Formal Verification of CHP Specifications with CADP Illustration on an Asynchronous Network-on-Chip

Gwen Salaün, Wendelin Serwe
INRIA Rhône-Alpes / VASY, Montbonnot, France
Email: Wendelin.Serwe@inria.fr

Yvain Thonnart, Pascal Vivet
CEA/Leti - MINATEC, Grenoble, France
Email: Yvain.Thonnart@cea.fr
Pascal.Vivet@cea.fr

Abstract

Few formal verification techniques are currently available for asynchronous designs. In this paper, we describe a new approach for the formal verification of asynchronous architectures described in the high-level language CHP*, by using model checking techniques provided by the* CADP *toolbox. Our proposal is based on an automatic translation from* CHP *into* LOTOS*, the process algebra used in* CADP*. A translator has been implemented, which handles full* CHP *including the specific probe operator.*

The CADP *toolbox capabilities allow the designer to verify properties such as deadlock-freedom or protocol correctness on substantial systems. Our approach has been successfully applied to formally verify two complex designs. In this paper, we illustrate our technique on an asynchronous Network-on-Chip architecture. Its formal verification highlights the need to carefully design systems exhibiting non-deterministic behavior.*

1. Introduction

Process calculi are acknowledged as appropriate formalisms to describe concurrent systems. For the description of asynchronous hardware, several VLSI programming languages based on process calculi have been proposed, as for instance CHP (*Communicating Hardware Processes*) [22], DI-Algebra [30], HASTE [26] (formerly TANGRAM [18]), or BALSA [9]. These languages allow the description of concurrent processes communicating via handshake synchronizations, and are supported by synthesis tools that can generate the implementation of a design from its process algebraic description. For instance, the TAST synthesis tool [28] can generate QDI (Quasi Delay Insensitive) netlists from CHP descriptions and is being used to elaborate complex designs. However, for CHP specifications, few formal verification means are available, although they are essential in an asynchronous context to reduce the risk of errors in the design.

Our goal is the verification of asynchronous hardware designs specified in CHP using existing model checking tools that are based on the exploration of LTSs (Labelled Transition Systems), also known as finite state machines. Examples of properties to be checked are the absence of deadlocks or the correct sequencing of communications in a protocol (some stimuli are always followed by the appropriate responses). In this work, we used the model checking tools available in the CADP toolbox [12]. CADP takes as input LOTOS [15] specifications, and provides an efficient tool for exploring the state space of a LOTOS description.

Our approach is based on the translation from CHP to LOTOS. Such a high-level translation has been chosen since CHP and LOTOS are both inspired by the process algebra CSP [14], i.e., they are based on the same kernel of behavioral operators, namely sequential composition, choice, and parallel composition. This translation is not trivial though because CHP has some hardware related specificities that do not exist in LOTOS. In particular, communication in CHP is not necessarily atomic (as it is in standard process calculi): the *probe* operator [21] of CHP allows to check whether the communication partner is ready for a communication or not, but without performing the communication actually.

A translator tool has been implemented and was used successfully for the verification of two industrial case studies designed at the CEA/Leti laboratory, namely an asynchronous implementation of the DES (*Data Encryption Standard*) [24] and ANOC, an asynchronous Network-on-Chip architecture [1]. This paper focuses on ANOC, the input controller of which is used for illustration purposes. We show how informal properties to be ensured are formalized and checked using the verification means available in CADP.

The organization of the paper is as follows. Section 2 overviews our translation from CHP into LOTOS. Section 3 shows how verification of hardware architectures is achieved using CADP tools. In Section 4, we present the

ANOC architecture, and we illustrate in Section 5 the application of our approach on a part of ANOC. Section 6 compares our proposal with related work. Section 7 ends the paper with some concluding remarks.

2. Translation of CHP into LOTOS

Translating CHP into LOTOS is quite straightforward for most CHP operators, since both languages are based on the process algebra CSP (see Appendices A and B for a brief summary of both languages). Thus, communication in both directions (emissions and receptions), guarded behavior, sequential and parallel composition, and nondeterministic choice exist in both languages. There are, however, some noticeable differences. For instance, CHP types (Booleans, natural numbers, bit vectors, etc.) are translated into LOTOS abstract data types. Similarly, CHP operations are translated into LOTOS operations, which are implemented either using algebraic equations or, more efficiently, by external C code, using the possibilities offered by CADP. CHP loops are translated into recursive processes in LOTOS because LOTOS does not have a similar loop construct.

A major difference between CHP and LOTOS is the hardware-specific probe operation of CHP, which has no equivalent in LOTOS. A previous version [29] of the translation focused on CHP behaviors using the probe only as a single operator in guards. To deal with the full specification of complex designs such as ANOC, we have extended our translation to handle probes in general expressions and to support complex datatypes and code specialization.

2.1. Translation of the Probe

A probe "$c\#$" (activity probe) allows a passive process (either emitter or receiver) to check if its active partner has already initiated a communication on channel c without performing the communication actually. The notation "$c\#V$" (value probe), which can only be used by a receiver, checks if the sender has initiated the emission of the particular value V. The probe operator enables a form of shared memory communication, and does not exist in classical process calculi based solely on message passing. Hence, its translation requires the generation of additional communications representing the access to the active channels to be probed.

To alleviate the overhead due to the general translation of probe operations, our approach uses code specialization, i.e., the translation of a communication channel c depends whether and how c is probed.

- *unprobed channel*: each CHP emission/reception is translated directly to its LOTOS equivalent.
- *general case*: the evaluation of an expression containing probes requires information on the state (communication pending or not) of the probed channels. The shared resource corresponding to a channel c is translated by an additional LOTOS process $channel\_c$ with two states, representing whether a communication has been initiated but not yet acknowledged; a probe operation allows to check the state of $channel\_c$.
- *single probe*: if a channel c is probed in a guard (containing no other probe operation), the additional LOTOS process can be avoided. In this case, the probe (passive side) is translated as a communication with an additional flag "`!Probe`" to distinguish it from a regular communication. The guarded statement can be executed as soon as the probe communication is possible. The communication on the active side is translated in a loop: both processes can interact on the probe communication[1] as often as necessary until the effective communication between them takes place. This translation is summarized in Figure 1 and detailed in [29].

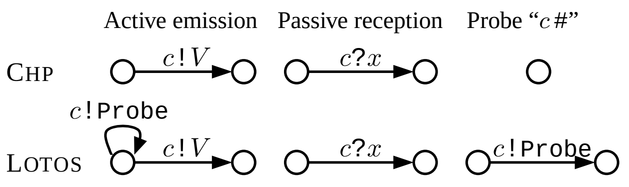

Figure 1. Single probe translation

In case of a probed channel, the translation scheme simulates a two-phase protocol with the effective communication corresponding to the acknowledgement. Otherwise (unprobed channel), such a protocol is avoided to reduce the corresponding state space. The examples in the remainder of this paper use only the unprobed and single probe case.

2.2. Tool Implementation

A translator from CHP to LOTOS has been developed using the SYNTAX and LOTOS NT compiler construction technologies [11]. It consists of 2,200 lines of SYNTAX, 13,400 lines of LOTOS NT, and 3,900 lines of C. This tool has been validated on more than 500 CHP specifications, corresponding to about 14,500 lines of CHP. We emphasize that this tool allows a completely automated translation for full CHP including probes and hierarchical components.

As regards semantics preservation, the translation of the probe operation requires the generation of additional hidden communications leading to internal transitions labelled by τ; thus it is impossible to preserve strong equivalence [23],

[1]The value-matching feature of LOTOS ensures that two offers "`!Probe`" will synchronize.

which handles τ transitions similarly to visible transitions, but only weak equivalences that allow additional τ transitions. Our translation preserves observational equivalence [23], as checked in practice on many examples.

3. Architecture Verification with CADP

The verification technique implemented in the CADP toolbox [12] is called *enumerative* since it is based on the explicit enumeration of states. To verify properties of large state spaces, CADP incorporates different techniques, such as static analysis, distributed, and compositional verification, that help to circumvent the so-called state explosion problem.

In this section we focus on the use of CADP for the verification of asynchronous architectures. In a first step, an LTS that is smaller, but equivalent to the complete state space, is generated compositionally. This LTS is then used in a second step for the verification of the interesting properties.

3.1. Compositional State Space Generation

The underlying principle of compositional state space generation is "divide and conquer": instead of a single monolithic generation step, the state space of the complete system is obtained by alternating generation, minimization, and sub-system-composition steps. This approach is particularly efficient if the subsystems contain internal τ transitions (representing details not required for the composition with the rest of the system), some (or even most) of which might be removed during minimization modulo a weak equivalence abstracting from τ transitions.

The successful use of compositional state space generation requires some care, since the state space of some part of the overall system might be larger than the state space of the complete system. For instance, the state space of a simple buffer generated separately contains *all* possible values, whereas only *some* particular values might be buffered in the complete system. Thus, the state space of a specific part of the system should be generated only if the constraints imposed by its environment are also taken into account, so that only the relevant part of its state space is generated.

Furthermore, restricting the input values of the system allows to reduce the size of the generated state space. Different techniques can be used for this purpose.

- Using the idea of data independence, if some parts of the input values have no influence with respect to the interesting properties, they can be chosen constant, so as to abstract from differences in the data values which otherwise would multiply the size of the state space accordingly.

- A *specific environment*, i.e., additional CHP processes implementing the environment of the system guaranteeing that only realistic stimuli are provided, avoids the enumeration of stimuli that will never occur in practice. Here and in the sequel, a stimulus is a communication with a passive port, i.e., either sending a message to a passive input port or requesting the emission of a message from a passive output port.

  A specific environment may also include additional ports on which messages are sent if an error has been detected.

- Exploiting symmetry arguments, the generation of a single state space taking into account all possible orderings of inputs can be replaced by the generation of several smaller state spaces corresponding to *scenarios* representing particular subsets (e.g., sequences) of the inputs. These scenarios should be expressed in CHP and integrated in the specific environment.

The compositional generation of the state space is facilitated by the use of SVL (Script Verification Language) [10], which allows a high-level and concise description of the calls to the different CADP tools. The order in which the different processes are combined has to be determined manually following the data path of the stimuli, requiring knowledge about the architecture. Since this order does not depend on the scenario, a single SVL script can be used for the compositional generation of the LTSs corresponding to several scenarios.

3.2. Property Verification

In the following, we describe the techniques for the verification of three different kinds of properties relevant in the validation of asynchronous hardware architectures.

3.2.1. Deadlock Freedom. Given the complete state space of a system, it is straightforward to check the absence of deadlocks by counting the number of successors of every state. In CADP, this property is printed as one of the standard characteristics of an LTS.

Notice that most of the remaining properties require a cyclic behavior. Thus their verification would fail if the system might deadlock.

3.2.2. Correct Stimulus-Response Protocol. Code inspection allows to verify that a single basic process of an asynchronous design conforms to the basic protocol, i.e., executes a cyclic behavior producing from a set of stimuli $\{S_1, \ldots, S_m\}$ a set of responses $\{R_1, \ldots, R_n\}$. To verify this property for more complex components, containing several subcomponents and processes, the following three-step approach relying on the equivalence checking tools of CADP can be used.

1. To check the cyclic occurrence of the whole set of stimuli, all transitions but the stimuli S_i are hidden (i.e., transformed to τ transitions) in the generated LTS, and the resulting LTS is then compared to an LTS describing the cyclic arrival of stimuli in any order; this second LTS can be given explicitly, or be generated from a LOTOS or CHP description, namely the CHP process "`[`$(S_1, \ldots, S_m)$`; loop]`".

2. To check the cyclic occurrence of the whole set of responses, the same technique can be applied as for the stimuli.

3. To check that the set of stimuli generates the set of responses, it is now sufficient to choose a single stimulus S and a single response R and to verify that S generates R; the advantage is to avoid to consider all possible orderings of stimuli and responses. This step can be achieved by comparing with the LTS generated from the CHP description "`[`S`;`R`;loop]`".

Notice that it is not sufficient to compare the LTS — after hiding all transitions but the stimuli and responses — with a second LTS corresponding to the CHP process "`[`$(S_1$`, ... ,`$S_m)$`;`$(R_1$`, ... ,`$R_n)$`;loop]`", since this would require that the first response occurs necessarily after the last stimulus, which might not be the case if a response R_j depends only on a subset of the stimuli.

3.2.3. Functional Properties. General functional properties can be verified in two ways. A first approach is to enrich the specific environment with observer processes that monitor the stimuli and responses, and signal any detected error (e.g., faulty responses) by a particular `Error` transition or block the system as soon as an error is detected. In this case, the verification of the property is tantamount to checking the reachability of an `Error` transition or the absence of deadlock.

A second possibility is classical model checking. In this case, the property is expressed formally using a temporal logic, e.g., regular alternation free μ-calculus in CADP. Then, a model checker, EVALUATOR [4], is used to verify if the generated state space, possibly after hiding some transitions, satisfies the formula.

The temporal logic used in CADP allows two modalities:

- the formula "`[` S `]` φ" is true in a state s, if all transition sequences of the form S and starting in s lead to a state where φ is true.
- the formula "`<` S `>` φ" is true in a state s, if there exists a transition sequence of the form S, starting in s, and leading to a state where φ is true.

In both cases, the pattern of a transition sequence S in the modal operator can use regular expressions on predicates of transition labels. In particular, the predicate `true` is true for every transition; thus the regular expression "`true*`" describes a possibly empty sequence of transitions.

4. An Asynchronous Network-on-Chip

The verification approach described in Sections 2 and 3 is illustrated hereafter on the CHP specification of an actual asynchronous design, namely ANOC, an Asynchronous Network-on-Chip architecture [1], which was used as the backbone of FAUST, a 4th Generation Wireless Telecom Baseband [8].

This Section presents the asynchronous node of ANOC with a specific focus on its input controller, which will be used in Section 5 for illustration of our approach.

4.1. ANOC Outline

ANOC relies on the GALS (Globally-Asynchronous Locally-Synchronous) paradigm, in which synchronous resources communicate through a fully asynchronous network. Resources are connected to the network via a synchronous network interface providing all higher-level network services (packetization, flow-control, ...) and application services (synchronization, scheduling, ...), which connects to the switching fabric through a GALS interface, as described in [2].

The five-port asynchronous network node is the elementary component of ANOC. It provides the lower-level network services, i.e., routing and arbitration of the transiting packets. It is actually made of five input controllers, which route incoming packets to one of the other four ports, and five output controllers, which arbitrate between packets heading for the same output.

Although no assumption is made on the topology of the network as regards the architecture of the node, it was chosen in the FAUST implementation of ANOC to connect the nodes in a 2D-Mesh topology, as seen in Figure 2.

It is crucial for any application that the interconnect structure transfers data without corruption, and with a sufficient level of QoS (Quality-of-Service). At a system level, a 2D-Mesh topology presents several advantages (scalable bandwidth, easier place and route on silicon), but also enables theorem proving thanks to its regular structure. For instance, using the "odd-even turn model" routing algorithm [6], a routing of the communications in the network that avoids any deadly embrace can be computed.

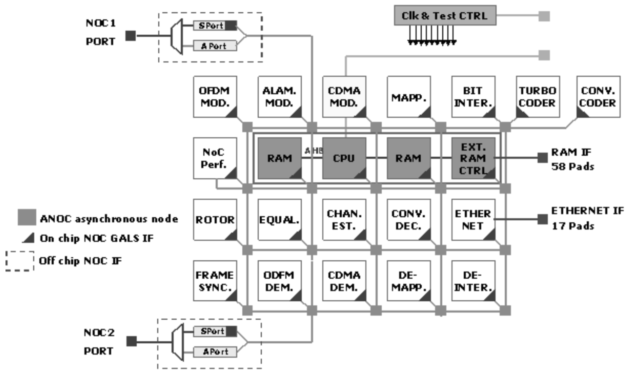

Figure 2. FAUST network architecture

Such analytical methods can often guarantee high-level properties, but always rely on an ideal operation of the asynchronous nodes. It is hence necessary to verify, at design level, that a node strictly complies to the ANOC protocol.

4.2. Functional Specifications: ANOC Protocol

Formal verification has to rely on rigorous functional specifications of the design. In the following, we describe the network protocol of ANOC that the asynchronous node should implement.

4.2.1. Data-Link Layer. The elementary piece of information transiting on the links of ANOC is called a *flit* (flow-control unit). In order to provide QoS in the network, ANOC was designed using two VCs (Virtual Channels) with different priority levels, used to transmit flits according to their precedence. A VC identifier is associated to every flit transiting in the network, and is propagated through a `Send` channel in the whole path, along with the actual `Data`.

The asynchronous node dispatches incoming flits inside its internal buffers according to this VC identifier, so that contention, i.e., the need for arbitration between two communications, may appear only between flits sharing the same VC. The arbitration between VCs was made static, so that VC0 always holds priority before VC1.

This mechanism actually creates two independent logical networks, one of which may be kept lightly loaded for low-latency signalling messages.

Additionally, a flit-based flow control mechanism had to be implemented to control the buffering capacity of each port on VC1. Indeed, the separation into VCs requires that the physical link between two nodes must be kept free for a new flit on VC0 after each flit on VC1. Since the link is shared by the two VCs, letting flits accumulate on VC1 more than the buffering capacity of the input controller would block flits of the other VC in the shared part. An `Accept1` channel was therefore introduced, indicating that a new flit can be buffered on VC1 in the input controller. This defines on VC1 a data-link protocol where a token on the `Accept1` channel is consumed by the emitter before each transmission on `Send` and `Data`, and is regenerated once the transmission is over. The protocol on VC0 is simpler, as it only requires simultaneous transmissions on `Send` and `Data`.

4.2.2. Network Layer. In order to minimize the overhead of routing information in every communication, the ANOC protocol is based on packet switching: several consecutive flits are gathered, using a single VC, into a packet, which will be transmitted, on this VC, without interruption. This requires an identification of the first and last flits of a packet: we introduced therefore two additional bits to the 32 bit data path, coding the *begin of packet* (*BOP*) and *end of packet* (*EOP*) informations.

Furthermore, in order not to handle complex routing tables, we used static source routing of the packets. Routing information (*path-to-target*) is provided in the first flit (*header*) of a packet, as a vector of dibits containing the successive directions to follow. Each node actually uses the two lower bits of this vector in the first flit to route the whole packet in the given direction, and shifts the *path-to-target* so that it can be used by the following node.

BOP	*EOP*	header flit payload		*path to target*	
33	32	31	18	17	0

Header flit

BOP	*EOP*	body flit payload	
33	32	31	0

Body flit

BOP	*EOP*	*flit id.*		fixed sequence		*dest.*	
33	32	31	30	29	2	1	0

Simplified flit for formal verification

Figure 3. Flit format

Figure 3 presents the structure of the actual flits transiting in the network, and how these flits can be constrained for the formal verification of the input controller of ANOC. Considering only the *VC identifier*, *BOP*, *EOP*, the *destination*, and a 2-bit *flit identifier*, we can reduce the potential state space of the input controller from about $5 * 10^{25}$ states to about $5 * 10^{16}$ states[2]. The fixed sequence used for the flit payload is made so that it reflects the data integrity, including the shift of the *path-to-target*.

[2]These numbers are obtained by multiplication of the numbers of states of the separate processes; due to synchronizations, the actual number might be slightly smaller.

4.3. CHP Design of the ANOC Input Controller

The asynchronous node was first developed using a coarse description level in SystemC/TLM, which was used to set up the various layers of the ANOC protocol at a system level [1]. Then, an equivalent functional CHP model of the node was written, and was later refined into elementary CHP processes that could be synthesized using a WHCB (Weak-Condition Half-Buffer [19]) circuit template, and mapped onto a specific standard cell library.

We chose to apply our formal verification technique on the refined CHP model of the input controller of the ANOC node, whose behavior is the closest to the actual implementation in the FAUST chip.

The internal micro-architecture of the input controller is given in Figure 4, and was described more precisely in [1]. In this figure, light grey processes are actually synthesized with combinational logic.

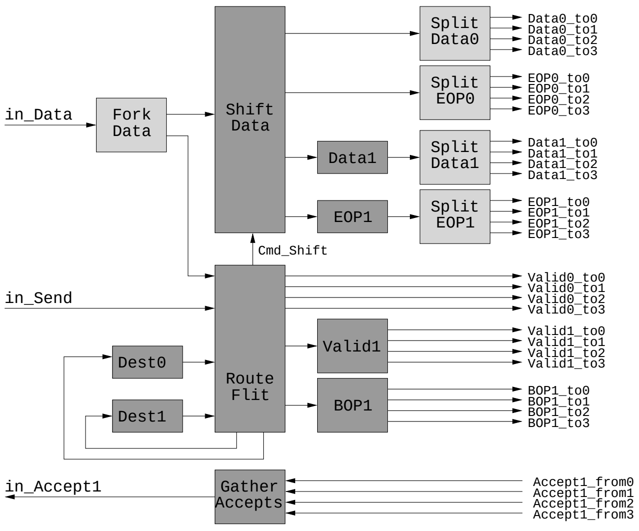

Figure 4. Input controller micro-architecture

The interface of the input controller complies, on the data-link side, to the ANOC protocol, and consists therefore of the signals `in_Data`, `in_Send` and `in_Accept1`. The internal control and data signals from an input controller to the output controllers are:

- `Data0` and `Data1`, buffered in the two VCs, that the input controller broadcasts to every output controller, only one of which will acknowledge the channel.
- `Valid`i`_to`d signals notifying the availability of a new flit on `Data`i to the output controller d.
- `BOP`i`_to`d signals, notifying the begin of a packet, necessary to trigger arbitration on the output controller.
- `EOP`i signals, notifying the end of a packet, broadcast in the same way as `Data`i, necessary to enable a new arbitration on the output controller.
- `Accept1_from`d signals from the output controller, indicating that a new flit can be sent on VC1.

A peculiarity in this design is the use of passive combinational splits at the output of the input controller, in order to reduce the gate count compared to a 34-bit passive-in / active-out commanded split. At implementation level, such a combinational split is indeed only a AND4 gate merging the “active low” acknowledgement signals, while the data wires are forked to every output controller.

This behavior was described in CHP using a passive output, which can be probed for communication requests by the output controllers. The actual notification of the output controllers is done by the `Route_Flit` process, using the `Valid`i`_to`d signals.

The CHP code of such a process is given below:

```
process Split_Data0 port (
    Data0     : in  DI passive MR[4][17];
    Data0_to0 : out DI passive MR[4][17];
    Data0_to1 : out DI passive MR[4][17];
    Data0_to2 : out DI passive MR[4][17];
    Data0_to3 : out DI passive MR[4][17])
variable x : MR[4][17];
begin
[
 @[
   Data0_to0# => Data0?x; Data0_to0!x; break
   Data0_to1# => Data0?x; Data0_to1!x; break
   Data0_to2# => Data0?x; Data0_to2!x; break
   Data0_to3# => Data0?x; Data0_to3!x; break
  ];loop
];
end;
```

5. Formal Verification of the Input Controller

The translation from CHP to LOTOS and compositional state space generation described in Sections 2 and 3 have been used to set up a formal model of the input controller of ANOC. Using this model, it is possible to check the functional properties describing its conformance to the ANOC protocol.

5.1. Translation of the CHP Design into LOTOS

First of all, for each considered scenario, the CHP design of the input controller (about 1,200 lines of code) is translated in less than one second into LOTOS (about 3,600 lines of code) using our translator tool. Let us show the LOTOS code obtained after translation of the CHP process

Split_Data0 shown above (find in Appendix B a summary of the LOTOS operators). The LOTOS process below is slightly simplified since we removed the explicit list of gates "[Data0_to$d$]" coming with every process call.

```
process Split_Data0: noexit :=
  Data0_to0 !Probe;
    Data0 ?data:MR4_17; Data0_to0 !data;
      Split_Data0
  []
  Data0_to1 !Probe;
    Data0 ?data:MR4_17; Data0_to1 !data;
      Split_Data0
  []
  Data0_to2 !Probe;
    Data0 ?data:MR4_17; Data0_to2 !data;
      Split_Data0
  []
  Data0_to3 !Probe;
    Data0 ?data:MR4_17; Data0_to3 !data;
      Split_Data0
endproc
```

The CHP process contains simple probes in its guards that are translated as communications in LOTOS with the specific offer "!Probe" to distinguish them from regular communications on the same channels. A branch of the choice structure in the LOTOS process can be executed if and only if an active process is ready to communicate on this particular channel with the process at hand.

5.2. State Space Generation

In order to provide realistic stimuli to the input controller of ANOC, several scenarios were used with packets of different length (one or more flits), emitted sequentially or overlapping on different channels and to different destinations. The composition order of the different processes used to build the state space of the input controller has been determined following the data path of the input values according to the architectural schema presented in Section 4.3.

The description of the 41 steps to generate the LTS corresponding to the input controller of ANOC for a particular cycle of four flits requires 483 lines of SVL. The script is generic in the sense that it can be used to generate the state space for all stimuli using only simplified flits as described in Figure 3. In the remainder of this section, we consider the scenario based on the sequence of four packets (each containing exactly one flit) sent on virtual channel VC0 alternatively to destination 0 and to destination 1. For this scenario, the SVL script generates, in about four minutes, the corresponding LTS with 1,300 states, 3,116 transitions, and 34 labels without any hidden internal transition. The largest intermediate LTS observed during the generation has 295,893 states and 812,283 transitions. Similar values are observed when the SVL script is used for the generation of LTS corresponding to different cycles of four flits.

5.3. Verification of the Functional Specifications

Even though the micro-architecture of the input controller was used during the state space generation, the properties to which it should conform were defined as if it were a blackbox. Thus, we enunciated functional properties, which describe the protocols at the interfaces, not biased by any structural hypothesis.

Similarly to the compositional state space generation, the verification of the properties uses a SVL script (of about 250 lines) which facilitates the calls to the different CADP tools, required for hiding and renaming of transitions, as well as equivalence and model checking. A further advantage of the script is the possibility to automate the verification of different scenarios. For a scenario with cyclic sequences of four flits, the execution of the script checking all the properties presented below takes about three minutes.

5.3.1. Infinite Occurrence. The input controller may not reach a state where a communication may not occur anymore on an asynchronous channel. For every signal S, it should have a "quasi-cyclic" behavior in a closed system.

These properties were checked with the CADP toolbox as explained in Section 3.2. Every state of the state space has a successor, which means there is no deadlock state. Furthermore, for a signal S, the "quasi-cyclic" behavior is verified using the observational equivalence-checking technique to compare the LTS of the input controller with the LTS of a CHP process "[S!;loop]", after hiding every other signal in the input controller.

5.3.2. Protocol Correctness. The input controller must comply at its inputs with the ANOC protocol, and transmit the incoming data to an output controller. On VC0, this means that an incoming flit will generate an emission to an output controller.

This was proven formally using a stimulus-response pattern, as described in Section 3.2: the two stimuli in_Data and in_Send together must lead to the three responses Valid$i$_to$d$, Data$i$, and EOPi, where the values of i — the virtual channel (0 or 1) — and d — the *destination* (0, 1, 2, or 3) — are obtained by inspection of the values communicated on in_Data.

On VC1, an accept token must be used and regenerated accordingly, but this is actually done in the output controller. Environment processes simulating the output controllers were described in the specific environment of the input controller, which perform this task. The property describing this behaviour actually ensured correctness of the combination of these additional processes to the input controller. Decorrelation of the accept process from the forward data flow of the input controller can be seen in Figure 4 of Section 4.3: the architecture of the input controller is divided in two unconnected parts.

5.3.3. Data Integrity. The content of the communications must be preserved by the input controller. Body flits should be transmitted without alteration, while header flits should be transmitted with their *path-to-target* field shifted, preserving the *BOP*, *EOP*, and *header payload* fields.

This functional property has been verified by integrating observer processes in the specific environment: the environment processes simulating the output controllers have been enriched in order to compare the data from the input controller with the expected result, determined according to the flit provided as a stimulus to the input controller.

5.3.4. Packet Routing. The input controller has to route all the flits of a packet in the right direction. A header flit will be routed to the output controller specified by its destination field, while a body flit will use the same direction as the previous header flit.

The following formula expresses that after every sequence of transitions such that after an unspecified start a transition labelled with a request to emit a priority packet (predicate "`on_channel(0)`") is followed by a transition labelled with a request to emit a data packet to destination `1` (predicate "`to_dest(1)`"), the next communication on one of the channels `Data0_to`d (predicate "`no_Data0_toD()`" is true for all other transitions) is necessarily on channel `Data0_to1`:

```
[true* . on_channel(0) . to_dest(1)]
<(no_Data0_toD())* . 'Data0_to1'>
true
```

Unfortunately, this formula does not hold for the LTS (after hiding all transitions on channels different from `in_Send`, `in_Data`, and `Data0_to`d). The counter-example provided by EVALUATOR [4], the model checker of CADP, shows the possibility that, contrary to the design assumptions, two flits for the virtual channel `0` can be present simultaneously in the input controller, which can lead to a routing error, i.e., a flit is transmitted to a wrong output.

This is confirmed by the following formula which expresses that in all transition sequences, after a communication on one of the channels "`Valid0_to`d" there has to be a communication on one of the channels "`Data0_to`d" before the next communication on one of the channels "`Valid0_to`d", which does not hold either:

```
[true* . 'Valid0_to.' .
 (not 'Data0_to.*')* . 'Valid0_to.']
false
```

This routing problem has also been found by simulating the CHP description of the input controller using the TAST simulator for about 500.000 steps. The actual issue with simulation-based validation is to decide when the verification was thorough enough to claim a correct behavior.

The encountered problem lies in the use of the passive split that simulates broadcasting to every output controller. EVALUATOR exhibits the following sequence of communications (unrelated transitions are hidden):

`in_Send!0;`	*In* $\longrightarrow$ *Route Flit*
`in_Data!flit0;`	*In* $\longrightarrow$ *Shift Data*
`Valid0_to0!;`	*Route Flit* $\longrightarrow$ *VC0 Out 0*
`Cmd_Shift!bop0;`	*Route Flit* $\longrightarrow$ *Shift Data*
`in_Send!0;`	*In* $\longrightarrow$ *Route Flit*
`in_Data!flit1;`	*In* $\longrightarrow$ *Shift Data*
`Valid0_to1!;`	*Route Flit* $\longrightarrow$ *VC0 Out 1*
`Data0_to1#true;`	*VC0 Out 1* $\longrightarrow$ *Split Data0*
`Data0!flit0;`	*Shift Data* $\longrightarrow$ *Split Data0*
`Data0_to1!flit0;`	*Split Data0* $\longrightarrow$ *VC0 Out 1*
`ERROR_1!;`	*VC0 Out 1* $\longrightarrow$

In this sequence, `flit0` is a header flit to output 0, and `flit1` is a header flit to output 1.

The issue is that the active signals `Valid0_to`d are acknowledged by the outputs before the probe is actually done in *Split Data0*. Once *Route Flit* has sent both its outputs `Valid0_to0` and `Cmd_Shift`, a new incoming flit can trigger `Valid0_to1` immediately afterwards. Both output controllers are therefore notified and may request the data, creating a non-exclusive guard in *Split Data0*, leading to an unwanted non-deterministic choice.

Nevertheless, this issue does not occur in the actual implementation of the node[3]. It is indeed an issue of interpretation of the CHP specification.

The circuit was synthesized using a WCHB [19] circuit template. The *slack* (maximal number of tokens a single process can accumulate) of each WCHB process is only 1/2, due to the four-Phase reshuffling. Besides, the *Split Data0* combinational process has a null slack. Hence, in the synthesized version, the `Cmd_Shift` token at the output of *Route Flit* is not freed before `Data0` was read, which happens only after *Split Data0* was notified. This guarantees that no other token can be emitted to `Valid0_to`x, preventing the routing error. Yet, this property holds only if the slack between *Route Flit* and *Split Data0* is less than 1. The system is said not to be *slack-elastic* [20].

However, on current implementations of the simulation and verification tools, communications on CHP channels are done atomically between two semicolons, and do not describe the acknowledgement of the data. This corresponds to a Full-Buffer (slack of 1) description of the asynchronous processes, except for *Split Data0*, thanks to the probe on

[3]This was actually checked in a SDF (Standard Delay File) back-annotated Verilog simulation of the node.

output mechanism.

The slack increase in the CHP description of this non slack-elastic process allows an unwanted behavior, namely the routing of a flit to a wrong output.

To investigate this further, we modified the CHP description so as to reduce the slack of the processes by doubling the communications on each channel, which mimics a four-phase protocol. A first communication describes the raising phase on the signal (falling on the acknowledge), while the second describes the falling phase on the signal (rising on the acknowledge), the guards being evaluated according to the first communication. With each process modified following this principle, the LTS contains no error transition: the routing properties hold.

As for VC1, the routing correctness was already verified on the original CHP description. Thanks to the `Accept1` flow-control mechanism, a single flit is allowed in the input controller. Even though an additional buffering stage is present on VC1, the overall slack is constrained to 1, and no interference can occur.

6. Related Work

As regards the application of model checking tools to asynchronous hardware designs in general, most approaches, e.g., [25, 7, 13, 16, 27, 31, 32], are based on a manual modelling of the design directly in the input language of the used verification tool, whereas we base the verification on design descriptions written by hardware designers — the same descriptions that are also used for synthesizing the circuits. As an example, [31] introduces a model for the low-level description of asynchronous circuits on the hardware gate level, and presents the verification of safety and progress properties using FDR2 (via a manual translation to CSP). On the contrary, our approach focuses on the verification of high-level descriptions, from which concrete implementations in hardware can be derived automatically, for instance using TAST.

As regards verification of process algebraic descriptions of asynchronous designs, we are only aware of [5] and [17]. [5] proposes a translation of CHP into networks of communicating automata. Contrary to our approach, [5] flattens CHP processes with intertwined sequential and parallel compositions, which is less efficient than our translation from CHP into LOTOS; we also observed reductions of the LTS generation time by factors up to four [29]. [17] presents techniques for equivalence and refinement checking of DI-Algebra descriptions. Our approach allows to verify a larger class of properties, i.e., alternation-free modal μ-calculus formulas.

7. Conclusion

In this paper, we have presented an approach for the verification of process algebraic descriptions of asynchronous architectures using the verification toolbox CADP originally designed for the verification of telecommunication protocols. Our approach is supported by a tool, capable of automatically translating full CHP into the international standard LOTOS. We validated our approach on two asynchronous designs, namely an implementation of the Data Encryption Standard [24] and ANOC, the input controller of which was used in this paper as an illustrating example.

The node of ANOC required a careful CHP design using non *slack-elastic* processes. Some of the node properties were thus verified while manually taking the handshake expansion into account. In order to ease this verification task, we suggest to enrich the high-level CHP language with pragmas indicating the chosen handshake expansion to the simulation, synthesis and verification tools.

As regards future work, we are currently using our approach to ensure the correctness, before synthesis, of architectures designed at the CEA/Leti laboratory, in particular the next implementation of ANOC. In this context, we rely on the possibility to use our approach hierarchically, by isolating subsystems that are verified separately and seen as blackboxes when verifying properties of the global system.

Acknowledgements

This collaborative work has been supported by the common laboratory of INRIA Rhône-Alpes and CEA/Leti as well as the Minalogic EmSoC MULTIVAL project.

The authors would like to thank their colleagues from the TIMA CIS group for providing the TAST CHP syntax, and for valuable discussions on the probe semantics.

References

[1] E. Beigné, F. Clermidy, P. Vivet, A. Clouard, and M. Renaudin. An asynchronous NoC architecture providing low latency service and its multi-level design framework. In *Proc. of ASYNC 2005*, pp. 54–63.

[2] E. Beigné and P. Vivet. Design of off-chip and on-chip interfaces for a GALS NoC architecture. In *Proc. of ASYNC 2006*, pp. 172–181.

[3] T. Bolognesi and E. Brinksma. Introduction to the ISO specification language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, Jan. 1988.

[4] R. Mateescu and M. Sighireanu. Efficient on-the-fly model-checking for regular alternation-free mu-calculus. *Science of Computer Programming*, 46(3):255–281, Mar. 2003.

[5] D. Borrione, M. Boubekeur, L. Mounier, M. Renaudin, and A. Sirianni. Validation of asynchronous circuit specifications using IF/CADP. In *Proc. of VLSI-SoC 2003*, pp. 86–91.

[6] G.-M. Chiu. The odd-even turn model for adaptive routing. *IEEE Trans. on Parallel and Distributed Systems*, 11(7):729–738, July 2000.

[7] G. Clark and G. Taylor. The verification of asynchronous circuits using CCS. Technical Report ECS-LFCS-97-369, University of Edinburgh, Oct. 1997.

[8] Y. Durand, C. Bernard, and D. Lattard. FAUST : On-chip distributed architecture for a 4G baseband modem SoC. In *Proc. of Design and Reuse IP-SOC 2005*, pp. 51–55.

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

[10] H. Garavel and F. Lang. SVL: a scripting language for compositional verification. In *Proc. of FORTE 2001*, pp. 377–392.

[11] H. Garavel, F. Lang, and R. Mateescu. Compiler construction using LOTOS NT. In *Proc. of CC 2002*, pp. 9–13.

[12] H. Garavel, F. Lang, and R. Mateescu. An overview of CADP 2001. *EASST Newsletter*, 4:13–24, Aug. 2002.

[13] J. He and K. J. Turner. Verifying and testing asynchronous circuits using LOTOS. In *Proc. of FORTE/PSTV 2000*, pp. 267–283.

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

[15] 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, Sept. 1989.

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

[17] H. K. Kapoor, M. B. Josephs, and D. P. Furey. Verification and implementation of delay-insensitive processes in restrictive environments. *Fundamenta Informaticæ*, 70(1–2):21–48, 2006.

[18] J. L. W. Kessels and A. M. G. Peeters. The Tangram framework (embedded tutorial): Asynchronous circuits for low power. In *Proc. of ASP-DAC 2001*, pp. 255–260.

[19] A. M. Lines. Pipelined asynchronous circuits. Master's thesis, California Institute of Technology, 1995 (rev 1998).

[20] R. Manohar and A. J. Martin. Slack elasticity in concurrent computing. In *Proc. of MPC 1998*, pp. 272–285.

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

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

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

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

[25] S. M. Nowick and D. L. Dill. Practicality of state-machine verification of speed-independent circuits. In *Proc. of ICCAD-89*, pp. 266–269. IEEE, 1989.

[26] A. Peeters and M. de Wit. *Haste Manual, Version 3.0*. Handshake Solutions, 2006.

[27] B. Rahardjo. SPIN as a hardware design tool. In *Proc. of SPIN 1995*.

[28] M. Renaudin. *TAST Compiler and TAST_CHP Language, Version 0.6*. TIMA Laboratory, CIS Group, 2005.

[29] G. Salaün and W. Serwe. Translating hardware process algebras into standard process algebras — illustration with CHP and LOTOS. In *Proc. of IFM 2005*, pp. 287–306.

[30] J. T. Udding. A formal model for defining and classifying delay-insensitive circuits. *Distributed Computing*, 1(4):197–204, 1986.

[31] X. Wang and M. Z. Kwiatkowska. On process-algebraic verification of asynchronous circuits. In *Proc. of ACSD 2006*, pp. 37–46.

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

A. Grammar of CHP

The behavior of a process B in CHP is described using assignments, communications, collateral and sequential compositions, and nondeterministic guarded commands (as implemented in the TAST tool [28]):

$B ::= \texttt{skip}$	*null action*
$\mid\ x\texttt{:=}V$	*assignment*
$\mid\ c\texttt{!}V$	*emission on channel c*
$\mid\ c\texttt{?}x$	*reception on channel c*
$\mid\ B_1\texttt{;}B_2$	*sequential composition*
$\mid\ B_1\texttt{,}B_2$	*collateral composition*
$\mid\ \texttt{[}V\texttt{->}B\texttt{;}\ T\texttt{]}$	*guarded command*
$\mid\ \texttt{@[}V_1\texttt{->}B_1\texttt{;}\ T_1\ \ldots\ V_n\texttt{->}B_n\texttt{;}\ T_n\texttt{]}$	*choice*
$T ::= \texttt{break} \mid \texttt{loop}$	*terminations*
$V ::= x \mid f(V_1, \ldots, V_n)$	*value expression*
$\mid\ c\texttt{\#}V \mid c\texttt{\#}$	*probe on passive channel*

B. Grammar of LOTOS

We only present in this appendix a very simplified grammar of the LOTOS notation, see [15, 3] for more details. The behavior of a process B in LOTOS is described using communications, sequence, guarded behavior, choice, parallel composition, and process call (to express a looping behavior):

$B ::= G\texttt{;}B$	*sequence*
$\mid\ \texttt{[}V\texttt{]->}B$	*guarded behavior*
$\mid\ B_1\texttt{[]}B_2$	*choice*
$\mid\ B_1\texttt{\|[}g_1, \ldots, g_n\texttt{]\|}B_2$	*parallel composition*
$\mid\ P\texttt{[}g_1, \ldots, g_n\texttt{]}$	*process call*
$G ::= \tau$	*internal action*
$\mid\ g\texttt{!}V$	*emission on gate g*
$\mid\ g\texttt{?}x : t$	*reception on gate g*
$V ::= x \mid f(V_1, \ldots, V_n)$	*value expression*

The parallel composition operator means that processes B_1 and B_2 synchronize on the gates $g_1, \ldots, g_n$. As LOTOS gates are untyped, the type of received values must be specified for receptions.