitted.

The semantics of a guarded command "**@**$[V_0 \Rightarrow B_0\mathbf{;}\ T_0\ \ldots\ V_n \Rightarrow B_n\mathbf{;}\ T_n]$" requires to check all guards simultaneously. Thus, P_B starts by reading the values of the variables x_c for all channels that have profile "*general*" and are probed by some guard V_j ($j \in \{0, \ldots, n\}$). Formally, this set of channels is defined as

$$\{c_1, \ldots, c_l\} = \big\{c \mid c \in \textstyle\bigcup_{j=0}^{n} chan_v(V_j) \ \wedge\ profile(c) = general\big\}$$

Then, P_B offers the possibility to execute any branch $B_i\mathbf{;}T_i$ the guard V_i of which is true. An additional branch handles the case where all the guards containing a probe of channel of profile "*general*" are false: in

this case, a *busy waiting* is implemented (using a recursive call to P_B), so as to allow to probe the channels c_1, ..., c_l once more until some guard becomes true.

The auxiliary process P_B is defined as follows:

process $P_B\,[\mathit{probe_enable}, \mathit{probe_disable}, \mathit{gate}(\hat{B}_i, H)]\,(x_1 : s_1, \ldots, x_m : s_m) :\ F$

$\mathit{probe_enable}$; $c_1_\mathit{probe}\ ?\overline{x_{c_1}} : \mathit{bool}\ ?x_{c_1} : \mathit{type}(x_{c_1})$; ... ; $c_l_\mathit{probe}\ ?\overline{x_{c_l}} : \mathit{bool}\ ?x_{c_l} : \mathit{type}(x_{c_l})$;

(

$c2l_g(V_0)\ \ \mathit{probe_disable};\ \ c2l_c\Big(B_0,\ T_0,\ H,\ D' \setminus \big(\mathit{def}(B_0) \setminus \mathit{use}(B_0)\big),\ U,\ \Delta'\Big)$

[] $\cdots$ []

$c2l_g(V_n)\ \ \mathit{probe_disable};\ \ c2l_c\Big(B_n,\ T_n,\ H,\ D' \setminus \big(\mathit{def}(B_n) \setminus \mathit{use}(B_n)\big),\ U,\ \Delta'\Big)$

[]

[$\displaystyle\bigwedge_{\substack{j \in \{0,\ldots,n\} \\ \wedge\ \mathit{cond}(V_j)}} \big(\neg c2l_v(V_j)\big)$] -> $\mathit{probe_disable}$; $P_B\,[\mathit{probe_enable}, \mathit{probe_disable}, \mathit{gate}(\hat{B}_i, H)]\,(x_1, \ldots, x_m)$

)

endproc

where

$$\Delta' = \Delta \cup \mathit{def}(B)$$

$$F = \begin{cases} \textbf{noexit} & \text{if } \mathit{inf}(B) \\ \textbf{exit}\big(\mathit{type}(x'_1), \ldots, \mathit{type}(x'_q)\big) \text{ such that } \mathit{func}(B, D, U) = \textbf{exit}(\{x'_1, \ldots, x'_q\}) & \text{otherwise} \end{cases}$$

and where the translation of a branch is defined as:

$$c2l_c(B_j, \textbf{break}, H, D, U, \Delta) = c2l_b(B_j, H, D, U, \Delta)$$

$$\begin{aligned} c2l_c(B_j, \textbf{loop}, H, D, U, \Delta) = {} & c2l_b\Big(B_j,\ H,\ D,\ D \cup \big(\mathit{def}(B_j) \setminus \mathit{use}(B_j)\big),\ D\Big) \\ & \text{>> } \textbf{accept}\ x_1 : s_1, \ldots, x_m : s_m\ \textbf{in} \\ & P_B\,[\mathit{probe_enable}, \mathit{probe_disable}, \mathit{gate}(\hat{B}_i, H)]\,(x_1, \ldots, x_m) \end{aligned}$$

Remark. As regards correction, we must distinguish between two parts in the LOTOS code generated for P_B. There is first a *prefix*, which corresponds to the first line of P_B, i.e. the sequence of transitions starting with *probe_enable* and ending with "$c_l_\mathit{probe}\ ?\overline{x_{c_l}} : \mathit{bool}\ ?x_{c_l} : \mathit{type}(x_{c_l})$". Notice that this prefix is similar to the LOTOS code generated by $\mathit{query_probe}(V_0, \ldots, V_n)$ in the translation of value expressions. This prefix is followed by a choice between $(n+1)$ *branches*: the first n branches are the lines starting with "$c2l_g(V_i)$" and the last branch is the line starting with the guard "$\bigwedge_{j \in \{0,\ldots,n\}\ \wedge\ \mathit{cond}(V_j)} \big(\neg c2l_v(V_j)\big)$".

Notice that the guards of these branches are not necessarily mutually exclusive, because CHP guarded commands themselves can be nondeterministic. However, the $(n+1)$ branches are exhaustive, since the guard of the last branch acts as a default case, so that always at least one branch can be taken. Moreover, the definition of P_B guarantees that the synchronisation on *probe_enable* is eventually followed by a synchronisation on *probe_disable*, because all branches contain a synchronisation on *probe_disable*, and all rendezvous between *probe_enable* and *probe_disable* are communications with the *channel_c* processes (see the translation of channels with $\mathit{profile}(c) = \mathit{general}$); these communications are always possible because each *channel_c* process always accepts to synchronise on its *c_probe* gate after a synchronisation on *probe_enable*. Thus, as regards the preservation of branching equivalence:

- The prefix preserves branching equivalence for the same reasons as for the definition of $\mathit{query_probe}(V_0, \ldots, V_n)$: after hiding, the transitions on the *probe_enable* and c_i_probe gates become confluent.
- Each of the first n branches starts with either one or two transitions that match, after hiding, the τ-transition generated by the CHP operational semantics. Indeed, the definition of $c2l_g(V_n)$ implies that the i-th branch starts either, if $\mathit{cond}(V_i)$, with a synchronisation on *probe_disable* or, if $\neg\mathit{cond}(V_i)$, with

a communication having the form "$c\,!Probe$" or "$c\,!Probe\ ?x : type(c)$" followed by a synchronisation on *probe_disable*.

– The last branch has no direct counterpart in the CHP operational semantics. However, it consists of a synchronisation on *probe_disable* followed by a recursive call to P_B (without modification of the value parameters x_1, ..., x_m). Thus, after hiding the *probe_disable* gate, this branch just amounts to a τ-cycle, which preserves branching equivalence.

□

4.6. *Translation of Several Concurrent Processes*

In a nutshell, the parallel composition "$\|$" of CHP is translated into the LOTOS operator "$|[\cdots]|$", because the underlying model of asynchronous concurrency (interleaving semantics) implemented by these two operators is basically the same. However, there are some "surface" differences between both languages worth being mentioned:

– The parallel composition operator of CHP is n-ary but allows only binary communications; to the contrary, the parallel composition operator of LOTOS is binary but allows n-ary communications.

– Although in the general case certain involved communication patterns cannot be expressed in LOTOS [29], translation into LOTOS is always possible for CHP descriptions, since CHP channels have pairwise distinct names.

– Although communication in CHP is always between two parties, the generated LOTOS code contains three-party communications. This is no problem, since in the generated LOTOS code any three-party communication is a communication on a channel c with $profile(c) = general$, which involves exactly two processes $\hat{B}_i$ and $\hat{B}_j$ (sender and receiver) plus one of the processes *channel_c* representing the shared variable x_c (see Section 4.5, translation of channels).

– The synchronisation rules are not identical: in CHP, two processes synchronise implicitly on all common channels; in LOTOS, processes do not synchronise on their common gates, but only on the set of gates g_1, ..., g_n explicitly listed inside the parallel composition operator "$|[g_1, ..., g_n]|$".

The translation of a CHP description $\langle \mathcal{C},\ \mathcal{X},\ \hat{B}_0 \parallel \cdots \parallel \hat{B}_n \rangle$ is defined as:

$$
\begin{aligned}
c2l(\text{“}\hat{B}_0 \parallel \cdots \parallel \hat{B}_n\text{”}) = {} & c2l_p(\text{“}\hat{B}_0 \parallel \cdots \parallel \hat{B}_n\text{”},\ \mathcal{C}) \\
& |[probe_enable, probe_disable, c_1, c_1_probe, \ldots, c_l, c_l_probe, c_{i_1}_init, \ldots, c_{i_p}_init]| \\
& (\\
& \quad channel_c_1\,[probe_enable, probe_disable, c_1, c_1_init, c_1_probe] \\
& \quad |[probe_enable, probe_disable]| \ \cdots\ |[probe_enable, probe_disable]| \\
& \quad channel_c_l\,[probe_enable, probe_disable, c_l, c_l_init, c_l_probe] \\
&)
\end{aligned}
$$

where function $c2l_p$ is defined as follows:

$$
c2l_p(\text{“}\hat{B}_0 \parallel \cdots \parallel \hat{B}_n\text{”},\ C) = \begin{cases} c2l_b(\hat{B}_0, H_0, \varnothing, \varnothing, \varnothing) & \text{if } n = 0 \\ c2l_b(\hat{B}_0, H_0, \varnothing, \varnothing, \varnothing)\ \ |[chan_{bin}(\hat{B}_0) \cap C]|\ \ c2l_p\big(\text{“}\hat{B}_1 \parallel \cdots \parallel \hat{B}_n\text{”},\ C \setminus chan_{bin}(\hat{B}_0)\big) & \text{otherwise} \end{cases}
$$

where $chan_{bin}(B) = \big\{c \mid c \in chan(B)\ \wedge\ \neg port(c)\big\}$ is the set of binary channels occurring in process B, $\{c_1, \ldots, c_l\} = \big\{c \mid c \in C\ \wedge\ profile(c) = general\big\}$ is the set of all channels of profile "*general*" in C, and $\{i_1, \ldots, i_p\} = \big\{i \mid i \in \{1, \ldots, l\}\ \wedge\ \neg port(c_i)\big\}$ are the indexes i such that $c_i \in \{c_1, \ldots, c_l\}$ is a binary channel (and no port) of profile "*general*". We distinguish between the constant $\mathcal{C}$ corresponding to the set of channels of the CHP description and the parameter C used in the recursive definition of $c2l_p$ to represent the set of channels for which a communication has yet to be generated.

4.7. *Optimisations of the Generated LOTOS Code*

In addition to the aforementioned code specialisation technique (i.e. different translations of a channel depending on the profile), several other optimisations are possible.

- A CHP guard that is always *true* can be dismissed in the corresponding LOTOS specification.
- Boolean expressions in LOTOS guards could be simplified using standard techniques (boolean algebra or BDD).
- The "**accept**" construct needs to be generated only if some variable values must be transmitted over a sequential composition >> (see translation of the **loop** construct in *Translation of Guarded Commands*, Section 4.5 as an example).
- If a guarded command is the left hand side of a sequential composition "**@**[...]**;** B'", we generate a second auxiliary process $P_{B'}$ (for B'); each "**break**" is translated into a call to $P_{B'}$ and the functionality F is computed with respect to B'. This optimisation avoids the introduction of a τ-transition due to the "**exit**" (generated by the translation of "**break**"). For each B_i such that $inf(B_i)$, T_i is not translated at all.
- In the definition of $query_probe(V_0, ..., V_1)$, if the set of channels of profile "*general*" probed by V_0, ..., V_n is empty, the synchronisations on gates *probe_enable* and *probe_disable* are not required.
- If none of the guards of a guarded commands probes a channel of profile "*general*", the synchronisation on gate *probe_enable* is unnecessary and can be removed. In this case, to match the τ-transition generated by the CHP operational semantics, the synchronisation on gate *probe_disable* must be replaced by a τ.

Such optimisations have been implemented in the translator we will present in Section 5.

4.8. *Example of the Arbiter without Priorities*

For the arbiter without priorities of Section 2.6, $profile(c) = unprobed$ (because c is never probed) and $profile(c_1) = profile(c_2) = single$ (because any probe operation on c_1 and c_2 is a guard). The LOTOS behaviour obtained by applying the translation rules of Section 4.6 is the following:

$$arbiter\,[c, c_1, c_2] \;\; |[\, c_1, c_2 \,]| \;\; \big(client\text{-}1\,[c_1] \;\; ||| \;\; client\text{-}2\,[c_2]\big)$$

Processes *client*-1, *client*-2, and *arbiter* are defined as:

process *client*-1[c_1] : **noexit** :=
  *channel*_c_1[c_1] >> *client*-1[c_1]
where
  process *channel*_c_1[c_1] : **exit** := c_1 **; exit** [] c_1 **!***Probe* **;** *channel*_c_1[c_1] **endproc**
endproc

process *client*-2[c_2] : **noexit** :=
  *channel*_c_2[c_2] >> *client*-2[c_2]
where
  process *channel*_c_2[c_2] : **exit** := c_2 **; exit** [] c_2 **!***Probe* **;** *channel*_c_2[c_2] **endproc**
endproc

process *arbiter*[c, c_1, c_2] : **noexit** :=
  (c_1 **!***Probe***;** (c **!1; exit** ||| c_1**; exit**) >> *arbiter*[c, c_1, c_2])
  []
  (c_2 **!***Probe***;** (c **!2; exit** ||| c_2**; exit**) >> *arbiter*[c, c_1, c_2])
endproc

Processes *client*-1 (respectively *client*-2) repeat forever the auxiliary process *channel*_c_1 (respectively *channel*_c_2), followed by a recursive call (the CHP **loop** construct being translated into a recursive LOTOS process call, see *Translation of Guarded Commands*, Section 4.5). Both processes *channel*_c_1 and *channel*_c_2 are obtained as described in Section 4.5 (translation of "c**!**V", "*single*" case). As regards the process *arbiter*, CHP guarded commands are translated to a LOTOS choice ([]). Guards consisting of a single probe are

translated as a communication with offer "**!***Probe*" (see *Translation of Guards* in Section 4.5). At last, the collateral composition is encoded as an interleaving " ||| " (of an emission on c and a communication on c_i). In both branches of the arbiter, the **loop** is translated in a recursive call to process *arbiter*.

The LTS generated from this LOTOS code by the CÆSAR tool of CADP has 72 states and 165 transitions after reduction modulo strong bisimulation. The LTS obtained after hiding all transitions containing a "**!***Probe*" offer was proved (by the BISIMULATOR tool [30] of CADP) to be branching equivalent to the one corresponding to the CHP operational semantics. Without code specialisation, the LTS generated by CÆSAR (also reduced modulo strong bisimulation) would have been about 20% larger, with 85 states and 198 transitions, still being branching equivalent.

4.9. *Example of the Arbiter with Priorities*

For the arbiter with priorities of Section 2.6, $profile(c) = unprobed$ and $profile(c_1) = profile(c_2) = general$ (because c_1 and c_2 are probed in an expression). The translation rules yield the following LOTOS behaviour:

$$\left(\begin{array}{l} arbiter\,[probe_enable, probe_disable, c, c_1, c_1_probe, c_2, c_2_probe] \;\; |[\, c_1, c_2 \,]| \\ \big(client\text{-}1\,[c_1, c_1_init] \;\; ||| \;\; client\text{-}2\,[c_2, c_2_init]\big) \end{array}\right)$$
$$|[\, probe_enable, probe_disable, c_1, c_1_init, c_1_probe, c_2, c_2_init, c_2_probe \,]|$$
$$\left(\begin{array}{l} channel_c_1\,[probe_enable, probe_disable, c_1, c_1_init, c_1_probe] \\ |[probe_enable, probe_disable]| \\ channel_c_2\,[probe_enable, probe_disable, c_2, c_2_init, c_2_probe] \end{array}\right)$$

where the processes $channel_c_1$ and $channel_c_2$ are obtained as described in Section 4.5 — *Translation of Channels*. The other processes are defined as follows:

process $client\text{-}1\,[c_1, c_1_init]$: **noexit** :=
  c_1_init **!***true***;** c_1 **!***true***;** $client\text{-}1\,[c_1, c_1_init]$
  []
  c_1_init **!***false***;** c_1 **!***false***;** $client\text{-}1\,[c_1, c_1_init]$
endproc

process $client\text{-}2\,[c_2, c_2_init]$: **noexit** :=
  c_2_init**;** c_2**;** $client\text{-}2\,[c_2, c_2_init]$
endproc

process $arbiter\,[probe_enable, probe_disable, c, c_1, c_1_probe, c_2, c_2_probe]$: **noexit** :=
  $probe_enable$**;** c_1_probe **?**$\overline{x_{c_1}}$: *bool* **?**x_{c_1} : *bool***;** c_2_probe **?**$\overline{x_{c_2}}$: *bool***;**
  (
    $[\overline{x_{c_1}}]$ -> $probe_disable$**;** $\big((c$ **!1; exit**$)$ ||| $(c_1$ **?**x**; exit**$)\big)$ >>
      $arbiter\,[probe_enable, probe_disable, c, c_1, c_1_probe, c_2, c_2_probe]$
    []
    $[\overline{x_{c_2}} \wedge \neg(\overline{x_{c_1}} \wedge x_{c_1})]$ -> $probe_disable$**;** $\big((c$ **!2; exit**$)$ ||| $(c_2$**; exit**$)\big)$ >>
      $arbiter\,[probe_enable, probe_disable, c, c_1, c_1_probe, c_2, c_2_probe]$
    []
    $[\neg(\overline{x_{c_1}}) \wedge \neg\big(\overline{x_{c_2}} \wedge \neg(\overline{x_{c_1}} \wedge x_{c_1})\big)]$ -> $probe_disable$**;**
      $arbiter\,[probe_enable, probe_disable, c, c_1, c_1_probe, c_2, c_2_probe]$
)
endproc

Since channel c_1 (respectively c_2) has profile "*general*", any active emission on c_1 (respectively c_2) is preceded by a communication on c_1_init (respectively c_2_init) with process $channel_c_1$ (respectively $channel_c_2$) to allow probes. Similarly, execution of process *arbiter* starts with communications on gates c_1_probe and

c_2*_probe* with processes *channel*_c_1 and *channel*_c_2 to check (or probe) whether a channel is ready for communication (first offer) and to retrieve the value possibly sent (second offer). Note that a third choice is added in process *arbiter*, whose guard corresponds to the negation of the two other guards, and whose body is a simple call to the process *arbiter* itself. This case is necessary to check again the possibility to probe channels c_1 and c_2, if none of the probes was successful the first time.

The LTS corresponding to this LOTOS specification, generated by the CÆSAR tool of CADP and reduced modulo strong bisimulation, has 138 states and 303 transitions. The LTS obtained after hiding all labels corresponding to communications on gates c_1*_init*, c_2*_init*, c_1*_probe*, and c_2*_probe* was proved (by the BISIMULATOR tool of CADP) to be branching equivalent to the one presented in Figure 3.

4.10. *Correction Concerns about the Translation from CHP into LOTOS*

Our translation algorithm from CHP into LOTOS is implemented in a translator (see Section 5) that is being used by hardware designers for validating several asynchronous circuits. Interestingly, when discussing correctness issues with these designers, we received the feedback that a formal proof of correction for our translator was a low priority issue. In addition to the usual constraints (lack of resources and time-to-market pressure), we can mention the following reasons:

(i) In the hardware community, circuit designers rely on closed-source commercial tools to perform semantic translations, for instance to transform a high-level description into RTL (*Register Transfer Level*), to synthesise a netlist from an RTL description, or to perform placement and routing for a netlist. In most cases, designers do not have any formal proof of correctness of for the complex algorithms present in these tools. There are even documented situations in which the tools are known to generate incorrect or inefficient outputs, which have to be corrected or optimised manually later.

(ii) Even if the tools were fully reliable, with correction proofs available, it is still unlikely that designers would trust them blindly because of the huge costs arising from mistakes in hardware circuits. Instead, designers routinely use independent cross-checking tools to verify whether commercial translation tools have produced correct outputs or not. Examples of such tools are symbolic model checkers, which verify that a generated netlist is equivalent to its high level design, or LVS (*Layout Versus Schematic*) checkers, which verify that an electric circuit obtained after placement and routing correctly implements its netlist.

(iii) The intended use of our translator is to enable model checking of CHP descriptions. In general, formal semantics is not of prime importance to model checkers — some famous model checkers do not even have a formal definition for their input languages! This is partly justified by the use of model checkers for "bug hunting": the model checker finds possible problems, which are then analysed manually to understand if they are "real" bugs in the design or "false positives" (caused by excessive abstractions or issues in the model checker itself). However, even if model checkers are not fully proven (with a non-null, yet small, probability of reporting false positives or even omitting the real bugs), the use of model checking is still beneficial, as it often uncovers real problems that would have remained undetected otherwise. This was the case with our translator (see Section 5); we can also mention that no translation bug has been reported so far.

Despite a formal proof of correction for our translator was considered of little practical interest by the end-users, the remainder of this section addresses this issue.

According to concurrency theory results, a natural approach to prove the correction of our translation would be to seek for an equivalence relation (such as a bisimulation) between the LTS (noted "LTS_{SOS}") obtained from a CHP description $\langle \mathcal{C},\ \mathcal{X},\ \hat{B}_0 \parallel \cdots \parallel \hat{B}_n \rangle$ using the proposed operational semantics for CHP and the LTS (noted "$\text{LTS}_{\text{LOTOS}}$") corresponding to the LOTOS specification *c2l*("$\hat{B}_0 \parallel \cdots \parallel \hat{B}_n$").

This task is not easy, since some CHP descriptions do not have a defined semantics, i.e. if they access variables that are neither initialised nor assigned before. For these CHP descriptions, our SOS semantics is undefined (as some variables evaluate to the undefined value "$\perp$"). To the contrary, LOTOS has a set of static semantic constraints that rule out all LOTOS descriptions in which a variable is used before being defined, so that each LOTOS description satisfying these constraints has a "well-defined" dynamic semantics. Thus,

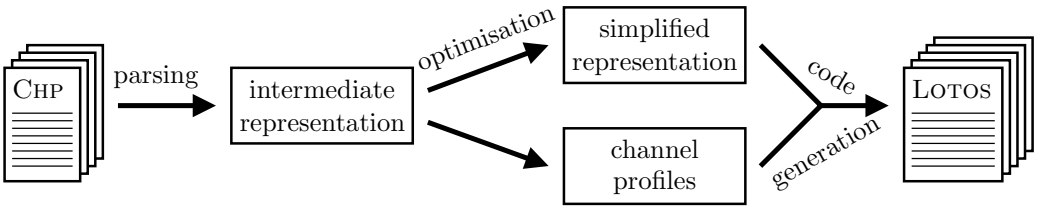

Fig. 6. Translation steps in CHP2LOTOS

only the subset of CHP that forbids accesses to undefined variables should be considered.

Moreover, finding an equivalence is difficult, as LTS_{SOS} and $\text{LTS}_{\text{LOTOS}}$ have different sets of actions (i.e. labels of transitions). Indeed, due to the translation of the probe operation, $\text{LTS}_{\text{LOTOS}}$ might contain additional gates (such as *probe_enable*, *probe_disable*, *c_init*, and *c_probe*) that do not exist in LTS_{SOS}. We suggest to hide these additional gates using the "**hide** G **in**" construct of LOTOS. However, this is not sufficient, since $\text{LTS}_{\text{LOTOS}}$ might also contain additional offers "**!***Probe*" which do not exist in LTS_{SOS}. We suggest to apply a renaming operator (which does not exist in LOTOS, but in other process calculi such as E-LOTOS [31]) to rename into τ all labels containing a "**!***Probe*" offer. Let $\text{LTS}'_{\text{LOTOS}}$ be the LTS obtained from $\text{LTS}_{\text{LOTOS}}$ after these hiding and renaming steps.

LTS_{SOS} and $\text{LTS}'_{\text{LOTOS}}$ are still not equivalent with respect to strong bisimulation, since the hiding and renaming steps introduce τ-transitions that do not exist in LTS_{SOS}. Besides, the symmetric sequential composition operator "**>>**" of LOTOS also introduces τ-transitions that do not exist in LTS_{SOS}. Thus, our translation can only preserve a weak equivalence, i.e. an equivalence that handles τ-transitions differently from ordinary transitions.

Furthermore, as mentioned before in Section 4.5, the translation of guarded commands creates τ-cycles (namely, in the last branch of process P_B, i.e. the line starting with "$\bigwedge_{j \in \{0,\ldots,n\} \wedge cond(V_j)} \big(\neg c2l_v(V_j)\big)$"). Thus, it is necessary to consider a weak equivalence relation that abstracts τ-cycles away. This suggests to consider branching bisimulation [20], the strongest of the weak equivalences found in the literature.

To prove that LTS_{SOS} and $\text{LTS}'_{\text{LOTOS}}$ are branching equivalent, one should proceed in two steps: (1) consider each CHP process $\hat{B}_i$ separately, apply the preliminary simplifications described in Section 4.4, and prove that its translation into LOTOS preserves branching equivalence — this can be done by induction on the syntax of $\hat{B}_i$; (2) extend this result to the parallel composition of several processes "$\hat{B}_0 \parallel \cdots \parallel \hat{B}_n$" taking into account that branching bisimulation is a congruence for the parallel composition operator of LOTOS. Proof arguments and remarks that substantiate this have been given throughout Sections 4.4, 4.5, and 4.6.

5. Implementation and Applications

We developed a translator from CHP into LOTOS, called CHP2LOTOS, which supports full CHP including value passing communication, ports open to the environment, and hierarchical components. CHP2LOTOS was developed using the SYNTAX and LOTOS NT compiler construction technologies [32] and consists of 2,200 lines of SYNTAX, 13,400 lines of LOTOS NT, and 3,900 lines of C. The translation is performed in several steps as depicted in Figure 6. The parsing and elaboration phases construct an intermediate representation, which is then optimised and simplified using the transformations described in Section 4.4. After computing the channel profiles, the translator generates LOTOS code according to the translation functions defined in Sections 4.5 and 4.6.

Taking advantage of the possibility offered by the CADP toolbox to implement LOTOS data types using external C code, the predefined CHP data types have been directly implemented as C libraries, thus avoiding the recompilation of the generic data types each time the LOTOS code is used for state space generation or verification.

We validated the CHP to LOTOS translator on more than 500 CHP specifications, corresponding to about 14,500 lines of CHP, which (after translation and pretty-printing) generate about 34,700 lines of LOTOS code. This increase in the number of generated lines is mainly due to the LOTOS pretty-printer of CADP,

which puts each communication action on a separate line. We also applied the CHP to LOTOS translator to two asynchronous circuits designed at the CEA/Leti laboratory in Grenoble (France) that we present with more details below.

Data Encryption Standard. The DES (*Data Encryption Standard*) [15] defines a method for encrypting information. It accepts as input two 64-bit words (a data and a key) and an operation mode (i.e. ciphering or deciphering), and outputs the (de)ciphered value. In this case study, we experimented CHP2LOTOS on a CHP description of an asynchronous implementation of the DES (25 processes, which correspond to about 1,600 lines of CHP), which has also been studied in [11]. We focused on the control part, abstracting 64-bit words into a single data value.

Since this case study contains many concurrent processes, direct generation of the LTS failed due to lack of memory (after 70 minutes, the generated LTS had more than 17 million states and 139 million transitions). However, using the compositional verification techniques (decomposing, minimising, and recomposing processes) [33] of the CADP toolbox, we generated an equivalent, but smaller LTS (16,910 states and 85,840 transitions) in 8 minutes on a SunBlade 100 (500 Mhz Ultra Sparc II processor, 1.5 GB of RAM). In a second step, several properties were proved on this LTS, such as deadlock freeness and some safety and liveness properties. As an example, we checked that after the reception of the three inputs (key, data, decrypt), an output is always returned. Once the LTS is generated from the LOTOS specification, only a few seconds suffice to verify each property.

Asynchronous Network on Chip Architecture. As a second case study, we considered the ANOC (Asynchronous Network On Chip) architecture [16,34], which is used as the backbone of FAUST, a 4th generation wireless telecom baseband [35]. ANOC implements the GALS paradigm, in which synchronous *resources* (e.g. memory, generic processor, dedicated hardware for Fourier transformation or MPEG decoding) are linked via an asynchronous communication network, consisting of a set of *nodes*, each of which is connected to a resource and four other nodes. Although no assumption is made on the topology of the network as regards the architecture of the node, the FAUST implementation of ANOC connects the nodes in a 2D-Mesh topology for several reasons (e.g. scalable bandwidth, easy placement and route on silicon).

Each network node provides lower-level network services, i.e. routing and arbitration of the transiting messages, called *flits*. Each node consists of five input controllers, which route incoming flits to one of the other four ports (a flit is never returned to the incoming port), and five output controllers, which arbitrate between flits heading for the same output.

In this case study, we dealt in particular with the verification of the most complex component of a node of ANOC, namely an input controller. As for the DES model, the ANOC model in CHP was abstracted by considering only four transmitted values (to identify different flits) plus the control information required for routing and handling *packets* (i.e. sequences of flits). Furthermore, we used a traffic generator (specified in CHP) to implement several realistic scenarios with packets of different length (one or more flits), either emitted sequentially or overlapping on different channels and to different destinations.

For each scenario, the CHP model of the input controller (about 1,200 lines of CHP) was translated into LOTOS (about 3,600 lines of code) using CHP2LOTOS (the translation takes less than one second). Then, we applied the compositional techniques of CADP to generate the state space. The generation of the LTS corresponding to the input controller of ANOC for a particular cycle of four flits was automated using an SVL script [36] (about 500 lines), which invokes automatically the different CADP tools for state space generation, hiding, minimisation, and composition. The SVL script is generic in the sense that it can be used to generate the state spaces for all scenarios. For a typical scenario, the SVL script generates in about four minutes the corresponding LTS (1,300 states and 3,116 transitions), the largest intermediate LTS observed during the generation having 295,893 states and 812,283 transitions.

Independently from any CHP model, we enunciated functional properties describing the protocol behaviour of the input controller, such as protocol correctness (the input controller must comply at its inputs with the ANOC protocol, and transmit the incoming data to an output controller), data integrity (the contents of the communications must be preserved by the input controller), and correct packet routing (the input

controller has to route all the flits of a packet in the right direction). These properties were verified using a generic SVL script (of about 250 lines), calling the model checking and equivalence checking tools of CADP.

During this verification stage, in collaboration with the authors of [16], we detected automatically a routing error — which previously had been found only manually and corrected in the synthesised and manufactured chip.

More details about the ANOC case study are presented in [37]. Recently, CEA/Leti managed the compositional verification of the output controller of ANOC using CHP2LOTOS and CADP.

Last but not least, as for the simple arbiter example, the code specialisation technique proved to be effective in both case studies, by reducing the size of the intermediate state spaces up to a factor of 89 as regards states and 156 as regards transitions, which had the effect of reducing the overall run-time of the compositional generation and verification by a factor of two.

6. Related Work

In general, the semantics of hardware process calculi is not given in terms of SOS rules (as it is usual in the concurrency theory community), but rather by means of a translation into another formalism, e.g. handshake circuits for TANGRAM [38], Petri nets for CHP [14], and CSP for BALSA [39]. In that respect, our SOS semantics is an original contribution allowing to bridge the gap between hardware process calculi and mainstream process calculi.

As regards the verification of asynchronous circuits and architectures, we distinguish two lines of related work:

- A first line of work focusses on the verification of asynchronous designs without paying attention to synthesis. In these approaches, the asynchronous designs are specified in the input language of the verification tool to be used, such as PROMELA for SPIN [40,41], CCS for CWB [42,43], LOTOS for CADP [44–46], or CSP for FDR2 [47,48]. These approaches target gate level descriptions of asynchronous circuits [40–42,44–48] or explicitly model of wires of unbounded delay [43]. This is different from our approach, which aims at verifying high level CHP descriptions from which gate level models can be synthesised automatically. Furthermore, a common issue with these approaches is the necessity of having two different descriptions of the circuit, one for verification and one for synthesis.
- A second line of work devises verification techniques for asynchronous designs described using hardware process calculi, for which synthesis tools exist. There are fewer works in this line; we can mention three such approaches.
  - Firstly, the process algebra CIRCAL (*Circuit Calculus*) and its associated tool [49] have been designed for the automatic verification of circuits. A particularity of CIRCAL is the possibility that several actions may happen simultaneously, which enables a precise modelling of systems combining synchrony and asynchrony. Recent extensions of CIRCAL allow to specify timing and performance properties by adding observer processes [50]. Contrary to CHP, CIRCAL provides no direct support for value passing communication, although this allows more abstract and concise models and is essential for describing large complex systems. Verification in CIRCAL mainly relies on equivalence checking. By translating to LOTOS and using CADP, our approach offers a wider variety of verification techniques, including equivalence checking, model checking of μ-calculus formulae, as well as on-the-fly techniques, static analysis, and distributed and compositional verification.
  - Secondly, the BALSA language can also be used together with CSP. [39] starts with a BALSA description of circuits, and sketches, but does not detail, a translation from BALSA into CSP. Contrary to our approach, the handling of probe-like operations is simpler in [39], since BALSA's handshake enclosure is more restrictive than CHP's probe operation (in particular, the equivalent of a probe can be used only in guards). Furthermore, the approach of [39] does not translate a BALSA process B independently of the other BALSA processes communicating with B, whereas our approach without code specialisation is modular in the sense that it allows to translate each CHP process into LOTOS regardless of its context.

· Thirdly, [11] is a precursor of our approach in the sense that it is also based on CHP and uses CADP too. Contrary to our approach, [11] translates CHP into networks of communicating automata. Thus, [11] cannot handle CHP processes with intertwined sequential and parallel compositions, except by flattening parallel processes, which is less efficient than our translation from CHP into LOTOS. Additionally, our approach brings support of the probe operation. As regards the efficiency of verification, we observed reductions of the LTS generation time by factors up to four on the same DES case study as [11].

Compared to a previous conference publication [17], the present article brings original material. In [17], the probe operations of CHP could only be used in guards; the present article lifts this restriction by supporting probe operations in any expression. Moreover, the present article introduces proof arguments about the correctness of the translation as well as code specialisation to optimise the translation. Finally, it reports on experiments on a much larger set of 500 examples, and features a second case study.

7. Conclusion

In hardware design, synchronous techniques have been predominating, but they face problems when implementing the global clock; asynchronous circuits and architectures are a promising approach to avoid these problems. A major issue is that asynchronous design is more complex than synchronous design, due to interleavings of concurrent processes. Thus, ensuring the correctness of asynchronous designs as early as possible is essential to avoid the cost of synthesising erroneous circuits. However, contrary to synchronous design, there does not yet exist an established methodology for the verification of asynchronous designs.

In this article, we proposed a formal semantics for the hardware process calculus CHP, including value-passing communication, ports open to the environment, and probe operations. Moreover, our semantics is independent of any particular (2- or 4-phase) handshake protocol expansion used for circuit implementation. This semantics has been approved by our colleagues from CEA/Leti and the developers of the TAST tool. Based on this semantics, we formalised a translation of CHP into the international standard LOTOS and implemented this translation in the CHP2LOTOS tool, which has been validated on many examples.

Our work prefigures a framework for the design of asynchronous circuits and architectures, enabling asynchronous designs to be specified using CHP, then translated into LOTOS, verified using the CADP model checking and equivalence checking tools, and finally synthesised using the TAST tool. This tool chain has been applied successfully to two industrial asynchronous circuits and is being used for validating other parts of the ANOC architecture.

Acknowledgements.

We are grateful to our colleagues Edith Beigné, François Bertrand, Yvain Thonnart, and Pascal Vivet (from CEA/Leti laboratory), as well as Marc Renaudin, Dominique Borrione, and Menouer Boubekeur (from TIMA laboratory) for interesting discussions on CHP and TAST, in particular their explanations regarding the probe operation. We also thank the same colleagues from CEA/Leti laboratory for their cooperation in the DES and ANOC case studies.

References

[1] S. Hauck, Asynchronous design methodologies: An overview, Proceedings of the IEEE 83 (1) (1995) 69–93.

[2] K. van Berkel, M. B. Josephs, S. M. Nowick, Scanning the technology: Applications of asynchronous circuits, Proceedings of the IEEE, special issue on Asynchronous Circuits and Systems 87 (2) (1999) 223–233.

[3] A. J. Martin, Compiling communicating processes into delay-insensitive VLSI circuits, Distributed Computing 1 (4) (1986) 226–234.

[4] D. Edwards, A. Bardsley, Balsa: An asynchronous hardware synthesis language, The Computer Journal 45 (1) (2002) 12–18.

[5] A. Peeters, M. de Wit, Haste Manual, Version 3.0, Handshake Solutions (2006).

[6] J. L. W. Kessels, A. M. G. Peeters, The Tangram framework (embedded tutorial): Asynchronous circuits for low power, in: Proceedings of the Asia and South Pacific Design Automation Conference ASP-DAC 2001 (Yokohama, Japan), ACM, 2001, pp. 255–260.

[7] J. A. Bergstra, A. Ponse, S. A. Smolka (Eds.), Handbook of Process Algebra, Elsevier, 2001.

[8] W. Fokkink, Introduction to Process Algebra, Texts in Theoretical Computer Science, Springer Verlag, 2000.

[9] A. J. Martin, The probe: An addition to communication primitives, Information Processing Letters 20 (3) (1985) 125–130.

[10] M. Renaudin, TAST Compiler and TAST_CHP Language, Version 0.6, TIMA Laboratory, CIS Group (2005).

[11] D. Borrione, M. Boubekeur, L. Mounier, M. Renaudin, A. Sirianni, Validation of asynchronous circuit specifications using IF/CADP, in: M. Glesner, R. A. da Luz Reis, H. Eveking, V. J. Mooney, L. S. Indrusiak, P. Zipf (Eds.), Proceedings of the International Conference on Very Large Scale Integration of System-on-Chip VLSI-SoC 2003 (Darmstadt, Germany), Darmstadt, 2003, pp. 86–91.

[12] H. Garavel, F. Lang, R. Mateescu, W. Serwe, CADP 2006: A toolbox for the construction and analysis of distributed processes, in: W. Damm, H. Hermanns (Eds.), Proceedings of the 19th International Conference on Computer Aided Verification CAV'2007 (Berlin, Germany), Vol. 4590 of Lecture Notes in Computer Science, Springer Verlag, 2007, pp. 158–163.

[13] ISO/IEC, LOTOS — a formal description technique based on the temporal ordering of observational behaviour, International Standard 8807, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève (Sep. 1989).

[14] M. Renaudin, A. Yakovlev, From hardware processes to asynchronous circuits via Petri nets: an application to arbiter design, in: Proceedings of the Workshop on Token Based Computing TOBACO'04 (Bologna, Italy), 2004.

[15] NIST, Data encryption standard (DES), Federal Information Processing Standards FIPS PUB 46-3, National Institute of Standards and Technology (Oct. 25 1999).

[16] E. Beigné, F. Clermidy, P. Vivet, A. Clouard, M. Renaudin, An asynchronous NoC architecture providing low latency service and its multi-level design framework, in: Proceedings of the 11th IEEE International Symposium on Asynchronous Circuits and Systems ASYNC'05 (New York, USA), IEEE Computer Society Press, 2005, pp. 54–63.

[17] G. Salaün, W. Serwe, Translating hardware process algebras into standard process algebras — illustration with CHP and LOTOS, in: J. van de Pol, J. Romijn, G. Smith (Eds.), Proceedings of the 5th International Conference on Integrated Formal Methods IFM'2005 (Eindhoven, The Netherlands), Vol. 3771 of Lecture Notes in Computer Science, Springer Verlag, 2005, pp. 287–306, full version available as INRIA Research Report RR-5666.

[18] F. S. de Boer, C. Palamidessi, A fully abstract model for concurrent constraint programming, in: S. Abramsky, T. S. E. Maibaum (Eds.), Proceedings of the International Joint Conference on Theory and Practice of Software Development TAPSOFT'91, Volume 1, Colloquium on Trees in Algebra and Programming CAAP'91 (Brighton, United Kingdom), Vol. 493 of Lecture Notes in Computer Science, Springer Verlag, 1991, pp. 296–319.

[19] W. Serwe, On concurrent functional-logic programming, Thèse de doctorat, Institut National Polytechnique de Grenoble (Mar. 2002).

[20] R. J. van Glabbeek, W. P. Weijland, Branching-time and abstraction in bisimulation semantics (extended abstract), CS R8911, Amsterdam, also in proc. IFIP 11th World Computer Congress, San Francisco, 1989 (1989).

[21] H. Garavel, J. Sifakis, Compilation and verification of LOTOS specifications, in: L. Logrippo, R. L. Probert, H. Ural (Eds.), Proceedings of the 10th International Symposium on Protocol Specification, Testing and Verification (Ottawa, Canada), IFIP, North Holland Publishing Company, 1990, pp. 379–394.

[22] R. Milner, A Calculus of Communicating Systems, Vol. 92 of Lecture Notes in Computer Science, Springer Verlag, 1980.

[23] R. Milner, Communication and Concurrency, Prentice-Hall, 1989.

[24] C. A. R. Hoare, Communicating Sequential Processes, Prentice-Hall, 1985.

[25] H. Ehrig, B. Mahr, Fundamentals of Algebraic Specification 1 — Equations and Initial Semantics, Vol. 6 of EATCS Monographs on Theoretical Computer Science, Springer Verlag, 1985.

[26] T. Bolognesi, E. Brinksma, Introduction to the ISO specification language LOTOS 14 (1) (1988) 25–59.

[27] H. Garavel, W. Serwe, State space reduction for process algebra specifications, Theoretical Comput. Sci. 351 (2) (2006) 131–145.

[28] J. Groote, J. Pol, State space reduction using partial τ-confluence, in: M. Nielsen, B. Rovan (Eds.), Proceedings of the 25th International Symposium on Mathematical Foundations of Computer Science MFCS'2000 (Bratislava, Slovakia), Vol. 1893 of Lecture Notes in Computer Science, Springer Verlag, Berlin, 2000, pp. 383–393, also available as CWI Technical Report SEN-R0008, Amsterdam, March 2000.

[29] H. Garavel, M. Sighireanu, A graphical parallel composition operator for process algebras, in: J. Wu, Q. Gao, S. T. Chanson (Eds.), Proceedings of the Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification FORTE/PSTV'99 (Beijing, China), IFIP, Kluwer Academic Publishers, 1999, pp. 185–202.

[30] D. Bergamini, N. Descoubes, C. Joubert, R. Mateescu, Bisimulator: A modular tool for on-the-fly equivalence checking, in: N. Halbwachs, L. Zuck (Eds.), Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2005 (Edinburgh, Scotland, UK), Vol. 3440 of Lecture Notes in Computer Science, Springer Verlag, 2005, pp. 581–585.

[31] ISO/IEC, Enhancements to LOTOS (E-LOTOS), International Standard 15437:2001, International Organization for Standardization — Information Technology, Genève (Sep. 2001).

[32] H. Garavel, F. Lang, R. Mateescu, Compiler construction using LOTOS NT, in: N. Horspool (Ed.), Proceedings of the 11th International Conference on Compiler Construction CC 2002 (Grenoble, France), Vol. 2304 of Lecture Notes in Computer Science, Springer Verlag, 2002, pp. 9–13.

[33] F. Lang, Compositional verification using SVL scripts, in: J.-P. Katoen, P. Stevens (Eds.), Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems TACAS'2002 (Grenoble, France), Vol. 2280 of Lecture Notes in Computer Science, Springer Verlag, 2002, pp. 465–469.

[34] E. Beigné, P. Vivet, Design of off-chip and on-chip interfaces for a GALS NoC architecture, in: Proceedings of the 12th IEEE International Symposium on Asynchronous Circuits and Systems ASYNC'06 (Grenoble, France), IEEE Computer Society Press, 2006, pp. 172–181.

[35] Y. Durand, C. Bernard, D. Lattard, FAUST : On-chip distributed architecture for a 4G baseband modem SoC, in: Proceedings of Design and Reuse IP-SOC'05 (France), 2005, pp. 51–55.

[36] H. Garavel, F. Lang, SVL: a scripting language for compositional verification, in: M. Kim, B. Chin, S. Kang, D. Lee (Eds.), Proceedings of the 21st IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems FORTE'2001 (Cheju Island, Korea), IFIP, Kluwer Academic Publishers, 2001, pp. 377–392, full version available as INRIA Research Report RR-4223.

[37] G. Salaün, W. Serwe, Y. Thonnart, P. Vivet, Formal verification of CHP specifications with CADP — illustration on an asynchronous network-on-chip, in: Proceedings of the 13th IEEE International Symposium on Asynchronous Circuits and Systems ASYNC 2007 (Berkeley, California, USA), IEEE Computer Society Press, 2007, pp. 73–82.

[38] K. van Berkel, Handshake Circuits: An Asynchronous Architecture for VLSI Programming, Vol. 5 of International Series on Parallel Computation, Cambridge University Press, 1993.

[39] X. Wang, M. Z. Kwiatkowska, G. Theodoropoulos, Q. Zhang, Towards a unifying CSP approach for hierarchical verification of asynchronous hardware, in: M. R. A. Huth (Ed.), Proceedings of the 4th International Workshop on Automated Verification of Critical Systems AVoCS'04 (London, UK), Vol. 128 of Electronic Notes in Theoretical Computer Science, 2004, pp. 231–246.

[40] G. Baulch, D. Hemmendinger, C. Traver, Analyzing and verifying locally clocked circuits with the concurrency workbench, in: Proceedings of the 5th Great Lakes Symposium on VLSI GLSVLSI'95 (Buffalo, USA), IEEE computer Society, 1995, pp. 144–147.

[41] B. Rahardjo, SPIN as a hardware design tool, in: J.-C. Gregoire (Ed.), Proceedings of the First SPIN Workshop SPIN 1995 (Quebec, Canada), 1995.

[42] G. Clark, G. Taylor, The verification of asynchronous circuits using CCS, Tech. Rep. ECS-LFCS-97-369, University of Edinburgh, Department of Computer Science (Oct. 1997).

[43] H. K. Kapoor, M. B. Josephs, Modelling and verification of delay-insensitive circuits using CCS and the Concurrency Workbench, Information Processing Letters 89 (6) (2004) 293–296.

[44] J. He, K. J. Turner, Verifying and testing asynchronous circuits using LOTOS, in: T. Bolognesi, D. Latella (Eds.), Proceedings of the Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification FORTE/PSTV'2000 (Pisa, Italy), IFIP, Kluwer Academic Publishers, 2000, pp. 267–283.

[45] M. Yoeli, A. Ginzburg, LOTOS/CADP-based verification of asynchronous circuits, Technical Report TR CS-2001-09, Technion, Computer Science Department, Haifa, Israel (Mar. 2001).

[46] M. Yoeli, R. Kol, Verification of Systems and Circuits Using LOTOS, Petri Nets, and CCS, Parallel and Distributed Computing, Wiley, 2008.

[47] X. Wang, M. Z. Kwiatkowska, On process-algebraic verification of asynchronous circuits, in: Proceedings of the 6th International Conference on Application of Concurrency to System Design ACSD'06 (Turku, Finland), IEEE Computer Society Press, 2006, pp. 37–46.

[48] M. B. Josephs, Gate-level modelling and verification of asynchronous circuits using CSPm and FDR, in: Proceedings of the 13th IEEE International Symposium on Asynchronous Circuits and Systems ASYNC 2007 (Berkeley, California, USA), IEEE Computer Society Press, 2007, pp. 83–94.

[49] A. Bailey, G. A. McCaskill, G. J. Milne, An exercise in the automatic verification of asynchronous designs, Formal Methods in System Design 4 (3) (1994) 213–242.

[50] A. Cerone, G. J. Milne, Property verification of asynchronous systems, Innovations in Systems and Software Engineering 1 (1) (2005) 25–40.