Model-checking Synthesizable SystemVerilog Descriptions of Asynchronous Circuits

Aymane Bouzafour[1], Marc Renaudin[1],
Hubert Garavel[2], Radu Mateescu[2], Wendelin Serwe[2]

[1]Tiempo Secure – SAS
Montbonnot-Saint-Martin, France
{First.Last}@tiempo-secure.com

[2]Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG
F-38000 Grenoble France
{First.Last}@inria.fr

***Abstract*—Asynchronous circuits have key advantages in terms of low energy consumption, robustness, and security. However, the absence of a global clock makes the design prone to deadlock, livelock, synchronization, and resource-sharing errors. Formal verification is thus essential for designing such circuits, but it is not widespread enough, as many hardware designers are not familiar with it and few verification tools can cope with asynchrony on complex designs. This paper suggests how an industrial design flow for asynchronous circuits, based upon the standard HDL SystemVerilog, can be supplemented with formal verification capabilities rooted in concurrency theory and model-checking technology. We demonstrate the practicality of our approach on an industrial asynchronous circuit (4000 lines of SystemVerilog) implementing a memory protection unit.**

Keywords—Asynchronous circuit, asynchronous design, asynchronous logic, computer-aided design, CADP, concurrency, formal verification, hardware design, LNT, memory protection unit, model checking, SystemVerilog.

I. INTRODUCTION

The synchronous design paradigm is currently ubiquitous in the electronic design industry. In a clocked circuit, the designer can assume a discrete notion of time, since the elements of the circuit evolve at periodic instants cadenced by a rhythmic clock. Although this approach has many advantages, it is undermined by the increasing complexity of VLSI systems and the ever-growing demand for more performance. Indeed, the worst-case timing assumptions induce, more often than not, timing overheads and energy overconsumption. Conversely, in an asynchronous circuit, the components evolve autonomously and operate on an "on-demand" basis. This indicates a potential for lower power dissipation, more harmonious electromagnetic emission, and better overall timing performance.

However, the elimination of the clock does not come without a price: in a clockless design, different orderings of independent events may lead to a different overall circuit behavior. Therefore, standard validation techniques based on simulation cannot provide sufficient coverage: a formal verification of all the possible combinations of signal transitions is required to guarantee correctness. Model checking is a standard technique for verifying properties of concurrent systems; it consists in exploring the reachable state space of a system by systematically examining all possible execution scenarios, and verifying on this state space a collection of desirable properties, either functional (i.e., correctness) or non-functional (i.e., performance, security, etc.), usually expressed as formulas of some temporal logic.

To describe asynchronous hardware, various dedicated VLSI programming languages have been defined, such as CHP [1], delay-insensitive algebra [2], Haste [3] (formerly Tangram [4]), and Balsa [5]. Although these languages are convenient for asynchronous circuits modeling and synthesis, their practical use in industry is as yet hampered by their problematical integration in mixed asynchronous-synchronous designs, and by their incompatibility with standard EDA tools for simulation.

Alternative approaches consist in selecting one of the mainstream HDLs (*Hardware Description Languages*) that are widespread for the design of synchronous circuits and supported by most of the commercial EDA tools, and adapting/extending this HDL to handle asynchronous circuits. Among Verilog, VHDL, SystemC, and SystemVerilog [6], the latter has been shown in [7] [8] to offer the best compromise between the design-abstraction level and the ability to describe concurrent processes communicating through channels.

This paper presents a comprehensive methodology adopted by Tiempo for the design of asynchronous circuits. These circuits are described using a subset of SystemVerilog (SV, for short) defined in [7] along with appropriate coding guidelines. Simulation, place-and-route, and static timing analysis are performed using standard commercial EDA tools. Synthesis is performed using a dedicated tool ACC developed by Tiempo. Formal verification is performed using the model-checking tools of the CADP toolbox [9] developed at Inria; this is achieved by translating SV descriptions into the input language LNT [10] of CADP and by expressing properties in the MCL temporal logic [11] supported by CADP.

The paper is structured as follows. Section II provides an overview of Tiempo's asynchronous design flow. Section III presents a translation scheme from the considered SV subset to LNT. Section IV gives key principles to guide a designer in the formal specification of system properties. Section V illustrates the proposed methodology on a real-world case study: an industrial asynchronous circuit implementing an MPU

(*Memory Protection Unit*). Finally, we conclude the paper by discussing related work (Section VI) and prospective endeavors (Section VII).

II. TIEMPO'S ASYNCHRONOUS DESIGN FLOW

Figure 1 describes Tiempo's industrial design flow for asynchronous circuits. This flow has three original features: modeling, synthesis and formal verification, which we present in turn.

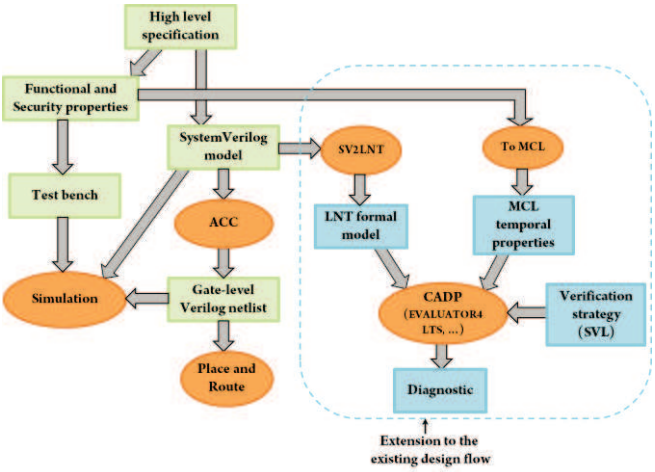

Figure 1: Tiempo's asynchronous design flow

A. Description language

In this design flow, asynchronous circuits are modeled using the standard SystemVerilog language [6] at the TLM (*Transaction Level Modeling*) level. More precisely, the designers are required to use only a subset of SV dedicated to asynchronous circuits. This is done by following a specific coding style [7] and by using predefined SystemVerilog packages.

This approach enables the designers to represent all the concepts that are necessary to model asynchronous circuits using SV processes ("**always**" blocks) and interfaces. This includes: parallelism ("**fork** ... **join**"), sequential composition ("**;**"), various communication and handshake mechanisms, as well as diverse memorization semantics. Similar concepts are also found in most of the existing formalisms for modelling asynchronous circuits (CHP, Balsa, etc.).

B. Synthesis

ACC (*Asynchronous Circuit Compiler*) is Tiempo's proprietary synthesis tool for asynchronous circuits. ACC is extensively used at Tiempo for product development and is also licensed to specific industrial partners. ACC takes as input descriptions written in the aforementioned SV subset and generates as output standard Verilog gate-level netlists, which can then be passed to commercial EDA tools for simulation, place-and-route, etc.

C. Formal verification

For a number of reasons discussed in Section VI, the existing commercial tools for verifying SystemVerilog descriptions are not appropriate for asynchronous circuits, the design paradigms of which significantly differ from those of mainstream (i.e., synchronous) circuits. Therefore, Tiempo's asynchronous design flow relies on the CADP toolbox [9] to perform formal verification. Although CADP was initially developed to analyze other kinds of concurrent systems (telecommunication protocols, distributed software, etc.), it has also found useful applications in hardware design, including, bus arbitration protocols, cache-coherent protocols, multiprocessor architectures, and asynchronous circuits.

As shown in Figure 1, connection with CADP requires designers to provide three inputs: (i) a formal model written in the LNT language, (ii) temporal properties expressed in the MCL language, and (iii) a verification strategy specified in the SVL language. We briefly introduce each in turn.

LNT [10] is the most recent language of CADP for the description of concurrent processes. LNT is a specification language based on the ideas of process calculi, such as CCS [12], CSP [13], and LOTOS [14]. Firmly grounded in the concepts of concurrency theory, LNT culminates a 25-year effort [15] to combine process calculi with language features borrowed from classical programming languages, in order to improve user friendliness and allow for a wide industrial dissemination. Tiempo's flow is based on the translation from the SV subset used for synthesis into an equivalent LNT model. This translation is further detailed in Section III.

The CADP tools are then applied to this LNT model so as to explore all reachable states of the asynchronous circuit. In the vocabulary of concurrency theory, the reachable state space is an LTS (*Labeled Transition System*), i.e., a finite, directed graph with a set of states (including an initial state) and a set of transitions describing the overall system changes from one state to another. The LTS model is *action-based*, meaning that the evolutions of the system can only be seen by observing the transitions, each of which corresponds to a communication event (namely, an input or output performed by the circuit, a handshake communication between two parts of the circuit, or even some "internal" action on which no precise information is available). The LTS model is not *state-based*, meaning that the internal contents of each state are not directly observable; such assumption fits well with asynchronous circuits: if one wants to observe a particular register, specific communications must be set to access this register, so that observable transitions corresponding to these communications will necessarily appear in the LTS.

The LTS provides a basis for evaluating all desirable properties that the asynchronous circuit should satisfy. These properties are expressed as temporal-logic formulas in the powerful language MCL [11] (further described in Section IV) and verified on the LTS using the EVALUATOR4 [11] model checker of CADP. In addition to a true/false verdict, this tool can generate, if requested by the user, a diagnostics explaining why a given MCL formula is true or false (e.g., an execution path starting from the initial state and leading to a state where the expected property is violated). Because observable information is attached to transitions rather than states, the LTS model makes it easy to examine the evolution of the system and trace problems back to the initial state.

In general, the LTS of a complex circuit is so large that it cannot be generated directly, i.e., using brute force only. The CADP toolbox implements many techniques that help alleviating this well-known *state-space explosion* problem. Some of these techniques are discussed in Section V below. In general, a successful verification scenario involves a number of steps (LTS generation, minimization, recombination, comparison, model checking, etc.). The CADP toolbox provides a high-level scripting language named SVL [16] that enables the designer to concisely specify how the many CADP tools should be orchestrated in order to perform the intended verification scenario.

III. MAPPING SYSTEMVERILOG TO LNT

A rigorous analysis of both SV and LNT languages enabled us to define comprehensive and systematic rules [17] for translating SV descriptions to LNT ones. Both languages are close, although not identical. To illustrate the similarity between SV and LNT, let us consider a simple asynchronous address decoder and its description in both languages.

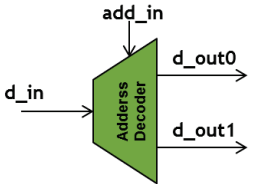

Figure 2: Simple address decoder

```
-- main SV module
module address_decoder (
  ch_bit.in add_in,
  ch_data_t.in d_in,
  ch_data_t.out d_out0,
  ch_data_t.out d_out1
);
  always begin
    bit address;
    data_t data;
    fork
      add_in.BeginRead(address);
      d_in.BeginRead(data);
    join
    case (address)
      1'b0: d_out0.Write(data);
      1'b1: d_out1.Write(data);
    end case
    fork
      add_in.EndRead();
      d_in.EndRead();
    join
  end
end module
```

```
-- main LNT process
process main[
  add_in : ch_bit,
  d_in,
  d_out0,
  d_out1 : ch_data_t]
is
  loop var
    address : bit,
    data : data_t in
      par
        add_in(?address)
      || d_in(?data)
      end par;
      case address in
        0 -> d_out0(data); d_out0
      | 1 -> d_out1(data); d_out1
      end case;
      par
        add_in
      || d_in
      end par
  end var end loop
end process
```

The correspondence between SV and LNT constructs is indeed manifest. The SV synthesizable subset can be entirely modeled in LNT [17]. In the subsequent sections, we explain the principles of our translation from SV to LNT.

A. Mapping of types, variables, and constants

Many SV basic types (integer, real, string, etc.) can be directly translated to LNT by leveraging the predefined data types in LNT. Other useful SV data types (bit and bit-array types) can be translated to LNT by creating equivalent LNT user-defined types built upon records, lists, sets, arrays, and enumerated types.

Each internal variable (declared within an SV process) is translated to an LNT variable with the same name. Each constant defined in the SV model is mapped to an LNT constant function returning the corresponding value.

B. Mapping of statements and behavioral operators

The SV conditional statements ("**case**" and "**if-else**") are translated to their LNT counterparts ("**case** … **end case**" and "**if** … **end if**"). However, if a "**case**" SV statement is used to express a non-deterministic choice [7], it is translated to a "**select** … **end select**" LNT statement.

SV sequential composition is translated to its counterpart LNT operator "**;**". The "**fork** … **join**" SV construct stipulates that the parent procedure is blocked until all the forked procedures have completed. This behavior is one-to-one equivalent to the LNT parallel composition operator "**par** … **end par**". Other SV composition operators such as "**fork** …. **join_none**" or "**fork** … **join_any**" are beyond the scope of this mapping, since these operators are not used to write synthesizable descriptions and their practicality is restricted to verification and testing procedures.

C. Mapping of communication channels

The asynchronous SV communication channels [7] are interface-like constructs allowing point-to-point communication between asynchronous design entities and processes. For each SV channel, a corresponding LNT channel with the same name is defined, which specifies the types of the data values exchanged through the SV channel. The communication protocol used in SV system descriptions is equivalent to a two-phase handshake protocol, and thus requires acknowledgements, which are modeled in LNT as pure (i.e., data-less) synchronizations; therefore, the corresponding LNT channels must include the empty-tuple type to support pure synchronization as well.

Another SV communication pattern is the *probe* operation [7] that allows a passive SV process to check the readiness of a channel (i.e., whether an active emitter process has initiated a communication on this channel) without performing the communication actually. The passive process can probe the communication channel as often as desired before performing a read operation and completing the communication. This is expressed in LNT using the nondeterministic choice operator "**select** … **end select**".

As an example, let C be an SV channel enabling communication between an active process P1 and a passive process P2 that, in a first scenario, probes C twice before performing a read operation and, in a second scenario, performs the exchange without probing. The following LNT code describes the exchange:

```
-- Active process P1
loop L in
  select
    C   -- probe
  []
    break L
  end select
end loop;
C(V); -- value emission
C     -- acknowledgement
```

```
-- Passive process P2
-- 1st scenario
C;        -- first probe
C;        -- second probe
C(?V);    -- value reception
C;        -- acknowledgement

-- 2nd scenario
C(?V);    -- value reception
C         -- acknowledgement
```

The LNT model of P1 allows the passive process P2 to probe C as often as desired. More often than not, the passive process will probe a channel only once before receiving the data value; hence, the use of the "**loop**" in the LNT model is rarely encountered. A formal semantics of the probe operator and its (possibly optimized) translation to a standard process calculus (such as LOTOS) that does not have a built-in concept of probe can be found in [18].

D. Mapping of processes and modules

Each individual SV process is translated into an LNT process with the same name; the communication channels of each individual SV process are identified and declared as formal gate parameters of the corresponding LNT process. The behavior of the process is translated as defined above. The global SV module regrouping all the individual SV processes is translated to a global LNT process, in which all these individual processes are composed in parallel using the LNT operator "**par** … **end par**". The ports of the global SV module and its interface are translated to formal gate parameters of this global LNT process. Finally, internal SV communication channels are declared as local LNT gates using the LNT operator "**hide** … **end hide**".

IV. EXPRESSION OF ASYNCHRONOUS PROPERTIES IN MCL

The properties to be verified are expressed in MCL (*Model Checking Language*) [11], an extended modal μ-calculus designed for specifying temporal properties of concurrent value-passing systems. MCL subsumes CTL, LTL, CTL*, and generalized Büchi automata, therefore being sufficiently powerful to capture all the properties expressible in SVA [6], the assertional language of SV. Note that basic (not temporal) SVA assertions can be directly encoded as assertions inserted in the LNT code.

It is worth reminding the classification of properties into *safety* and *liveness* properties [19]. Intuitively, safety properties specify that "something bad never happens", whereas liveness properties specify that "something good eventually happens", i.e., a desirable event (or sequence of events) will occur. Any property is either a safety property, a liveness property, or a combination of both.

We now present four property patterns frequently encountered in the verification of asynchronous hardware, and explain how such patterns can be expressed in MCL. In practice, MCL is a rather straightforward language, provided the designer is acquainted with the semantics of the main MCL operators.

A. Deadlock freedom

A characteristic feature of asynchronous circuits is that they are susceptible to deadlocks, but a correct asynchronous system should be free of deadlocks for all possible combinations of transitions. The absence of deadlocks in an LTS is expressed in MCL by imposing that every reachable state has at least one successor transition, whatever the label of this transition:

[true*] < true > true

B. Livelock freedom

Even if it has no deadlocks, an asynchronous circuit might get stuck into a livelock, i.e., an infinite loop of internal ("non-progress") actions that prevent any "progress" action (namely, delivering outputs to the environment or other concurrent processes) to occur any more beyond a certain execution point. Therefore, another desirable property of asynchronous circuits is livelock freedom. The presence of livelocks can be characterized, using the fairness operator "**@**" of MCL, as the existence of an infinite sequence made by concatenating subsequences satisfying some regular formula R (the simplest case R = "**i**" corresponding to a livelock consisting of internal actions only):

< true* > < R > @

C. Mutual exclusion

Various properties frequently encountered in the validation of asynchronous circuits relate to mutual exclusion between concurrent accesses. Indeed, the inherent concurrency of the asynchronous circuits makes these circuits vulnerable to failures arising from undesirable concurrent accesses to memories, peripherals, etc. Concurrency is represented in an LTS by considering all the possible interleaving sequences of parallel actions. The two-phase communication protocol provides a convenient basis for characterizing forbidden concurrent accesses. For instance, given two mutually exclusive actions A1 and A2, A1 (resp. A2) must not occur between A2 (resp. A1) and its acknowledgement $A2_{ack}$ (resp. $A1_{ack}$). This is expressed in MCL as the conjunction of two safety properties, each given as a regular expression (the use of regular expressions being a distinctive feature of MCL):

[true*. A2 **. (not** $A2_{ack}$**)* .** A1 **. (not** $A2_{ack}$**)* .** $A2_{ack}$ **] false and**

[true*. A1 **. (not** $A1_{ack}$**)* .** A2 **. (not** $A1_{ack}$**)* .** $A1_{ack}$ **] false**

D. Correct input-output behavior

Another useful property is that, for a given sequence of stimuli (I1, ..., In), the asynchronous circuit should eventually produce a definite sequence of responses (O1, ... , On). Notice that if the stimuli (resp. responses) occur concurrently, then the MCL property should take into account all possible interleaving sequences.

To illustrate this property, let us consider the asynchronous address decoder described in Section III. Our goal is to express in MCL the following liveness property: if the bit value offer received on the channel "add_in" is a '0' and if a data input "data0" is received on the "d_in" channel, then a data output "data0" is inevitably emitted on the channel "d_out0".

One first needs to express the interleaving of inputs, i.e., that the two receipt events on the "d_in" and "add_in" channels are allowed to occur in any order. Again, this can be done using a regular expression:

```
("add_in !0"."d_in !data0") | ("d_in !data0"."add_in !0")
```

One then needs to express the inevitability of the output response. To denote that an action A inevitably occurs, there is a standard pattern INEV (A), which can be defined as an MCL macro-definition using a minimal fixed point operator "**mu**":

```
macro INEV (A) =
        mu X . ((< A > true or < not A > X) and [ not A ] X)
```

Using this macro, the property stating that each sequence of transitions matching a regular formula R is inevitably followed by an action A can be expressed as follows:

```
[ true* . R ] INEV (A)
```

Finally, the initial property is expressed as follows:

```
[ true*. (("add_in !0"."d_in !data0") | ("d_in !data0"."add_in !0")) ]
INEV ("d_out0 !data0")
```

V. CASE STUDY: MEMORY PROTECTION UNIT CIRCUIT

We applied Tiempo's asynchronous design flow to several complex components of asynchronous circuits, among which we selected an MPU (*Memory Protection Unit*) as the main case study for the present article. Two reasons motivate this choice. Firstly, the MPU block realistically reflects the asynchronous circuits designed at Tiempo and exhibits a considerable level of complexity (state-space wise).

Secondly, the SV model of the MPU block involves most of the SV constructs needed to model asynchronous circuits, as well as all the design patterns that typically appear in transaction-level models of asynchronous systems. The MPU block thus enabled us to devise design rules that help optimizing the formal modeling and verification efforts.

A. Architecture of the SystemVerilog model

The MPU implements an access control policy. Figure 3 shows the overall architecture of the block. It consists of four principal modules: the decoder, the MPU configuration registers (MPU_CFR), the multiplexer, and the demultiplexer.

The key purpose of the decoder is to decode the address inputs sent by the microcontroller in order to identify the targeted peripheral. Once the targeted peripheral is identified by the decoder, the current subject access rights are sent to the MPU_CFR, which evaluates the access rights of the user against the access configuration of the peripheral to be accessed, before granting or denying access. In the former case, control orders to the multiplexers and demultiplexers are issued in order to forward/fetch data to/from the targeted peripheral.

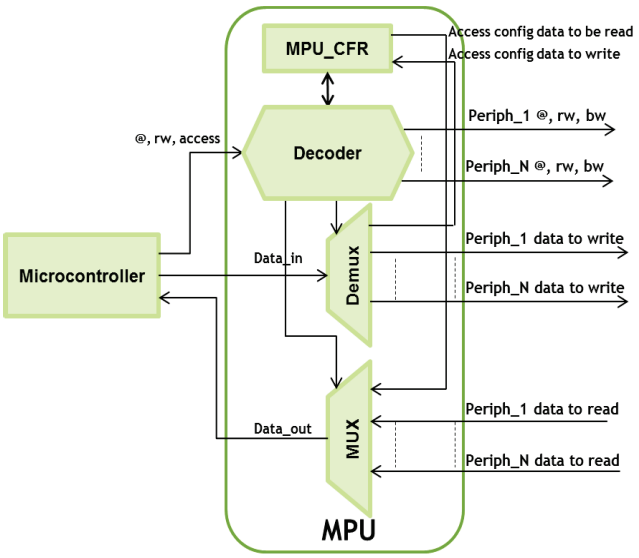

Figure 3: MPU architecture

B. LNT model

The LNT model was obtained by manual, yet systematic translation from the original SV model, following the aforementioned translation rules. Thus, the LNT has exactly the same architecture as the SV model [17].

The original SV code has 4386 lines. The resulting LNT model comprises: 6682 lines of code representing the direct translation of the SV model, 1591 lines of code representing the environment and auxiliary processes, and 673 lines of code for the types and channels definitions; this amount to 8946 lines of LNT code in total.

The resulting LNT model is thus more verbose than the SV model. This is due to several reasons: (i) some of the standard data types of SV (bit vectors and array types) have to be defined explicitly in LNT; (ii) when composing parallel processes in LNT, the synchronization rules must be explicitly defined, whereas ACC infers synchronization rules by intersecting the sets of communication channels; (iii) in LNT, a process declaration must include a description of its interface while SV does not require this; (iv) compositional state-space generation [20] requires auxiliary LNT processes to constrain the environment and generate the state spaces corresponding to system components; (v) unlike LNT, the asynchronous SV coding style supports compiler directives and macro-definitions, which allow for a more compact code.

The MPU block is composed of 146 concurrent processes communicating through 250 internal channels. These concurrent processes may in turn contain concurrent statements (or blocks of statements). This results in a high degree of internal concurrency; indeed, the CAESAR compiler of the CADP toolbox indicates 660 units running concurrently.

C. State-space reduction techniques

One of the major limitations of formal verification approaches is the state-space explosion problem, which, in an action-based verification setting, corresponds to the generation of LTSs with a huge number of states and transitions. For real-life systems, such as the MPU block, the state space is indeed large and cannot be merely handled by brute-force generation of exhaustive LTSs. To overcome this problem, more sophisticated approaches are needed, which we discuss in the following sections.

1) Abstraction of data types and variables

One of the most influential factors in the state-space size is the number of variables defined in the system and the size of their value domains, both of which may exponentially increase the number of reachable states. Hence, the system under verification should be constrained as much as possible by its environment (inputs, outputs, environment variables). Also, data variables that do not have any influence on the overall function of the system should be abstracted away, and those that really intervene in the system operation should be kept to a minimum.

In the case of the MPU block, we chose to abstract away all data that is not specifically processed inside the module. Indeed, the data forwarded/fetched by the decoder to/from the controlled peripherals is irrelevant to the operation of the MPU and does not undergo any modification inside the MPU. Hence, we redefined the LNT types for data inputs, replacing eight-bit vectors by enumerated types having only two possible values:

```
type data_bit8_t is
        data_token0, data_token1
        with "=="
end type
```

As the size of all data values manipulated in the MPU is a multiple of 8 bits, this abstraction can easily be extended to larger bit vectors, which can be defined as arrays of "data_bit8_t". This abstraction only requires limited changes to the original LNT code, considerably reduces the overhead of enumerating all possible values for data inputs, and can be rolled back without effort.

To further simplify the formal verification endeavor, this abstraction technique could be automated by annotating "irrelevant" variables in the SV description, so that a translation tool could recognize these variables and implement them using abstracted LNT data types.

As for the address inputs, we decided to structure the verification process by checking the access properties for each controlled peripheral separately. This allowed to reduce the value domain of each address input variable and, subsequently, the overall state space.

2) Reduction w.r.t. branching bisimulation

Another approach to tackle the state-space explosion problem is to reduce the LTS w.r.t. some of the numerous equivalence relations between LTSs that have been defined in the framework of concurrency theory [21]: strong bisimulation, branching bisimulation, divergence-sensitive branching bisimulation, safety equivalence, etc. Many equivalences have been implemented in the CADP toolbox, e.g., in the BCG_MIN tool, which minimizes an LTS already generated, or the REDUCTOR tool, which performs reduction on the fly, i.e., while an LTS is being generated.

The choice of the most appropriate equivalence relation primarily depends on the properties to be verified. Weaker equivalences reduce the LTS more, but do not necessarily preserve the validity of all the properties to be verified on the reduced LTS. For instance, safety equivalence yields large reductions but preserves only the validity of safety properties, not that of liveness properties (e.g., the absence of deadlocks).

For the MPU case study, we chose to use branching bisimulation [22], which allows significant LTS reductions, good algorithmic performance, and preserves the validity of all properties of interest.

3) Compositional state-space generation

Compositional state-space generation [20] approaches employ the "divide-and-conquer" principle to address the complexity of the system under verification. Typically, given a system having several parallel components, it may be judicious to generate the LTS of each component separately, reduce each LTS (e.g., w.r.t. branching bisimulation), and recombine in parallel all the minimized LTSs to obtain a reduced LTS of the global system. Such approaches only work for well-designed concurrent languages having a suitable formal semantics compatible with the chosen equivalence relation, which is the case for LNT and branching bisimulation.

In general, there are different ways of splitting a system into components, and there are different possible orders for recombining the minimized LTSs. Finding a proper solution is crucial for the success of compositional approaches. In the case of the MPU block, a solution was found empirically, using trial-and-error iterations to determine the feasibility and the CPU time needed for generating the LTS of the various components.

Also, the components of the system often constrain each other and/or are constrained by the environment in which they operate. Hence, the isolated LTS generation for an unconstrained component may lead to a severe growth in the state space of the behavior of this component. Therefore, in order to generate the LTS of a component, it is necessary to take into account its interface constraints and leverage them when generating the LTSs of the components. These constraints often enable states and transitions that are globally unreachable (i.e., in the LTS of the complete system) to be eliminated locally.

One solution to generate interface constraints consists in manually creating LNT processes that mimic the interface of each component. However, this method is time-consuming and requires calculating the appropriate constraints manually.

Another solution for constraining a given component, provided its environment is already modeled in LNT, is to use the semi-composition operator [23] [20], which represents interface constraints using a regular language that gives an over-approximation of all possible interactions between the component and its environment.

For the MPU, our verification strategy consisted in constraining the decoder module at first (see Figure 3). The LTS generated for the decoder was then used as an interface for generating the LTSs of the remaining components via semi-composition. Finally, these intermediate LTSs were reduced w.r.t. branching bisimulation and composed in order to produce an LTS to be used for the verification of the MPU block.

D. State-space generation results

We now present experimental data about the size of the generated LTSs and the CPU time required for generating them. Keeping in mind that the MPU controls the accesses to memories (NVM, RAM, ROM), interfaces, and coprocessors, we analyzed various scenarios in which the decoder handles a wide range of accesses to peripherals and memories. For each type of access, e.g., when accessing the crypto-coprocessor, the NVM, or the MPU_CFR (the latter denoting a read or write access to the security configuration registers), we generated the corresponding LTS compositionally.

Tables 1 and 2 give, for each type of access, the number of intermediate LTSs generated, the size of the largest of these intermediate LTSs, the total CPU time needed for compositional generation, and the size of the LTS finally obtained. All the measurements were made on a server equipped with an Intel Xeon E5520 CPU, 8MB of cache memory, and 32 GB of RAM memory, using a single CPU core for LTS generation.

Table 1: Compositional state-space generation

Access type	Number of intermediate LTSs	Largest intermediate LTS		Total runtime
		States	Transitions	
Coprocessor	20	6,623,272	53,110,205	13 min
MPU_CFR	20	26,967,252	355,426,207	4h33min
NVM	20	116,812,174	862,147,515	3h34min

Table 2: Final LTS

Access type	States of the final LTS	Transitions of the final LTS	File size (MB)
Coprocessor	5,540,008	42,113,629	92
MPU_CFR	26,967,244	355,426,197	692
NVM	21,030,280	143,670,959	296

E. Formal verification of MPU properties

On these generated LTSs, we verified not less than 184 MCL properties. These properties can be classified into functional and security properties.

1) Functional properties

We first verified deadlock freedom (see Section IV.A) and livelock freedom (see Section IV.B).

We verified (using the expression templates described in Section IV.C) that read and write operations are mutually exclusive for certain configuration registers.

We verified various stimuli-response properties (see Section IV.D) to validate the overall correct behavior of the circuit. For instance, we verified that, for each access to peripherals, memory, and MPU_CFR, the decoder extracts the correct output address, and that the multiplexer (resp. demultiplexer) fetches (resp. forwards) the data from (resp. to) the component being accessed.

We also verified the correct initialization of the MPU configuration registers. In accordance with the action-based paradigm, the LTS model does not directly export the values of the configuration registers in the initial state of the system; however, the current value of each configuration register is observable on all the transitions that perform a read operation on this register. Thus, as long as the initial value of the register is not explicitly modified by a transition performing a write operation, any transition that reads this register should carry its initial value. We therefore expressed the intended property in MCL as the conjunction, for each configuration register, of a liveness property and a safety property. The liveness property checks that there exists at least a read transition not preceded by any write transition, and that the value carried by this read transition is indeed the correct initial value for that register. The safety property checks that all the read transitions not preceded by any write transition do not carry a value that is different from the initial configuration.

2) Security properties

We also verified by model checking the security requirements related to the many access-control policies enforced by the MPU. For instance, we verified that, each time a subject requests access to a controlled object (peripheral, memory, or MPU_CFR), an access-right evaluation operation must take place before access is granted.

On an LTS, this property expresses that any transition on an object's input or output channel (meaning that access to that object was granted) must be preceded by both a transition performing a read operation on the access-right channel of the subject and a transition on the read channel of the corresponding configuration register. In MCL, this property amounts to a combination of a liveness property (which checks the existence of an execution sequence containing the required transitions) and a safety property (which ensures that there is no execution sequence violating the access-control property).

VI. Related work

Our approach is based on a standard HDL, SystemVerilog, a synthesizable subset of which is used, with a few adaptations intended for asynchronous circuits. As mentioned in Section I, this differs from prior approaches based on specific languages dedicated to asynchronous circuits. Rather than requiring designers trained in specific languages and particular tools, we remain in a mainstream design flow that smoothly fits in with commercial EDA tools. This idea of using SystemVerilog to design asynchronous circuits was already put forward in [7] [8] but with an emphasis on simulation and synthesis, rather than formal verification.

Our formal verification approach focuses on system-level models of asynchronous circuits, and thus differs from other approaches that operate on detailed, gate-level descriptions using various modeling formalisms: Petri nets [24] or a hardware-targeted restricted version of Petri nets called signal transition graphs [25] [26], delay-insensitive algebra [27], trace theory [28], CCS [29], CSP [30] [31], LOTOS [32] [33],

conditional partial order graphs [34], continuous-time differential equations [35], timed automata [36], etc. VHDL libraries for asynchronous circuits have also been proposed [37], with a translation of handshake protocols to Petri nets for formal verification. Such detailed models allow fine analyses of the timing behavior (using, e.g., model checking, refinement checking, reachability checking [38], theorem proving [39] [40], etc.), so as to study the propagation of glitches [41] [42] or verify the correctness of speed-independent circuits [43]; however, the use of very detailed models often limits the size of circuit that can be analyzed.

Instead, our approach relies on more abstract, system-level models, which are akin to distributed systems with two key differences: the presence of hardware-specific operations (e.g., probing whether a communication has been initiated), and the constraints imposed by hardware-synthesis tools. Many approaches have also addressed such high-level models using, e.g., Petri nets [44], signal transition graphs [45] [46], communicating automata [47], state graphs [48], and process calculi such as CSP [49] [50], CHP [51], LOTOS [52], and LNT [52]. It has been pointed out [49] that process calculi provide a unifying framework for modeling asynchronous circuits from system to gate level. In our industrial design flow, however, descriptions are written using a standard HDL, from which formal models can be derived automatically. Beyond asynchronous circuits, one can also mention another approach that provides formal semantics to a subset of SystemVerilog by mapping this subset to a timed process calculus suitable for clocked circuits [53].

Regarding the specification of properties, we cannot rely on the PSL [54] and SVA [6] property languages, which are too much oriented towards a state-based, linear-time setting. Although SVA can, in theory, express properties of asynchronous circuits, this endeavor is complex [55], making SVA more suitable for RTL properties than TLM ones [56]. Also, unconstrained accesses to data variables in PSL or SVA assertions are not applicable to asynchronous circuits, in which global signals must be referenced using channels. We therefore preferred MCL, an action-based, branching-time property language that offers suitable theoretical properties, such as adequacy w.r.t. bisimulation equivalences and proper support for compositional verification.

VII. Conclusion

In this paper, we proposed a complete flow for the design of asynchronous circuits: this flow is based on SystemVerilog and includes synthesis and formal verification tools. As in the pioneering works of [47] [51] [18], our approach relies upon the CADP toolbox [9], yet uses its most recent features, namely the property specification language MCL [11] and the modeling language LNT [10], which combines salient concurrency-theory concepts with user-friendly programming notations, and which can be derived from SystemVerilog descriptions using systematic mapping rules we defined.

We have shown the validity of our approach on an industrial circuit, a Memory Protection Unit, the complexity of which was successfully dealt with using abstractions and compositional minimization techniques. These results prompted Tiempo to apply the described flow to several other asynchronous circuits, including a DES crypto-processor and an Asynchronous Serial Link circuit [57].

As regards forthcoming work, we plan to implement a SystemVerilog-to-LNT translator in order to accelerate the formal verification task, while eliminating the potential errors that may arise from manual translation. In particular, we intend to develop LNT libraries implementing the usual SystemVerilog data types and operations. We also plan to formulate SystemVerilog design guidelines for promoting abstraction/reduction strategies and fighting state-space explosion. We will also investigate how the proposed flow can be extended towards lower-level (e.g., gate-level) descriptions, so as to establish the formal verification link advocated in [49] between the various design abstraction levels. Our aim is to enable the verification of quantitative properties (timing, pipeline saturation, etc.) on gate-level formal models, and to perform equivalence checking between models at different levels of design abstraction.

Acknowledgements

The present work has been partly funded by BPI France and FEDER (*Fonds Européen de Développement Économique Régional*) Rhône-Alpes Auvergne under national project "SecurIoT-2" supported by the four competitiveness clusters Minalogic, SCS, Systematic Paris-Région, and Derbi, and by the European Community under project "THINGS2DO" of the ENIAC Nanoelectronics Joint Technology Initiative.

References

[1] A. J. Martin, "Compiling communicating processes into delay-insensitive VLSI circuits". Distributed Computing, vol.1, no. 4, pp. 226–234, 1986, Springer.

[2] J. T. Udding. "A formal model for defining and classifying delay-insensitive circuits". Distributed Computing vol. 1, no. 4, pp. 197– 204, 1986, Springer.

[3] A. Peeters, M. de Wit. "Haste Manual, Version 3.0". Handshake Solutions, 2006.

[4] J. L. W. Kessels, A. M. G. Peeters. "The Tangram framework (embedded tutorial): Asynchronous circuits for low power". Proc. of Asia and South Pacific Design Automation Conference ASP-DAC 2001, pp. 255–260, IEEE.

[5] D. Edwards, A. Bardsley. "Balsa: An asynchronous hardware synthesis language". The Computer Journal, vol. 45, no. 1, pp. 12-18, 2002, Oxford University Press.

[6] IEEE 1800-2012. "IEEE Standard for SystemVerilog--Unified Hardware Design, Specification, and Verification Language". Feb. 21, 2013.

[7] M. Renaudin, A. Fonkoua. "Tiempo Asynchronous Circuits System Verilog Modeling Language". Proc. of ASYNC 2012, pp. 105-112, IEEE.

[8] A. Saifhashemi, P. A. Beerel. "SystemVerilogCSP: Modeling Digital Asynchronous Circuits Using SystemVerilog Interfaces". Proc. of Communicating Process Architectures CPA 2011, pp. 287-302, IOS Press.

[9] H. Garavel, F. Lang, R. Mateescu, W. Serwe. "CADP 2011: a toolbox for the construction and analysis of distributed processes". International Journal on Software Tools for Technology Transfer 2013, vol. 15, no. 2, pp. 89-107, Springer.

[10] D. Champelovier et al, "Reference Manual of the LNT to LOTOS Translator (Version 6.7)", INRIA/VASY and INRIA/CONVECS, 2017.

[11] R. Mateescu, D. Thivolle. "A Model Checking Language for Concurrent Value-Passing Systems". Proc. of Formal Methods FM 2008, LNCS, vol. 5014, pp. 148-164, Springer.

[12] R. Milner. "Communication and Concurrency". Prentice Hall, 1989.
[13] S. D. Brookes, C. A. R. Hoare, A. W. Roscoe. "A Theory of Communicating Sequential Processes". Journal of the ACM vol. 31, no. 3, pp. 560-599, 1984.
[14] ISO international standard 8807:1989. "LOTOS: A formal description technique base on the temporal ordering of observational behaviour".
[15] H. Garavel, F. Lang, W. Serwe. "From LOTOS to LNT". LNCS, vol. 10500, pp. 3-26, 2017, Springer.
[16] H. Garavel, F. Lang. "SVL: a Scripting Language for Compositional Verification". Proc. of Formal Techniques for Networked and Distributed Systems FORTE 2001, pp. 377-392, Springer.
[17] A. Bouzafour. "Asynchronous SystemVerilog Semantics and their translation to LNT". Tiempo internal document. May, 2017.
[18] H. Garavel, G. Salaün, W. Serwe. "On the semantics of communicating hardware processes and their translation into LOTOS for the verification of asynchronous circuits with CADP". In Science of Computer Programming, vol. 74, no. 3, pp.100-127, 2009, Elsevier.
[19] Z. Manna, A. Pnueli. "The Temporal Logic of Reactive and Concurrent Systems I (Specification)". Springer, 1992.
[20] H. Garavel, F. Lang, R. Mateescu. "Compositional Verification of Asynchronous Concurrent Systems using CADP". Acta Informatica vol. 52, no. 4-5, pp. 337-392, 2015, Springer.
[21] R. van Glabbeek. "The Linear Time - Branching Time Spectrum I". Handbook of Process Algebra, chp. 1, 2001, North Holland.
[22] J. F. Groote, F. Vaandrager. "An Efficient Algorithm for Branching Bisimulation and Stuttering Equivalence". Proc. of ICALP 1990, LNCS, vol. 443, pp. 626-638, Springer.
[23] J.-P. Krimm, L. Mounier. "Compositional State Space Generation from LOTOS Programs". Proc. of TACAS'97, LNCS, vol. 1217, pp. 239-258, Springer.
[24] K. Masukura, M. Tomisaka, T. Yoneda. "Verification of asynchronous circuits based on zero-suppressed BDDs". Systems and Computers in Japan, vol.32, no. 2, pp. 43-54, 2001, Wiley.
[25] I. Poliakov, A. Mokhov, A. Rafiev, D. Sokolov, A. Yakovlev, "Automated Verification of Asynchronous Circuits Using Circuit Petri Nets". ASYNC 2008, pp. 161-170, IEEE.
[26] I. Blunno, L. Lavagno. "Automated synthesis of micro-pipelines from behavioral Verilog HDL". Proc. of ASYNC 2000, pp. 84-92, IEEE.
[27] H. Park, A. He, M. Roncken, X. Song, I. E. Sutherland. "Modular Timing Constraints for Delay-Insensitive Systems". Journal of Computer Science and Technology, vol. 31, no. 1, pp. 77-106, 2016, Springer.
[28] D. L. Dill. "Trace Theory for Automatic Hierarchical Verification of Speed- Independent Circuits". MIT press, 1988.
[29] G. Clark, G. Taylor. "The Verification of Asynchronous Circuits using CCS". Technical Report ECS-LFCS-97-369, University of Edinburgh, 1997.
[30] S. X. Wang, M. Z. Kwiatkowska, "On Process-algebraic Verification of Asynchronous Circuits". Fundamenta Informaticae, vol. 80, no. 1-3, pp. 283-310, 2007, IOS press.
[31] M. B. Josephs. "Gate-level modelling and verification of asynchronous circuits using CSPM and FDR". Proc. of ASYNC 2007, pp. 83-94, IEEE.
[32] M. Yoeli, A. Ginzburg. "LOTOS/CADP-based verification of asynchronous circuits". TR CS-2001-09, Department of Computer Science, Technion, 2001.
[33] J. He, K. J. Turner. "Verifying and Testing Asynchronous Circuits using LOTOS". Proc. of of Formal Techniques for Networked and Distributed Systems FORTE 2000, pp.267-283, Springer.
[34] A. Mokhov, A. Yakovlev, "Conditional Partial Order Graphs: Model, Synthesis, and Application". IEEE Transactions on Computers vol. 59, no. 11, pp. 1480-1493, 2010.
[35] C. Yan, M. Greenstreet, J. Eisinger. "Formal Verification of an Arbiter Circuit". ASYNC 2010, pp. 165-175.
[36] M. Bozga, H. Jianmin, O. Maler, S. Yovine. "Verification of Asynchronous Circuits using Timed Automata". Electronic Notes in Theoretical Computer Science, vol. 65, no.6, pp. 47-59, 2002, Elsevier.
[37] C. J. Myers. "Asynchronous Circuit Design". Wiley-Interscience, 2004.
[38] C. Yan, F. Ouchet, L. Fesquet and K. Morin-Allory, "Formal Verification of C-element Circuits". Proc. of ASYNC 2011, pp. 55-64, IEEE.
[39] C. K. Chau, W. A. Hunt Jr., M. Roncken, I. E. Sutherland. "A Framework for Asynchronous Circuit Modeling and Verification in ACL2". Proc. of Haifa Verification Conference, pp. 3-18, 2017, Springer.
[40] P. N. Loewenstein. "Formal verification of counterflow pipeline architecture". LNCS, vol. 971, pp. 261-276, 1995, Springer.
[41] Y. Peng, I. W. Jones, M. R. Greenstreet. "Finding Glitches Using Formal Methods". Proc. of ASYNC 2016, pp. 45-46, IEEE.
[42] M. Függer, T. Nowak, U. Schmid. "Unfaithful Glitch Propagation in Existing Binary Circuit Models". IEEE Transactions on Computers vol. 65, no.3, pp. 964-978, 2016.
[43] D. Sokolov, V. Khomenko, A. Mokhov, A. Yakovlev, D. Lloyd. "Design and Verification of Speed-Independent Multiphase Buck Controller". Proc. of ASYNC 2015, pp. 29-36, IEEE.
[44] A. Kondratyev, M. Kishinevsky, A. Taubin, J. Cortadella, L. Lavagno. "The Use of Petri Nets for the Design and Verification of Asynchronous Circuits and Systems". Journal of Circuits, Systems, and Computers vol.8, pp. 67-118, 1998, World Scientific.
[45] J. Beaumont, A. Mokhov, D. Sokolov, A. Yakovlev. "Compositional design of asynchronous circuits from behavioural concepts". Proc. Of Formal Methods and Models for System Design MEMOCODE 2015, pp. 118-127, IEEE.
[46] V. Khomenko, A. Mokhov, D. Sokolov, A. Yakovlev. "Formal Design and Verification of an Asynchronous SRAM Controller". Proc. Of Application of Concurrency to System Design ACSD 2017, pp. 59-67, IEEE.
[47] M. Boubekeur, D. Borrione, L. Mounier, A. Siriani, M. Renaudin. "Modeling CHP descriptions in Labeled Transitions Systems for an efficient formal validation of asynchronous circuit specifications". In "Languages for System Specification", 2004, Springer.
[48] H. Zheng, E. Rodriguez, Y. Zhang, C. J. Myers. "A Compositional Minimization Approach for Large Asynchronous Design Verification". Proc. of SPIN 2012, LNCS, vol. 7385, pp.62-79, Springer.
[49] X. Wang, M. Kwiatkowska, G. Theodoropoulos, Q. Zhang. "Towards a Unifying CSP approach to Hierarchical Verification of Asynchronous Hardware". Electronic Notes in Theoretical Computer Science, vol. 128, no. 6, pp. 231-246, 2005, Elsevier.
[50] H. K. Kapoor. "A Process Algebraic View of Latency-Insensitive Systems". IEEE Transactions on Computers vol. 58, no.7, pp. 931-944, 2009.
[51] G. Salaün, W. Serwe, Y. Thonnart, P. Vivet. "Formal Verification of CHP Specifications with CADP - Illustration on an Asynchronous Network-on-Chip". Proc. of ASYNC 2007, pp. 73-82, IEEE.
[52] W. Serwe. "Formal Specification and Verification of Fully Asynchronous Implementations of the Data Encryption Standard". Electronic Proceedings in Theoretical Computer Science, vol. 196, pp. 61-147, 2015, Elsevier.
[53] K. L. Man, C. U. Lei, H. K. Kapoor, T. Krilavicius, J. Ma, N. Zhang, "PAFSV: A Formal Framework for Specification and Analysis of SystemVerilog". Computing and Informatics, vol. 35, pp. 143-176, 2016.
[54] IEEE 1850-2010. "IEEE Standard for Property Specification Language (PSL)". June 28, 2012.
[55] D. Smith. "Asynchronous Behaviors Meet Their Match with SystemVerilog Assertions". Proc. of Design and Verification Conference and Exhibition DVCon'2010, Feb. 2010.
[56] D. Korchemny. "SystemVerilog Assertions for Formal Verification". Tutorial at the 9th Haifa Verification Conference HVC'2013, Nov. 2013.
[57] M. Renaudin, R.Wilson, S.Engels. "Clockless circuit design in FDSOI", IP-SoC 2017 conference, Grenoble, https://www.design-reuse.com/ipbasedsocdesign/v_2017-tiempo.html