Automatic synthesis of digital circuits from temporal specifications

Fatemeh Javaheri

UNIVERSITÉ GRENOBLE ALPES

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES
Spécialité : **Nanoélectronique et Nanotechnologies**

Arrêté ministériel : 7 août 2006

Présentée par

Fatemeh (Negin) JAVAHERI

Thèse dirigée par **Mme. Dominique BORRIONE**
et codirigée par **Mme. Katell MORIN-ALLORY**

Préparée au sein du **Laboratoire TIMA**
Dans **l'École Doctorale Electronique, Electrotechnique, Automatique** & **Traitement du Signal (E.E.A.T.S)**

Synthèse automatique de circuits numériques à partir de spécifications temporelles

Thèse soutenue publiquement le **1 octobre 2015**,
devant le jury composé de :

M. Philippe COUSSY
Professeur, Université de Bretagne Sud, Président
M. Paolo PRINETTO
Professeur, Politecnico di Torino, Rapporteur
M. Rolf DRECHSLER
Professeur, Universität Bremen, Rapporteur
Mme. Dominique BORRIONE
Professeur, Université Joseph Fourier, Directrice de thèse
Mme. Katell MORIN-ALLORY
Maître de Conférences, Grenoble INP, Co-Directrice de thèse

Acknowledgments

Firstly, I would like to express my sincere gratitude to my supervisors Prof. Dominique BORRIONE and Dr. Katell MORIN-ALLORY for their continuous support of my thesis studies and research, for their patience, motivation, enthusiasm, and immense knowledge. Their guidance helped me throughout the execution of this research, and the writing of this thesis.

Besides my supervisors, I would like to thank the rest of my thesis committee: Prof. Philip COUSSY for being the president, and Prof. Rolf DRECHSLER and Prof. Paolo PRINETTO for accepting our invitation to evaluate and examine this thesis and their useful suggestions and comments.

Also I thank the head of the VDS group, Prof. Laurence PIERRE, and administration team in the TIMA laboratory: Laurence BENTITO, Anne-Laure FOURNERET-ITIE, Sophie MARTINEAU, Youness RAJAB, Frederic CHEVROT, Lucie TORELLA, and Alexandre CHAGOYA for supporting and helping me kindly.

I would like to extend my appreciation to my colleagues and friends: Maryam BAHMANI, Alexandre PORCHER, Zeineb BELHADJ AMOR, Ladan AMINI, Hoda BARENIA, Paria SALMANZADE, Leila GRAYELI, Ahmad BIJAR, Laila DAMRI, Hamed SHEYBANI, and Sahar FOROUTAN for their kindness, support, and all the fun we have had during the last three years.

I would like to acknowledge the help of Guillaume PLASSAN, Sebastian CORZO, and Hugo ROYNETTE in collecting some experimental results.

I also thank my best friends Somayeh, Paria, Mona, Nastaran, and Marzieh for all their supports from a very long distance.

I specially thank Prof. Zain NAVABI who taught me the research attitude. My appreciation for his fatherly advises and supports cannot be expressed in words.

I would like to express my deepest gratitude to my parents, who support me spiritually throughout my life, for all of the sacrifices that they have made on my behalf. Without their encouragement and support, I would not have a chance to be here. All I have today is because of their unconditional love and support. I thank my lovely sister Negar, and my dear brother Ali for their love and encouragements. Although we were far apart, I felt them every moment. I would also like to thank my mother, father and sister in-laws for their kindness and support.

Last but not the least; I would also like to extend my appreciation to my beloved, Sina NAKHJAVANI for his unconditional love, patience, kindness, and support. Without his support and patience, I could not complete this journey. Words cannot express how grateful I am to him.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

1.1 Preface

Day by day the influence of technical systems on our life increases. They have emerged in all aspects of our life, ranging from entertainment to communication, business, transport, and medicine, where they affect human life directly. Digital circuits, as processors or controllers, are a crucial part of such systems. However, for justifying our dependence on such systems we should be able to answer this question: "are these systems *reliable* and *safe*?". Circuit verification answers to this question.

Integrated circuit capacity follows Moore's law. The Intel's 4004 microprocessor introduced in 1971 had 2300 transistors. Due to the advances in semiconductor technology, today's complex systems having a wide variety of functionalities are being integrated in a single chip as system-on-a-chip (SoC), containing billions of transistors. For example, Intel's 15-core Xeon Ivy Bridge-EX is a commercially available CPU (in one chip) with over 4.3 billion transistors on a single chip [TRA].

The circuits that used to be considered a system, now are just one core among hundreds of components on a single chip. The increasing size and complexity of the designs, and the time to market considerations, are creating new verification challenges; the verification problem is getting very huge and it has become increasingly difficult to identify all the design bugs in such a large and complex system before the chips are fabricated. Providing the appropriate testbenches is another challenge.

Design error detection and correction in hardware is too expensive. Specially, if it is after fabrication; all the malfunctioned chips should be collected and replaced by the new ones. This situation occurred with the FDIV bug in the floating point unit of Intel's Pentium processor. The company had to spend about 475 million US dollars to replace the faulty processors [Kro99]. Therefore, design errors should be detected as soon as possible. Earlier bug detection means shorter time to market, less cost, and more success.

However, obtaining first time right digital circuits is a hard to reach objective when considering the current architectures. The verification problem of such systems is addressed by a combination of methods and technologies that include high-level simulation, property checking, step by step refinement verification, prototyping, automatic synthesis and equivalence checking. At some point in the design flow, the use of pre-designed and fully verified modules allows to stop the refinement and verification process for these modules. However, their interconnections should still be verified.

The work reported in this thesis proposes a method and a prototype tool to help the verification of the control and communication protocols between modules. A drawback of the formal verification methods is that they only can be used after the system is designed. The main idea of this work is to propose methods for generating a system from its specifications and verifying its correctness during the process of hardware generation, *correct-by-construction.*

In this chapter, we briefly review the design process of SoCs, and discuss its limitations that are the motivation of this work.

1.2 Classical design flow

Figure 1.1 shows a very abstract view of the traditional design flow. A typical design process starts by considering the informal behavior of the system. This informal behavior is generally given in a document written in a human language, for example in English. Then, the design teams develop an implementation. It may be required to partition a design into software and hardware, and implement them concurrently. In the next step, the implementation should be verified to consider if it conforms to the given specifications. To formally verify the design, the formal specifications should be extracted from the informal behavior of the system. After verification, several refinements of the circuit may be required due to the detected errors.

This refinement and debugging process is very time consuming for today's extremely large and complex circuits. Even if faults are detected prior to fabrication, the required time for correcting bugs may be high, which can delay the time of introducing the product to the market. Studies show that a delay of one week equals a revenue loss of at least tens of millions of dollars [Kro99].

Consequently, a significant amount of time during the design process is spent for error finding, usually by simulation or emulation. A recent study shows that the total project time spent in verification in 2014 was 57% in average, while it was 46% in 2007. In addition, the number of projects that spend more than 80% of their time in verification has increased [Fos15].

Moreover, the flow of Fig. 1.1 assumes design and verification to be performed by different teams, which brings added difficulties:

- *Communication between design and verification engineers.* Providing the formal specifications for verification is difficult both for design engineers and verification engineers. It is usually difficult for the designers to write temporal declarative assertions. Conversely, it is difficult for a verification engineer to extract the designer's intent from a conventional Hardware Description Language (HDL) code.

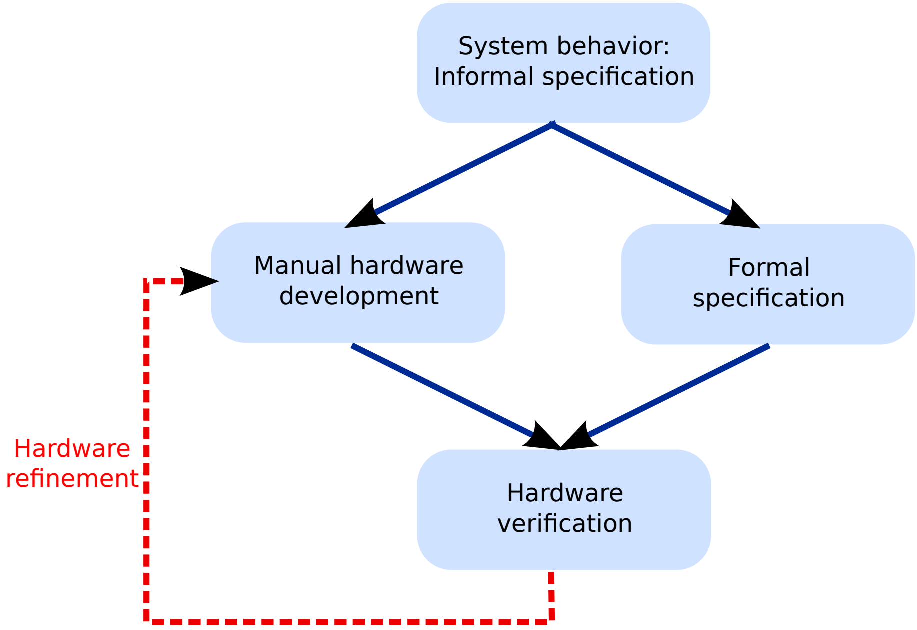


FIGURE 1.1: Traditional design flow

- *Testing environment.* Providing a testbench for a complex exiting design is itself a challenge. It is difficult to identify all the possible scenarios for a big and complex design, by analyzing its HDL code. In addition, it is a formidable task to modify the testbench after identifying a new bug and modification in the design. All the previously examined scenarios should be verified again, and the testbench should be revised to consider the possible new scenarios.

1.3 The proposed design flow

The above mentioned difficulties have brought us to the context of *Assertion Based Synthesis* (ABS), i.e. the direct production of compliant (control and communication) modules from a set of assertions. A property is seen as the specification of the module to be designed. The objective is then to directly produce the synthesizable Register Transfer Level (RTL) design from its assertions.

In ABS, properties about the behavior of a component (*assertions*) or its environment (*assumptions*) specify the input-output functional characteristics of the modules and the communications between system parts.

Generally, a design assertion (property) expresses the design's intent. Assertions are concise, declarative, expressive, and unambiguous specifications of desired system behavior.

A complete set of assertions can unambiguously characterize how a module reacts to signals sent to it, logically and temporally.

In the proposed design flow we start the design in a more abstract level, and incorporate the verification into the design process. The design behavior is expressed formally using assertions (see Fig. 1.2). In this method, the design and verification tasks have been unified; a *correct-by-construction* circuit is generated from the formal specifications directly.

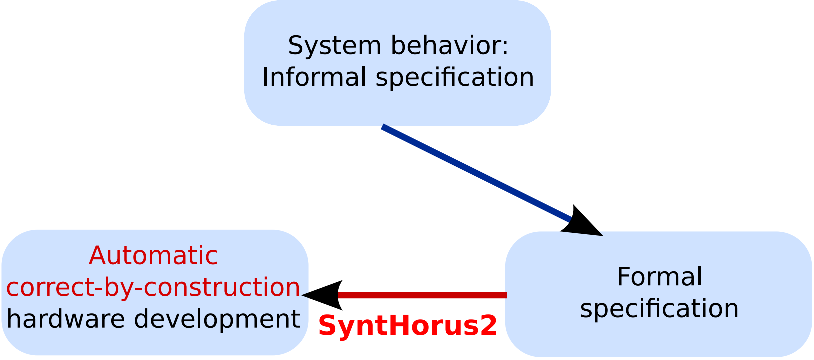


FIGURE 1.2: Our proposed design flow

The generated circuit is called *reactant*: it *reacts* to waveforms on its inputs and produces waveforms on its outputs, in compliance with the assertions. Compiling assertions into reactants, and checking the compliance of the reactant behavior with the design is very efficient.

A *reactant* may be used to replace a non available module by a fast prototype of it, to check a more comprehensive design; it may also replace the (complex) environment of a designed module by a fast prototype of just the part of the environment that interacts with it.

Our new proposed flow has the following steps:

- providing the *formal specification* of the circuit: It is the first step in the design process, and it is a challenging task to generate the formal specifications from the informal description of the circuit, and it is out of the scope of this thesis. In this context, we assume that we have the formal specification of the circuits.

- deriving an *implementation* from the specification: this thesis proposes a method for synthesizing a circuit from its formal specification. We have provided a tool, SyntHorus2 to synthesize the circuits from specifications.

- correctness relation and proof: we should check if the properties are complete and consistent, and also we should check if the implementation is equivalent to the specification.

1.4 Overview of the thesis

The rest of this manuscript is organized as follow: Chapter 2 introduces the principles of circuit verification. In addition, it addresses Property Specification Language (PSL), and the subset of PSL that we deal with in this thesis.

The previous works that have been done in the context of assertion-based verification and synthesis are summarized in Chapter 3.

Chapter 4 introduces our global synthesis flow and a running example, the IBM Generalized Buffer: this is the simplest example that exhibits all the problems to be solved.

Chapter 5 and Chapter 6 explain how to construct the correct-by-construction library of primitive reactants for FL operators (Chapter 5) and SERE operators (Chapter 6). The concept of dependency relation is introduced, and a dependency relation is defined for each FL and SERE operator. Then, for each operator, its hardware interpretation is given in accordance with the trace semantics.

Using the dependency relations, Chapter 7 provides an *annotation* algorithm to decide (annotate) the signal's direction in a single property: the signal is an input to a property (observed), or is an output of the property (constrained).

Having the library of primitive reactants, and also the signal directions on each property, Chapter 8 explains how to generate a property reactant, i.e. a complex reactant. It is the interconnection of primitive reactants.

Some signals may be constrained by several properties. Moreover, after annotation some signals may still exist without any directions. The direction of such signals cannot be determined by considering a property alone. Chapter 9 addresses how to identify the value of such signals. The method extracts the dependency among all the properties that include the signal, and then generates *solver*(s) to specify (resolve) the value of the signal. Then, the final circuit is the interconnection of the complex reactants and the solvers.

Practical results and an analysis of the performance of the method are provided in Chapter 10. Compared to other existing works, our proposed method is fast, scalable, and produces reasonably small reactants.

Finally, Chapter 11 concludes the work, and addresses future research.

Chapter 2

Assertion-based Verification

Contents

2.1 Introduction

One of the challenges of SoC design is verification, and it is very important to reduce the verification time. The need for an advanced verification methodology, with improved observability of design behavior has increased significantly. This requires new *design* and *verification* techniques. Here, we introduce an *assertion-based* methodology that enables designers to deal with the complex and large circuits, and also meet the time-to-market goals. This method takes advantage of both *simulation-based* and *formal* verification: formal verification can check scenarios that are hard to cover in simulation, while simulation can verify the designs that are too big for any formal verification methods. Hence, the assertion-based verification can ensure higher design quality and faster time to market.

Generally, a design *assertion* (property) expresses the design's intent, and refers to the properties that should be verified. Assertions are concise, declarative, expressive, and unambiguous specifications of desired system behavior, which are used to guide the verification process.

Assertions can be checked both in simulation and formal verification. Thus, a common environment should be provided for both. The belief that formal verification does not need a testbench is a myth. This can be handled by using *constraints*. Constraints are the required conditions for a verification. They are used in the testbenches to model the environment of a DUV. Therefore, constraints are used in simulation as *generators* of stimuli [YAP10]. Assertion are checked in simulation usually as *monitors* of simulation traces.

In contrary to the black-box testing approach, assertion-based approach adds assertions that monitor internal points within the DUV. This avoids missing an internal error for a given stimulus; and increases the observability of the design. Using assertions, it is possible to detect when and where bug occurs and isolating bugs closer to the actual source. Therefore, design teams save debug time since an engineer does not have to backtrack through large simulation trace files and multiple blocks of logic to identify the exact location of the bug. Experiences demonstrate that assertions can save up to 50 percent of debug time [ABG$^+$00, Fos08].

In addition, assertions facilitate reusing *Intellectual Property* (IP) components. If the IP component comes with the assertions that describe its interface behavior, it is much easier to use this component inside the design environment. Additionally, the support effort required by the IP supplier company is reduced because assertions tell the users when they are using the IP incorrectly.

Assertions that describe the system behavior can be verified using various formal techniques in early stages of the design. Verifying the assertions, some design errors, such as inconsistencies, may be captured early.

Another advantage of assertions is specifying correct behavior of the design unambiguously. Other engineers can review the assertions to understand the specifics of how to interface with another block.

Furthermore, assertions formally document protocols, interfaces, and assumptions in an unambiguous form that clarifies a designer's interpretation of the specification and design intent [FKL03].

Assertions can be expressed using a temporal property language. In the following sections, we review the verification technology. Then, we consider the importance of the assertion languages, and then, introduce two formal languages that are commonly used in assertion-based verification.

2.2 Review of verification technology

Functional verification approaches can be classified as being either dynamic (simulation) or static (formal). Simulation is the first verification step, whereas static verification is based on mathematical proofs and plays a complementary and also very important role.

2.2.1 Simulation-based verification

Simulation is the experimental process that mimics the dynamic behavior of a design through time [Mil94]. Simulation-based verification is applied to a representative subset of variable values and behaviors of a circuit, to check if an implementation behaves correctly with regard to its specification [Kro99]. Actually, this approach is a testing approach: the designer implements the circuit using HDLs, provides a testbench that instantiates the design under verification (DUV), applies the input vectors (simulation stimuli) to the DUV one by one, and compares the outputs to the expected behavior.

Using this method, some errors may be masked due to the stimuli; they may appear using another stimulus, or by running the simulation for a few more cycles. However, verifying all the possible stimuli and all the internal properties of a design is not possible, because of the exponential number of stimuli with respect to the number of inputs and states.

Symbolic simulation is a way to speed up the simulation. The key idea of symbolic simulation is representing the arbitrary input values by symbols using the mathematical techniques. In contrast to conventional simulation, the symbolic simulation propagates symbols. Symbolic simulation differs from logic simulation since it builds Boolean expressions instead of the scalar values, as a result of circuit simulation. In symbolic simulation, the state space of a synchronous circuit is explored iteratively by means of symbolic expressions. At each step of simulation a Boolean expression is assigned to each output signal and present state signal. The simulation proceeds by deriving the appropriate Boolean expression for each internal signal of the combinational part of the network, based on the expressions at the inputs of each logic gate and the functionality of the gate. The procedure is equivalent to propagating the symbolic expressions through a time-unrolled version of the circuit, where the combinational part is duplicated as many times as there are simulation steps.

Formal verification is an alternative solution that uses mathematical proof to show that an implementation conforms to its specification for all time instances and all input combinations. In the following, each verification method is explained briefly.

2.2.2 Formal verification

The goal of formal verification is considering formally if an *implementation* satisfies a *specification*. The term *implementation* refers to the design description that is to be verified, while the term *specification* refers to the design description or the property with respect to which correctness is to be determined.

In formal verification, both specification and design descriptions are translated into mathematical models. The degree of the confidence obtained by formal verification depends on the power of the underlying modeling formalism and the accuracy of the specifications [Mil94].

The mathematical model can be expressed in various ways: data flow graphs, process algebras, finite state machines, temporal logic, and etc. A design can be modeled directly using these formalisms, or these models can be extracted from the HDL description of a design. Using the mathematical model of the circuit and its behavior, formal verification should prove that the design satisfies the specification of its intended behavior through mathematical proofs; it is verified if there is a relationship between the implementation and the specifications. If there exists a design bug, formal verification techniques produce a counter-example to facilitate the debugging process.

Almost all the formal verification techniques can be classified in one of two categories: *model-based* or *proof-theoretic.*

The model-based techniques use a formalization based on propositional temporal logics (see Section 2.2.2.3) or finite state machines (see Section 2.2.2.1) [Gup92, CBE^{+}92]. The algorithms are based on the brute-force exploration of the whole solution space. The effective data structures for propositional logic are *decision diagrams* (BMD, BDD, ...). Alternatively, the problem can be converted to a Boolean satisfiablility problem, and SAT solvers can be used to determine if an interpretation of a system satisfies the given Boolean formula. Model-based techniques can be categorized as checking the properties over the design: *model checking* and checking if two implementations are equivalent: *equivalence checking.*

The other family of formal verification techniques are proof-theoretic methods that are based on abstractions and hierarchical methods to prove the correctness of a system. This method uses theorem prover software to provide support in reasoning and deriving proofs about the specifications and the developed model of a design.

Based on the above discussion, having a formalism is inevitable. *Temporal logics* and *regular expressions* specify a set of behaviors in a rigorous formalism. Almost all the design verification methods are essentially the process of deciding if the design behavior conforms to the properties. Here, we first introduce some notations and terminologies that are required in formal verification. Then, the concepts of regular expression and temporal logic are reviewed. Finally, we review the various formal verification methods.

2.2.2.1 Terminology and notations

A *proposition* is a statement that can be either *true* or *false*. An *Atomic Proposition* (AP) cannot be broken into simpler propositions. In a circuit, APs include all the signals in the design. *Propositional formulas* are composed from APs with Boolean connectives such as conjunction, disjunction, and negation. The truth value of a propositional formula can be calculated from the truth values of the atomic propositions that it contains.

A way of expressing a sequential system mathematically is to represent it as a *Finite State Machine* (FSM).

The FSM is modeled by means of APs. So, it is possible to process it with Boolean operations. We assume that the set of alphabet is $B = \{0, 1\}$. A state machine M can be formally described by a 7-tuple $M = (S, s_0, F, I, O, \delta, \lambda)$ as follows [DB95]:

- S is a power of B, and represents the set of states of the machine.

- $s_0 \in S$ represents the initial state of M.

- F is a subset of S, and represents the set of the final states of the machine. F is partitioned into a set of *accepting* states and a set of *rejecting* states; $F = \mathcal{A}ccept \cup \mathcal{R}eject$.
- I is a power of B, and represents the set of inputs of the machine.
- O is a power of B, and represents the set of outputs of the machine.
- $\delta : S \times I \to S$, δ represents the next state function. $\delta_i : S \times I \to B$ is the transition function of the state variable s_i.
- $\lambda : S \times I \to O$, λ represents the output function. $\lambda_i : S \times I \to B$ is the output function of the variable o_i.

Any state of the machine is binary encoded into some valuation of the state variables of the model. Two states are equal of they are represented by the same valuation.

A machine *configuration* is represented by a unique valuation $c = (s, i, o)$ of the variables of the model, where s is the current state, and i is the input, such that $o = \lambda(s, i)$. If no ambiguity exists, an arbitrary configuration associated to the initial state can be denoted by $c_0 = s_0$.

A machine state s' is a *successor* of a machine state s, if and only if: $\exists i \in I, s' = \delta(s, i)$. This relation can be denoted with the *Succ* predicate: $Succ(s, s')$.

A *state path* is a possibly infinite sequence of machine states, $(s_0, s_1, ..., s_n, ...)$, such that $Succ(s_i, s_{i+1})$. For a finite path $(s_0, ..., s_n)$, the length of the path is n. Similarly, a *configuration path* is a possibly infinite sequence of machine configurations $(c_0, c_1, ..., c_n, ...)$, such that $Succ(c_i, c_{i+1})$.

A machine state s_n is *reachable* if there is a finite path $(s_0, s_1, ..., s_n)$. In other words, a reachable state is a state that is reachable for some input sequences from a given set of possible initial states.

The set of the reachable states is defined inductively as follows:

$$R_0 = s_0 \tag{2.1}$$

$$R_{n+1} = R_n \cup \{s' | \exists s, s \in R_n \wedge Succ(s, s')\} \tag{2.2}$$

The machine has a finite number of states; therefore, there exists a k such that $R_{k+1} = R_k$. R_k is the set of reachable states. It is the smallest fixed point of (2.2).

2.2.2.2 Regular Expression

Here, the *Regular Expressions* (REs) are considered in the contexts of language theory, and also system specification.

Language theory viewpoint. Regular expressions define formal languages as sets of strings over a finite alphabet. An *alphabet*, Σ, is a finite set of symbols that form words in a language. For example the set $\{0, 1\}$ is an alphabet. A *string* (word) over Σ is several number, or zero, elements of Σ that are placed in order. For example, $w =$ ”001” is a string over $\Sigma = \{0, 1\}$. The *null string*, denoted by ϵ, is always a string over Σ (no matter what Σ is). For an alphabet Σ, Σ^* shows the set of all possible strings over Σ. A *language* over Σ, $L(\Sigma) \subset \Sigma^*$, is a set of strings over Σ. New languages can be constructed from existing ones by applying these three operations: union, concatenation, and closure.

Starting from the simplest possible languages, consisting a single string with length 1 or the ϵ string, and then applying any combination of the above operators, *regular languages* can be constructed. Regular languages can be recognized by FSMs. A string w is accepted by state machine M inductively:

$$\begin{array}{lll}
w = \ell, & s_1 = \delta(s_0, \ell), & \textit{and } s_1 \textit{ is an accepting state} \\
w = \ell_0\ell_1 \ldots \ell_{n-1}, & s_1 = \delta(s_0, \ell_0) & \\
 & s_2 = \delta(s_1, \ell_1) & \\
 & \ldots & \\
 & s_n = \delta(s_{n-1}, \ell_{n-1}), & \textit{and } s_n \textit{ is an accepting state}
\end{array}$$

The language accepted or recognized by M, $L(M)$, is the set of all strings w accepted by M: for each string w there is a finite state path $(s_0, \ldots, s_{|w|})$ such that $s_{|w|} \in \mathcal{A}ccept$. Regular languages can be described by formulas called *Regular Expressions* (REs).

Definition 1. *RE. A regular expression over the alphabet Σ is defined as follow:*

1 $\emptyset$ is a regular expression that corresponds to the empty language $\emptyset$.

2 ϵ is a regular expression that corresponds to the language $\{\epsilon\}$.

3 For each symbol $l \in \Sigma$, l is a regular expression corresponding to the language $\{l\}$.

4 For any regular expressions $p \in L(p)$ and $q \in L(q)$ over Σ ($L(p)$ and $L(q)$ are the corresponding languages to p and q), each of the following is a regular expression:

4-1 pq: corresponds to the language $L(p)L(q)$, and gives the concatenations *of the strings in the $L(p)$ and $L(q)$.*

4-2 $p + q$: corresponds to $L(p) \cup L(q)$.

4-3 p^*: corresponds to the language $L^*(p)$

Interpretation in the context of circuit specification. Let $\mathbf{P}$ be a non-empty set of atomic propositions. In practice it is the set of signal names in the specification. The set of all possible valuations of $\mathbf{P}$ is denoted $\Sigma = 2^{\mathbf{P}}$. An element of $\ell \in \Sigma$ is called "letter", it is a valuation of all the propositions in $\mathbf{P}$. A "word" w is a sequence of "letters": in practice it stands for the succession over time of the signal values, i.e. an execution trace. For a finite or infinite word $w = \ell_0\ell_1\ell_2...$ and integers i and j, $w^i = \ell_i$ is the $(i+1)^{th}$ letter of w; $w^{i..j} = \ell_i\ell_{i+1} \ldots \ell_j$ is the finite word starting at ℓ_i and ending at ℓ_j; $w^{i\cdots} = \ell_i\ell_{i+1} \ldots$ is the suffix of w starting at w^i.

The semantics of a Boolean expression *exp* over $\mathbf{P}$ is the set of all the letters of Σ on which *exp* takes value *true*. $\ell \vdash exp$ reads: "*exp* is true in ℓ", meaning that *exp* takes value true if all its variables take their value as in ℓ.

2.2.2.3 Temporal Logic

Temporal logic is a formal logic used to reason about sequences of events that describes the design behaviors over time. *Linear Temporal Logic* (LTL) and *Computational Tree Logic* (CTL) are the two types of temporal logics used in practice in circuit verification (there are many more). In LTL, operators are provided for describing system behavior along a single computation path. LTL properties can be checked both in simulation and formal verification. CTL models behavior as execution trees, which can be only checked in formal verification. The building blocks of both logics are the atomic propositions. LTL formulas are inductively defined as follows.

Definition 2. *LTL. All APs are LTL formulas; if p and q are LTL formulas, the followings are also LTL formulas:*

- $!p$*: is true iff p is false*
- $p \wedge q$*: is true iff p and q are both true*
- $p \vee q$*: is true iff either p or q is true*
- Xp*: is true iff p is true in the next step*
- pUq*: is true iff p is true until q is true and q must be eventually true*
- pWq*: is true iff p is true as long as q is not true, and q does not have to be true in the future*

There are also shorthand ways of expressing commonly used formulas: Fq (q becomes true eventually) stands for $trueUq$, and Gp (p is always true) stands for $!F\ !q$.

CTL was first proposed by Clark and Emerson as a branching-time temporal logic [CE82]. CTL formulas are composed of path quantifiers and temporal operators. The path quantifiers are used to describe the branching structure in the computation tree. There are two path quantifiers:

- A:“for all” paths,
- E: there “exists” a path or for “some” paths

There are four basic temporal operators in CTL:

- X: next time
- F: eventually or in the future
- G: always or globally
- U: until

In CTL, every quantifier is followed by a temporal operator. Therefore, there are eight basic CTL operators. Using the relations between the path quantifiers and temporal operators, each of the eight basic CTL operators can be expressed in terms of only three operators: EX, EG, and EU [LT10].

2.2.2.4 Model checking

The model checking technique was originally developed in 1981 by Clarke, Emerson, and Sifakis [CE82, Sif82]. Model checking is an automatic technique for verifying finite-state reactive systems, such as sequential circuit designs and communication protocols. The requirements of model checking are a model of the system, a temporal logic framework, and a model checking procedure. Briefly, model checking has the following steps:

1 Modeling the system as a *state-transition graph*, in which nodes are the states of the system and the edges are the transitions of the system.

2 Expressing the system specification, which may initially be in a natural language, in temporal logic

3 Verifying if the model satisfies the properties: the model checker will terminate with the answer true, if the model satisfies the specification, or give a counterexample that shows why the formula is not satisfied.

The properties are generally *safety* and *liveness* properties. Safety means that nothing bad ever happens. In this case, the verification problem is a reachability problem: finding a trace which violates the property. Liveness means a good thing eventually happens; the verification problem is cycle detection: finding a run in which the "good thing" is postponed indefinitely.

Model checking can be done *explicitly* where all the state space is enumerated. State space describes all the possible behaviors of the model. Therefore, it is impossible to handle very large examples because there is an exponential relationship between the number of the states and the number of memory elements in a system. The complexity of the algorithms grows exponentially with the number of memory elements in a system. This problem is called the *state space explosion* problem.

Model checking can also be done *implicitly*, by representing the state space with special symbolic data structures such as BDDs. Implicit (symbolic) model checking, is usually more powerful. Recently, by the application of propositional satisfiability (SAT) [Cim08] solving techniques, model checking has been considerably enhanced. *Bounded Model Checking* (BMC) is a model checking approach that uses SAT methods on a finite number of cycles. In the rest of this section, various model checking methods are explained briefly.

Enumerative model checking

The enumerative method uses an explicit representation of the states. To prove the properties, first a global state machine that represents the combined behavior of all the components of the system should be constructed. Then, through explicit state search and checking one state at a time, model checkers search for a trace falsifying the specification. If there is such trace, the property does not hold.

The model checking algorithms for LTL and CTL are different; the computational complexity of CTL model checking is polynomial, while it is exponential for LTL. Let |M| be the size of the system model in terms of state space and $|\varphi|$ indicate the size of the specification (the total number of propositions, logical connectives, and temporal operators). Then the model checking algorithm for CTL runs in time $O(|M||\varphi|)$ [CES86], while for LTL it runs in time $|M|\cdot 2^{O(\varphi)}$ [OLP85]. This is because CTL is state-based (i.e. reasoning over states in time), and this set of states is easily converted into an automaton, whereas the path-based model of LTL (where many possible paths may pass through a single state) must be expanded.

Symbolic model checking

Symbolic model checking is an alternative approach to enumerative model checking that operates on sets of states instead of individual states [McM93]. In contrast to the enumerative model checking that considers one state in each step, this considers a set of states (a symbol) in each step. Initially, *Binary Decision Diagram* (BDD) was the only method used to realize symbolic model-checking systems, and symbolic model checking had been synonymous with BDD-based model checking. The overall approach in BDD-based symbolic model checking is to represent state sets and the transition relation of the FSM as BDDs and realize the state traversal algorithms through suitable Boolean operations on these BDDs [PH09]. The complexity of symbolic model checking is the complexity of BDD operations, and its efficiency highly depends on the state space representation.

Bounded model checking

Bounded model checking was introduced by Biere *et al.* in [BCC+99]. It is based on satisfiability (SAT) methods. The method is mainly used for design error detection instead of an approach for a full correctness proof. The essential idea for verifying a property on a finite transition system is to search for a counter examples in the space of all executions of the system whose length is bounded by some integer k. There are two steps in bounded model checking: 1) encoding the sequential behavior of a transition system over a finite interval as a propositional formula, and 2) using a propositional decision procedure, i.e. a satisfiability solver, to either obtain a satisfying assignment or to prove there is none. Briefly, the transition relations are unrolled symbolically up to k times steps. Then, the checking problem is reduced to show satisfiability of an expression using SAT solvers.

Bounded model checking has been implemented by many commercial tools; PROVER[Bor97], SATO[Zha97], GRASP[MS95], EBMC[EBM], SATRennesPA[SAT], MiniSAT[MIN], MaxSAT[MJML14], and Z3[MB08] are some examples of SAT solvers.

Language containment

Language containment treats the property and the design as two finite state automata. Then, it is verified if the formal language of the property automaton contains the formal language of the design automaton. In fact, model checking of properties in LTL is modeled as a language containment problem.

2.2.2.5 Equivalence checking

Equivalence checking is a model-based method that checks if two descriptions of a design specify the same behavior, which means that they produce identical output sequences for all valid input sequences. These descriptions can be in different abstraction levels.

There are three basic approaches for combinational equivalence checking: *structural*, *functional*, and *random simulation*. The structural methods look for a counter-example, and they are usually implemented using SAT solvers. Similarly, random simulation looks for a counter-example by random search. Functional methods are based on a canonical function representation. BDDs are widely used for functional methods.

General methods for sequential equivalence require the reachable states of both designs to be computed and their corresponding outputs compared at each equivalent pair of states. It is required to explore the state space of the circuits. However, performing such a traversal is computationally expensive, and has the state space explosion problem, the main disadvantage of this method. However, the equivalence checking tools provide a high degree of automation.

2.2.2.6 Theorem proving methods

Theorem proving approach is used where the verification problem is described as a theorem in a formal theory: both the design and the properties are expressed as formulas using mathematical logic. A formal theory consists of a language in which the formulas are written, a set of axioms, and a set of inference rules, which are the transformation rules for the formulas. These rules together with the axioms are used for proving the theories. A property is proved if it can be derived from the design in a logical system of axioms and a set of inference rules.

Theorem proving is a very strong verification method, since the formal theory can support reasoning at all the levels of abstraction. In addition, it supports a powerful proof technique such as induction, and allows the direct verification of parametric designs without having to instantiate the parameters. However, the main disadvantage is that in contrast to all the previously mentioned methods, the verification process is not totally automatic. Theorem provers require detailed and explicit human guidance even for relatively simple problems. Therefore, these methods need a deep understanding of the design and formal proofs.

Although, this approach seems impractical, there are several successful real case studies such as Motorola MC68020 microprocessor object code [BY96], the AMD K86's division algorithm [KMB97], and the SRT division algorithm [CGZ99].

Logic for Computable Functions (LCF) [Mil72], Higher-Order Logic (HOL) [Gor88], Otter [McC03], Prototype Verification System (PVS) [ORS92], and ACL2 [KM96] are some examples of theorem provers.

2.3 Assertion languages

As was discussed earlier in this chapter, all the formal methods, and also assertion-based verification need a formal specification of the design. In addition, it is necessary to have an approach in order to communicate with the designer and understand the design structure and its functionality. There are various approaches to achieve these requirements. For instances, schematics have been used to specify the structure, programming languages have been used to specify the behavior, and timing diagrams involving waveforms have been used to specify timing information. In all of these types of specifications, the natural language is used [Mil94]. However, describing the behavior informally is usually ambiguous, incomplete, and hard to analyze. Moreover, there is always the risk that some scenarios and properties are not covered or considered.

A formal specification is a concise and abstract description of the behavior and properties of a system written in a mathematically based-language, stating what a system is supposed to do. So, specifications are written in a language with a well-defined semantics that supports formal deduction.

Our focus is on assertions, which state properties that can be used both in simulation and formal verification. Some of the requirements for the assertion languages are the following [Mil94]:

- The ability to represent the structure
- The ability to represent the concurrent and sequential behaviors
- Supporting hierarchical design

- The ability of presenting a design in different abstraction levels
- The ability of mixing the design description of different abstraction levels
- Supporting the simulation and verification techniques by the presence of an underlying formal system giving a semantics and the language syntax

Not long ago most verification activities were performed using Hardware Description Languages (HDLs). HDLs include constructs that support assertion specification. For instance, VHDL includes a keyword "assert" that enables designers to embed some checkers to model description code. This language construct expresses that the associated user-specified condition should evaluate to *true*. Figure 2.1 shows a VHDL assertion that fires when (*not req and gnt*) evaluates to *true*.

```
assert not (not req and gnt) report "Grant received without any request"
    severity failure;
```

FIGURE 2.1: A VHDL assertion example

However, HDLs are not appropriate specification formalism for the formal verification techniques. As the verification problem began to grow, High Level Verification (HLV) languages emerged. Multiple verification oriented languages such as IBM Sugar [BBDE+01], Motorola CBV [AAH+03], Intel ForSpec [AFF+02], Synopsys Open Vera Assertion (OVA) [OVA], Open Verification Library (OVL) [OVL], System Verilog Assertion (SVA) [SMB+05], and Property Specification Language (PSL) [FG05] have been developed.

Here, we review the PSL and SVA assertion languages, particularly useful for their deep embedding in the VHDL and SystemVerilog HDLs.

2.3.1 Property Specification Language (PSL)

Property Specification Language (PSL) [FG05] is the standardization by Accelera, then by IEEE, of the Sugar property language originally developed by IBM [BBDE+01]. Like Sugar, it includes and extends with more concise operators, both LTL and CTL.

A specification written in PSL is both easy to read and mathematically precise, which makes it ideal for both documentation and verification. PSL can be used both in simulation and formal verification. Unlike the SystemVerilog assertion construct, which are used predominantly during RTL implementation, the PSL property language is suited for specifying architectural properties before and during RTL implementation.

PSL comes in four flavors, one for each of the hardware description languages SystemVerilog, Verilog, VHDL, and GDL. The syntax of each flavor conforms to the syntax of the corresponding HDL.

The properties that are expressed using PSL are generally safety and liveness properties. For example, the property "whenever signal *req* is asserted, signal *gnt* is asserted within 4 cycles" is a safety property; and, the property "whenever signal *req* is asserted, signal *gnt* is asserted sometime in the future" is a liveness property.

A PSL property consists of four layers: *Boolean*, *temporal*, *verification*, and *modeling* (see Fig. 2.2). Here, each layer is introduced briefly.

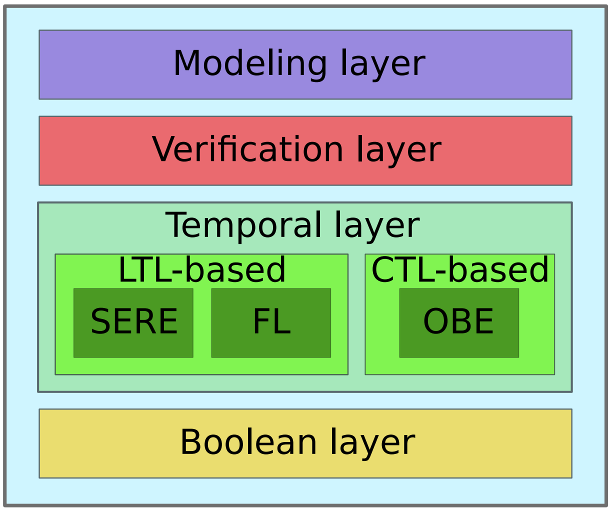


FIGURE 2.2: PSL layers

2.3.1.1 PSL Boolean layer

This layer specifies propositions, or expressions over design and auxiliary signals that evaluate to true or false in a single evaluation cycle. The expressions are written in the HDL that describes the design. This is equivalent to a condition being evaluated within an if statement in Verilog or VHDL. Additionally, PSL provides a number of predefined functions. There are two classes of built-in functions: the first group including `prev`, `next`(), `stable`(), `rose`(), and `fell`() deal with the value of the expression over time. The second group including `isunknown`(), `countones`(), and `onehot`() deals with the values of bits in a vector at a given instant. For example, `prev` takes an expression of any type as argument and returns a previous value of that expression. As another example of the first group, consider `rose`(): it takes a Bit expression as argument and produces a Boolean result that is true if the argument's value is 1 at the current cycle and 0 at the previous cycle. As an example of the second group, the `countones`() function takes a BitVector as argument. It returns a count of the number of bits in the argument that have the value 1.

2.3.1.2 PSL temporal layer

The temporal layer is the heart of PSL. The temporal layer is used to define properties that describe the behavior of the design or environment over time. It is used to describe the temporal behaviors built up with Boolean layer propositions and temporal operators. This layer consists of both properties that use linear semantics (LTL-based) as well as those that use branching semantics (CTL-based). The LTL-based subset includes the *Foundation Language (FL)* and *Sequential Extended Regular Expression* (SERE), while the CTL-based subset includes the Operational Branching Extension (OBE) (see Fig. 2.2). FL and OBE cannot be mixed in one property.

Properties with linear semantics reason about computation paths in a design and can be checked in simulation, as well as in formal verification. Properties with branching semantics reason about computation trees and can be checked only in formal verification. Here, we just consider the properties with linear semantics.

Sequential Extended Regular Expressions (SEREs)

Sequential Extended Regular Expression (SERE) is an extension to RE introduced in Section 2.2.2.2. SEREs describe single- or multi-cycle behavior built from a series of Boolean expressions. The most basic SERE is a Boolean expression. A SERE enclosed in braces is another form of a sequence. A SERE is not a property on its own; it is a building block of a property; properties are built from temporal operators applied to SEREs and Boolean expressions. Table 2.1 gives the description of the SERE operators. In this table, `s` represents a sequence, and `b` represents a Boolean.

The SERE operators can be categorized as follow:

1 Simple SERE: represent a single thread of subordinate behaviors, occurring in successive cycles. This subset of SEREs consists of the ‘;’ and ‘:’ operators.

2 Compound SERE: represent a set of one or more threads of subordinate behaviors, starting from the same cycle, and occurring in parallel. This subset of SEREs consists of the ‘|’, ‘&’, “&&”, and `within` operators.

Here, we introduce some terms that relate to SEREs:

- *hold tightly*: Satisfaction of a SERE on a finite path requires an exact match, and is referred to as the SERE *holds tightly* on the finite path. For example, `SERE1` (see Fig 2.3) holds tightly on a path iff the path is of length six, where *req* is *true* in the first cycle, *busy* is *true* in cycles ♯2, ♯3, ♯4, ♯5, and *gnt* is true in cycle ♯6.

- *hold*: A weak sequence *holds* on a path iff the corresponding SERE holds tightly on an extension or on a prefix of the path. A strong sequence *holds* on a path iff the corresponding SERE holds tightly on a prefix of the path [FKL03]. For example, `SERE1` holds if *req* holds on the one-cycle path. `SERE2` is the strong form of `SERE1`. `SERE2` does not hold in the one-cycle path. `SERE2` holds, if *req* is followed by 4 repetitions of *busy*, which is followed by *gnt*.

- *start*: A sequential expression starts at the first cycle of any behavior for which it holds. In addition, a sequential expression starts at the first cycle of any behavior that is the prefix of a behavior for which it holds. For example, if *req* holds at cycle ♯1 and *busy* holds from cycle ♯2 to cycle ♯5, and *gnt* holds at cycle ♯6, then the sequential expression `SERE1` starts at cycle ♯1.

- *completes*: A sequential expression completes at the last cycle of any design behavior on which it holds tightly. For example, `SERE1` completes at cycle ♯6 (see Fig. 2.3).

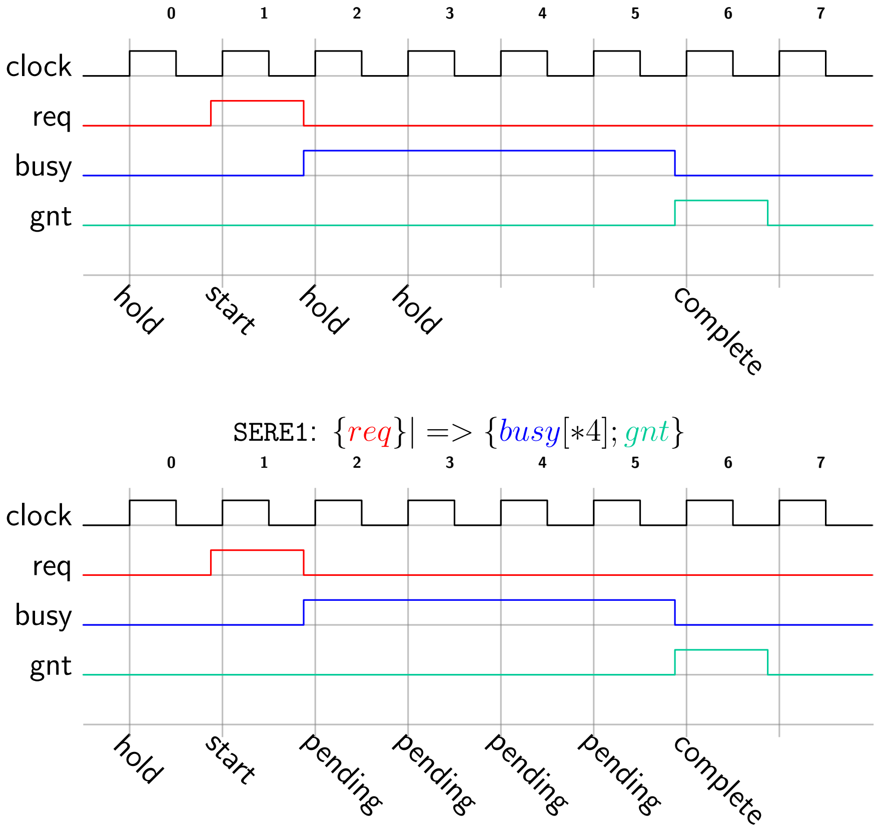
For finding out more about the formal syntax and semantics, refer to IEEE manual of PSL [FG05].

Foundation Language (FL)

The Foundation Language (FL) of PSL is LTL that is extended with SERE [FG05]. A PSL FL property can be compiled down to a LTL formula, possibly with some auxiliary HDL code. FL properties, describe single- or multi-cycle behavior built from Boolean expressions, sequential expressions, and subordinate properties. The most basic FL property is a Boolean expression. An FL Property enclosed in parentheses is also an FL property.

Table 2.1: Definition of the SERE operators

SERE operator	Name	Description
`s1`; `s2`	concatenation	`s2` starts one cycle after `s1` completes
`s1` : `s2`	fusion	`s2` starts in the cycle that `s1` completes
$[*n]$	count consecutive repetition	skips n cycles
$[*]$	consecutive repetition	skips 0 or more cycles
$[+]$	consecutive repetition	skips 1 or more cycles
`s`$[*n]$	count consecutive repetition	`s` repeats n times consecutively (n concatenations of `s`)
`s`$[*]$	consecutive repetition	`s` repeats 0 or more times consecutively
`s`$[+]$	consecutive repetition	`s` repeats 1 or more times consecutively
`s`$[*m\ to\ n]$	consecutive repetition	`s` repeats between m and n times consecutively
`b`$[= n]$	nonconsecutive repetition	`b` repeats n times, not necessarily consecutively
`b`$[= m\ to\ n]$	nonconsecutive repetition	`b` repeats between m and n times
`b`$[->n]$	Goto repetition	`b` repeats n times, the last `b` occurs at the end of the path
`b`$[->m\ to\ n]$	Goto repetition	Boolean repeats between m and n times, the last `b` occurs at the end of the path
`s1` | `s2`	or	holds if either `s1` or `s2` holds
`s1` & `s2`	non-length-matching and	`s1` and `s2` hold at the point of observation, they have the same starting point, and may complete in different cycles
`s1` && `s2`	length-matching and	`s1` and `s2` hold at the point of observation, they have the same starting point, and should complete in the same cycles
`s1 within s2`		`s2` contains `s1`, `s2` holds at the point of observation, `s1` starts at or after the cycle in which `s2` starts; `s1` completes at or before the cycle in which `s2` completes
`s1` |->`s2`	suffix implication	`s2` starts at the ending cycle of `s1`
`s1` |=>`s2`	suffix next implication	`s2` starts the cycle after the ending cycle of `s1`



`SERE1`: $\{req\}| => \{busy[*4]; gnt\}$

`SERE2`: $\{req\}| => \{busy[*4]; gnt\}!$

FIGURE 2.3: The trace for property `SERE1` and `SERE2`

FL properties can be connected using the logical unary (not), binary (or and and), and implication operators and generate more complex FL formulas. In addition to the logical operators, more complex FL properties are built from Boolean expressions, sequential expressions, and subordinate properties using various temporal operators. These operators, consistent to the VHDL flavor of PSL, are shown in Table 2.2. In this table, p represents an FL property, and b represents a Boolean.

Table 2.2: Definition of the FL temporal operators (in VHDL flavor)

FL operator	Description
eventually! p	p holds eventually (it holds some time in the future)
always p	p must hold at all times
p abort b	p holds unless b evaluates to *true* first
never p	p must never hold
next(p)	p holds in the next cycle
next[*n*](p)	p holds *n* cycles later
next_a[*m to n*](p)	p holds in all the cycles in the range
next_e[*m to n*](p)	p holds in at least one cycle in the range
next_event(b)[*n*](p)	p holds at the n^{th} occurrence of b
next_event_a(b)[*m to n*](p)	p holds in all the cycles in the specified range of occurrences of b
next_event_e(b)[*m to n*](p)	p holds in at least once in range of occurrences of b
p1 until p2	p2 holds up to the cycle p2 holds (exclusive)
p1 until_ p2	p1 holds up to and including the cycle p2 holds (inclusive)
p1 before p2	p1 holds before p2 holds (exclusive)
p1 before_ p2	p1 holds before and at the same cycle as p2 holds (inclusive)

In this table, eventually! and abort are *strong* operators, and all the other operators are *weak* operators. Strong operators require the ending condition to eventually occur, while the weak operators do not. Each of the weak operators that are listed in Table 2.2, except for always has a strong version with the indicator '!' appended to its keyword.

The FL operators can be categorized as the following groups:

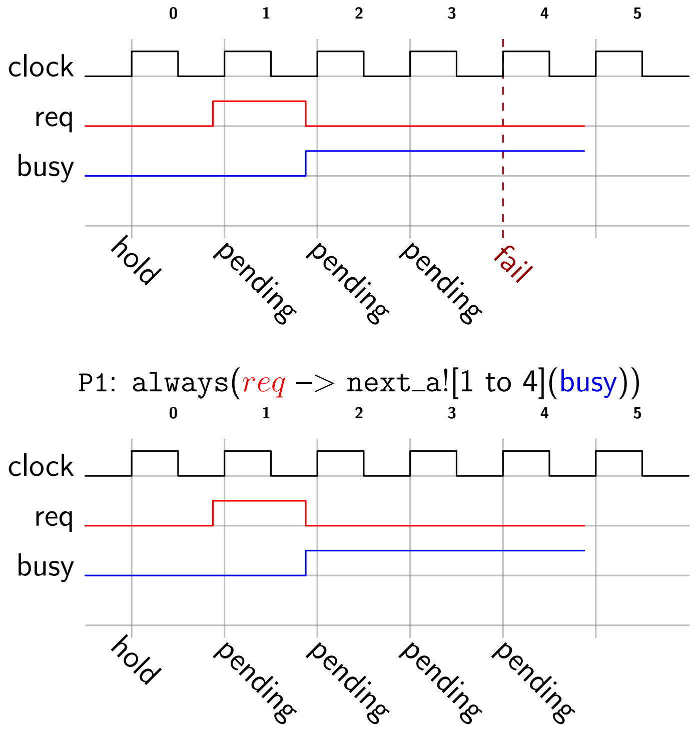
1 Simple FL properties include the always, never, eventually!, and next! operators.

2 Extended next FL properties include the next_a, next_e, next_event, next_event_a, and next_event_e operators.

3 Compound FL properties include the abort, before family, and until family operators.

4 Sequence-based FL properties include the suffix implication and suffix next implication operators.

5 Logical FL properties include the logical implication, logical iff, and, or, and not operators.



P1: always(*req* –> next_a![1 to 4](busy))

P2: always(*req* –> next_a[1 to 4](busy))

FIGURE 2.4: The difference between PSL weak and strong operators

6 LTL operators

PSL defines four levels of satisfaction of a property [FG05]

- *holds strongly*: 1) no bad states have been seen, 2) all future obligations have been met, and 3) the property will hold on any extensions of the path
- *holds*: 1) no bad states have been seen, 2) all future obligations have been met, and 3) the property may or may not hold on any given extensions of the path
- *pending*: 1) no bad states have been seen, 2) future obligations have not been met, and 3) the property may or may not hold on any extensions of the path
- *fails*: 1) a bad state has been seen, 2) future obligations may or may not have been met, and 3) the property will not hold on any extensions of the path

A property that is defined with a weak operator holds if the computation path is truncated inappropriately before the expected cycles or events can happen. For example, consider property P2 and its simulation trace in Fig. 2.4. If the simulation stops at cycle ♯4, the property holds, but not strongly. If the simulation continues and *busy* remains high, P2 holds strongly at cycle ♯6.

The strong operators demand that the property holds unconditionally. As an example, property P1 (see Fig. 2.4) fails if the simulation stops at cycle ♯4.

In this work we focus on the temporal layer of PSL. At the end of this section, we will introduce a simple subset of PSL, PSL$_{simple}$, that we can deal with in our synthesis method.

2.3.1.3 PSL verification layer

The verification layer tells the verification tools what to do with the properties described by the temporal layer. In addition, the verification layer provides constructs that group related directives and other PSL statements.

Verification directives

There are seven verification directives: `assert`, `assume`, `assume_ guarantee`, `restrict`, `restrict_ guarantee`, `cover`, and `fairness`. Here, some verification directives are explained.

- The `assert` directive: tells the verification tool to verify that a property holds.
- The `assume` directive: tells the verification tool to constrain the verification (e.g., the behavior of the input signals) so that a property holds. Assumptions are often used to specify the operating conditions of a property by constraining the behavior of the design inputs.
- The `cover` directive: tells the tool to indicate if a property has been exercised by the test inputs or given constraints.

The other directives are meaningful in the context of formal verification only, and are not recalled here.

Verification units

PSL statements can be used individually in the code, or they can be grouped into the *verification units*. There are three types of verification units: `vprop`, `vmode`, and `vunit`. `vprop` groups assertions to be verified. `vmode` groups the constraints with the `assume`/ `restrict` directives. Finally, `vunit` combines the two, which enables grouping assertions and assumptions together. Verification units may also contain modeling layer constructs that are used by the assertions or constraints. In addition, a verification unit can inherit other verification unit by using the *inherit* statement [EF06].

2.3.1.4 PSL Modeling layer

The modeling layer makes it possible to model the behavior of design inputs, and to declare and give behavior to auxiliary signals and variables. The modeling layer enables writing some extra code from the underlying language to model auxiliary combinational signals, state machines etc. that are not part of the actual design but are required to express the property concisely. For example, the modeling layer could be used to provide an input. The Verilog (VHDL) flavor of the modeling layer consists of the synthesizable subset of Verilog (VHDL) [FG05].

Figure 2.5 shows the four layers of a PSL property that have been discussed.

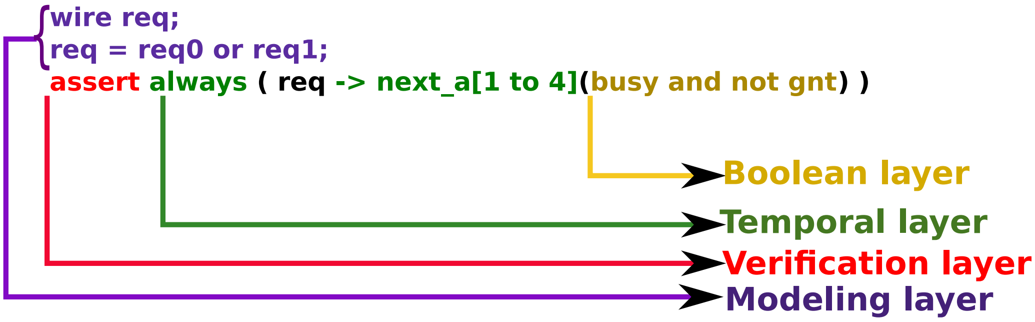


FIGURE 2.5: Different layers of a PSL property

2.3.1.5 PSL simple subset (PSL$_{simple}$)

PSL can express properties that cannot be evaluated in simulation, where time advances monotonically along a single path, although such properties can be addressed by formal verification methods. The simple subset of PSL, PSL$_{simple}$, is a subset that conforms to the notion of monotonic advancement of time, left to right through the property, which ensures that properties within the subset can be simulated easily. Any FL property in the simple subset should meet all of the following conditions [FG05]:

- The operand of a negation operator is a Boolean.
- The operand of a `never` operator is a Boolean or a sequence.
- The operand of an `eventually`! operator is a Boolean or a sequence.
- The left-hand side operand of a logical `and` operator is a Boolean.
- The left-hand side operand of a logical `or` operator is a Boolean.
- The left-hand side operand of a logical implication (–>) operator is a Boolean.
- Both operands of a logical iff (<–>) operator are Boolean.
- The right-hand side operand of a non-overlapping `until` operator (`until` and `until`!) is a Boolean.
- Both operands of an overlapping `until` operator (`until_` and `until_`!) are Boolean.
- Both operands of the `before` family operators are Boolean.

All other operators not mentioned above are supported in the simple subset without restriction. In particular, all of the `next_event` operators and all forms of suffix implication are supported in the simple subset.

2.3.2 System Verilog Assertion (SVA)

SystemVerilog [SMB$^+$05] has integrated a set of constructs that helps to specify a system behavior using assertions. SystemVerilog assertions are part of the language, which means that they can be used inline with other language constructs. SVA was defined at the same time as PSL, also based on the concepts and semantics of Sugar, restricted to SEREs. With a different syntax, it shares most of its basic primitives with PSL.

SystemVerilog assertions are either *immediate* or *concurrent*.

Immediate assertions follow simulation event semantics for their execution. They describe a design behavior at an instant of time. An immediate assertion is evaluated whenever the value of a variable in the expression changes. These assertions are executed like a statement in a procedural block. Immediate assertions are an easy way to create an assertion and are generally used with simulation.

Concurrent assertions are based on clock semantics and use sampled values of variables. They specify a design behavior over a period of time. Concurrent assertions are associated with clock edges. A concurrent assertion is evaluated right before the clock edge, and any timing or event behavior between clock edges is ignored. A concurrent assertion can occur within a procedural block or within a module.

This section provides an overview of SystemVerilog Assertion (SVA).

2.3.2.1 Operators

Table 2.3 shows the SystemVerilog operators and their descriptions. The SystemVerilog operators are available for relating Boolean and vector expressions within sequence and property definition [FKL03]. A SystemVerilog sequence is often described using regular expressions. The sequence operators that are defined for SystemVerilog allow us to compose expressions into temporal sequences. These sequences are the building blocks of properties and concurrent assertions [FKL03].

As is shown in Table 2.3, the repetition counts and temporal delay can be specified as either a range or a single constant expression.

2.3.2.2 Verification directives

Property directives define how to use properties (and sequences) for specific works. SVA has three verification directives: *assert*, *assume*, and *cover*. These directives are similar to the *assert*, *assume*, and *cover* verification directives of PSL (see Section 2.3.1.3).

2.3.2.3 Built-in functions

Assertions are commonly used to evaluate certain specific characteristics of a design implementation, such as whether a particular signal is onehot. The following system functions are included to facilitate this common assertion functionality:

- `$onehot`(): returns *true* when exactly one bit of a multi-bit expression is one.
- `$onehot0`(): returns *true* when zero or one bit of a multi-bit expression is one.
- `$stable`(): returns *true* when the previous value of the expression is the same as the current value of the expression.
- `$rose`(): returns *true* when an expression was previously zero and the current value is one.
- `$fell`(): returns *true* when an expression was previously one and the current value is zero.

Table 2.3: Definition of the SVA operators

SVA operator	Name	Description
`s1`[$*m : n$]	consecutive repetition	repetition of `s1` n times, or between n to m times
`s1`[$= m : n$]	nonconsecutive repetition	`s1` repeats between m and n times
`s1`[$->m : n$]	Goto repetition	`s1` repeats between m and n times, the last cycle of `s1` occurs at the end of the path
`s1`##[$m : n$] `s2`	temporal delay	concatenation of `s1` and `s2` with a delay between m and n
`not p1`	logical not	inverts the result of the evaluation of `p1`
`s1 and s2`	and	is similar to the non-length-matching and of PSL
`s1 intersect s2`	intersection	is similar to the length-matching and of PSL
`s1 or s2`	or	Either `s1` or `s2` occurs
`if` (*exp*) `p1 else p2`	condition	based on the evaluation of *expr*, evaluates property `p1` or `p2`
`b throughout s1`	Boolean until	`b` must be *true* until sequence `s1` completes
`s1 within s2`	within	`s2` contains `s1`, `s1` and `s2` must occur, the length of `s1` should be less than or equal than/to the size of `s2`, and `s1` may start later than `s2`
`s1.ended`	ended	is *true* if sequence `s1` completes at this time
`s1.matched`	matched (from different clock domains)	is *true* is sequence `s1` (on another clock) completes at this time
`first_matched`(`s1`)	first match	is *true* in the first completion of `s1`
`s1`|-> `p1`	overlapping implication	if `s1` occurs, `p1` must occur starting at the ending cycle of `s1`
`s1`|=> `p1`	non-overlapping implication	if `s1` occurs, `p1` must occur starting the cycle after the ending cycle of `s1`

2.4 Summary

In this chapter, the importance of the verification and its role in today's design process is discussed. The dynamic and static verification techniques are introduced. Although dynamic verification is often the first step of verifying a circuit, it is not exhaustive. Static verification is an alternative approach to verify all the possible scenarios of a system and prove its correctness using mathematical proofs. Generally, formal methods are either based on theorem proving or based on the model of the system (model-based techniques). A design methodology that is becoming popular is assertion-based verification that benefits of both dynamic and static methods. It uses assertions that can be simulated and also provide a path to formal verification. An assertion language is required to specify the system behavior concisely and unambiguously. In this chapter two assertion languages are introduced: PSL and SVA. The core of both languages is temporal logic. However they are different in some aspects. PSL is divided into the FL and OBE. SVA is a linear temporal logic that can be compared to the FL of PSL, while SVA does not have most of the FL operators. In the rest of the document, only PSL will be used as input specification language. Yet the methods developed in the thesis are applicable to SVA as well.

Chapter 3

State of the art

Contents

3.1 Introduction

In this chapter we review some works in the area of assertion-based verification and design.

In Section 3.2 the existing methods, modular and automaton-based, for synthesizing checkers are considered. Section 3.3 reviews the related works in synthesizing a design, a correct-by-construction circuit, from its formal specification.

Finally, the existing tools for automatically generating the checkers or synthesizing a design from its specifications are introduced in Section 3.4.

3.2 Property synthesis as monitors

A monitor surveys the state of the design during simulation. It observes the signals that are operands in a property, and outputs the status of the property. Therefore, all the operand variables are inputs of the monitor. Monitors are generated either in a modular way or from automata. Here, each of these methods is explained.

3.2.1 The automaton-based approach

In this method all the simulation traces are considered as the words of a language built over the alphabet Σ of all the possible combinations of values of the design variables (each valuation is a *letter* of Σ). Monitors are finite state machines that accept or reject certain simulation traces. Some states in the monitor are initial states, some of them are *accepting* states, and some of them are *rejecting* states. A simulation trace that drives the monitor into an accepting state exhibits a good behavior. In the automaton-based method, monitors use the "language-theoretic" concept to analyze the formal languages. The automaton's transitions are labeled with letters from the alphabet. A state may have several outgoing transitions labeled with the same letter. A word runs over the automaton by starting from all initial states, following the transitions that correspond to the sequence of letters in the word. Since each state may have several outgoing transitions labeled with the same letter, there may be several paths for a word. In the case of Regular Expression (RE), if the final state on any of the paths is an accepting state, then the word is accepted. For Linear Temporal Logic (LTL), a word is accepted if it has an accepting state. If a word is not accepted, then it is rejected. However, it is hard to trace multiple paths, and testing the automaton accepting the condition for LTL is very difficult. To solve these problems, the automaton needs to be deterministic which means having one initial state, and each state has just one successor state for any letter.

The construction of language-recognizing automata for REs and LTL has a long history. Early RE to automata translations were given in [MY60] and [Tho68]. LTL to automata translation was considered in [WVS83]. The method is a *tableau-based* approach, in which the satisfaction of a temporal formula is decomposed both logically (across Boolean connectives) and temporally (obligations in the next time). However the tableau-based approach has some limitations. As an example, consider the negation of an RE. To negate an RE, its Non-deterministic Finite Automaton (NFA) should be constructed and should be converted to a Deterministic Finite Automaton (DFA). The negated RE is then constructed by complementing the DFA. Therefore, tableau-based approaches are not suitable for constructing the DFA of a negated RE.

Sidhu and Prasanna presented a hardware implementation of RE matchers for FPGA [SP01]. This approach uses the method of McNaughton-Yamada [MY60] for constructing NFAs from REs. In the proposed method, the actual NFA construction can be performed in hardware. Moreover, the Self-Reconfigurable Gate Array (SRGA) can be reconfigured automatically in real time to match the pattern of a new expression.

Floyd and Ullman worked on synthesizing REs into integrated circuit for hardware RE matching [FU82]. A regular expression can be converted into a NFA. Instead of converting the NFA to a DFA, two methods are proposed for direct implementation of the NFA. One approach is based on producing a Programmable Logic Array (PLA). The PLA has approximately n rows and $2n$ columns, where n is the number of RE's operands. The states of the NFA are represented by the columns. Another approach is using McNaughton-Yamada algorithm to produce automata from REs. The hierarchical structure of these automata can guide the layout structure of the circuit. The experimental results show that the area of the generated circuit grows linearly with the size of the regular expression.

Both works in [FU82] and [SP01] implement NFAs in hardware to perform RE matching. However, the intersection and complementation operators in REs are not supported.

Since the standardization of Property Specification Language (PSL), several works have been done to convert PSL to automata [GHS03, GG05, BFH05, CRST06]. Some of them propose a two-step conversion: 1) encoding the PSL property into an Alternating Büchi Automaton (ABA)[1]; 2) converting the ABA into a Non-deterministic Büchi Automaton (NBA) with variants of Miyano-Hayashi's construction [MY60]. In practice, such approaches are inefficient because of the conversion time.

Gordon *et al.* have modeled PSL in higher order logic for the HOL theorem prover [GHS03]. HOL can also be used to produce a DFA from a PSL expression. The DFA can be used to process a simulation trace in HOL to evaluate if a finite trace satisfies a PSL formula. In another application that is mentioned in [GHS03], a DFA can be converted to HDL to produce an assertion checker.

The "PROSYD" project has published methodologies for the use of PSL, and reports on the tools that are developed in the project [BCE$^+$04]. The PSL algorithms are introduced in the context of generating checkers for simulation. The conversion of an NFA to a Discrete Transition System (DTS) is presented as a central result. A DTS is a symbolic program that represents an NFA, and is used during simulation for performing the assertion monitoring. For the conversion of PSL assertions to NFAs it is referred to [BdFR04], in which the automata are developed for model checking. However, in [BCE$^+$04] it is not mentioned how these automata are adapted to be used in checkers. In [BdFR04], which is the basis for PSL to NFA in PROSYD, length-matching intersection of Sequential Extended Regular Expressions (SEREs) is not supported.

Gascard in [Gas05] proposes a method for transforming SEREs to DFAs. The work is based on derivatives of REs introduced by Brzozowski in [Brz64]. The derivative of a RE is a way of removing a given prefix in the language that is described by the RE. This technique can be used to create a DFA from a RE. Then, monitors are generated from DFAs. In most cases, monitors do not take sequence overlapping into account. In addition, no results have been provided, neither the construction time, nor the synthesis

[1] A Büchi automaton extends a finite automaton to infinite inputs. It accepts an infinite input sequence if there exists a run of the automaton that visits (at least) one of the final states infinitely often. It recognizes the ω-regular languages, the infinite word version of regular languages.

metrics.

The work presented in [GG05] deals with the translation of a subset of PSL SEREs into monitors. For each operator of this subset, a function is implemented that builds the corresponding non-deterministic automaton. The monitors can be generated in E and Verilog. This method is faster than the conversion method based on HOL [GHS03], but is slower than FoCs [ABG$^+$00].

In [CRST06] Cimatti *et al.* propose an effective method to transform a PSL property into a normal form that separates the LTL and the SERE components. Then, each of them is processed separately to generate the corresponding NBA of the original PSL property. The aim of this approach is principally automata construction for model-checking, but it is also possible to build monitors. This approach reduces the construction time of the NBA, as well as the overall verification time. In addition, the correctness of the transformation is proved.

To the best of our knowledge, the most effective approach in synthesizing monitors from PSL SEREs is done by Boule and Zilic [BZ07, BZ08b, BZ08c]. In this work, the SERE base cases are introduced. Then, the automata algorithms are developed for these cases. In addition, a complete set of rewrite rules has been proposed in [MABBZ08] and applied for all other operators, to rewrite them using the base cases. The automaton for a complex property is obtained by combining primitive automata. Using this method, the automata are constructed for the left and right hand side of an implication. Then, they are connected to represent the property by a single automaton. It is shown that the generated monitors are resource efficient. The approach also enhances debug capability.

The authors in [EFP09] introduce the SynPSL tool, and also a method similar to the method of Marc Boule for generating synthesizable HDL code from PSL assertions. However, it does not support general Boolean layer expressions, it can just consider simple Boolean expressions. The method does not support unbounded repetition in SEREs. In addition, it can just be applied to the *std_logic* type.

Despite all the attempts that have been put in this area, the automaton-based approach is still too expensive. Although there are approaches [SP01] for constructing NFAs using hardware, NFA is still inappropriate for hardware implementation, because of the large number of concurrent transitions required by NFA. In addition, transforming NFA to DFA is costly; it is exponential in the number of non-deterministic decision states.

Additionally, the automaton-based approaches generally indicate the status of the property at the end of the simulation, and cannot be used for debugging purposes. It would be more useful if the method could provide a dynamic trace of the assertion and indicate each assertion failure. This goal can be reached through the modular method.

3.2.2 The modular approach

In [Ray96], Raymond proposed a modular method for building a Boolean dataflow network (sequential circuit) to recognize the language described by a regular expression. A safety property that is expressed using regular expression constructs is translated into a synchronous program, such as Boolean network. A tool, reglo, has been designed that translates a set of REs into an equivalent Boolean dataflow network that is expressed in the Lustre language. The construction time, and the size of the resulting network are linear with respect to the size of the regular expression.

Oliviera and Hu worked on generating interface monitors for verifying the interconnection protocols between design modules [OH02]. The goal is providing an easy way to generate monitors for common interface protocols. This work demonstrates that although regular expressions work well for specifying simple IP interface monitors, they cannot be easily used to specify complex interfaces. To overcome this problem, two new extensions to REs have been proposed: defining storage variables, and a pipe-lining operator. Using these concepts, a specification style has been created that can easily specify the full behavior of complex IP interface monitors. This style is called PREMiS (Pipelined Regular Expression Monitor Specification). Then, algorithms are proposed, and a prototype tool is developed to translate these specification into Verilog/VHDL monitor circuits. The method is modular and works by passing the token from one sub-circuit to the next one. The usefulness of the method has been shown by applying it to ARM AMBA AHB bus protocol, and Open Core Protocol (OCP). However, neither the construction time nor the combinational synthesis metric have been reported.

Pellauer *et al.* worked on the implementation of System Verilog Assertion (SVA) assertion checkers [PLN05]. The "first-match" operator is used as a basis to implement sequences in the right-hand side of suffix implications. Checkers are produced in the BlueSpec SystemVerilog language. It is an unclocked language; its models are subsequently translated into sequential hardware. The implementation is not fully modular. The SERE matching is performed using FSMs; a single FSM is used to implement the left-hand side of a suffix implication, and multiple FSMs are used in the right-hand side. To process the matches that are triggered by a left-hand side sequence, a finite number of FSMs in the right-hand side are used. Therefore, unbounded repetition is not allowed in the left hand-side sequence, since it needs an infinite number of FSMs of the right-hand side. A case study on a cache controller is presented.

Checker generation for SVA is performed by Das *et al.* in [SMDC06]. The idea is breaking the sequence expressions as a sequence of expressions concatenated with the corresponding time range expressions. Then, for each of the smaller sequence expressions a sub-module is generated. These modules are interconnected in a way to determine a match or fail of the actual sequence expression; every generated checker has the *start* input that triggers the start of checking, and the *match* output that shows the match of an expression. RE operators are classified into subsets, and a different synthesis approach is proposed for each subset. The method cannot deal with some unbounded repetitions of a sequence since there are cases where the expressions cannot be synthesized into a finite amount of hardware resources. For detecting the “not” of a sequence, if the sequence fails, separate rules are given for each operator. However, there is no proof or evidence showing that the rules are correct. In this work, assertions for the AMBA AHB bus are used, and the corresponding checkers have been generated. The results have been compared with the results from Synopsys OVA checker. Synopsys VCS simulator is used for simulating the generated checkers and OVA checkers; the generated checkers are simulated faster than the OVA checkers. In addition, the checkers are synthesized, and the area overhead is reported.

Implementing PSL SEREs using the modular approach was performed by Morin-Allory *et al.* in [MAGB07] for online fault detection. In contrast to the other reviewed modular methods, the proposed approach covers properties with both finite and infinite state sequences over time. In addition, the method supports both weak and strong operators. In this method, a library is provided that consists of the synthesizable VHDL module for each SERE operator, which is called SERE connector. A property sequence is built recursively by interconnecting the SERE connectors, based on the abstract syntax tree of the SERE. Both the library and the construction of complex monitors are proven correct with respect to the trace semantics of SEREs. The connectors have a common interface. They are synchronized by a clock signal, and initialized by reset. They take one or two tokens as input, and output a token. Tokens are passed from one connector module to the next one. A monitor is triggered each time a token is transmitted to its input. The presence of a token on the output of a monitor means that the sequence starting at the cycle when the monitor was triggered has been recognized. Two types of token have been introduced in [MAGB07]: *monochrome* and *polychrome*. The polychrome tokens are used for dealing with several simultaneous evaluations of a sequence. Each color corresponds to one evaluation of the sequence. However, the drawback is when multiple concurrent matches happen, a large number of colors should be supported in a token, which affects the hardware overhead significantly.

The modular approach introduced by Morin-Allory and Borrione in [KAB06] generates checkers for PSL temporal properties. In this method, the simple subset of PSL is considered. Each PSL operator in this subset is implemented as a synthesizable VHDL module, with a generic interface. A PSL property is generated by interconnecting the operators' sub-modules based on the abstract syntax tree of the property. A prototype tool, HORUS, was developed for the automatic construction of a test environment for the design [OMAB08, Odd09]. In these works, monitors can be triggered several times, and are able to trace concurrently the evolution of the property for the successive triggers: when a property succeeds or fails, the particular starting point for this particular result is known. Therefore, this method is useful in debugging. The method was then improved by Oddos *et al.* in [OMAB07] to prototype generators for on-line test vector generation.

3.3 Property synthesis as correct-by-construction circuits

Contrary to the previous section where properties are verified over an existing design model, in this section a property is seen as the specification of the module to be designed. The objective is then producing the synthesizable RTL design from its assertions directly. In contrast to the monitors, some operand variables are inputs to the module and others are outputs.

The synthesis of control-type sequential circuits from formal formulas is not new. The functional specification of controller circuit involves describing sequences of events and their interactions. The studies on automatic synthesis of a circuit from its specifications started more than 50 years ago, with the following question raised by Church [Chu57, Chu62]:

"Given a requirement which a circuit is to satisfy, we may suppose the requirement expressed in some suitable logistic system which is an extension of restricted recursive arithmetic. The synthesis problem is then to find recursion equivalences representing a circuit that satisfies the given requirement (or alternatively, to determine that there is no such circuit)."

Almost all the solutions to this problem are automaton-based. However, the synthesis method of our work is modular. Here, some of the related works of each groups are reviewed.

3.3.1 The automaton-based approach

In this method, an automaton is defined as a set of states and transitions between them that is specified by a given specification. The goal is the construction of a finite-state procedure that transforms any input sequence into an output sequence such that a given specification is satisfied. The underlying formalization of the specifications (regular expressions or temporal logic formulas) that specify sequences of events is based on either language theory (grammar-based) [SB94, Öbe99] or automata theory [FKTMo86, PR89, ABBSV00, SM02, BGJ$^+$07a, FJR09, BJP$^+$12, EKH12]. The grammar-based specifications do not have the limitations of the procedural specification style, that is the dependency of the specification implementation on time and also dependency to the size of input and output signals.

Church's problem was first addressed by Büchi [BL69], and then by Rabin [Rab72]. These approaches build automata of the properties, and reduce the synthesis problem to the emptiness problem of automata. If a non-empty automaton can be found for the specifications, its corresponding circuit is produced.

Pnueli and Rosner reconsidered the synthesis problem from LTL specifications in [PR89]. The proposed method starts constructing a Büchi automaton for a given LTL specification and then converts it into a deterministic Rabin automaton. The complexity of the synthesis algorithm is double exponential in the length of the given specification.

It is the origin of "synthesis of reactive systems", with the algorithmic theory of two or multi-player games. Some of the works use this approach, and formalize the synthesis as a two-player game between the environment (that provides the inputs) and the system that responds on its outputs [FJR09, BJP$^+$12, EKH12, BCG$^+$10, BGJ$^+$07a]. The specifications are realizable if the system can always win the game. In this case, the corresponding circuit is extracted. Some recent works [Gre04] [BEK$^+$14] use SAT-based method for constructing the final circuit.

The Clairvoyant tool was developed by Seawright and Brewer [SB94] to automatically synthesize HDL RT level descriptions from a specification written in Production Based Specification (PBS). PBS is a language that is similar to REs in many points. In a PBS, the control part of the design is specified as a hierarchical set of productions. Each production is viewed as a non-deterministic automaton. The idea is to transform the specification into BDD, then synthesize this BDD to RTL. Experimental results are good for simple circuits. In Clairvoyant, a design entity with a single process and a well defined boundary and interface is specified. All interactions with inputs and outputs are described at clock cycle level.

Öberg synthesized data communication protocols that are expressed using Backus-Naur Form (BNF) [2] grammars [Öbe99]. To do this, he developed a language, ProGram, and its compiler. The ProGram language is based on a regular LL(1) grammar and uses a BNF-like notation to code both input and output sequences, targeted for specification of data communication protocols. The language supports a description style that is independent of the port sizes. The implementation is generated so that sizes are a generic parameter that is fixed in a subsequent step. The ProGram Compiler takes the ProGram description as its input. It then parses the language and produces a RT-level VHDL implementation of the interface protocol, by partitioning the input sequences into

[2] BNF is a formal notation for the specification and documentation of programming language syntax. Many programming languages, communication protocols or formats have a BNF description in their specification.

a sequence of tokens, and output sequences into a sequence of output assignments. The method uses a Directed Acyclic Graph (DAG) and explores the state space to analyze all possible behaviors of the circuit. Some experiments have been done to evaluate the design space exploration strategy of the ProGram compiler; a ProGram description of a reduced F4 OAM protocol is implemented to generate different designs by various port-size constraints of the inputs and outputs. To evaluate the quality of the produced designs, a set of designs are coded in ProGram, High Level Synthesis (HLS) style VHDL code and RTL styleVHDL code. Then, the code sizes are compared. The results show that ProGram generates more compact designs. The same set of designs is synthesized using a commercial HLS tool and the ProGram compiler followed by standard logic synthesis. The results show that ProGram generates smaller circuits.

Heymans uses Answer Set Programming (ASP) to synthesize synchronization skeleton programs [HNV05]. In contrast to most of the methods, the method uses CTL for expressing properties of concurrent programs. First, a model of the CTL specification is built using ASP. Next, a coherency analysis is performed and the synchronization skeleton is extracted.

Aziz *et al.* [ABBSV00] describe how to synthesize sequential circuits from S1S logical formulas. S1S is a second order logic that allows effectively describing the sequential systems. The formula is transformed into a single finite automaton, to be synthesized into a gate-level hardware. The proposed method is automatic. It provides a systematic way to reduce the problem of optimizing interacting FSMs to optimizing a single FSM. Additionally, the approach can be easily extended to different interconnection topologies. Moreover, the approach generalizes to the synthesis of safety and liveness properties. Any specification provided in S1S owns a Büchi automaton that can be synthesized into a netlist. The method suffers from high complexity due to the use of negations and determinizations (if necessary) in Büchi automata. Although some optimizations have been applied to reduce the algorithmic complexity, these automaton-based approaches cannot process complex designs.

Kukula and Shiple describe in [KS00] how to effectively synthesize the mathematical relations in combinational circuit. The approach transforms the equation that specifies the relation between inputs and outputs into a FBDD (Free-BDD). Then, the final circuit is extracted from FBDD. The circuit size is proportional to the FBDD.

Müller and Siegmund [SM02] also worked on the synthesis of communication interfaces from protocols. They used a SystemC extension formalism, SV, for protocol specification. SV allows writing the communication protocol between the components in high level. The communication is done through abstract channels, through read/write actions. The idea is to start with a description of the communication protocol between two SV components. The description is then synthesized into a SystemC description. The SV part is analyzed and the Protocol Flow Graph (PFG) is built. The PFG provides the definition of the communication protocol. The SystemC synthesizable descriptions are obtained by transforming the PFG into two FSMs, one for each interface. They used automaton-based methods to build the final design.

Although automatic synthesis from the specifications is not so new, it has recently been applicable to real circuits, through the development of prototype tools [PPS06, BGJ$^+$07a, RAT, FJR11, EKH12].

Bloem *et al.* defined a subset of LTL named “GR(1)” from which properties are translated to automata [BGJ$^+$07a, BGJ$^+$07b]. They use the two-player game method. The game theory algorithms compute all the correct behaviors of the design under all admissible interactions with the environment. It is shown how to build a winning strategy and extract a system from it. The method is polynomial in N^3, where N is the sequential complexity of the specification. In [BGJ+07a] the method is applied to the Generalized Buffer (GenBuf) from IBM [IBM] and the AMBA AHB Bus arbiter. The PSL properties of these examples are presented in [BGJ+07a], and the synthesis results are shown. Increasing the number of senders/receivers in GenBuf, or the number of masters/slaves in the AMBA AHB arbiter, increases the synthesis time and the size of the generated hardware. In addition, the generated gate-level circuit is very complicated and cannot be changed manually. The work is later extended and improved in [BJP+12], to obtain smaller circuits.

Ehlers *et al.* present a game approach for synthesizing circuits from their formal specifications [EKH12]. A general strategy is introduced as the characterization of the set of moves that lead the system player wins the game. There may be more than one solution for the system player to win the game. The goal is selecting a good circuit for this strategy among the several possible extracted circuits.

These methods have been implemented and some prototype tools have been provided for property synthesis. Briefly, Lily and Anzu were implemented based on the researches in [JB06, PPS06], and then improved to Ratsy. In addition, Unbeast was developed based on the works of [Ehl11, EKH12] (see Section 3.4).

All the related works that have been reviewed so far extract the final circuits from various forms of BDDs. Some other works propose a SAT-based method for synthesizing the circuit.

In the approach that is proposed by Greaves in [Gre04], the small components are synthesized using SAT-based methods. The specification is provided as input. A SAT solver is used to generate the programming bit-stream on a pseudo-FPGA architecture to comply with the formal specifications of the system. This method is taking advantage of the fact that the basic component of the FPGA is the LUT. LUTs are defined as functions, having some free variables (the input signals) and some variables whose values should be determined by SAT solvers (the intermediate or some output signals). In addition, the design is expressed using some logical rules that consider the value of a signal in various cycles. Both the FPGA program and the design specifications are expanded into CNF form. Then, a SAT solver is used to find an appropriate solution. The properties are in the form of $A \rightarrow next(B)$. A few experiments have been tried to show the feasibility of this method. However, the approach is not automatic.

In [BEK+14] Bloem *et. al* uses a SAT-based method for synthesizing circuits from safety specifications. The proposed SAT-based learning method combines quantifier elimination with computational learning. The method generates smaller circuits in shorter time in comparison to BDD-based methods. The basic algorithms that are used in this work are not new, but new optimizations have been presented for safety specification. It is shown how the idea of interpolation for circuit extraction can be combined with learning to compute the interpolants more efficiently.

3.3.2 The modular approach

Some other works have been done to synthesize the temporal properties of PSL or SVA based on completely different principles [SNBE07, EFP09].

Eveking *et al.* introduce "Cando-Objects", and address how they can be incorporated in synthesizing modules from PSL properties [SNBE07]. Cando-Objects can do anything allowed by their property, i.e. can show all the possible behaviors of the properties from which they were generated. The properties are rewritten in a normalized form. The normalization procedure identifies the potential inconsistencies between properties and disjoints them so that only one property specifies a signals' value in a specific state. Then, the VHDL description of the Cando-Object is generated. The original models can then be replaced by the corresponding Cando-Objects. In order to allow all possible behaviors, additional signals are used to generate signal values for the cases when a signal value is not defined. If the specification is logically non-determined, e.g. *A or B*, free inputs are used. Non deterministic behaviors are admitted, and translated into the addition of more input signals connected to random sources. The Cando-Objects are a fault-conserving abstraction of the original modules; therefore, if the full design including the replacements can be verified, the design is correct. Also, it proves that the set of module properties is complete with respect to the architectural properties. The approach is limited to bounded time properties. The method has been applied to AMBA AHB master, PCL Local Bus and a MIPS core, and the generation time has been reported. However, there is no result on hardware metrics.

The subject was reconsidered by Oddos *et al.* in [OMAB09], and a preliminary solution was proposed to synthesize the control circuits from PSL temporal properties [Odd09]. The method is modular, each property is turned into a component combining monitor and generator features: the extended-generator. A synthesizable VHDL sub-module is provided for each operator in PSL_{simple}. These operator sub-modules are proven to be correct. Each property is the interconnection of its operators' sub-modules. The final design is the interconnection of the property modules, and it is correct-by-construction. The approach synthesizes circuits specified by hundreds of temporal properties in a few seconds. The idea extended HORUS, which was used for synthesizing checkers, to SyntHorus. It could synthesize the circuits from FL temporal properties in PSL_{simple}. The method supports both strong and weak operators. It does not imply any limitations to the PSL_{simple} FL operators. It is not necessary for a designer to specify the assumptions. In addition, the method enhanced the debugging capability of the design. However, the method was not totally automatic. The designer should have annotated the properties, which means in each property, the designer should have made the decision about the signal directions. Then, the input of SyntHorus was the annotated properties. Moreover, the solution did not support duplicated signals; it was limited to the cases in which a signal is constrained just by a single property. In addition, it was not possible to consider the consistency and completeness of the properties. Moreover, it did not support SEREs.

These shortcomings have been resolved in this thesis. The signals in the properties are automatically annotated, and the duplicated signals are resolved automatically. In addition, some complementary properties are generated that can be used both in simulation and formal verification tools to verify the consistency and completeness of the set of the properties.

3.3.3 Synthesizing from Regular Expressions

All of the above reviewed methods, both automaton-based and modular, synthesize the temporal properties and not the sequences. There are just a few works in synthesizing

REs, some of them are very old and go back to more than 40 years ago [BP63, Brz65, Cur68, LJ88, BL88].

Brzozowski presents a method in [BP63] for modular synthesis of REs over the Boolean alphabet. The described designs should be synchronous, deterministic, and finite. For each basic operator, a sub-module is constructed that implements the operator. Then, the final circuit is constructed recursively by interconnecting the sub-modules. The paper introduces the notion of recursive realization, and proves that the construction is valid if proper assumptions are considered. He then introduces the Linear Sequential Circuits in [Brz65], and addresses how to obtain the REs that are accepted by such circuits directly. In addition, he gives a method for interpreting the REs to construct a word description of the circuit behavior. The method supports unbounded repetition. However, neither a tool has been provided to implement the method nor any experimental results are given.

Curtis in [Cur68] considers how to obtain directly the realizations of synchronous finite automata from their specification that are expressed in REs. He defines a polylinear sequential circuit[3] realization, and proves that every synchronous finite automaton has such a realization. A finite automaton is realized by a polylinear sequential circuit if its next state variables and output have the polylinear property; i.e. they can be expressed as a linear function of the present state variables for each of the inputs. In this method the polylinear sequential circuit realizations do not require special initial circuitry. In contrast to the indirect methods that need some combinational logics to obtain the next state and output equations after state minimization, in this method these equations are being generated automatically. The drawback is the size of the generated circuits.

Luk and Jones present an approach to derive regular synchronous circuits from their RE specification [LJ88]. In this method, some common structures are defined. In the first step, the specification should be rewritten based on the predefined structures. From this, a draft architecture is obtained. Then algebraic theorems are used for optimizing the draft architecture. The method is applied to a “rank evaluation circuit” taken as case study. First, a preliminary architecture is obtained. Then, it is optimized in several ways, each has its own trade-off, and may affect the latency, frequency or area of the obtained architecture.

Brown and Leeser propose a method in [BL88] for synthesizing a correct sequential circuit from its specification. The approach is developing a circuit as a program. After verifying the program, it is compiled to a sequential machine description. The program specifies the assignment statements of the data-path, the data-path branching conditions, and also the structure of the controller that implements this. This program can be represented using a state transition system. Then, the states and transitions are partitioned into a controller and data-path. Each program’s statement is labeled uniquely. The states of the transition system consists of an assignment of values to program variables, and a program label that specifies the control part. The focus of this paper is on generating the controller. The data-path can then be implemented automatically or manually. It has been proven that the generated circuits are correct. The method is demonstrated on the design of a multiplier.

In this thesis, we revisit this old problem, and propose an approach for synthesizing PSL SEREs. To the best of our knowledge, it is the only work that addresses synthesizing a design from SEREs, and none of the previously developed tool support SEREs; they

[3] In a polylinear sequential circuit the next state variables are linear functions of the present state variables for each of the inputs.

just consider the temporal operators.

3.4 Existing tools

In this section, we briefly review some of the existing tools in the area of ABV, and compare these tools.

There are a large variety of tools for formal verification. Based on the tool being used, the properties can be expressed as PSL, LTL, or CTL. OneSpin [ONE], Mentor Graphics 0-In, Cadence Incisive [CAD], and RuleBased from IBM [HIL04] are some of the most well-known tools that can be used for formal verification. We exercised RuleBased, 0-In, and OneSpin, and selected OneSpin for formally verifying the generated circuits.

For compiling assertions into monitors some industrial tools exist. The first industrial tool for construction checkers of PSL properties is IBM FoCs [ABG$^+$00]. The details have not been published for commercial reasons. FoCs uses automata to generate HDL checkers from PSL assertions. An "end-of-simulation" signal should be provided by the user to mark the end of time when strong properties are used. This signal is used by the checkers to report any unfulfilled obligations as errors when the cycles are truncated (there exists no further cycles). FoCs does not support all the operators, and supports very few strong operators.

Another tool that has recently been developed by Atrenta is BugScope [BUG]. It uses design and testbench information, and automatically generates assertions and functional coverage properties. BugScope takes an RTL design and also its testbench; then, it synthesizes automatically the high assertions that capture key design constraints and specifications. In addition, it generates functional coverage properties. The coverage properties are functional and are independent of the syntax of the RTL. BugScope has sufficient capacity to support assertion synthesis for full SoC designs, with run-time performance scaling linearly with respect to design complexity. It can generate the assertions in IEEE standard formats such as SVA, PSL or synthesizable Verilog.

Dolphin integration also provides a tool, SLED SDG (Synthesizable Detector Generator), for synthesizing assertions as checkers [SDG]. It generates RTL checkers (Verilog or VHDL) from PSL assertions. It also integrates RTL synthesizable hardware checkers into circuits for real-time verification.

In addition to the mentioned industrial tools, there are academic tools for compiling assertions as checkers. MBAC is an academic tool that has been developed by Boule [BZ08a] in McGill university. It uses an automaton-based approach to generate assertions from PSL properties. The MBAC checker generator produces assertion-monitoring circuits from PSL statements and augments these checkers with debug-assist circuitry. Other forms of debug information, such as signal dependencies, can also be sent to the front-end applications. Boule compares the checkers produced by MBAC to the checkers that are produced by FoCs. The comparison involves generating checkers for a suite of assertions, and then synthesizing the checkers using FPGA implementation tools. The circuit size of the checkers are compared using the number of flip-flops and also combinational logic. The results in [BZ08a] shows that the MBAC checker generator outperforms FoCs.

Horus [OMAB08] is another academic tool that has been developed in the VDS group of TIMA Lab. It uses the modular method to generate monitors from PSL FL properties. It can also be used for the automatic generation of a test environment. This tools is the basis of SyntHorus and SyntHorus2.

In the scope of correct-by-construction, there are just a few tools.

Acacia$^+$ [ACA] is based on the works in [FJR09, FJR11]. It inputs LTL specifications, and outputs a design in *dot* format. The tool is based on the two-player game approach. It provides several options, for example: backward or forward state traversal; the circuit player or the environment player has the initial move. However, the problem is that it can only synthesize very small, and non realistic, circuits.

Unbeast [UNB] is based on the works in [EKH12]. It inputs LTL specifications in XML syntax and produces an intermediate NuSMV file that is turned to an *aig* format by AIGER. ABC [ABC] is used to translate *aig* into Verilog. The tool can be applied to small examples, and it times out for complex or large circuits.

Ratsy (Requirements Analysis Tool with Synthesis) [RAT, BCG$^+$10], is an update of Rat that is developed by Bloem *et al.* in the University of Gratz. It contains the Lily [JB06] and Anzu [PPS06] previous tools of the same research group. A graphical user interface has been provided for Ratsy. Ratsy inputs *GR(1)* PSL properties. The properties must be partitioned into a *guarantee* and an *assume* part. It produces a Verilog design. The tool can also take a Büchi automaton as its input. In this system the environment player moves first. Ratsy performs an on-the-fly verification of the properties.

SyntHorus is the extended version of Horus that is developed in TIMA Lab [Odd09]. In contrast to other existing methods, the tool is based on the modular approach, and could synthesize PSL$_{simple}$ FL properties to VHDL. However, it had some limitations. It could just support scalar signals of type *std_logic*. In addition, it could not deal with the modeling layer of PSL. Moreover, the process was not totally automatic. In addition, there were some difficulties for specifying the value of the signals that are being generated in several properties. However, the synthesis results and hardware generation time are better than Ratsy.

In this thesis, SyntHorus has been improved to SyntHorus2. This new version inputs PSL properties and generates the synthesizable VHDL circuit automatically. It supports FL properties, and also partially supports SEREs. The tool can automatically decide about the signal directions in each property (see Chapter 7). Moreover, it can resolve the value of the signals that are generated in several properties (see Chapter 9). Additionally, SyntHorus2 supports *std_logic_vector* signals. Additionally, it partially supports the modeling layer. Finally, it generates complementary properties to verify if the properties are consistent and complete.

3.5 Summary

In this chapter we briefly reviewed the related works in assertion-based verification and design. We considered both the automaton-based and modular approaches.

In the area of synthesizing assertion checkers for REs, the modular methods have some limitations. Assertion checkers should be able to handle multiple simultaneous sequences of events that can overlap temporally. In general, the modular methods indicate difficulties for implementing all SERE operators, especially intersection and unbounded repetition [SMDC06, PLN05, MAGB07, BZ05], while the automaton-based methods overcome these difficulties [BZ08b].

Most of the works in designing correct-by-construction circuits are based on automaton. Such methods are expensive. In addition, almost all the proposed methods only deal with the temporal properties, not the sequences.

Our synthesis approach is modular for temporal operators of PSL_{simple}. For synthesizing SEREs, to avoid the shortcomings of the modular method, a *hybrid* method is introduced. Using this method, the left-hand side of a property sequence is generated using the automaton-based method, while the modular method is used for synthesizing the right-hand side of the suffix implication.

Finally, some of the tools in the area of assertion-based verification and design are introduced in this chapter. In Chapter 10 the synthesis results of our tool, SyntHorus2 are compared to the results of Ratsy.

Chapter 4

Fast prototyping from assertions: the overall synthesis flow

Contents

4.1 Introduction

In this chapter, the overall synthesis flow is introduced. The details of each step of this flow will be explained in the following chapters. In addition, a running example is introduced and used in the following chapters to show the applicability of the proposed synthesis method.

4.2 Reactant synthesis

In this work a correct-by-construction method is proposed to directly produce the synthesizable RTL design from its assertions.

The method is modular; i.e. the reactant of each property is the interconnection of its operators' modules. Therefore, operators' modules are the building blocks of a property, and are called *primitive reactants*, since they do not consist other reactant modules. Then, the final circuit is the interconnection of the properties' reactants.

Fig. 4.1 shows the overall synthesis flow that produces a circuit from a set of properties.

The initial step is providing a library of primitive reactants for FL[1] and SERE[2] operators. It is done by considering the formal semantics of the operators, and interpreting these mathematical semantics to hardware (see Chapter 5 and Chapter 6). The library of primitive reactants for FL operators has been already implemented by Morin-Allory *et al.* in [OMAB09]. During this thesis, the library of primitive reactants for SERE operators has been provided.

At the beginning, properties are processed one by one. A conventional front-end produces the *Abstract Syntax Tree* (AST) from the source text of each property. The implementation of the subsequent processing steps makes use of no further compilation tool.

For each property it should be identified which signal occurrences are read (*monitored*), and which ones are constrained (*generated*) by the property. We refer to this step as *annotation*. To annotate the signals we use the dependency relations of the temporal operators (see Section 5.2 of Chapter 5, and Section 6.3 of Chapter 6). Based on these dependencies, an algorithm is proposed for annotation (see Chapter 7): a direction is given to the edges of each AST, and the *Directed Abstract Syntax Tree* (DAST) of each property is generated.

Then, using the library of primitive reactants for the operators, a complex reactant is built for each fully annotated DAST, by applying the principles that are explained in Chapter 8, Sections 8.3.2 (for FLs) and 8.3.3 (for SEREs).

Considering the DASTs, a signal may be constrained by several properties. In addition, there may be cases that a signal's direction cannot be decided just by considering a property alone. Therefore, we need to consider all the properties together to identify the dependency among properties. To this goal, a graph that reflects a global view of the signals in the circuit is constructed from all the DASTs (See *Dependency Graph* in Chapter 9). The following information can be extracted from this graph:

1 identifying which properties constrain a specific signal, *duplicated* signal. Using this information we generate a *simple solver* that specifies the value of the signal.

[1] Foundation Language

[2] Sequential Extended Regular Expression

2 identifying the signals whose direction has not been specified in a property, *unannotated* signals, and identifying the dependency of these signals on other signals. Using this information, we generate a component, *complex solver*, that specifies the value of an unannotated signal based on the value of the other signals that affect it.

We refer to this procedure as *resolution* (see Chapter 9). The final circuit is constructed as the interconnection of the reactants for all the properties, together with solvers (see Chapter 9, Section 9.7).

This is a register transfer level model that is input to a conventional industrial synthesis tool to obtain the final implementation, either on FPGA or on an ASIC.

From the information provided by the dependency graph, a set of complementary properties can be generated, which can be used by an industrial verification tool, to verify if the set of the properties are complete and consistent.

We have provided a prototype tool, SyntHorus2 that implements the above synthesis process: it takes a set of PSL properties as its input, and generates the final circuit in VHDL. It also generates some complementary properties to verify the completeness and coherency of the set of the specification (see Fig. 4.1). The proposed method is applicable to the controller parts of a design.

4.3 Running Example: Generalized Buffer

Here, we introduce IBM Generalized Buffer [IBM] (*GenBuf*) as our running example.

4.3.1 Presentation

The generalized buffer GenBuf is an arbiter that sequentializes requests coming from *nbsend* senders, and transmits them one at a time to *nbrec* receivers (*nbsend* and *nbrec* are generic parameters). Each sender has its own bus, and the receivers share the same bus. A FIFO (with the length of four 32 bit data) stores the incoming data waiting to be sent to the receivers.

A controller communicates with all the modules and the FIFO: it enforces a round-robin selection policy on the receivers side, it blocks the senders when the FIFO is full, and blocks the receivers when the FIFO is empty. Figure 4.2 displays the architecture of the system and the interface control signals that are used for the communication.

- From each sender S_i, GenBuf receives a request input *StoB_REQ(i)* and replies with an acknowledge output *BtoS_ACK(i)*.
- To each receiver R_j, GenBuf outputs a request *BtoR_REQ(j)* and gets an acknowledge *RtoB_ACK(j)*.
- GenBuf gets the *FULL* and *EMPTY* signals from the FIFO and provides the *ENQ* and *DEQ* signals for writing (reading) appropriate data to (from) the FIFO.

The set of FL properties that specify the GenBuf controller are taken from [BGJ$^+$07a]. These properties have been rewritten and completed in order to be used by our prototype tool, SyntHorus2. In addition, the properties have been rewritten as SEREs, using the rewriting rules that are introduced in [MABBZ08]. In the following sections, the communication between the controller, the senders, the receivers, and the FIFO, together with the corresponding properties are explained in details.

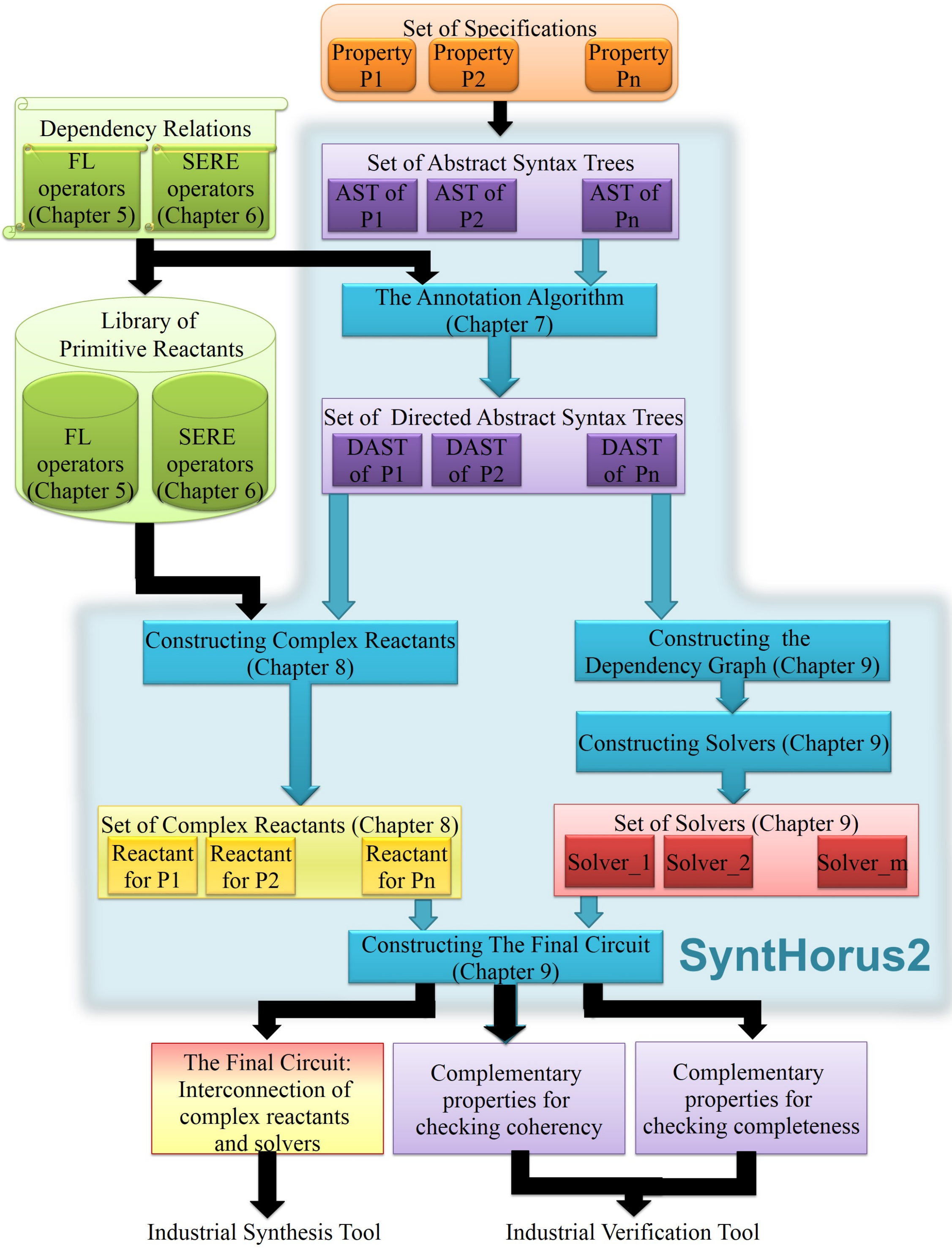

Set of Specifications
Property P1
Property P2
Property Pn
Dependency Relations
FL operators (Chapter 5)
SERE operators (Chapter 6)
Set of Abstract Syntax Trees
AST of P1
AST of P2
AST of Pn
The Annotation Algorithm (Chapter 7)
Library of Primitive Reactants
FL operators (Chapter 5)
SERE operators (Chapter 6)
Set of Directed Abstract Syntax Trees
DAST of P1
DAST of P2
DAST of Pn
Constructing Complex Reactants (Chapter 8)
Constructing the Dependency Graph (Chapter 9)
Constructing Solvers (Chapter 9)
Set of Complex Reactants (Chapter 8)
Reactant for P1
Reactant for P2
Reactant for Pn
Set of Solvers (Chapter 9)
Solver_1
Solver_2
Solver_m
Constructing The Final Circuit (Chapter 9)
SynthHorus2
The Final Circuit: Interconnection of complex reactants and solvers
Complementary properties for checking coherency
Complementary properties for checking completeness
Industrial Synthesis Tool
Industrial Verification Tool

FIGURE 4.1: Overall Synthesis Flow

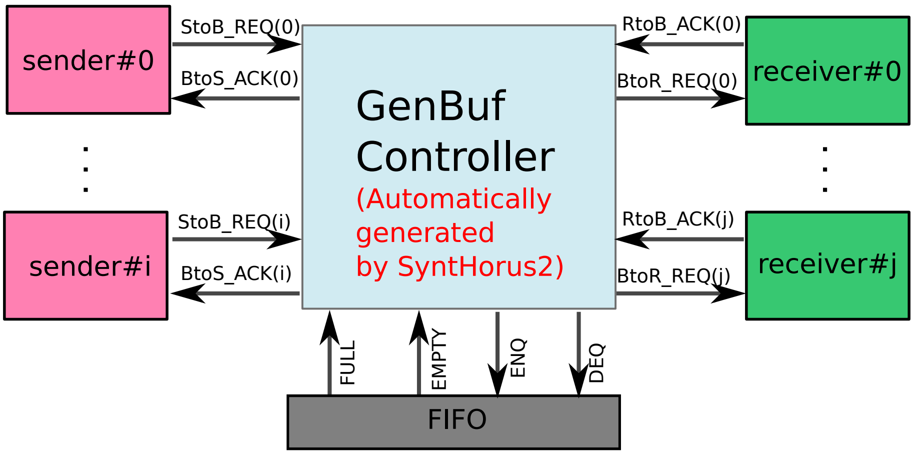


FIGURE 4.2: GenBuf circuit interface

4.3.2 Communication with FIFO

Data are read/written from/to the FIFO buffer by activating the *DEQ*/*ENQ* signals. The *EMPTY* and *FULL* signals show the status of the FIFO. The properties that are shown in Fig 4.3 guarantee that the FIFO works correctly. The properties have the following meaning:

- `P0_FIFO`: When the FIFO is full and no data is read from the FIFO, no data can be written into the FIFO.
- `P1_FIFO`: When the FIFO is empty, no data can be read from it.

```
vunit genbuf_FIFO
{
  P0_FIFO:
    always (FULL and not DEQ -> not ENQ);

  P1_FIFO:
    always (EMPTY -> not DEQ);
}
```

FIGURE 4.3: FL specification that guarantees the correct behavior of FIFO

The FIFO should be selected, and the data should be put into it, whenever the GenBuf controller sends an acknowledgment to the corresponding sender (see properties under “FIFO interface” in Fig. 4.5). The FIFO cannot be selected and get the data before being sure that the request from the sender is acknowledged by the buffer. So, the *ENQ* and *SLC* signals follow the behavior of the acknowledgment signals to the senders.

The data should be read from the buffer after completion of the transfer. Hence, after getting the acknowledgment from a receiver, the *DEQ* signal becomes high (see properties under “FIFO interface” in Fig. 4.8).

4.3.3 Communication with the senders

The interface between the senders and GenBuf is a 4-phase handshaking protocol. Let S_i be any one of the senders.

- S_i asserts *StoB_REQ(i)*, then puts data on the next cycle
- GenBuf asserts *BtoS_ACK(i)* after reading the data
- In the next cycle, the sender deasserts *StoB_REQ(i)*
- GenBuf specifies the end of the transaction by deasserting *BtoS_ACK(i)*

We assume that signals take a default value 0, GenBuf maintains FIFO order, and senders are never starved. Figure 4.4 shows an example timeline of a GenBuf to sender handshake.

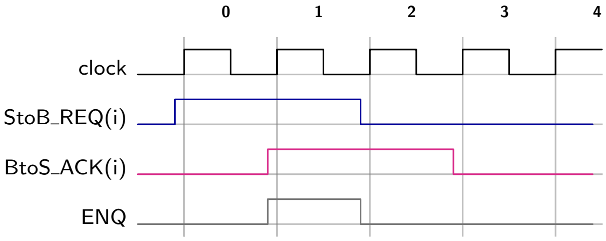


FIGURE 4.4: An example timeline of a GenBuf to sender handshake

4.3.3.1 Formal FL specification

Figure 4.5 shows the FL properties that describe the communication between the GenBuf controller, the senders, and the FIFO (the properties are shown for two senders).

The properties have the following meaning:

- `P0_sender_i`: Once low, the acknowledge to S_i remains low as long as S_i sends no request.
- `P1_sender_i`: The acknowledge to S_i remains high as long as S_i keeps its request high.
- `P2_sender_i`: A new request may be raised by S_i only if its acknowledge signal is low.
- `P3_sender`: The senders may send simultaneous requests to GenBuf, but at most one acknowledge signal is high.
- `P4_FIFO_sender`: Either *ENQ* is 0, or one of the acknowledge signals is 1.
- `P5_FIFO_sender`: If signal *ENQ* is low, none of the acknowledge signals has just been raised. To generate signals, SyntHorus2 requires that all properties be temporally aligned from present to future. Property `P5_FIFO_sender` has to be rewritten as:

```
P5_sere_FIFO_sender_0:
  always (not BtoS_ACK(0) -> next!(ENQ or not BtoS_ACK(0)) );

P5_sere_FIFO_sender_1:
  always (not BtoS_ACK(1) -> next!(ENQ or not BtoS_ACK(1)) );
```

- `P6_FIFO_sender`: If there is a request from the senders and the FIFO is not full and signal *ENQ* is low, *ENQ* must be high in the next cycle and go back to low in the following cycle.

- `P7_FIFO_sender_i`: If there is an acknowledgment to S_i, then the i^{th} data is pushed into the FIFO ($SLC = i$).

4.3.3.2 Formal SERE specification

The set of SERE properties that specify the communication between the GenBuf controller, the senders and the FIFO are shown in Fig. 4.6 (for 2 senders). Here, only two properties are explained as examples.

- `P1_sere_sender_i`: The acknowledge to S_i remains high as far as S_i keeps its request high, and it is deasserted one cycle after deactivation of the request signal.

- `P6_sere_FIFO_sender`: If there is a request from the senders and the FIFO is not full and signal *ENQ* is low, *ENQ* must be high in the next cycle, and then, it will be deasserted in the following cycle.

4.3.4 Communication with the receivers

GenBuf interacts with the receivers through a 4-phase handshake protocol. The arbitration mechanism that is used by GenBuf is round-robin: GenBuf does not request the same receiver consecutively. In addition, it does not request both receivers at the same time.

A request signal from GenBuf remains active, until receiving the acknowledgement.

One cycle after asserting the acknowledge signal, the request will be deasserted and cannot be asserted again until one cycle after deactivation of the acknowledge signal.

Figure 4.7 shows an example timeline of a GenBuf to receiver handshake.

4.3.4.1 Formal FL specification

The set of FL properties that specify the communication between the GenBuf controller, the receivers, and the FIFO are shown in Fig. 4.8 (for 2 receivers).

The properties have the following meaning:

- `P0_rec`: When the FIFO is not empty, GenBuf should send a request to one of the receivers; which receiver should be requested is not specified.

- `P1_rec`: When the FIFO is empty, none of the receivers should be requested.

- `P2_rec`: Two receivers cannot be requested at the same time.

```
vunit genbuf_sender
{
  P0_sender_0:
    always ((not BtoS_ACK(0)) and (not StoB_REQ(0)) -> next! (not BtoS_ACK
        (0)));

  P0_sender_1:
    always ((not BtoS_ACK(1)) and (not StoB_REQ(1)) -> next! (not BtoS_ACK
        (1)));

  P1_sender_0:
    always ((BtoS_ACK(0) and StoB_REQ(0)) -> next! (BtoS_ACK(0)));

  P1_sender_1:
    always ((BtoS_ACK(1) and StoB_REQ(1)) -> next! (BtoS_ACK(1)));

  P2_sender_0:
    always (rose(StoB_REQ(0)) -> not BtoS_ACK(0));

  P2_sender_1:
    always (rose(StoB_REQ(1)) -> not BtoS_ACK(1));

  P3_sender:
    always ( not BtoS_ACK(0) or not BtoS_ACK(1) );

-- FIFO interface
  P4_FIFO_sender:
    always (not ENQ or BtoS_ACK(0)  or BtoS_ACK(1));

  P5_FIFO_sender:
    always (not ENQ -> not rose(BtoS_ACK(0)) and not rose(BtoS_ACK(1)));

  P6_FIFO_sender:
    always ( (StoB_REQ(0) or StoB_REQ(1)) and (not FULL) and (not ENQ) ->
        next! (ENQ) and next![2](not ENQ) );

  P7_FIFO_sender_0:
    always (rose(BtoS_ACK(0)) -> SLC = 0);

  P7_FIFO_sender_1:
    always (rose(BtoS_ACK(1)) -> SLC = 1);
}
```

Figure 4.5: FL specification of GenBuf communication with senders in the case of two senders

```
vunit genbuf_sender_sere
{
  P0_sere_sender_0:
    always ({not BtoS_ACK(0) and not StoB_REQ(0)} |=> {not BtoS_ACK(0)}!);

  P0_sere_sender_1:
    always ({not BtoS_ACK(1) and not StoB_REQ(1)} |=> {not BtoS_ACK(1)}!);

  P1_sere_sender_0:
    always ({(BtoS_ACK(0) and StoB_REQ(0))} |=> {(BtoS_ACK(0))}!);

  P1_sere_sender_1:
    always ({(BtoS_ACK(1) and StoB_REQ(1))} |=> {(BtoS_ACK(1))}!);

  P2_sere_sender_0:
    always ({not StoB_REQ(0); StoB_REQ(0)} |-> {not BtoS_ACK(0)});

  P2_sere_sender_1:
    always ({not StoB_REQ(1); StoB_REQ(1)} |-> {not BtoS_ACK(1)});

  P3_sere_sender:
    always ( not BtoS_ACK(0) or not BtoS_ACK(1) );

---------- FIFO interface
  P4_sere_FIFO_sender:
    always (BtoS_ACK(0)  or BtoS_ACK(1) or not ENQ );

  P5_sere_FIFO_sender_0:
    always ({not BtoS_ACK(0)} |=> {ENQ or not BtoS_ACK(0)}!);

  P5_sere_FIFO_sender_1:
    always ({not BtoS_ACK(1)} |=> {ENQ or not BtoS_ACK(1)}!);

  P6_sere_FIFO_sender:
    always ({(StoB_REQ(0) or StoB_REQ(1)) and not FULL and not ENQ} |=> {
       ENQ; not ENQ}!);

  P7_sere_FIFO_sender_0:
    always ({not BtoS_ACK(0); BtoS_ACK(0)} -> SLC = 0);

  P7_sere_FIFO_sender_1:
    always ({not BtoS_ACK(1); BtoS_ACK(1)} -> SLC = 1);
}
```

FIGURE 4.6: SERE properties of GenBuf communication with senders in the case of two senders

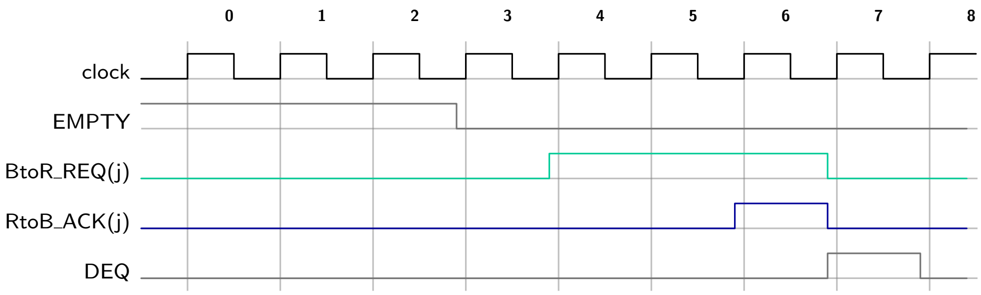


FIGURE 4.7: An example timeline of a GenBuf to receiver handshake

```
vunit genbuf_receiver
{
----- receiver side
  P0_rec:
    always(not EMPTY -> next!(BtoR_REQ(0) or ( BtoR_REQ(1))));

  P1_rec:
    always(EMPTY -> next!(not BtoR_REQ(0) and (not BtoR_REQ(1))) );

  P2_rec:
    always (not BtoR_REQ(0) or not BtoR_REQ(1));

  P3_rec_0:
    always ( rose (BtoR_REQ(0)) -> next! (next_event! (prev(not BtoR_REQ(0)
        )) (not BtoR_REQ(0) until_ (BtoR_REQ(1)))));

  P3_rec_1:
    always ( rose (BtoR_REQ(1)) -> next! (next_event! (prev(not BtoR_REQ(1)
        )) (not BtoR_REQ(1) until_ (BtoR_REQ(0)))));

  P4_rec_0:
    always ((BtoR_REQ(0)) and (not RtoB_ACK(0))-> next! (BtoR_REQ(0)));

  P4_rec_1:
    always ((BtoR_REQ(1)) and (not RtoB_ACK(1))-> next! (BtoR_REQ(1)));

  P5_rec_0:
    always ( (RtoB_ACK(0)) -> (next! (not BtoR_REQ(0))));

  P5_rec_1:
    always ( (RtoB_ACK(1)) -> (next! (not BtoR_REQ(1))));

---------- FIFO interface
  P6_FIFO_rec:
    always (( fell(RtoB_ACK(0)) or (fell(RtoB_ACK(1))) and not EMPTY) -> (
        DEQ));

  P7_FIFO_rec:
    always (not fell(RtoB_ACK(0)) and not fell(RtoB_ACK(1)) -> (not DEQ));
}
```

FIGURE 4.8: FL specification of GenBuf communication with receivers, in the case of two receivers

- `P3_rec_0`: Describes the round-robin scheme on the receiver side. Once receiver R_0 has been requested, it cannot be requested again before receiver R_1 is requested.
- `P4_rec_j`: The request to R_j remains high as long as the acknowledgment from R_j is not received.
- `P5_rec_j`: The request to R_j is deasserted one cycle after *RtoB_ACK(j)* is set by R_j.
- `P6_FIFO_rec`: If there is an acknowledgment from one of the receivers, and the buffer is not empty, then a data is read from FIFO (*DEQ* becomes high) in the falling edge of the acknowledgment signal.
- `P7_FIFO_rec`: No data is read from the FIFO as long as none of the acknowledgments has not been deasserted.

4.3.4.2 Formal SERE specification

The set of SERE properties that specify the communication between the GenBuf controller, the receivers, and the FIFO are shown in Fig. 4.9.

Here, a property is explained as an example.

- `P3_sere_rec_0`: Describes the round-robin scheme on the receiver side. Once receiver R_0 has been requested, we wait until deassertion of the request signal. Then, this request signal remains low until requesting R_1.

```
vunit genbuf_receiver_sere
{
  P0_sere_rec:
    always({not EMPTY} |=> {BtoR_REQ(0) or BtoR_REQ(1)}!);

  P1_sere_rec:
    always({EMPTY} |=> {not BtoR_REQ(0) and not BtoR_REQ(1)}! );

  P2_sere_rec:
    always (not BtoR_REQ(0) or not BtoR_REQ(1));

  P3_sere_rec_0:
    always ( {not BtoR_REQ(0); BtoR_REQ(0); {BtoR_REQ(0)[*]; not BtoR_REQ
        (0)}} |=> {(not BtoR_REQ(0))[*]; (prev(BtoR_REQ(1)))}!);

  P3_sere_rec_1:
    always ( {not BtoR_REQ(1); BtoR_REQ(1); {BtoR_REQ(1)[*]; not BtoR_REQ
        (1)}} |=> {(not BtoR_REQ(1))[*]; (prev(BtoR_REQ(0)))}!);


  P4_sere_rec_0:
    always ( {BtoR_REQ(0) and (not RtoB_ACK(0))} |=> {BtoR_REQ(0)}!);

  P4_sere_rec_1:
    always ( {BtoR_REQ(1) and (not RtoB_ACK(1))} |=> {BtoR_REQ(1)}!);

  P5_sere_rec_0:
    always ( {RtoB_ACK(0)} |=> {not BtoR_REQ(0)}!);

  P5_sere_rec_1:
    always ( {RtoB_ACK(1)} |=> {not BtoR_REQ(1)}!);

  P6_sere_FIFO_rec:
    always ((fell(RtoB_ACK(0)) or (fell(RtoB_ACK(1))) and not EMPTY) -> (
        DEQ));

  P7_sere_FIFO_rec:
    always (not fell(RtoB_ACK(0)) and not fell(RtoB_ACK(1)) -> (not DEQ));

}
```

Figure 4.9: SERE properties of GenBuf communication with receivers in the case of two receivers

Chapter 5

Synthesizing FLs

Contents

5.1 Introduction

In this chapter, we address how to synthesize an FL temporal operator, and provide a library of primitive reactants for FL operators. First, the dependency relation concept is introduced. Then, for each operator, the dependency relation between its operands are considered, and formalized. Based on the dependency relations and the formal semantics of the operators, a generic format is proposed for FL operators. Then, it is shown how each of the FL temporal operators can be mapped into this generic format, and be synthesized.

5.2 Formalization of the annotation

The purpose of this section is to formally define the notion of dependency between operands of a PSL operator, in order to have a synthezisable VHDL description of each operator and to mark signals. To this aim, we define a dependency relation which characterizes the conditions under which an argument of a temporal operator is free or constrained to a value, based on the value of the other argument.

The materials of this section are originally provided by Katell Morin-Allory and are published in [MAJB15]. Since they are the underlying formalism of the annotation, they have been brought here. Later, in Chapter 7 it is shown how these dependency relations can be used to annotate the signals in a property.

5.2.1 Dependency relation: definition and notations

For proving the dependency relations, we use the PSL semantic definitions in Appendix B of the IEEE Standard [FG05]. The essential concepts and notations have been already introduced in Chapter 2. Here, we review some of the required notations that were introduced in Chapter 2, Section 2.2.2.2.

- $\mathbf{P}$: a non-empty set of atomic propositions, in practice the set of signal names in a property
- $\Sigma = 2^{\mathbf{P}}$: the set of all possible valuations of $\mathbf{P}$
- letter: a letter, $\ell \in \Sigma$, is a valuation of all the propositions in $\mathbf{P}$.
- word: a word, w, is a sequence of letters ($w = \ell_0\ell_1\ell_2...$), and it stands for the succession over time of the signal values, i.e. an execution trace. If i and j are integers, $w^i = \ell_i$ is the $(i+1)^{th}$ letter of w; $w^{i..j} = \ell_i\ell_{i+1}\ldots\ell_j$ is the finite word starting at ℓ_i and ending at ℓ_j; and $w^{i\cdots} = \ell_i\ell_{i+1}\ldots$ is the suffix of w starting at w^i.
- The semantics of a Boolean expression *exp* over $\mathbf{P}$ is the set of all the letters of Σ on which *exp* takes value *true*.
- $\ell \vdash$ *exp* (*exp* is *true* in ℓ): means that *exp* takes value true if all its variables take their value as in ℓ.
- $w \models$ `property` (“`property`” is true on word w): is the extended semantics by structural induction over FL properties to words.

Example 1. Notations.

Consider property `P0_sender_0` from GenBuf sender.

```
P0_sender_0:
  always(not BtoS_ACK(0) and (not StoB_REQ(0)) -> next!(not BtoS_ACK(0)));
```

The set of atomic propositions is $\mathbf{P} = \{StoB\_REQ(0), BtoS\_ACK(0)\}$. Then, $\Sigma = \{<0,0>, <0,1>, <1,0>, <1,1>\}$, which is the set of all possible valuations of $\mathbf{P}$. Each element of Σ, ex.$<1,1>$, denotes a letter, ℓ. $w = (<1,0>, <1,1>, <0,1>, <0,0>)$ is an execution trace, which is the sequence of letters. Property `P0_sender_0` holds on trace w, since the $\ell \vdash$ exp implication holds on each letter of w (see Fig.5.1).

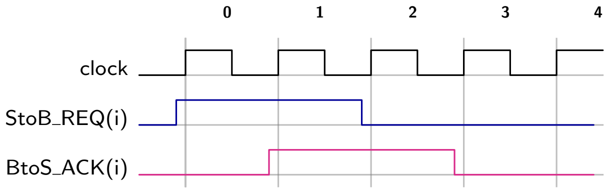


FIGURE 5.1: An execution trace for `P0_sender_0`

Definition 1. *Let w be a trace, A and B two FL formulas. The dependency relation between A and B is defined as follows:*

$$\lfloor A \lhd B \rfloor_w \iff w \models B \Rightarrow w \models A$$

When $\forall w$, $\lfloor A \lhd B \rfloor_w$ we can write: $A \lhd B$.

The relation $\lfloor A \lhd B \rfloor_w$ reads: on a trace w, the value of A depends on the value of B. For a trace w, if B is satisfied on w, A must be satisfied on w.

Based on this definition, we express some properties that are useful in proving the dependency relations in Section 5.2.2.

Property 1. $\lfloor A \lhd B \rfloor_w \wedge \lfloor A \lhd C \rfloor_w \Leftrightarrow \lfloor A \lhd (B \,\texttt{or}\, C) \rfloor_w$

PROOF.

$$\begin{aligned}
&\Leftrightarrow (w \models B \Rightarrow w \models A) \wedge (w \models C \Rightarrow w \models A)\\
&\Leftrightarrow (w \models \neg B \vee w \models A) \wedge (w \models \neg C | w \models A)\\
&\Leftrightarrow w \models A \vee (w \models \neg B \wedge w \models \neg C)\\
&\Leftrightarrow w \models A \vee (w \models (\neg B \,\texttt{or}\, \neg C))\\
&\Leftrightarrow w \models A \vee (w \models \neg(B \,\texttt{or}\, C))\\
&\Leftrightarrow w \models (B \vee C) \Rightarrow w \models A\\
&\Leftrightarrow \lfloor A \lhd (B \,\texttt{or}\, C) \rfloor_w
\end{aligned}$$

□

Property 2. $\lfloor A \lhd B \rfloor_w \vee \lfloor A \lhd C \rfloor_w \Leftrightarrow \lfloor A \lhd (B \,\texttt{and}\, C) \rfloor_w$

PROOF.

$$\begin{aligned}
&\Leftrightarrow (w \models B \Rightarrow w \models A) \vee (w \models C \Rightarrow w \models A)\\
&\Leftrightarrow (w \models \neg B \vee w \models A) \vee (w \models \neg C | w \models A)\\
&\Leftrightarrow w \models A \vee (w \models \neg B \vee w \models \neg C)\\
&\Leftrightarrow w \models A \vee (w \models (\neg B \,\texttt{or}\, \neg C))\\
&\Leftrightarrow w \models A \vee (w \models \neg(B \,\texttt{and}\, C))\\
&\Leftrightarrow w \models (B \,\texttt{and}\, C) \Rightarrow w \models A\\
&\Leftrightarrow \lfloor A \lhd (B \,\texttt{and}\, C) \rfloor_w
\end{aligned}$$

□

Property 3. $\lfloor (A \,\texttt{and}\, B) \lhd C \rfloor_w \Leftrightarrow \lfloor A \lhd C \rfloor_w \wedge \lfloor B \lhd C \rfloor_w$

PROOF.

$$\begin{aligned}
&\Leftrightarrow w \models C \Rightarrow (w \models A \,\texttt{and}\, B)\\
&\Leftrightarrow w \models C \Rightarrow (w \models A \wedge w \models B)\\
&\Leftrightarrow w \models \neg C \vee (w \models A \wedge w \models B)\\
&\Leftrightarrow (w \models \neg C \vee w \models A) \wedge (w \models \neg C \vee w \models B)\\
&\Leftrightarrow (w \models C \Rightarrow w \models A) \wedge (w \models C \Rightarrow w \models B)\\
&\Leftrightarrow \lfloor A \lhd C \rfloor_w \wedge \lfloor B \lhd C \rfloor_w
\end{aligned}$$

□

Property 4. $\lfloor (A \,\texttt{or}\, B) \lhd C \rfloor_w \Leftrightarrow \lfloor A \lhd (C \wedge \neg B) \rfloor_w$

PROOF.

$$\begin{aligned}
&\Leftrightarrow w \models C \Rightarrow (w \models A \,\texttt{or}\, B)\\
&\Leftrightarrow w \models C \Rightarrow (w \models A \vee w \models B)\\
&\Leftrightarrow w \models \neg C \vee (w \models A \vee w \models B)\\
&\Leftrightarrow w \models (\neg C \,\texttt{or}\, B) \vee w \models A\\
&\Leftrightarrow w \models \neg(C \wedge \neg B) \vee w \models A\\
&\Leftrightarrow (w \models (C \wedge \neg B) \Rightarrow w \models A\\
&\Leftrightarrow \lfloor A \lhd (C \wedge \neg B) \rfloor_w
\end{aligned}$$

□

Property 5. $\lfloor A \lhd B \rfloor_w \Leftrightarrow \lfloor \neg B \lhd \neg A \rfloor_w$

PROOF.

$$\begin{aligned}
&\Leftrightarrow w \models B \Rightarrow w \models A\\
&\Leftrightarrow w \models \neg B \vee w \models A\\
&\Leftrightarrow w \models \neg A \Rightarrow w \models \neg B\\
&\Leftrightarrow \lfloor \neg B \lhd \neg A \rfloor_w
\end{aligned}$$

□

Definition 2. *Let φ be a FL formula. A and B are two operands of φ. Let w be a trace. A depends on B in φ if: $\forall w, \lfloor \varphi \lhd true \rfloor_w \Leftrightarrow \lfloor A \lhd B \rfloor_w$.*

Property 6. *For any trace w, $\lfloor \lhd \rfloor_w$ is a partial order*

Proof.

Reflexivity: $A \lhd A$

$$\forall w, |w| > 0, (w \models A \Rightarrow w \models A)$$

This is evidently true.

No Symmetry: $\lfloor A \lhd B \rfloor_w \not\Rightarrow \lfloor B \lhd A \rfloor_w$

$$\begin{aligned} \forall w,&(w \models B \Rightarrow w \models A) \\ &\Rightarrow (w \models A \Rightarrow w \models B)? \end{aligned}$$

We replace $w \models A$ with a and $w \models B$ with b. Then, we should prove: $(b \Rightarrow a) \Rightarrow (a \Rightarrow b)$
This statement is not a tautology. Therefore, $\lfloor \lhd \rfloor_w$ is not symmetric.

Antisymmetry: $\lfloor A \lhd B \rfloor_w \wedge \lfloor B \lhd A \rfloor_w \Rightarrow A \Leftrightarrow B$

$$\begin{aligned} \forall w,&(w \models B \Rightarrow w \models A) \wedge (w \models A \Rightarrow w \models B) \\ &\Rightarrow (A \Leftrightarrow B)? \end{aligned}$$

By changing the variables as $a = w \models A$ and $b = w \models B$, we should prove: $(b \Rightarrow a) \wedge (a \Rightarrow b) \Leftrightarrow (a \Leftrightarrow b)$
This statement is a tautology.

Transitivity: $\lfloor A \lhd B \rfloor_w \wedge \lfloor B \lhd C \rfloor_w \Rightarrow \lfloor A \lhd C \rfloor_w$

$$\begin{aligned} \forall w,&(w \models B \Rightarrow w \models A) \wedge (w \models C \Rightarrow w \models B) \\ &\Rightarrow (w \models C \Rightarrow w \models A)? \end{aligned}$$

With the same variable changes as above, we need to prove:

$$(b \Rightarrow a) \wedge (c \Rightarrow b) \Rightarrow (c \Rightarrow a)$$

This formula is a tautology; $\lfloor \lhd \rfloor_w$ is transitive. The relation $\lfloor \lhd \rfloor_w$ is thus a partial order for any trace w. □

5.2.2 Dependency relation between operands of FL operators

For an FL formula the dependency between its operands is stated using the general $\lfloor \lhd \rfloor_w$ relation.

In the following, the dependency relation for each FL operator is presented.

5.2.2.1 Always

Dependency Rule 1. Always
Let $\varphi = \texttt{always}\, A$*, then*
$\lfloor \varphi \lhd \mathit{true} \rfloor_w$ *iff* $\forall i < |w|, \lfloor A \lhd \mathit{true} \rfloor_{w^{i\cdots}}$

Proof. We replace `always` by its semantic definition (see Appendix B of [FG05]).

$$\begin{aligned}
&\forall w, \lfloor \varphi \lhd \mathit{true} \rfloor_w \\
&\quad \Leftrightarrow w \models \mathit{true} \Rightarrow w \models \texttt{always}\, A \\
&\quad \Leftrightarrow w \models \mathit{true} \Rightarrow \forall i \in \mathbb{N}, |w| > i, w^{i\cdots} \models A \\
&\quad \Leftrightarrow \forall i < |w|, \lfloor A \lhd \mathit{true} \rfloor_{w^{i\cdots}}
\end{aligned}$$

□

5.2.2.2 Eventually!

Dependency Rule 2. Eventually!
Let $\varphi = \texttt{eventually!}\, A$*, then*
$\lfloor \varphi \lhd \mathit{true} \rfloor_w$ *iff* $\exists i < |w|, \lfloor A \lhd \mathit{true} \rfloor_{w^{i\cdots}}$

Proof. In the second line, we replace `eventually!` by its semantic definition.

$$\begin{aligned}
&\forall w, \lfloor \varphi \lhd \mathit{true} \rfloor_w \\
&\quad \Leftrightarrow w \models \mathit{true} \Rightarrow w \models \texttt{eventually!}\, A \\
&\quad \Leftrightarrow w \models \mathit{true} \Rightarrow \exists k < |w|, w^{k\cdots} \models A \wedge \forall i < k, w^{i\cdots} \models \mathit{true} \\
&\quad \Leftrightarrow w \models \mathit{true} \Rightarrow \exists k < |w|, w^{k\cdots} \models A \\
&\quad \Leftrightarrow \exists i < |w|, \lfloor A \lhd \mathit{true} \rfloor_{w^{i\cdots}}
\end{aligned}$$

□

5.2.2.3 Next family

Dependency Rule 3. Next![k]
Let $\varphi = \texttt{next!}[k]A$*, then*
$\lfloor \varphi \lhd \mathit{true} \rfloor_w$ *iff* $\lfloor A \lhd \mathit{true} \rfloor_{w^{k\cdots}}$

Proof. In the second line, `next!` is replaced by its semantic definition.

$$\begin{aligned}
&\forall w, \lfloor \varphi \lhd \mathit{true} \rfloor_w \\
&\quad \Leftrightarrow w \models \mathit{true} \Rightarrow w \models \texttt{next!}[k]A \\
&\quad \Leftrightarrow w \models \mathit{true} \Rightarrow |w| > k \wedge w^{k\cdots} \models A \\
&\quad \Leftrightarrow \lfloor A \lhd \mathit{true} \rfloor_{w^{k\cdots}}
\end{aligned}$$

□

Dependency Rule 4. Next_a!
Let $\varphi = \texttt{next\_a!}[i \text{ to } j]A$, *then*
$\lfloor \varphi \lhd \mathit{true} \rfloor_w$ *iff* $\forall k \in [i..j]$, $\lfloor A \lhd \mathit{true} \rfloor_{w^{k\ldots}}$

Proof. $\texttt{next\_a!}[i \text{ to } j]A$ can be rewritten as:

$$(\texttt{next!}[i]A) \wedge (\texttt{next!}[i+1]A) \wedge .... \wedge (\texttt{next!}[j]A)$$

Using the Dependency Rule 3, and substituting each next! operator, the dependency relations is proved easily. □

Dependency Rule 5. Next_e!
Let $\varphi = \texttt{next\_e!}[i \text{ to } j]A$, *then*
$\lfloor \varphi \lhd \mathit{true} \rfloor_w$ *iff* $\exists k \in [i..j]$, $\lfloor A \lhd \mathit{true} \rfloor_{w^{k\ldots}}$

Proof. $\texttt{next\_e!}[i \text{ to } j]A$ can be rewritten as:

$$(\texttt{next!}[i]A) \vee (\texttt{next!}[i+1]A) \vee .... \vee (\texttt{next!}[j]A)$$

The proof is done simply by mathematical rewriting of the above formula. □

5.2.2.4 Until family

Dependency Rule 6. Until!
Let $\varphi = A \ \texttt{until!} \ B$, *then*
$\lfloor \varphi \lhd \mathit{true} \rfloor_w$ *iff* $\exists k < |w|$, $\lfloor B \lhd \mathit{true} \rfloor_{w^{k\ldots}} \wedge \forall i < k$, $\lfloor A \lhd \neg B \rfloor_{w^{i\ldots}}$

Proof.

$$\begin{aligned}
&\forall w \lfloor \varphi \lhd \mathit{true} \rfloor_w \\
&\quad \Leftrightarrow w \models \mathit{true} \Rightarrow w \models A \,\texttt{until!}\ B \\
&\quad \Leftrightarrow w \models \mathit{true} \Rightarrow \exists k < |w|, w^{k\cdots} \models B \wedge \forall i < k, w^{i\cdots} \models A \\
&\quad \Leftrightarrow w \models \mathit{true} \Rightarrow \exists k < |w|, w^{k\cdots} \models B \wedge \forall i < k, w^{i\cdots} \models A \,\texttt{or}(B \,\texttt{and}\, \neg B) \\
&\quad \Leftrightarrow w \models \mathit{true} \Rightarrow \exists k < |w|, w^{k\cdots} \models B \wedge \forall i < k, (w^{i\cdots} \models A \vee w^{i\cdots} \models B) \\
&\quad \wedge (w^{i\cdots} \models A \vee w^{i\cdots} \models \neg B) \\
&\quad \Leftrightarrow w \models \mathit{true} \Rightarrow \exists k < |w|, w^{k\cdots} \models B \wedge \forall i < k, w^{i\cdots} \models \neg B \Rightarrow w^{i\cdots} \models A \\
&\quad \wedge (w^{i\cdots} \models A \vee w^{i\cdots} \models \neg B) \\
&\quad \Leftrightarrow w \models \mathit{true} \Rightarrow \exists k < |w|, \lfloor B \lhd \mathit{true} \rfloor_{w^{k\ldots}} \wedge \forall i < k, \lfloor A \lhd \neg B \rfloor_{w^{i\ldots}} \\
&\quad \wedge (w^{i\cdots} \models A \vee w^{i\cdots} \models \neg B)
\end{aligned}$$

In the last equivalence, if we assume that k is the least integer such that $w \models B$, $(w^{i\cdots} \models A \vee w^{i\cdots} \models \neg B)$ can be simplified to *true*. Then, the last equivalence is simplified to:

$$\begin{aligned}
&\Leftrightarrow w \models \mathit{true} \Rightarrow \exists k < |w|, \lfloor B \lhd \mathit{true} \rfloor_{w^{k\ldots}} \wedge \forall i < k, \lfloor A \lhd \neg B \rfloor_{w^{i\ldots}} \\
&\Leftrightarrow \exists k < |w|, \lfloor B \lhd \mathit{true} \rfloor_{w^{k\ldots}} \wedge \forall i < k, \lfloor A \lhd \neg B \rfloor_{w^{i\ldots}}
\end{aligned}$$

□

In this dependency relation, if A and B are both Boolean, the dependency relation $\lfloor A \lhd \neg B \rfloor_{w^{i\ldots}}$ can be reversed (see Property 5).

Dependency Rule 7. Until
Let $\varphi = A$ `until` B*, then*

$$\forall w, \begin{cases} \exists k < |w|, \lfloor B \lhd \mathit{true} \rfloor_{w^{k\ldots}} \wedge \forall i < k, \lfloor A \lhd \neg B \rfloor_{w^{i\ldots}} \\ \text{or} \\ \forall i < |w|, \lfloor A \lhd \mathit{true} \rfloor_{w^{i\ldots}} \end{cases}$$

Proof. The first dependency is like the dependency rule of the `until!` operator. Since `until` is a weak operator, B may never occur, and $\forall i < |w|, \lfloor A \lhd \mathit{true} \rfloor_{w^{i\ldots}}$.
□

5.2.2.5 Before family

Dependency Rule 8. Before!
Let $\varphi = A$ `before!` B*, then*
$\lfloor \varphi \lhd \mathit{true} \rfloor_w$ *iff* $\exists k < |w|, \lfloor \neg B\, \mathtt{and}\, A \lhd \mathit{true} \rfloor_{w^{k\ldots}} \wedge \forall i < k, \lfloor \neg B \lhd \mathit{true} \rfloor_{w^{i\ldots}}$

Proof. `before!` can be rewritten using the `until!` operator.

$$\forall w \lfloor \varphi \lhd \mathit{true} \rfloor_w$$
$$\Leftrightarrow w \models \mathit{true} \Rightarrow w \models (\neg\, B)\, \mathtt{until!}\, (A\ \mathtt{and}\, \neg B)$$

Using the Dependency Rule 6 for `until!` we have:

$$\forall w \lfloor \varphi \lhd \mathit{true} \rfloor_w$$
$$\Leftrightarrow w \models \mathit{true} \Rightarrow \exists k < |w|, \lfloor A\, \mathtt{and}\, \neg B \lhd \mathit{true} \rfloor_{w^{k\ldots}} \wedge \forall i < k, \lfloor \neg B \lhd \neg A\, \mathtt{or}\, B \rfloor_{w^{i\ldots}}$$

The above statement can be rewritten using Property 1, as follows:

$$\Leftrightarrow w \models \mathit{true} \Rightarrow \exists k < |w|, \lfloor A\, \mathtt{and}\, \neg B \lhd \mathit{true} \rfloor_{w^{k\ldots}} \wedge \forall i < k, \lfloor \neg B\, \mathtt{or} (\neg B\, \mathtt{and}\, A) \lhd \mathit{true} \rfloor_{w^{i\ldots}}$$
$$\Leftrightarrow w \models \mathit{true} \Rightarrow \exists k < |w|, \lfloor A\, \mathtt{and}\, \neg B \lhd \mathit{true} \rfloor_{w^{k\ldots}} \wedge \forall i < k, \lfloor \neg B \lhd \mathit{true} \rfloor_{w^{i\ldots}}$$
$$\Leftrightarrow \exists k < |w|, \lfloor \neg B\, \mathtt{and}\, A \lhd \mathit{true} \rfloor_{w^{k\ldots}} \wedge \forall i < k, \lfloor \neg B \lhd \mathit{true} \rfloor_{w^{i\ldots}}$$

□

5.2.2.6 Next_event family

Dependency Rule 9. Next_event!
Let $\varphi =$ `next_event!`$(B)A$*, then* $\lfloor \varphi \lhd \mathit{true} \rfloor_w$ *iff*
$\exists k < |w|, \lfloor B \lhd \mathit{true} \rfloor_{w^{k\ldots}} \wedge \forall i \leq k, \lfloor A \lhd B \rfloor_{w^{i\ldots}}$

Proof. We rewrite `next_event!` using the `until!` operator, as follows:

$$\forall w \lfloor \varphi \lhd \mathit{true} \rfloor_w$$
$$\Leftrightarrow w \models \mathit{true} \Rightarrow w \models (\neg\, B)\, \mathtt{until!}\, (A\ \mathtt{and}\, B)$$

Using the Dependency Rule 6 for until! we have:

$$\forall w \lfloor \varphi \lhd true \rfloor_w$$
$$\Leftrightarrow w \models true \Rightarrow \exists k < |w|, \lfloor A \texttt{ and } B \lhd true \rfloor_{w^{k \dots}} \wedge \forall i < k, \lfloor \neg B \lhd \neg A \texttt{ or } \neg B \rfloor_{w^{i \dots}}$$

The above statement can be rewritten using Property 3, and Property 1, as follows:

$$\Leftrightarrow w \models true \Rightarrow \exists k < |w|, \lfloor A \lhd true \rfloor_{w^{k \dots}} \wedge \lfloor B \lhd true \rfloor_{w^{k \dots}} \wedge \forall i < k, \lfloor \neg B \lhd \neg B \rfloor_{w^{i \dots}}$$
$$\wedge \lfloor \neg B \lhd \neg A \rfloor_{w^{i \dots}}$$
$$\Leftrightarrow w \models true \Rightarrow \exists k < |w|, \lfloor A \lhd true \rfloor_{w^{k \dots}} \wedge \lfloor B \lhd true \rfloor_{w^{k \dots}} \wedge \forall i < k, \lfloor \neg B \lhd \neg A \rfloor_{w^{i \dots}}$$

Using Property 5, we have:

$$\Leftrightarrow w \models true \Rightarrow \exists k < |w|, \lfloor B \lhd true \rfloor_{w^{k \dots}} \wedge \forall i \leq k, \lfloor A \lhd B \rfloor_{w^{i \dots}}$$

□

All the other FL operators are defined as expressions involving the operators above. Their dependency rules are derived from the rewrite rules provided by the PSL standard[FG05].

5.3 Dependency relation synthesis

Each operator is a primitive reactant. In order to synthesize PSL$_{simple}$ properties into circuits, a library of FL primitive reactants is provided. To this goal, we give a hardware interpretation of the dependency relation $\lfloor \varphi \lhd true \rfloor_w$, where φ stands for a call to any of the FL operators, and Ω stands for the FL temporal operator (dependency Rules 1 to 9 above). For example if $\varphi = A \texttt{ until! } B$, Ω is the operator until!.

Here, we address how to implement the FL primitive reactants. The complex reactants are constructed recursively from these primitives (see Chapter 8).

5.3.1 Principles of the primitive reactant construction

The primitive reactants have a general interface: they take *clock* and *reset* as the synchronization signals. Each primitive reactant has a *start* signal for its activation (see Fig.5.2).

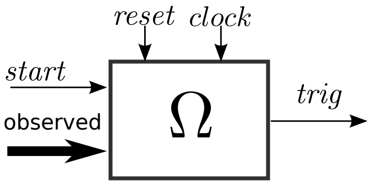


FIGURE 5.2: Generic interface of a primitive reactant

The operands of an FL operator, Ω, are *observed* or *constrained* by the primitive reactant of Ω during its activity. Thus, the output of a reactant is *not* the value of a signal, but the *trigger* that will *start* the primitive hardware component in charge of the signal value generation or observation (see Fig. 5.2).

In particular, for a Boolean signal S, triggering $S = 0$ and $S = 1$ is done by two distinct signals $Trig_S$ and $Trig_{\neg S}$. In the following, without loss of generality, we shall only consider the positive case $S = 1$. $Trig_S$ is set to 1 by the reactant at all cycles when the dependency $\lfloor S \lhd true \rfloor_w$ holds.

Let $\mathbf{P}$ be the set of the signal names in φ; i.e. the operands of Ω. The operands may be observed or constrained. Therefore, we split $\mathbf{P}$ in two sets $\mathbf{P}_{in}$ and $\mathbf{P}_{out}$ ($\mathbf{P} = \mathbf{P}_{in} \cup \mathbf{P}_{out}$), that contain observed and constrained operands respectively.

Circuit $\mathcal{C}$ is the circuit that implements a primitive reactant.

The set $\mathbf{P}_\mathcal{C}$ of atomic propositions for the circuit $\mathcal{C}$ is given by $\mathbf{P}_\mathcal{C} = \mathbf{P}_{in} \cup \{reset, start\} \cup \{Trig_S, Trig_{\neg S} \mid S \in \mathbf{P}_{out}\}$. We denote $\Sigma_\mathcal{C}$ the alphabet built on $\mathbf{P}_\mathcal{C}$.

Let $w_\mathcal{C}$ be a trace built on $\Sigma_\mathcal{C}$ and w_φ (or w) a trace built on Σ.

Definition 3. *$w_\mathcal{C}$ is equivalent to w on $\mathbf{P}_{in}$, denoted $w_\mathcal{C} \equiv_{|\mathbf{P}_{in}} w$ iff $\exists j$ such that*

- $w_\mathcal{C}^j \vdash start$ *and* $\exists k < j$ *such that* $w_\mathcal{C}^{k-1} \vdash reset$ *and* $\forall l, k < l \leq j$, $w_\mathcal{C}^l \vdash \neg reset$ *and* $w_\mathcal{C}^l \vdash \neg start$

- $\forall i, {w_\mathcal{C}^{j+i}}_{|\mathbf{P}_{in}} = {w^i}_{|\mathbf{P}_{in}}$

Circuit $\mathcal{C}$ is activated after being reset, when the *start* signal becomes 1. It means that there is a time point in the trace ($w_\mathcal{C}^j$) where *start* becomes 1. Before this point, $\mathcal{C}$ has been reset and it is not active. After this point, $w_\mathcal{C}$ and $w1$ should have identical values for the atomic propositions in $\mathbf{P}_{in}$; i.e. circuit trace $w_\mathcal{C}$ is equivalent to w on $\mathbf{P}_{in}$. For the sake of simplicity, we assume that $j = 0$.

Definition 4. *$w_\mathcal{C}$ is equivalent to w, denoted $w_\mathcal{C} \equiv w$, iff*

- $w_\mathcal{C} \equiv_{|\mathbf{P}_{in}} w$

- $\forall S \in \mathbf{P}_{out}, \forall i, w_\mathcal{C}^i \vdash Trig_S \implies w^i \vdash S$

- $\forall S \in \mathbf{P}_{out}, \forall i, w_\mathcal{C}^i \vdash Trig_{\neg S} \implies w^i \vdash \neg S$

A trace $w_\mathcal{C}$ is equivalent to w, if they are equivalent on inputs. Additionally, for each output signal if its trigger takes value *true* in $w_\mathcal{C}^i$, the signal takes value *true* in w^i.

Definition 5. *A circuit $\mathcal{C}$ implements the dependency $\lfloor \varphi \lhd True \rfloor_w$ iff $w_\mathcal{C} \equiv w$ and $\forall i, w_\mathcal{C}^i \vdash start \rightarrow \lfloor \varphi \lhd true \rfloor_{w^{i\ldots}}$*

Definition 6. *A circuit $\mathcal{C}$ is a primitive reactant circuit that implements the temporal formula φ, iff $\mathcal{C}$ implements the dependency $\lfloor \varphi \lhd true \rfloor_w$ for all traces w, and we write: $\mathcal{C} \Vdash \varphi$.*

In this chapter, we address how to synthesize $\mathcal{C}$ for FL primitive reactants.

5.3.1.1 Boolean reactant

The simplest primitive reactant implements Boolean expressions. A Boolean expression can be written as one of the following: a constant in {0,1}, a signal or the negation of a

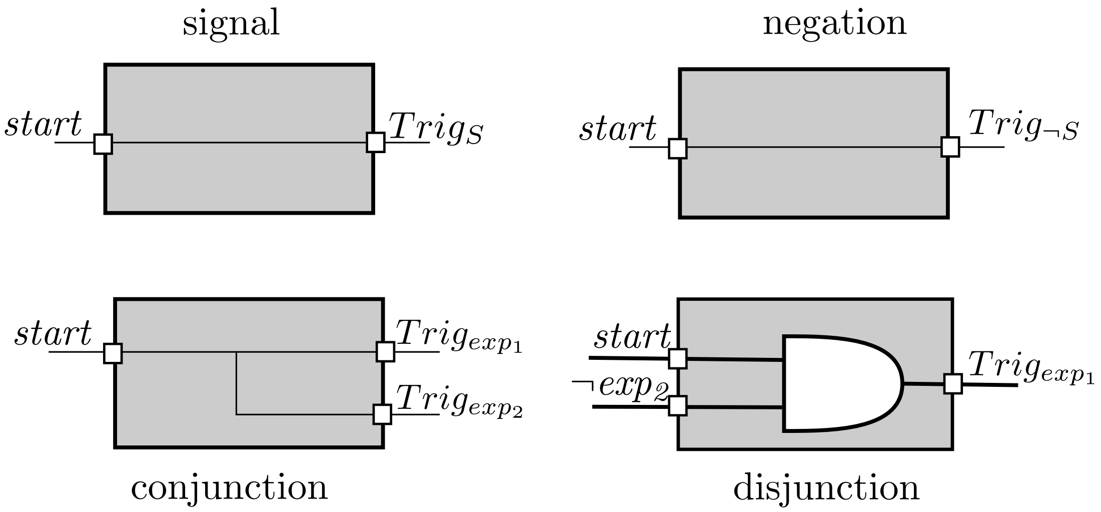


Figure 5.3: Boolean reactant

signal, the conjunction or disjunction of two Boolean expressions. Fig. 5.3 shows the four different implementations for a Boolean reactant.

In the case of a disjunction, when the reactant is started, if all the signals of one operand are in $\mathbf{P}_{in}$ (exp_2 in Fig. 5.3), the operand is observed: in that case, there is no constraint on the other operand if exp_2 is known to be 1. Due to the commutativity of OR, the roles of exp_1 and exp_2 can be exchanged. It may happen that deciding which operand is observed cannot be decided locally to a property: this is discussed in Chapter 9.

5.3.2 Generic format of a FL operator

We proposed a generic format for all FL operators [MAJB15]. This format is based on the operator semantics definition. It is shown that all primitive reactants for the temporal operators can be built from a few number of elements that are the circuit counterpart of the constituting elements of the operator semantic definition.

Considering the dependency rule of the temporal operator shows that each dependency is a special case of one of two following generalized expressions

1 the *"forall"* family includes: `always`, `until`, `next!`, `next_a`, `next_event`, `next_event_a`.

2 the *"exists"* family includes: `eventually!`, `before`, `next_e`, `next_event_e`.

The "forall" and "exists" generalized expressions have the following format:

$$\forall i \in [k_{min}, k_{max}], \lfloor exp \lhd cond \rfloor_{w^{k_i}} \tag{5.1}$$

$$\exists i \in [k_{min}, k_{max}], \lfloor exp \lhd cond \rfloor_{w^{k_i}} \tag{5.2}$$

In the above formulas, exp and $cond$ are two Booleans, and min and max are two naturals such that $max \geq min$. k_{min} and k_{max} are computed using a counting function, I_{th}. The I_{th} function returns the number of times that its operand, a formula F computed on trace w, has been $true$ on $w^{0..k}$. We are interested in the first time point for each number of occurrences of formula $F = true$. This consists in numbering the time points of trace w that satisfy:

$I_{th}(\lfloor F \lhd true \rfloor_{w^{k_i}}) = i \wedge \lfloor F \lhd true \rfloor_{w^{k_i}}, \forall i \in \mathbb{N}$

The sequence $\{k_0, k_1, ..., k_i ...\}$ is the set of these time points (see Fig. 5.4).

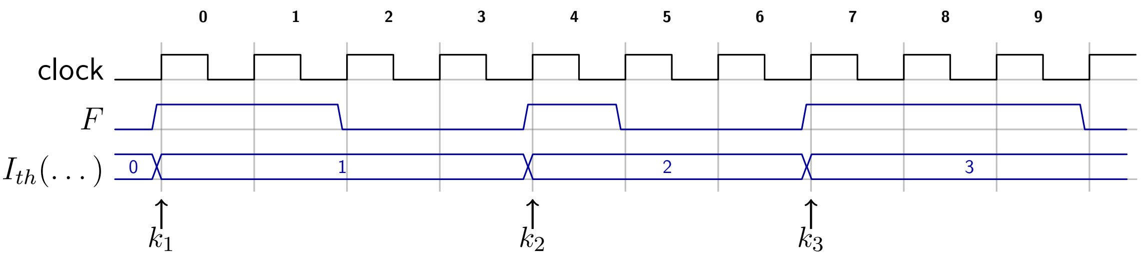


FIGURE 5.4: Illustration of function $I_{th}(\lfloor F \lhd true \rfloor_{w^{k_i}})$

The values of min, max, exp, $cond$ and F depend on the temporal operator. Table 5.1 gives their value for each PSL FL operator. Column F specifies what is counted by function I_{th}. When $F = true$, I_{th} counts clock cycles. Otherwise, it counts each time point k in the trace where a dependency relation holds. The two columns opt. _ and opt. !, indicate if the operator can have overlap/non_overlap and strong/weak options. As was mentioned earlier, based on the dependency rules, operators $Until$ and $Before$ have two versions, depending on whether their left operand A is observed or generated.

Temporal Operator	F	*min*	*max*	*cond*	*exp*	opt. _	opt. !
`always` A	*true*	0	$\mid w \mid$	*true*	A	no	no
A `until` B (1)	B	0	1	$\neg B$	A	yes	yes
A `before` B (1)	$\neg B$	0	1	$\neg A$	$\neg B$	yes	yes
`next!`$[i]A$	*true*	i	i	*true*	A	no	yes
`next_a`$[i\ to\ j]A$	*true*	i	j	*true*	A	no	yes
`next_event`$[i](B)A$	B	i	i	B	A	no	yes
`next_event_a`$[i\ to\ j](B)A$	B	i	j	B	A	no	yes
`eventually!` A	A	0	1	*true*	A	no	no
A `until` B (2)	B	0	1	$\neg A$	B	no	yes
A `before_` B (2)	$\neg B$	0	1	B	A	no	yes
`next_e`$[i\ to\ j]A$	*true*	i	j	*true*	A	no	yes
`next_event_e`$[i\ to\ j](B)A$	B	i	j	*true*	A	no	yes

Table 5.1: Values of parameters for forall (top) and exists (bottom) expressions

In the following, the implementation of the I^{th} function is explained. Then, the implementation of the "forall" and "exists" expressions will be discussed.

Implementation of function I_{th}. The counting function is implemented as a generic shift register, instantiated with a parameterized number of cells (Fig. 5.5). The shift register reads data one bit at a time on input s_in. The content of the register can be read serially on output s_out, and in parallel on output p_out. The *shift* input control signal shifts the register. The other input control signal, *clear*, clears the register. By default, *shift* is 1 (it shifts at all cycles) and *clear* is 0 (the input value is propagated).

Since PSL operators are re-entering, the shift register may count events simultaneously starting from several distinct start times.

5.3.2.1 Implementation of an operator of the "forall" group

Fig. 5.6 illustrates the implementation of the "forall" expression. It is based on the interconnection of 4 components: Min, Max, ForAll, and Dep. In the following, in case of ambiguity, we prefix a signal name with the component name when we do not mean the global module interface signal.

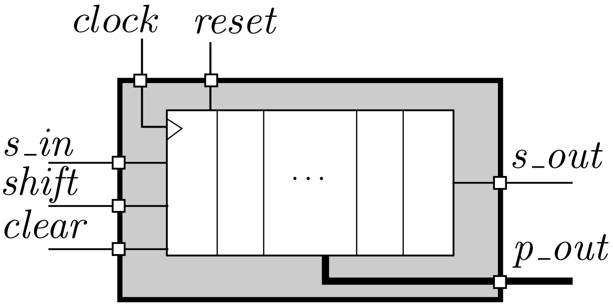


Figure 5.5: Interface of the shift register

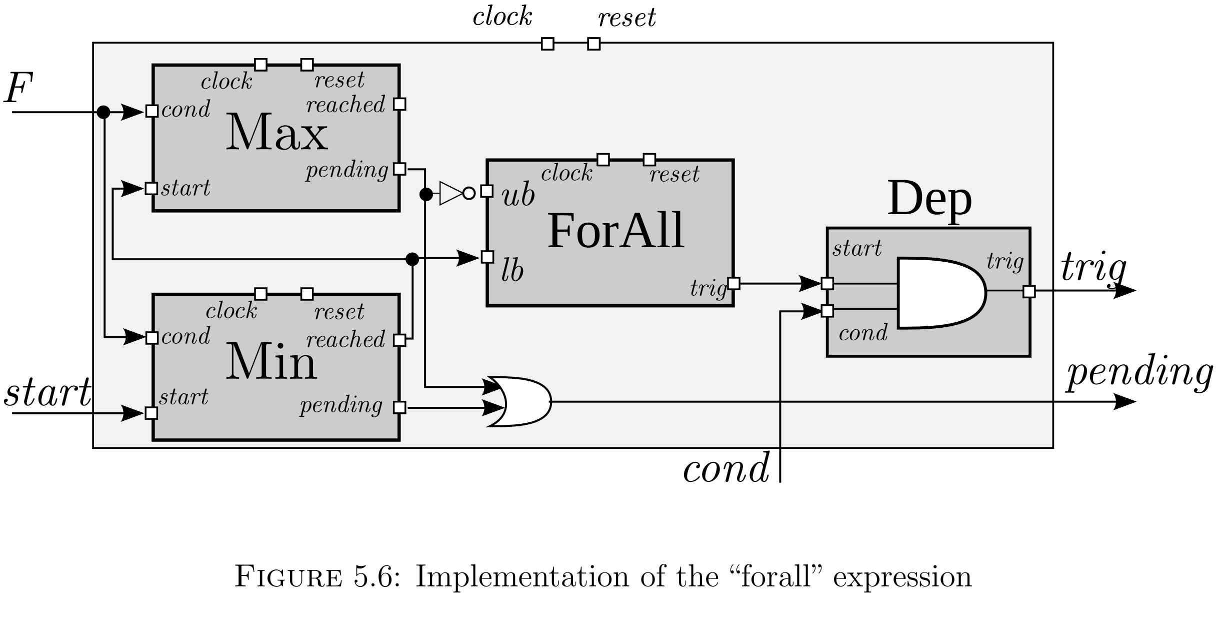
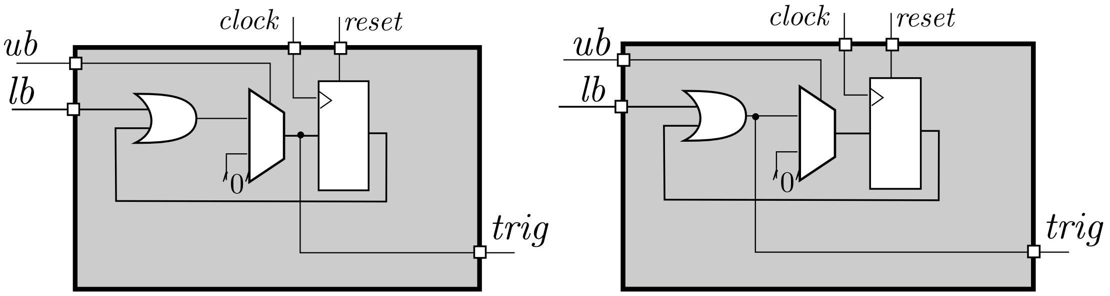


Figure 5.6: Implementation of the "forall" expression

- Component Dep (for the dependency $\lhd$) is a mere AND gate that implements the expression $\lfloor exp \lhd cond \rfloor_{w^k}$. It triggers the evaluation of exp depending on the value of $cond$.

- Component ForAll implements the $\forall i \in \lfloor k_{min}, k_{max} \rfloor$ expression. It is used to trigger the evaluation of the dependency relation at all times between the two bounds k_{min} and k_{max} that are connected to its inputs lb and ub. The signal ForAll.$trig$ is asserted at all times between lb and ub. Depending on whether the operator is overlapping or not, two versions are used (see Fig. 5.7).



(a) non overlapping

(b) overlapping

FIGURE 5.7: Implementation of $\forall i \in [lb, ub]$

- Component Min(Max) takes $start$ and $cond$ as its inputs. The $start$ signal initiates counting of the occurrences of F on its input Min.$cond$ (Max.$cond$). The Min and Max components embed a shift register of size min for Min, of size $max - min$ for Max. If min is 0, the shift register is just a wire. If max is unbounded, component max is the ground. (see columns min and max of Table 5.1 for finding the value of min and max for each operator). The $reach$ output signal takes value 1 when the min (max) value is found. The Min.$pending$ (Max.$pending$) output signal is the output of the embedded shift register in Min(Max), and is 1 as long as the min (max) value is not found. The global output $pending$ is the OR of Min.$pending$ and Max.$pending$, and is 1 between the min-th and the max-th occurrences of F after $start = 1$.

All the FL operators of this group observe F, which can be an operand of an operator (e.g. B in `next_event`$(B)A$), and constrain exp (ex. A in `next_event`$(B)A$).

Example 2. Implementation of operator `next!`$[i]$.

$\varphi =$ `next!`$[i]A$ is a special case of the "forall" expression (see Dependency Rule 3):

$$\forall i \in [k_{min}, k_{max}], \lfloor exp \lhd cond \rfloor_{w^{k_i}}$$

In the above expression, $[min = max = i]$, and formula $F = B = true$ (see Table 5.1). Therefore, the shift register of Min has i cells, and component Max is a wire, and Min counts clock cycles. Fig. 5.8 shows the implementation of φ.

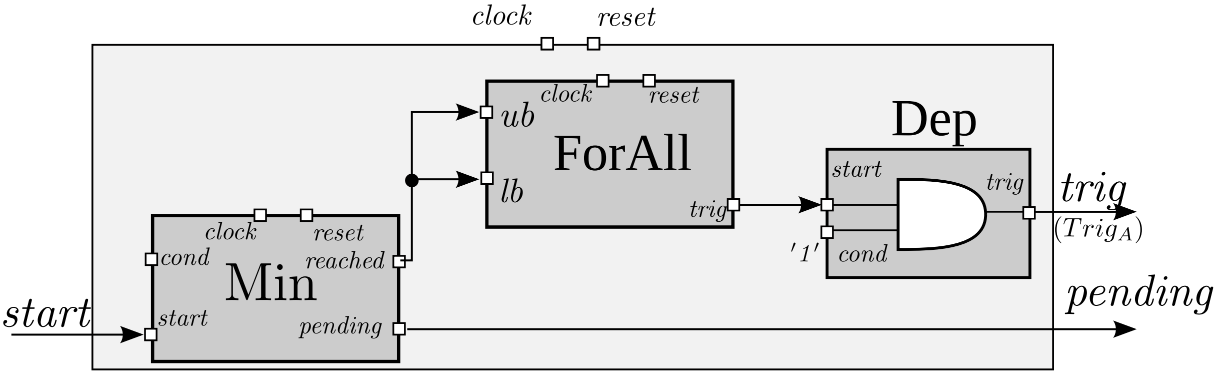


FIGURE 5.8: Implementation of `next!`$[i]$

Example 3. Implementation of `next_event_a!`$[i\ to\ j](B)A$.

$\varphi =$ `next_event_a!`$[i\ to\ j](B)A$ corresponds to the general "forall" expression, where $[min = i, max = j]$, and $F = B$. Therefore, two full Min and Max components are used (see Fig. 5.9).

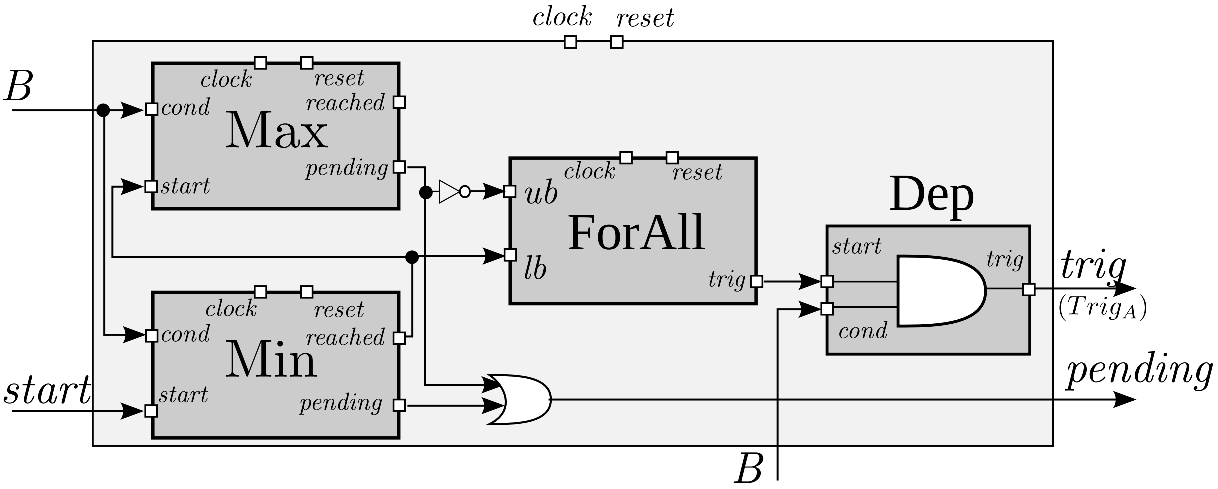


FIGURE 5.9: Implementation of `next_event_a!`$[i\ to\ j](B)A$

Example 4. Implementation of A `until!` B.

$\varphi = A$ `until!` B corresponds to the general "forall" expression, where $[min = 0, max = 1]$, and $F = B$. Thus, the Max component is 1-bit register. Once B is observed, the Dep component is activated, and implements the $\lfloor A \lhd \neg B \rfloor$ dependency relation (see Fig. 5.10). So, the operator constrains A.

5.3.2.2 Implementation of an operator of the "exists" group

The implementation of the "exists" expression is shown in Fig. 5.11. Component Min and Max observe the formula F, and count the numbers of its occurrence in the $[k_{min}, k_{max}]$ interval.

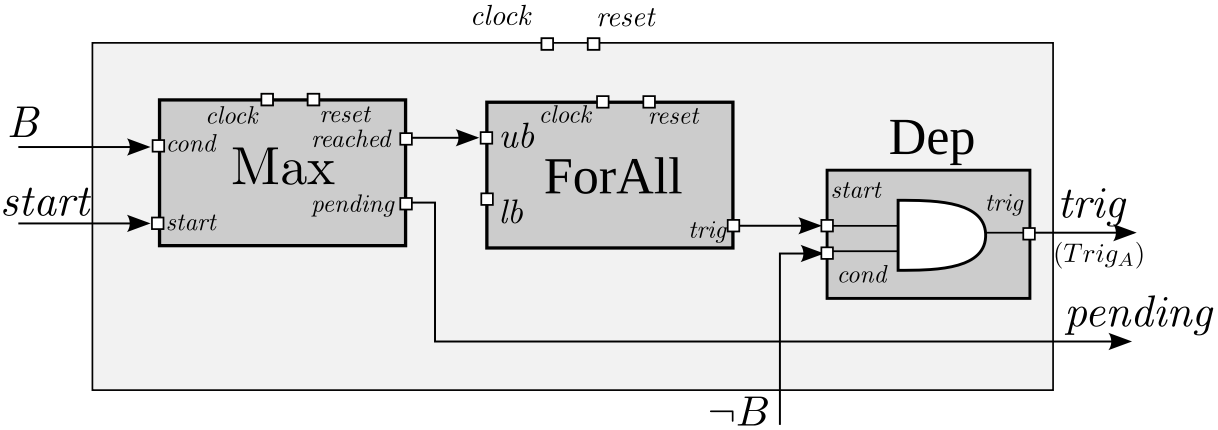


FIGURE 5.10: Implementation of A `until`! B

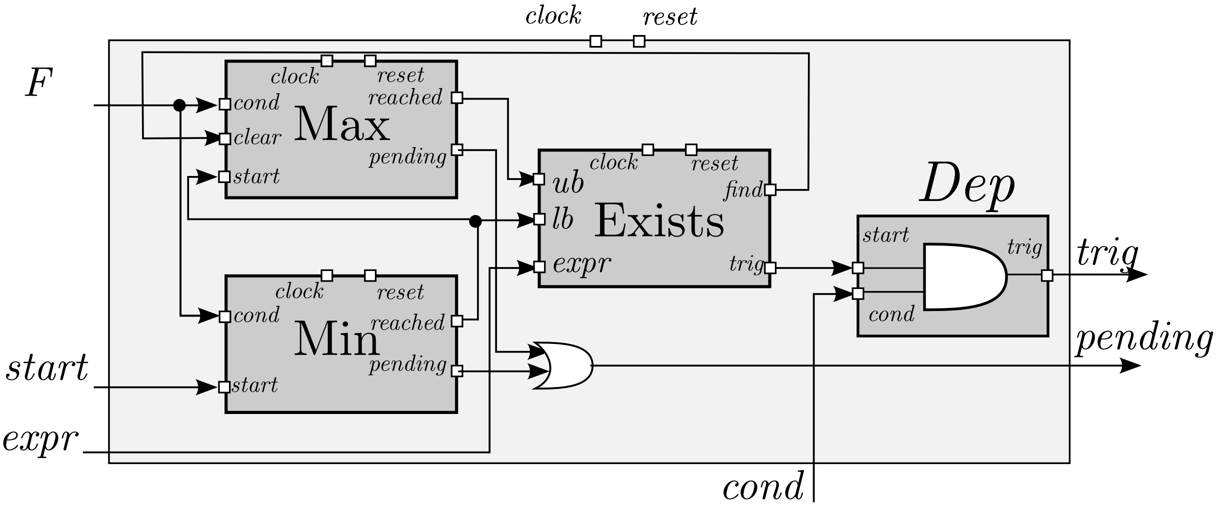


FIGURE 5.11: Implementation of the "exists" expression

The Exists component inputs *exp*, *lb*, and *ub* (see Fig. 5.12). If $exp = 1$ has not been met in the $[k_{min}, k_{max}]$ interval, Exists triggers the evaluation of the dependency relation at time k_{max} (Exists.$trig = 1$). Otherwise, if *exp* is 0 in this interval, the Exists component has no external effect, and Exists.$find = 0$. Output Exists.$find$ is connected to the *clear* input of component Max to stop the counting when $exp = 1$ has been met.

Simultaneous executions of an "exists" expression that has been started several times may be cleared by a single occurrence of $exp = 1$.

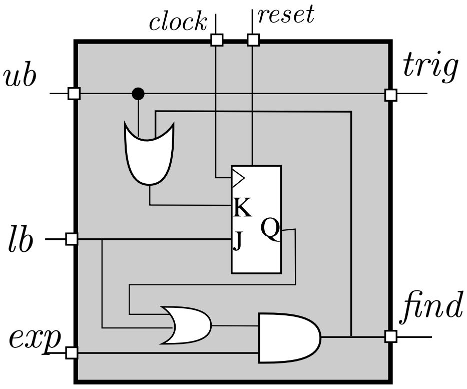


FIGURE 5.12: Implementation of $\exists i \in [lb, ub]$

Example 5. *Implementation of operator* `next_e`$[i\ to\ j]$.

$\varphi =$ `next_e`$[i\ to\ j]A$ is a simple case of the "exists" expression. Since $F = B = true$, the clock cycles should be counted between $min = i$ and $max = j$, while $exp = A = 0$. Whenever A becomes 1, Exists.$trig$ is set to 1. Since the *cond* input of the Dep component is is 1, the rightmost AND gate is eliminated, and $trig =$ Exists.$Trig$ (see Fig. 5.13)

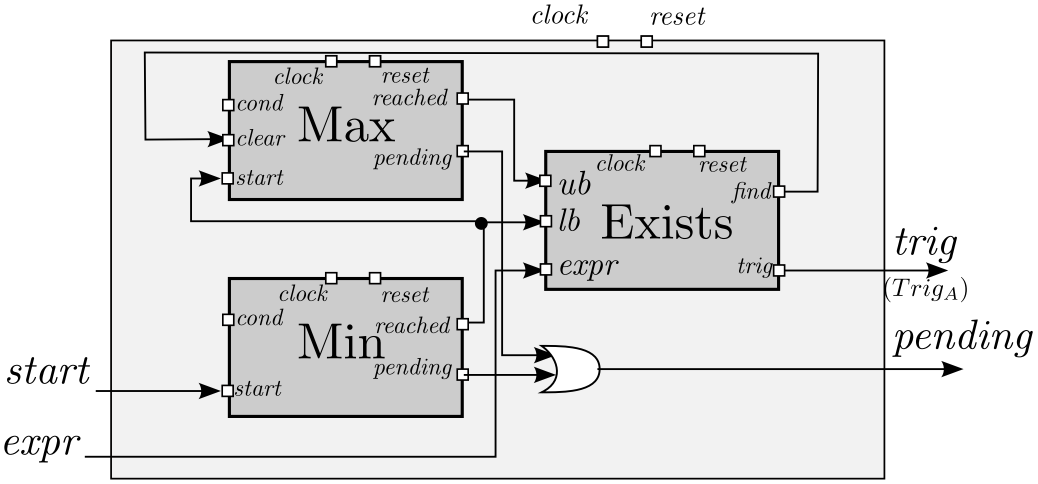


FIGURE 5.13: Implementation of `next_e`$[i\ to\ j]A$

Remark.

It should be mentioned that this implementation of the temporal operators is not optimal. However, it facilitates the proof of correctness of their compliance with the formal trace semantics that is provided in the PSL standard document [FG05]. In the library of our prototype system, all the operators have been simplified, and their optimized version proven equivalent to the one presented here.

5.4 Summary

In this chapter, we addressed how to provide a correct-by-construction library of primitive reactants for FL temporal operators of PSL_{simple}. First, the concept of the dependency relation $\lfloor\triangleleft\rfloor_w$ is introduced. For each FL temporal operator of PSL_{simple}, the relation between its operands are defined formally based on the formal trace semantics of the operator. Based on these semantics, a generic format is proposed for the operators. All the FL temporal operators of PSL_{simple} are special cases of a general dependency expression, either universally (“forall” expression) or existentially (“exists” expression) quantified. Then, some elements are introduced for constructing the quantifiers, and “forall” and “exists” expressions. Based on these constructions, a hardware interpretation for the mathematical formulas of the temporal FL operators’ dependency relations is given. The presented dependency relations are the underlying formalism of the annotation (see Chapter 7). Later in Chapter 8, we show how these primitive reactants are used to construct the complex reactant of a property.

Chapter 6

Synthesizing SEREs

Contents

6.1 Introduction

In this chapter we discuss the synthesis principles of SEREs[1]. SEREs are very similar to the sequences in SVA[2]. SEREs are a convenient way to express the signals' waveforms, by writing simple properties of the form:

{*observe*} |=> {*observe*}
or
{*observe*} |=> {*generate*}

These properties can represent the environment behavior or a communication protocol.

First, using some examples we demonstrate some of the difficulties and challenges that we should deal with in considering SEREs.

Then, we address how to provide a library of primitive reactants for SERE operators. We start by formalizing the relationships between the operands of a SERE operator based on its trace semantics. Then, we categorize SERE operators, and show with some examples how a SERE operator of each category can be implemented.

Later in Chapter 8 we show how these primitive reactants are used to construct the complex reactant of a SERE property.

Finally, we introduce a synthesizeable subset of SEREs.

6.2 Challenges and motivations

We can rewrite some SEREs as FLs[3] (see the rewriting rules in [MABBZ08]). In this case, we do not need to synthesize SEREs, and the provided library of FLs is sufficient. However, SEREs cannot be always translated to FLs. Properties that involve some special form of counting cannot be expressed in PSL without using SEREs. If we want to rewrite such properties in FLs, we may need to define auxiliary variables and properties. Moreover, some behaviors or English specification can be expressed using SEREs more easily, and in a more compact way.

Example 1. FL or SERE?

Assume that we want to write a property that states: "signal *a* is asserted on every even cycle"[4]. The assertion may be written as `assertion_FL`:

```
assertion_FL:
  assert (a and always (a -> next[2](a)));
```

This assertion expresses that *a* should be asserted in cycle ♯0, and then, whenever *a* is 1, it should be asserted 2 cycles later. Now, consider the trace shown in Fig. 6.1. *a* is 1 in cycle ♯5. Based on `assertion_FL`, *a* should be 1 in cycle ♯7. Since it is 0 in cycle ♯7, `assertion_FL` fails. However, in the English property, we did not say anything about *a* in odd cycles. Therefore, `assertion_FL` is checking a condition that is not intended to be checked. Briefly, the trace shown in Fig. 6.1 is correct based on the English property, however, it is not correct based on `assertion_FL`.

[1] Sequential Extended Regular Expression

[2] System Verilog Assertion

[3] Foundation Language

[4] The example is taken from [EF06]

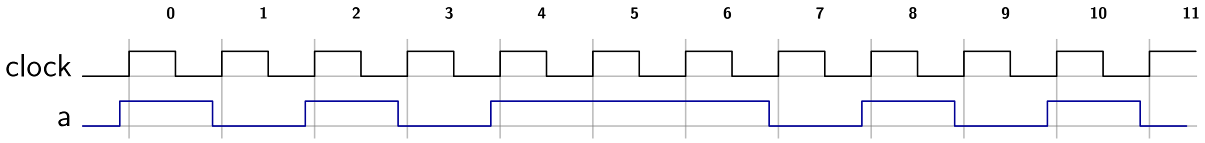


FIGURE 6.1: An example timeline for "*a* is asserted on every even cycle"

We can fix this problem by defining an auxiliary variable and writing a code in the modeling layer (see Fig. 6.2). Signal *even* becomes 1 in every even cycle, and signal *a* should be asserted whenever *even* is 1.

```
vunit check_even{
  signal even:std_logic:='1';
  process (clk'event and clk = '1')
  begin
    even <= not even;
  end process;

  assert always (even -> a);
}
```

FIGURE 6.2: Using modeling layer to check "*a* is asserted on every even cycle"

However, this property can be easily written using SEREs:

```
assertion_SERE:
  assert {[*1]; {[*2]}[*] } |-> {a};
```

Additionally, rewriting a SERE property as FL may cause that it does not fit in PSL_{simple} anymore.

Example 2. Rewriting SEREs as FLs.

Consider `HDLC_200` from Fig. 6.3[5]. It states the behavior of the controller in the case of having an abort command. In this situation, 7 consecutive 1's should be put on the output, followed with one or more flags, which is "01111110". It can be easily written as a SERE. Conversely, property `HDLC_200` reflects this behavior easily: once an abort command is received, i.e. *TxSendAbort* changes from 0 to 1, while the transmitter is enabled the output *TxDout* becomes 1 for at least 7 consecutive cycles, and finally it is followed by one or more flags.

As another example consider `HDLC_300`. It can be easily converted to an FL property; however, it is not in PSL_{simple}(see Fig. 6.4), since the left-hand side of the implication operator is not Boolean.

Since all the SEREs cannot be expressed using FLs, in the rest of this chapter we propose a method for synthesizing SEREs. Before going to the synthesis method, here we bring some examples that reveal the difficulties and challenges that we should consider. These difficulties impose some limitations to the subset of SEREs that we are able to synthesize.

[5] The properties describe High-level Data Link Controller (HDLC), and are taken from [PPSQ13]

```
HDLC_200 :
  always ({not TxSendAbort and TxEnable; TxSendAbort and TxEnable}
          |-> {TxDout[*7]; TxDout[*]; {not TxDout; TxDout[*6]; not TxDout
              }[+]});

HDLC_300 :
   always({not BuffEmpty and TxEnable; (BufEmpty and not TxDataWr and
       TxEnable); (not TxDataWr and TxEnable)[*7]}
               |->  next(TxUnderRun) );
```

FIGURE 6.3: Sample SERE properties of High-level Data Link Controller

```
HDLC_300_FL:
  always(not BuffEmpty and TxEnable and next((BuffEmpty and not not
     TxDataWr and TxEnable) and next_a[2 to 8](not TxDataWr and TxEnable))
     ->  next[9](TxUnderRun) );
```

FIGURE 6.4: FL version of `HDLC_300`

It should be mentioned that when we constrain a signal, we assign a specific value to that signal. Generating a signal means constraining the signal value. When a signal is not constrained, its value is *don't care*. However, in the following examples we consider value 0 for unconstrained signals.

Example 3.

Consider property `P1`, where a, b, and c are Boolean:

```
 P1: always {a} |=> {b[*]; c}
```

Assume that a is observed and we generate b and c. Then, the question is: "when should we stop constraining b to 1, and start constraining c to 1"? If we want to generate c, the property is not deterministic since c can be constrained to 1 in any cycle after $a = 1$. If we observe b and generate c, when should c be constrained to 1? It may depend on other properties.

If we observe c and generate b, b is no longer constrained as soon as c becomes 1.

Example 4.

In this example, we assume value 0 for unnamed signals. Consider property `P2`, where a, b, and c are Boolean:

```
 P2: always {a} |=> {b[+]; c; not c}
```

Assume that a is observed, and b and c are generated. Some questions may arise: "when should we stop constraining b to 1?", "If a is 1 in two consecutive cycles, how should we constrain b and c to avoid an inconsistency?"

Figure 6.5 shows two possible traces. Sequence 1 starts at cycle ♯0. b is constrained to 1 for three cycles. In cycle ♯4, c is constrained to 1, and it is constrained to 0 in cycle ♯5, and sequence 1 completes in this cycle.

Assume a is 1 in two consecutive cycles ♯6 and ♯7. Sequence 2 starts when a is asserted in cycle ♯6. Then, b is constrained to 1 in cycles ♯7 and ♯8, c is constrained to 1 at cycle ♯9 and it is constrained to 0 at cycle ♯10. Sequence 2 completes in cycle ♯10. Sequence 3

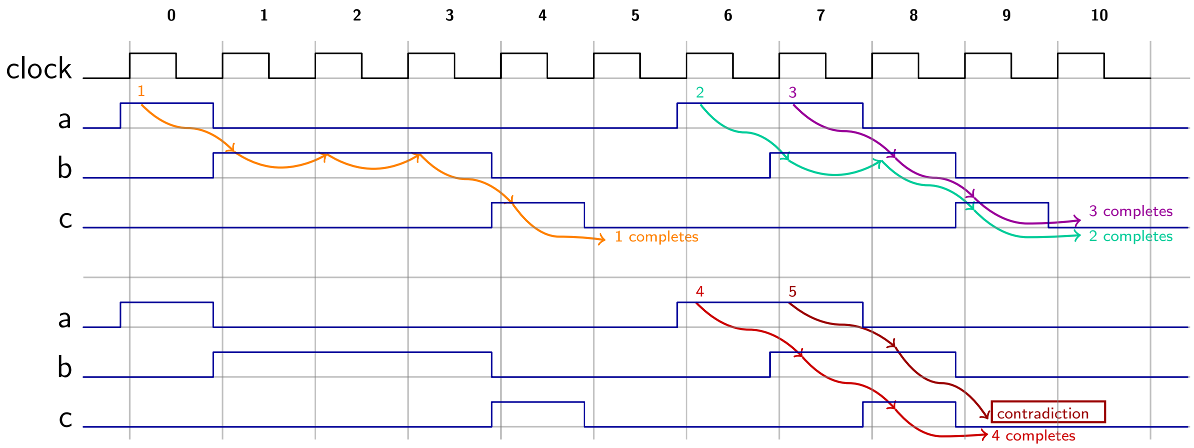


FIGURE 6.5: Timing diagram of P2, where b and c are generated (Example 4)

starts when a is asserted in cycle $\sharp 7$. Then, b is constrained to 1 in cycle $\sharp 8$, c is constrained to 1 at cycle $\sharp 9$ and it is constrained to 0 at cycle $\sharp 10$. Sequence 3 completes in cycle $\sharp 10$.

Now, consider the bottom trace of Fig. 6.5. Sequence 4 starts in cycle $\sharp 6$. Then, b is constrained to 1 in cycle $\sharp 7$, c is constrained to 1 at cycle $\sharp 8$ and it is constrained to 0 at cycle $\sharp 9$; hence, sequence 4 completes in this cycle. Sequence 5 starts in cycle $\sharp 7$. Then, b is constrained to 1 in cycle $\sharp 8$. If we stop constraining b to 1 at this cycle and want to constrain c to 1 at cycle $\sharp 9$, there is a contradiction with the value that is being generated at the same cycle for c by sequence 4. This example shows that when we should stop generating b, for each run of the property is an issue.

Assume that b is generated. We observe c, and constrain b to 1 while c is 0. Whenever c is asserted, in the following cycle, we constrain c to 0. In brief, we generate b, we observe c, and we generate $\texttt{not}\, c$. It implies that other properties should constrain c. Our prototype tool, SyntHorus2, can identify if the signal is constrained by any property. However, constraining c by other properties may cause inconsistency. As it will be explained in Chapter 9, SyntHorus2 generates some complementary properties to identify if there are any inconsistencies. Figure 6.6 shows a possible trace. Sequence 1 starts in cycle $\sharp 0$, and completes in cycle $\sharp 3$. Sequence 2 starts in cycle $\sharp 1$. This sequence cannot be completed, since c is 1 in cycle $\sharp 2$, and hence, b cannot be constrained to 1.

Sequence 3 starts in cycle $\sharp 5$. c is 0 in cycles $\sharp 6$, $\sharp 7$, and $\sharp 8$. Therefore, b is constrained to 1 in these cycles. In cycle $\sharp 9$, c is asserted. Consequently, we should stop constraining b to 1 at cycle $\sharp 9$, and we should constrain c to 0 in cycle $\sharp 10$. However, c is constrained to 1 by other properties, and there is an inconsistency. If we keep constraining b to 1 in cycle $\sharp 9$, there would be no contradiction, and the property holds. So, we may conclude that we should monitor $\{c; \texttt{not}\, c\}$ instead of observing c and generating $\texttt{not}\, c$. In the following, we explain why it is not a good solution.

Now, assume that we generate b, and observe $\{c; \texttt{not}\, c\}$. We should stop constraining b to 1 as soon as $\{c; \texttt{not}\, c\}$ occurs. However, it is not possible. The limitation is shown in Fig. 6.7. In this figure, signal *ended* indicates the completeness of $\{c; \texttt{not}\, c\}$. Signal b is constrained to 1 as far as *ended* is 0. The sequence starts in cycle $\sharp 0$. In cycles $\sharp 1$, $\sharp 2$, $\sharp 3$, and $\sharp 4$ the *ended* signal is 0; i.e. $\{c; \texttt{not}\, c\}$ has not been observed, so b is constrained to 1. In cycle $\sharp 5$, $\{c; \texttt{not}\, c\}$ completes, and we stop constraining b to 1. However, it is not correct and P2 does not hold. Because the last occurrence of b should be followed by $\{c; \texttt{not}\, c\}$, which is not the signals' behavior here (remember our assumption that the

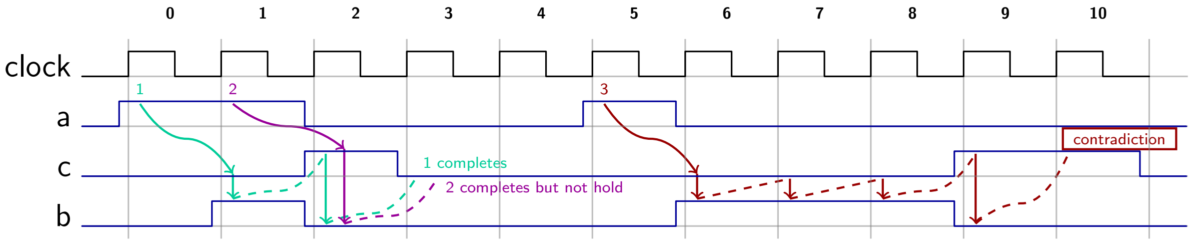


FIGURE 6.6: Timing diagram of P2, where b is generated, c is observed, and $\texttt{not}\, c$ is generated (Example 4)

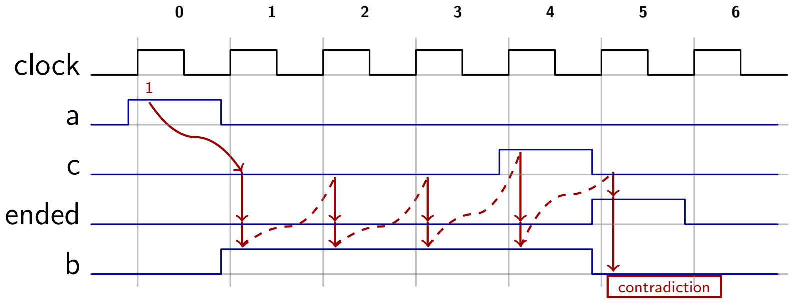


FIGURE 6.7: Timing diagram of P2, where b is generated, and $\{c, \texttt{not}\, c\}$ is observed (Example 4)

default value of the signals are 0 when they are not constrained). The problem is that we cannot have the length of the sequence that should be observed, and we cannot move backward and correct the value of the generated signal.

To solve this problem, we have to put a limitation: each unbounded repetition should be followed by a Boolean signal that is observed and is the stopping condition for the constraint.

Example 5.

Consider property P3, where a, b, and c are Boolean:

```
P3: always {a;b}[*3] |=> {c};
```

In this property, we observe the left-hand side of the implication, and constrain c. Assume that there is a signal *start* that starts the evaluation of P3, and a signal *ended* that shows when $\{\{a;b\}[*3]\}$ completes in order to constrain c. To compute when *ended* becomes 1, we should count the number of times that $\{a;b\}$ occurs, therefore we need a counter. If *start* is 1 in 2 consecutive cycles, we need 2 counters. Generally, re-entering the property is a challenge. We need several instances of the input and output tokens. We can solve this problem by rewriting $\{\{a;b\}[*3]\}$ as three concatenations of $\{a;b\}$.

Example 6.

Consider property P4, where a, b, and c are Boolean:

```
P4: always {a} |=> {b} | {c};
```

First, assume that a is observed, and we want to generate the right-hand side of the property. If both b and c are output signals, we cannot constrain them in this property.

The solution of this problem is explained in Chapter 9. Now, suppose that b is an input. Therefore, we observe b, and constrain c whenever b is 0, i.e. the value of c depends on $\neg b$.

Example 7.

Consider property P5, where a, b, and c are Boolean:

```
P5: always {a} |=> {b; not b} | {c};
```

First, assume that a, b and c are observed. We start observing $\{b; not\ b\}$ and $\{c\}$ at the same time. P5 holds and completes when either $\{b; not\ b\}$ or $\{c\}$ completes.

Now assume that a is observed, and we want to generate the right-hand side of the implication. If both b and c are output signals, we cannot constrain them in this property. In addition, the left operand, $\{b; not\ b\}$, is a sequence. In this case we cannot synthesize the property.

If c is the input, and b is the output, we observe c and if it is 0, we generate $\{b; not\ b\}$.

However, if b is input and c is the output, we are not able to constrain c, since we should wait for the completeness of $\{b; not\ b\}$, and then constrain c based on that. However, this is not possible, since two sequences $\{c\}$ and $\{b; not\ b\}$ should start at the same time.

Based on the last two examples, we put a limitation that we can only synthesize the hardware of disjunction if: 1) both operands are observed, or 2) both operands are generated and are Boolean, or 3) one operand is generated, and the other one is observed and is Boolean.

Example 8.

Consider property P6, where a, b, c, and d are Boolean:

```
P6: always {a;b;c} & {{b;c}[*2]} |=> {d};
```

In this property, we should observe a, b, and c. The sequences $\{a; b; c\}$ and $\{\{b; c\}[*2]\}$ should start at the same time. There may be also an re-entering condition, i.e. the start signal remains 1 for more than a cycle (it is permanently 1 after `always`). In this case, we need to use polychrome tokens [MAGB07], or an automata-based method [BZ08c] for observing the sequence. This problem does not exist in the generation mode, since we start generating $\{a; b; c\}$ and $\{\{b; c\}[*2]\}$ at the same time when $start = 1$.

These difficulties show that some restrictions may apply in order to produce deterministic hardware. It may also be necessary to perform some sanity checks using a model checker to make sure that a set of properties is consistent to lead to meaningful hardware. SyntHorus2 generates complementary properties for consistency checking.

6.3 Formalization of the annotation

The purpose of this section is to formally define the notion of dependency between the operands of a SERE operator, in order to have a synthezisable VHDL description of each operator. To this aim, two dependency relations are introduced for each SERE operator: a dependency relation for expressing when a sequence holds, and a dependency relation for expressing when the sequence completes.

In FLs (see Chapter 5) we do not need to determine the end of a temporal operator trace. Because, in PSL$_{simple}$ a FL property can just be dependent on a Boolean, not on another FL. However, in SEREs, a sequence can depend on another sequence.

Example 9.

Assume $\varphi 1 = \{b1; b2\}$, where $b1$ and $b2$ are Boolean. We assume that $b1$ is 1 at cycle $\sharp 0$, therefore, $\varphi 1$ starts at this cycle. In the following cycle, if $b2$ is 1, $\varphi 1$ completes. Now, assume that $\varphi 2 = \{\varphi 1; b3\}$. $\varphi 2$ starts when $\varphi 1$ starts. To complete $\varphi 2$, $\varphi 1$ should complete, and in the following cycle $b3$ should occur. Therefore, we need to identify when $\varphi 1$ completes.

In this section, first some terminology and notations are introduced. Then, the dependency relations for each operator are defined.

6.3.1 Dependency relation: definition and notations

Here, we use the SERE semantic definitions in Appendix B of the IEEE Standard [FG05], and also the same notations and definitions as Chapter 5. We only introduce one notation here, necessary for SEREs.

- $w \models\!\!\equiv$ `property`: is the extended semantics by structural induction over SERE properties to words, and means "`property`" holds tightly on word w (w models tightly `property`).

For defining the dependency relation, we use Definition 1 given in Chapter 5.

Briefly, the dependency relation $\lfloor A \lhd B \rfloor_w$ means that on a trace w, the value of A depends on the value of B. Semantically, for a trace w, if B is satisfied on w, A must be satisfied on w. Let φ be a sequence that is composed of sub-sequences A and B using a SERE operator Ω, and w be a trace. Then, A depends on B in φ if
$\forall w, \lfloor \varphi \lhd true \rfloor_w \Leftrightarrow \lfloor A \lhd B \rfloor_w$.

Definition 1. *Let φ be a SERE, and $Ended_\varphi$ be a Boolean that becomes 1 when φ completes. w is a trace, and ℓ is the j^{th} letter of w ($\ell = w^j$), so that $\ell \vdash Ended_\varphi$. Then, for each sequence φ we can say:*

$$\forall w, \lfloor \varphi \lhd true \rfloor_{w^{i \ldots j}} \Leftrightarrow \lfloor Ended_\varphi \lhd true \rfloor_{w^j}$$

We refer to the dependency relation $\lfloor Ended_\varphi \lhd true \rfloor_{w^j}$ as $E_\varphi Relation$.

Here, the rules that were discussed for FLs in Chapter 5, are applicable (see properties 1 to 5 of Chapter 5).

6.3.2 Dependency relation between operands of SERE operators

In this section, two dependency relations are given for each SERE operator. The first dependency relation is referred to as "$\varphi Relation$", and expresses the dependency between sub-sequences of φ in order that φ holds. This dependency between operands is stated using the general $\lfloor \lhd \rfloor_w$ relation. The second dependency relation is referred to as "$E_\varphi Relation$" and expresses the dependency between the sub-sequences of φ at the cycle that φ completes. This dependency is stated using $\lfloor \lhd \rfloor_{w^i}$ (See Definition 1).

In the following, φ represents a sequence, A and B are the left and right operands (sub-sequences). A and B are assumed to be not empty (they do not satisfy the empty sequence). Ended_φ, Ended_A and Ended_B are the Boolean variables that specify the end of φ, A, and B sequences respectively.

6.3.2.1 Base cases

Let *exp* be a Boolean expression, and A be a SERE:

$$\begin{aligned}&\lfloor \mathit{exp} \lhd \mathit{true} \rfloor_w \Leftrightarrow \lfloor \mathit{exp} \lhd \mathit{true} \rfloor_{w^0} \wedge \lfloor \mathit{Ended}_{\mathit{exp}} \lhd \mathit{true} \rfloor_{w^0}\\ &\lfloor \{A\} \lhd \mathit{true} \rfloor_w \Leftrightarrow \lfloor A \lhd \mathit{true} \rfloor_w\end{aligned}$$

These relations are the direct rewriting of the semantic definitions.

6.3.2.2 Concatenation

Dependency Rule 1. *φRelation for concatenation*
Let $\varphi = A;B$, then:
$\lfloor \varphi \lhd \mathit{true} \rfloor_w$ *iff* $\exists i < |w|, \lfloor \mathit{Ended}_A \lhd \mathit{true} \rfloor_{w^i} \wedge \lfloor B \lhd \mathit{true} \rfloor_{w^{i+1\ldots}}$

Proof. We replace *concatenation* by its semantic definition.

$$\begin{aligned}&\forall w, w \models \mathit{true} \Rightarrow w \models A;B\\ &\Leftrightarrow \exists w_1, w_2, w = w_1w_2, w_1 \models A \quad \wedge \quad w_2 \models B\\ &\Leftrightarrow \exists i < |w|, w_1 = w^0 \ldots w^i, w_2 = w^{i+1} \ldots w^{|w|-1}, w_1 \models A \quad \wedge \quad w_2 \models B\\ &\Leftrightarrow \exists i < |w|, w_1 = w^0 \ldots w^i, w_2 = w^{i+1} \ldots w^{|w|-1}, \lfloor \mathit{Ended}_A \lhd \mathit{true} \rfloor_{w^i} \quad \wedge \quad \lfloor B \lhd \mathit{true} \rfloor_{w_2}\end{aligned}$$

The proof is immediate by replacing w_1 and w_2. □

Dependency Rule 2. *E_φRelation for concatenation*
Let $\varphi = A;B$, then:
$\exists j < |w|, \lfloor \mathit{Ended}_\varphi \lhd \mathit{true} \rfloor_{w^j}$ *iff* $\exists k < j, \lfloor \mathit{Ended}_A \lhd \mathit{true} \rfloor_{w^k} \wedge \lfloor \mathit{Ended}_B \lhd \mathit{true} \rfloor_{w^j}$

Proof. The proof is immediate, by considering the *φRelation*:

$$\exists i < |w|, \lfloor \mathit{Ended}_A \lhd \mathit{true} \rfloor_{w^i} \wedge \lfloor B \lhd \mathit{true} \rfloor_{w^{i+1\ldots}}$$

We expect that the evaluation of B completes at some point in the trace, which means:

$$\begin{aligned}&\Leftrightarrow \exists i < j < |w|, \lfloor \mathit{Ended}_B \lhd \mathit{true} \rfloor_{w^j}\\ &\Leftrightarrow \exists j < |w|, \lfloor \mathit{Ended}_\varphi \lhd \mathit{true} \rfloor_{w^j}\end{aligned}$$

□

In a special case where B is a Boolean, then $j = k + 1$.

6.3.2.3 Fusion

Dependency Rule 3. *φRelation for fusion*
Let $\varphi = A : B$, then:
$\lfloor \varphi \lhd true \rfloor_w$ *iff* $\exists i < |w|, \lfloor Ended_A \lhd true \rfloor_{w^i} \wedge \lfloor B \lhd true \rfloor_{w^{i\ldots}}$

Proof. In the second line of the rule, we replace *fusion* by its semantic definition.

$$\forall w, w \models true \Rightarrow w \models A : B$$
$$\Leftrightarrow \exists w_1, w_2, l, w = w_1 l w_2, w_1 l \models A \wedge l w_2 \models B$$
$$\Leftrightarrow \exists w_1, w_2, l, w = w_1 l w_2, \lfloor Ended_A \lhd true \rfloor_l \wedge \lfloor B \lhd true \rfloor_{l w_2}$$

Let $i = |w_1|$, we can write:

$$\exists i < |w|, w_1 l = w^{0\ldots i-1} l = w^{0\ldots i} \wedge l w_2 = l w^{i+1\ldots} = w^{i\ldots}$$

So, by replacing l and w_2 in the proof, we have:

$$\Leftrightarrow \exists i < |w|, \lfloor Ended_A \lhd true \rfloor_{w^i} \wedge \lfloor B \lhd true \rfloor_{w^{i\ldots}}$$

□

Dependency Rule 4. *E_φRelation for fusion*
Let $\varphi = A : B$, then
$\exists j < |w|, \lfloor Ended_\varphi \lhd true \rfloor_{w^j}$ *iff* $\exists k \leq j, \lfloor Ended_A \lhd true \rfloor_{w^k} \wedge \lfloor Ended_B \lhd true \rfloor_{w^j}$

Proof. Considering the *φRelation*, we have:

$$\exists i < |w|, \lfloor Ended_A \lhd true \rfloor_{w^i} \wedge \lfloor B \lhd true \rfloor_{w^{i\ldots}}$$

B starts to be evaluated in the last cycle of A, and its evaluation should complete, which means:

$$\Leftrightarrow \exists j \geq k, \lfloor Ended_B \lhd true \rfloor_{w^j}$$

□

In the case where B is a Boolean, $j = k$.

6.3.2.4 Length-matching conjunction

Dependency Rule 5. *φRelation for length-matching conjunction*
Let $\varphi = A$ && B, then
$\lfloor \varphi \lhd true \rfloor_w$ *iff* $\lfloor A \lhd true \rfloor_w \wedge \lfloor B \lhd true \rfloor_w$

Proof. The proof is straightforward, just by replacing *length-matching conjunction* with its semantic definition.

$$\forall w, w \models true \Rightarrow w \models A \;\&\&\; B$$
$$\Leftrightarrow \forall w, w \models A \wedge w \models B$$
$$\Leftrightarrow \forall w, \lfloor A \lhd true \rfloor_w \wedge \lfloor B \lhd true \rfloor_w$$

□

Dependency Rule 6. *E_φRelation for length-matching conjunction*
Let $\varphi = A$ && B, then
$\exists j < |w|, \lfloor Ended_\varphi \lhd true \rfloor_{w^j}$ *iff* $\lfloor Ended_A \wedge Ended_B \lhd true \rfloor_{w^j}$

Proof. Considering the *φRelation*, we have:

$$\Leftrightarrow \forall w, \lfloor A \lhd true \rfloor_w \wedge \lfloor B \lhd true \rfloor_w$$

Therefore, we expect at some points in the trace, the evaluation of A and B completes, which means:

$$\Leftrightarrow \exists j < |w|, \lfloor Ended_A \lhd true \rfloor_{w^j} \wedge \exists k < |w|, \lfloor Ended_B \lhd true \rfloor_{w^k}$$

In *length-matching conjunction* the sequences start at the same time and should complete at the same time. Therefore, $k = j$. Using Property 3 from Chapter 5 we have:

$$\Leftrightarrow \exists j < |w|, \lfloor Ended_A \wedge Ended_B \lhd true \rfloor_{w^j}$$

□

6.3.2.5 Non length-matching conjunction

Dependency Rule 7. *φRelation for non length-matching conjunction*
Let $\varphi = A$ & B, then
$\lfloor \varphi \lhd true \rfloor_w$ *iff* $\exists i < |w|, (\lfloor A \lhd true \rfloor_w \wedge \lfloor B \lhd true \rfloor_{w^{0...i}}) \vee (\lfloor A \lhd true \rfloor_{w^{0...i}} \wedge \lfloor B \lhd true \rfloor_w)$

Proof. The proof is straightforward, just by replacing *non length-matching conjunction* with its semantic definition.

$$\begin{aligned}
&\forall w, w \models true \Rightarrow w \models A \ \&\ B \\
&\Leftrightarrow \forall w, w \models true \Rightarrow w \models \{A \ \&\&\ \{B; true[*]\}\} \mid \{\{A; true[*]\} \ \&\&\ B\} \\
&\Leftrightarrow \forall w, (\lfloor A \lhd true \rfloor_w \wedge \lfloor \{B; true[*]\} \lhd true \rfloor_w) \\
&\quad \vee (\lfloor B \lhd true \rfloor_w \wedge \lfloor \{A; true[*]\} \lhd true \rfloor_w) \\
&\Leftrightarrow \forall w, (\lfloor A \lhd true \rfloor_w \wedge \exists i < |w|, \lfloor B \lhd true \rfloor_{w^{0...i}}) \\
&\quad \vee (\lfloor B \lhd true \rfloor_w \wedge \exists i < |w|, \lfloor A \lhd true \rfloor_{w^{0...i}}) \\
&\Leftrightarrow \exists i < |w|, (\lfloor A \lhd true \rfloor_w \wedge \lfloor B \lhd true \rfloor_{w^{0...i}}) \vee (\lfloor B \lhd true \rfloor_w \wedge, \lfloor A \lhd true \rfloor_{w^{0...i}})
\end{aligned}$$

□

Dependency Rule 8. *E_φRelation for non length-matching conjunction*
Let $\varphi = A$ & B, then
$\exists j < |w|, \lfloor Ended_\varphi \lhd true \rfloor_{w^j}$ *iff*
$(\lfloor Ended_A \lhd true \rfloor_{w^j} \wedge \exists k < j, \lfloor Ended_B \lhd true \rfloor_{w^k}) \vee$
$(\lfloor Ended_B \lhd true \rfloor_{w^j} \wedge \exists k < j, \lfloor Ended_A \lhd true \rfloor_{w^k})$

Proof. Considering the *φRelation*, we have:

$$\exists i < |w|, (\lfloor A \lhd true \rfloor_w \wedge \lfloor B \lhd true \rfloor_{w^{0...i}}) \vee (\lfloor A \lhd true \rfloor_{w^{0...i}} \wedge \lfloor B \lhd true \rfloor_w)$$

Therefore, we expect at some point in the trace, the evaluation of A and B completes. Since the sequences may not have the same length, we consider two cases: 1) A is longer, 2) B is longer, then:

$$\begin{aligned}&\Leftrightarrow 1)\ \exists j < |w|, \lfloor Ended_A \lhd true \rfloor_{w^j} \wedge \exists k < j, \lfloor Ended_B \lhd true \rfloor_{w^k}\\ &\quad \vee\\ &\quad 2)\ \exists j < |w|, \lfloor Ended_B \lhd true \rfloor_{w^j} \wedge \exists k < j, \lfloor Ended_A \lhd true \rfloor_{w^k}\end{aligned}$$

□

6.3.2.6 Disjunction

Dependency Rule 9. *φRelation for disjunction*
Let $\varphi = A \mid B$, then
$\lfloor \varphi \lhd true \rfloor_w$ *iff* $\lfloor A \lhd \neg B \rfloor_w \vee \lfloor B \lhd \neg A \rfloor_w$

Proof. The proof is straightforward, just by replacing *star* with its semantic definition.

$$\begin{aligned}&\forall w, w \models true \Rightarrow w \models A \mid B\\ &\Leftrightarrow \forall w, w \models A \vee w \models B\\ &\Leftrightarrow \forall w, \lfloor A \lhd true \rfloor_w \vee \lfloor B \lhd true \rfloor_w\end{aligned}$$

By rewriting using Property 4 from Chapter 5 ($\lfloor (A\ or\ B) \lhd C \rfloor_w \Leftrightarrow \lfloor A \lhd (C \wedge \neg B) \rfloor_w$) we have:

$$\lfloor A \lhd \neg B \rfloor_w \vee \lfloor B \lhd \neg A \rfloor_w$$

□

Dependency Rule 10. *E_φRelation for disjunction*
Let $\varphi = A \mid B$, then
$\exists j < |w|, \lfloor Ended_\varphi \lhd true \rfloor_{w^j}$ *iff* $\lfloor Ended_A \lhd true \rfloor_{w^j} \vee \lfloor Ended_B \lhd true \rfloor_{w^j}$

Proof. Considering the *φRelation*, we have:

$$\Leftrightarrow \forall w, \lfloor A \lhd true \rfloor_w \vee \lfloor B \lhd true \rfloor_w$$

Therefore, we expect at some point in the trace, the evaluation of A and B completes, which means:

$$\Leftrightarrow \exists i < |w|, \lfloor Ended_A \lhd true \rfloor_{w^i} \wedge \exists k < |w|, \lfloor Ended_B \lhd true \rfloor_{w^k}$$

Let $j = min(i, k)$. Then:

$$\Leftrightarrow \exists j < |w|, \lfloor Ended_A \lhd true \rfloor_{w^j} \vee \lfloor Ended_B \lhd true \rfloor_{w^j}$$

□

6.3.2.7 Kleene closure

Dependency Rule 11. *φRelation for star*
Let $\varphi = A[*0]$, then
$\lfloor \varphi \lhd \mathit{true} \rfloor_w$ *iff* $|w| = 0$
Let $\varphi = A[*]$, then
$\lfloor \varphi \lhd \mathit{true} \rfloor_w$ *iff* $|w| = 0 \vee \exists i < |w|, \lfloor \mathit{Ended}_A \lhd \mathit{true} \rfloor_{w^i} \wedge \lfloor \varphi \lhd \mathit{true} \rfloor_{w^{i+1\ldots}}$

Proof. The *star* operator is replaced with its semantic definition.

$$\begin{aligned}&\forall w, w \models\!\equiv \mathit{true} \Rightarrow w \models\!\equiv A[*]\\ &\Leftrightarrow w \models\!\equiv [*0] \vee \exists w_1, w_2, |w_1| > 0, w = w_1 w_2, w_1 \models\!\equiv A \wedge w_2 \models\!\equiv A[*]\end{aligned}$$

Let $w_1 = w^{0\ldots i}$. We can write:

$$\Leftrightarrow |w| = 0 \vee \exists i < |w|, \lfloor \mathit{Ended}_A \lhd \mathit{true} \rfloor_{w^i} \wedge \lfloor \varphi \lhd \mathit{true} \rfloor_{w^{i+1\ldots}}$$

□

6.3.2.8 Plus

Dependency Rule 12. *φRelation for plus*
Let $\varphi = A[+]$, then
$\lfloor \varphi \lhd \mathit{true} \rfloor_w$ *iff* $\exists i < |w|, \lfloor \mathit{Ended}_A \lhd \mathit{true} \rfloor_{w^i} \wedge \lfloor A[*] \lhd \mathit{true} \rfloor_{w^{i+1\ldots}}$

Proof. The *plus* operator is replaced with its semantic definition.

$$\begin{aligned}&\forall w, w \models\!\equiv \mathit{true} \Rightarrow w \models\!\equiv A[+]\\ &\Leftrightarrow \exists w_1, w_2, w_1 \neq \epsilon, w = w_1 w_2, w_1 \models\!\equiv A \wedge w_2 \models\!\equiv A[*]\end{aligned}$$

□

Dependency Rule 13. *E_φRelation for plus*
Let $\varphi = A[+]$, then
$\exists j < |w|, \lfloor \mathit{Ended}_\varphi \lhd \mathit{true} \rfloor_{w^j}$ *iff* $\lfloor \mathit{Ended}_A \lhd \mathit{true} \rfloor_{w^j}$

We assume that each time A completes, Ended_φ becomes 1, not only the last time.

Proof. Considering the φ*Relation*, we have:

$$\exists i < |w|, \lfloor \mathit{Ended}_A \lhd \mathit{true} \rfloor_{w^i} \wedge \lfloor A[*] \lhd \mathit{true} \rfloor_{w^{i+1\ldots}}$$

The proof is immediate; A occurs at least once, and then its evaluation completes every $j = |A|$ cycles:

$$\Leftrightarrow \exists j, 0 \leq j < |w|, \lfloor \mathit{Ended}_A \lhd \mathit{true} \rfloor_{w^j}$$

□

6.4 Dependency relation synthesis

In order to construct a library of primitive reactants for SERE operators, we give a hardware interpretation of the two dependency relations $\varphi Relation$ ($\lfloor \varphi \lhd true \rfloor_w$), and $E_\varphi Relation$ ($\lfloor Ended_\varphi \lhd true \rfloor_{w^i}$).

6.4.1 Principles of the primitive reactant construction

The primitive reactants have a general interface: they take *clock* and *reset* as the synchronization signals. Each primitive reactant has a *start* signal for its activation. The corresponding circuit of each SERE operator is the interconnection of the circuits of the $\varphi Relation$ and $E_\varphi Relation$ dependency relations. This interconnection is shown in Fig. 6.8.

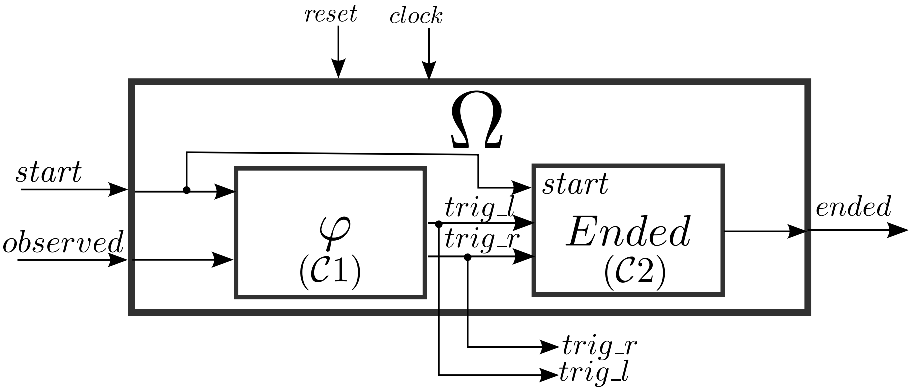


FIGURE 6.8: Generic interface of a SERE operator

The operands of a SERE operator, Ω, are *observed* or *constrained* by the primitive reactant of Ω during its activity. Thus, the output of a primitive reactant is *not* the value of a signal, but the *trigger* that will *start* the primitive hardware component in charge of the signal value generation or observation. In particular, for a Boolean signal S, triggering $S = 1$ and $S = 0$ is done by two distinct signals $Trig_S$ and $Trig_{\neg S}$. In the following, without loss of generality, we shall only consider the positive case $S = 1$. $Trig_S$ is set to 1 by the reactant at all cycles when the dependency $\lfloor S \lhd true \rfloor_w$ holds.

The left circuit in Fig. 6.8, $\mathcal{C}1$ implements $\varphi Relation$ and based on this dependency it generates two trigger signals: *trig_l* and *trig_r*. These signals start the primitive hardware components that are in charge of generating or observing left or right sub-sequences. For example, if S is the left sub-sequence, then $trig\_l = Trig_S$ means that *trig_l* constrains S to 1.

Then, the right circuit $\mathcal{C}2$, which implements $E_\varphi Relation$, generates a signal that indicates if φ completes. We call the output of this module *ended*. It is equivalent to the $Ended_\varphi$ Boolean that has already been introduced (see Definition 1).

Here, we define $\mathcal{C}1$ and $\mathcal{C}2$ that implement $\varphi Relation$ and $E_\varphi Relation$ respectively.

Let $\mathbf{P}$ be the set of the signal names in φ. The signals may be observed or be constrained. Therefore, we split $\mathbf{P}$ in two sets $\mathbf{P}_{in}$ and $\mathbf{P}_{out}$ ($\mathbf{P} = \mathbf{P}_{in} \cup \mathbf{P}_{out}$), that contain observed and constrained signals in the sub-sequences respectively.

Circuit $\mathcal{C}$ is the circuit that implements a primitive reactant, and is the interconnection of $\mathcal{C}1$ and $\mathcal{C}2$.

The sets $\mathbf{P}_{\mathcal{C}1}$ and $\mathbf{P}_{\mathcal{C}2}$ of atomic propositions for the circuits $\mathcal{C}1$ and $\mathcal{C}2$ are given by $\mathbf{P}_{\mathcal{C}1} = \mathbf{P}_{in} \cup \{reset, start\} \cup \{trig\_l, trig\_r\}$, and $\mathbf{P}_{\mathcal{C}2} = \{start\} \cup \{trig\_l, trig\_r\} \cup \{ended\}$. Then, $\mathbf{P}_{\mathcal{C}} = \mathbf{P}_{\mathcal{C}1} \cup \mathbf{P}_{\mathcal{C}2}$.

We denote $\Sigma_{\mathcal{C}}$ the alphabet built on $\mathbf{P}_{\mathcal{C}}$. In a similar way , we denote $\Sigma_{\mathcal{C}1}$ and $\Sigma_{\mathcal{C}2}$ the alphabets built on $\mathbf{P}_{\mathcal{C}1}$ and $\mathbf{P}_{\mathcal{C}2}$.

Let $w_{\mathcal{C}1}$ be a trace built on $\Sigma_{\mathcal{C}1}$ and w_{φ} (or $w1$) a trace built on Σ. Similarly, let $w_{\mathcal{C}2}$ be a trace built on $\Sigma_{\mathcal{C}2}$ and $w_{Ended_{\varphi}}$ (or $w2$) a trace built on Σ.

Definition 2. *$w_{\mathcal{C}1}$ is equivalent to $w1$ on $\mathbf{P}_{in}$, denoted $w_{\mathcal{C}1} \equiv_{|\mathbf{P}_{in}} w1$ iff $\exists j$ such that*

- $w_{\mathcal{C}1}^{j} \vdash start$ *and* $\exists k < j$ *such that* $w_{\mathcal{C}1}^{k-1} \vdash reset$ *and* $\forall l, k < l \leq j$, $w_{\mathcal{C}1}^{l} \vdash \neg reset$ *and* $w_{\mathcal{C}1}^{l} \vdash \neg start$
- $\forall i, {w_{\mathcal{C}1}^{j+i}}_{|\mathbf{P}_{in}} = {w^{i}}_{|\mathbf{P}_{in}}$

This definition is very similar to Definition 3 of Chapter 5. Briefly, it means that there is a time point in the trace ($w_{\mathcal{C}1}^{j}$) where *start* becomes 1. Before this point, $\mathcal{C}1$ has been reset and it is not active. After this point, $w_{\mathcal{C}1}$ and $w1$ should have identical values for the atomic propositions in $\mathbf{P}_{in}$. For the sake of simplicity, we assume that $j = 0$.

Definition 3. *Let $\varphi = A \Omega B$, where A and B are the left and right sub-sequences:*

- *$w_{\mathcal{C}1}$ is equivalent to $w1$, denoted $w_{\mathcal{C}1} \equiv w1$, iff*
  - $w_{\mathcal{C}1} \equiv_{|\mathbf{P}_{in}} w$
  - $\forall i, w_{\mathcal{C}1}^{i} \vdash trig\_l \implies w1^{i} \vdash A$
  - $\forall i, w_{\mathcal{C}1}^{i} \vdash trig\_r \implies w1^{i} \vdash B$
- *$w_{\mathcal{C}2}$ is equivalent to $w2$, denoted $w_{\mathcal{C}2} \equiv w2$, iff*
  - $\exists m \geq j$ *such that* $w_{\mathcal{C}2}^{m} \vdash \mathcal{C}2.start$
  - $\forall i, w_{\mathcal{C}2}^{i} \vdash ended \implies w2^{i} \vdash Ended_{\varphi}$

Briefly, $w_{\mathcal{C}1}$ is equivalent to $w1$ if they are equivalent on inputs, and for the left and right operands of Ω, if they take value *true* in $w1$, their corresponding trigger should take value *true* in $w_{\mathcal{C}1}$. Additionally, $w_{\mathcal{C}2}$ is equivalent to $w2$, if $\mathcal{C}2$ starts with or after $\mathcal{C}1$, and if *ended* takes *true* in $w_{\mathcal{C}2}$, $Ended_{\varphi}$ takes *true* in $w2$. The value of m in the above definition depends on the SERE operator, that is discussed later in this section.

Definition 4. *A circuit $\mathcal{C}$ implements the $\varphi Relation$ and $E_{\varphi}Relation$ dependencies; it is the interconnection of two sub-circuits $\mathcal{C}1$ and $\mathcal{C}2$ as in Fig. 6.8, and:*

- *Circuit $\mathcal{C}1$ implements the dependency $\lfloor \varphi \lhd true \rfloor_{w1}$ ($\mathcal{C}1 ||\!\!\vdash \varphi$) iff $w_{\mathcal{C}1} \equiv w1$ for all traces $w1$ and $\forall i, w_{\mathcal{C}1}^{i} \vdash start \rightarrow \lfloor \varphi \lhd true \rfloor_{w1^{i \ldots}}$.*
- *Circuit $\mathcal{C}2$ implements the dependency $\lfloor Ended_{\varphi} \lhd True \rfloor_{w2^{n}}$ ($\mathcal{C}2 ||\!\!\vdash Ended_{\varphi}$) iff $w_{\mathcal{C}2} \equiv w2$ for all traces $w2$ and $\forall m \geq i, w_{\mathcal{C}2}^{m} \vdash \mathcal{C}2.start \rightarrow \exists n \geq m, \lfloor Ended_{\varphi} \lhd true \rfloor_{w2^{n}}$*

Simply, $\mathcal{C}1$ implements *$\varphi Relation$* if $w_{\mathcal{C}1}$ is equivalent to $w1$ on $\mathbf{P}_{in}$, and also after starting the circuit at $w_{\mathcal{C}1}^{i}$, the $\lfloor \varphi \lhd true \rfloor_{w1^{i \ldots}}$ dependency holds. Additionally, $\mathcal{C}2$ implements *$E_{\varphi}Relation$* if it starts after $\mathcal{C}1$, and zero or more cycles later, $Ended_{\varphi}$ becomes *true*.

In this chapter, we explain how to synthesize $\mathcal{C}$ for SERE primitive reactants.

Based on the syntactic abstract tree of the operators and also the dependency relation between the operands, we have categorized SERE operators into three groups: *simple* SEREs, *compound* SEREs, and *unbounded* SEREs.

Here, we first explain each SERE category briefly, and then discuss how to build their corresponding hardware intuitively.

6.4.1.1 Simple SEREs

A simple SERE operator is a binary operator, whose left sub-sequence should complete, and then, the right sub-sequence starts. The sequence completes when the right sub-sequence completes. The set of simple SERE operators is denoted by $\mathit{SimSERE} = \{;, :\}$. For a simple SERE operator we have the following dependency relations:

$$\exists i < |w|, \lfloor \mathit{Ended}_A \lhd \mathit{true} \rfloor_{w^i} \wedge \lfloor B \lhd \mathit{true} \rfloor_{w^{i+1\ldots}} \text{ for ';'}$$
$$\exists i < |w|, \lfloor \mathit{Ended}_A \lhd \mathit{true} \rfloor_{w^i} \wedge \lfloor B \lhd \mathit{true} \rfloor_{w^{i\ldots}} \text{ for ':'}$$

Limitations and remarks. The following remarks should be considered:

1 The count repetition operator, $[*n]$, can be categorized as a simple SERE operator, since we can rewrite it as n concatenations of its operand.

2 If the left sub-sequence is an unbounded repetition (e.g. $\varphi = A[*]; b$), the concatenation operator belongs to *SimSERE*, however, a new limitation is added: the right sub-sequence should be a Boolean expression.

6.4.1.2 Compound SEREs

A compound SERE operator is a binary operator, whose left and right sub-sequences start at the same time. The completeness of the sequence depends on the operator. The set of compound SERE operators is denoted by $\mathit{CompSERE} = \{\&\&, \&, |\}$. For compound SERE operators we have the following dependency relations:

$$\lfloor A \lhd \mathit{true} \rfloor_w \wedge \lfloor B \lhd \mathit{true} \rfloor_w \text{ for '\&\&'}$$
$$\exists i < |w|, (\lfloor A \lhd \mathit{true} \rfloor_w \wedge \lfloor B \lhd \mathit{true} \rfloor_{w^{0\ldots i}}) \vee (\lfloor A \lhd \mathit{true} \rfloor_{w^{0\ldots i}} \wedge \lfloor B \lhd \mathit{true} \rfloor_w) \text{ for '\&'}$$
$$\lfloor A \lhd \neg B \rfloor_w \vee \lfloor B \lhd \neg A \rfloor_w \text{ for '|'}$$

Limitations and remarks. Based on the operator and also the direction and type of the operands, some limitations may apply to observe or constrain a compound SERE.

1 The left and right sub-sequences are not an unbounded repetition.

2 If all the signals of both operands are in $\mathbf{P}_{out}$, both operands should be constrained. In this case, Ω cannot be the && operator. If $\Omega = |$, deciding which operand is observed cannot be made locally to a property. This will be explained in Chapter 9. In addition, if $\Omega = |$, both operands should be Boolean expressions, possibly within curly brackets, i.e. SEREs of length 1.

4 If all the signals of operand A are in $\mathbf{P}_{in}$, A is observed. If some of the signals of the other operand, B, are in $\mathbf{P}_{out}$, B should be constrained, based on the value of the observed operand. In this case, the observed operand should be a Boolean expression (see Example 7). Due to the commutativity of &, &&, and |, the roles of A and B can be exchanged. In this case, if $\Omega \in \{|, \&\&\}$, both operands should be Boolean.

6.4.1.3 Unbounded SEREs

An unbounded SERE operator is a unary operator. The set of unbounded SEREs is defined as $\mathit{UnbSERE} = \{*, +\}$.

Limitations and remarks. Assume that $\varphi = A[\Omega]$, $\Omega \in \{+, *\}$. Based on the type of the operands and their signal directions some limitations may apply to observe or constrain an unbounded SERE.

1 When evaluating[6] A, we need to know when the evaluation should be terminated (see Example 3). Therefore, φ should be followed by a Boolean expression. Based on this limitation, we assume that $\varphi = \{A[\Omega]; b\}$, b is observed, and we have the following dependency relation:

$$\exists i < |w|, \lfloor b \lhd \mathit{true} \rfloor_{w^{i \ldots}} \wedge \forall j < i, \lfloor A \lhd \neg b \rfloor_{w^{j \ldots}}$$

2 If $\Omega = +$, we assume that $\lfloor \neg b \lhd \mathit{true} \rfloor_{w^0}$. Because based on the semantic definition of the *plus* operator, its operand should occur at least once ($|w| > 0$).

Considering the mentioned limitations, the synthesizable subset of SEREs, *SynSERE*, is defined as:
$\mathit{SynSERE} = \mathit{SimSERE} \cup \mathit{CompSERE} \cup \mathit{UnbSERE}$.

6.4.2 Implementation of primitive reactants of SERE operators

In this section we address how a SERE primitive reactant can be implemented intuitively using a simple example for each SERE category.

6.4.2.1 Simple SEREs

Example 10. Implementation of $\varphi = \{b1; b2\}$

First, assume φ is observed; hence, $b1$ and $b2$ are observed. The hardware is shown in Fig. 6.9. When circuit starts, $\mathit{start} = 1$, $b1$ should be observed. If it is 1, *trig_l* becomes 1, and in the next cycle, we observe $b2$. If $b2$ is 1, *trig_r* becomes 1, and the sequence completes ($\mathit{trig} = 1$).

If φ is generated, we generate $b1$ when the circuit starts, and we generate $b2$ in the next cycle (see Fig. 6.10). Therefore, *trig_l* constrains $b1$ ($\mathit{Trig}_{b1} = \mathit{trig\_l}$), and in the next cycle, *trig_r* constrains $b2$ ($\mathit{Trig}_{b2} = \mathit{trig\_r}$), and the sequence completes.

[6] here evaluation means either observation or generation

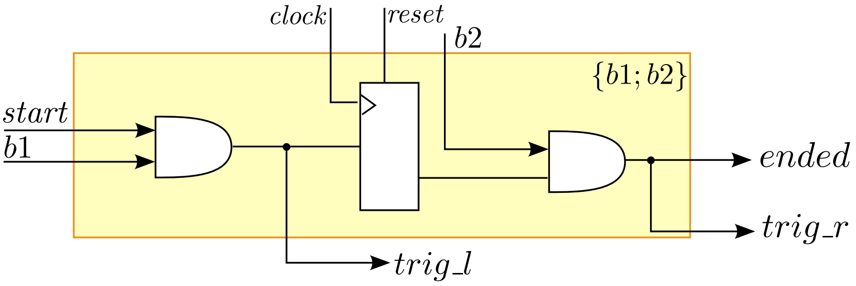


FIGURE 6.9: Implementation of $\{b1;b2\}$ ($b1$ and $b2$ are observed)

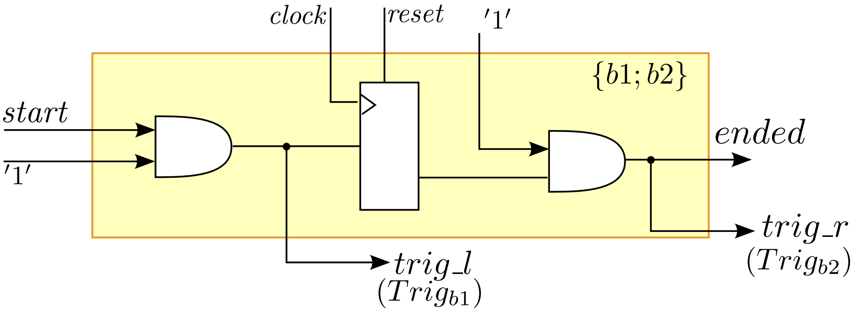


FIGURE 6.10: Implementation of $\{b1;b2\}$ ($b1$ and $b2$ are generated)

Example 11. Implementation of $\varphi = \{q;b\}$

Assume that we want to generate φ; therefore, we generate q and b. The circuit of q is activated when the circuit of φ starts ($start = 1$). Then, we should wait for the completeness of q. Simply, we should register the start signal to indicate a sequence has been already started and is in progress. *trig_l* becomes 1 when q completes and the registered start signal is 1. In the next cycle, b is generated (*trig_r* $= 1$), and the sequence completes (*ended* $= 1$).

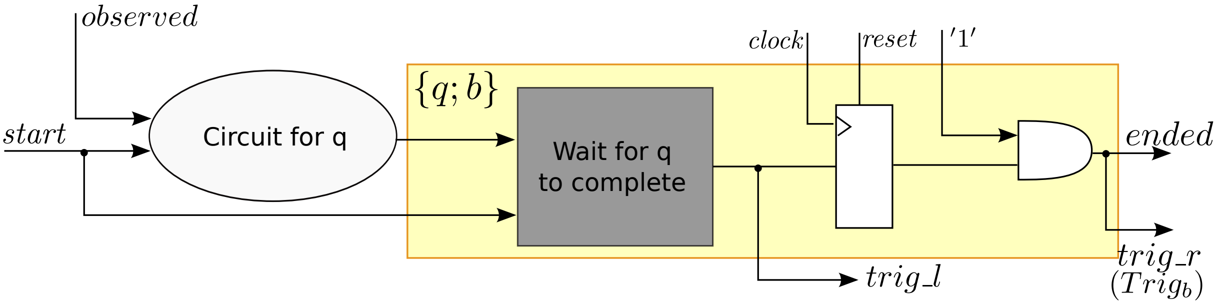


FIGURE 6.11: Implementation of $\{q;b\}$ (q and b are generated)

For clarifying this discussion, assume that $q = \{b1;b2\}$. Figure 6.12 shows the corresponding trace. If the circuit starts at cycle ♯0, then q takes two cycles to complete. In this period, the circuit that is waiting for the completeness of q, registers the start signal (see *start_reg* in Fig. 6.12). In cycle ♯1, q completes, and *start_reg* is 1, therefore, *trig_l* becomes 1 that shows the completeness of q. In the next cycle b is generated and the sequence completes.

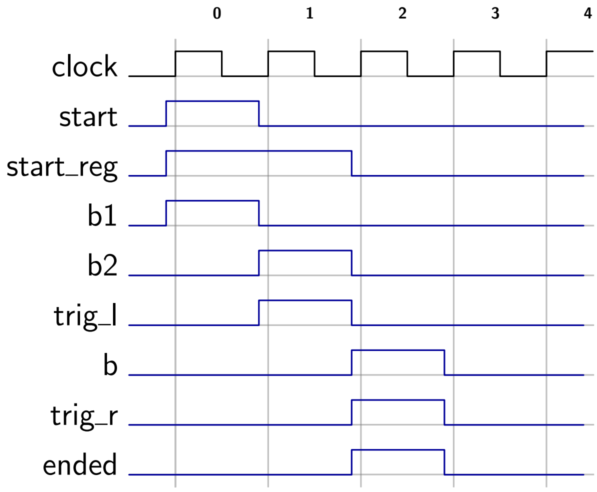


FIGURE 6.12: Timing diagram of $\{\{b1; b2\}; b\}$

Example 12. Implementation of $\varphi = \{b; q\}$

Assume that we want to generate φ. Therefore, we generate b and q. The implementation is shown in Fig. 6.13. When the circuit starts, *trig_l* becomes 1 that constrains b. In the next cycle, *trig_r* starts the circuit of q. Then, we should wait for the completeness of q; and *ended* becomes 1 when q completes.

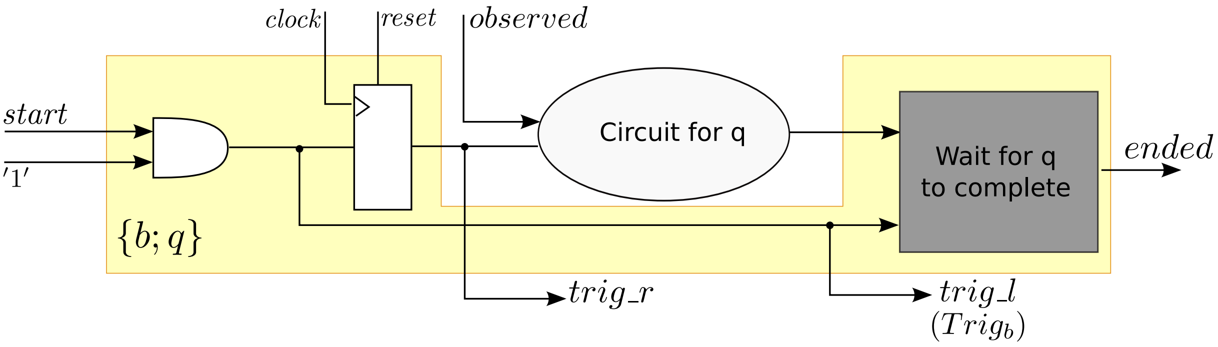


FIGURE 6.13: Implementation of $\{b; q\}$ (b and q are generated)

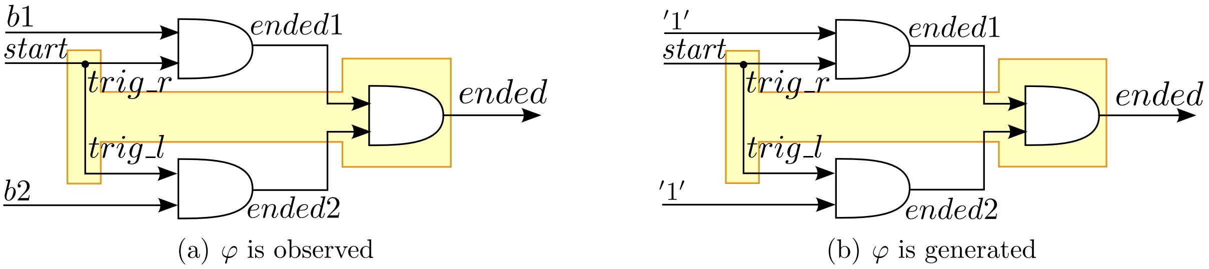
As is shown in these examples, when the left sub-sequence is a SERE, we should wait for its completion and register the start signal. The right sub-sequence starts when the left sub-sequence completes. In order to identify when the sequence completes, we need to implement a circuit to wait for completion of the right sub-sequence, while registering the *trig_l* signal.

The same discussion is valid for the *fusion* operator. The only difference is that the right sub-sequence starts at the last cycle of the left sub-sequence. Therefore, it is not necessary to have a flip-flop between *trig_l* and the start of the right sub-sequence.

6.4.2.2 Compound SEREs

Example 13. Implementation of $\varphi = \{b1\}\&\{b2\}$

φ is implemented like a logical AND. Suppose that φ is observed; then $b1$ and $b2$ are observed (see Fig. 6.14 (a)). In the case of generation, both operands should be generated at the same time (see Fig. 6.14 (b)). Then, $ended1$ and $ended2$ constrain $b1$ and $b2$ respectively. $ended$ becomes 1 if both operands are 1 (in the case of observation), or both are generated (in the case of generation).



(a) φ is observed (b) φ is generated

FIGURE 6.14: Implementation of $\{b1\}\&\{b2\}$

Example 14. Implementation of $\varphi = \{q\}\&\{b\}$

Assume that we want to generate φ. Therefore, both sub-sequences should be generated, and they should start at the same time ($trig\_l$ = $trig\_r$ = $start$). The right sub-sequence is a Boolean, it is constrained by $ended2$ when the circuit starts. The left sub-sequence is a SERE. We should wait for its completeness. The φ sequence completes ($ended$ = 1), when q completes ($ended1$ = 1). Figure 6.15 shows the implementation. For clarifying the discussion, suppose that $q = \{b1; b2\}$. The trace is shown in Fig. 6.16.

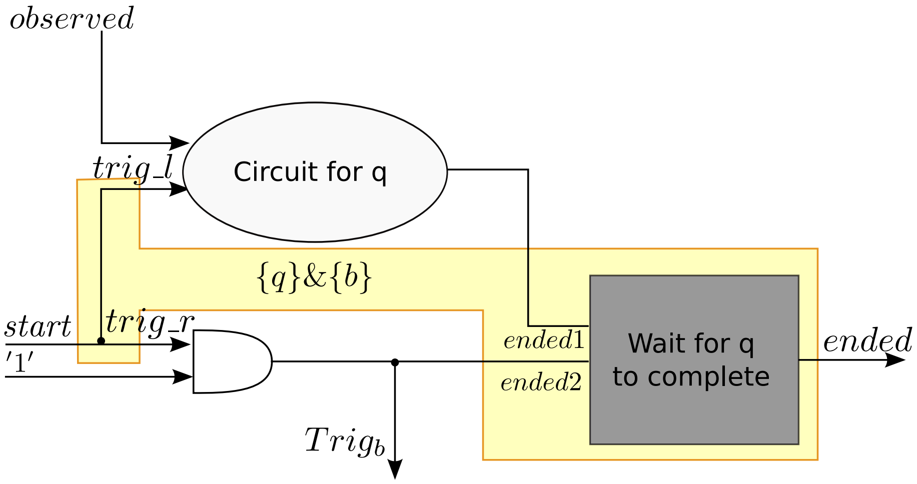


FIGURE 6.15: Implementation of $\{q\}\&\{b\}$ (q and b are generated)

Both sub-sequences start in cycle ♯0. $ended2$ becomes 1 in cycle ♯0 and constrains b. The circuit of q starts at the same cycle; it generates $b1$ in cycle ♯0. We should wait for completeness of q, while registering $ended2$ (see $ended2\_reg$ in Fig. 6.16, it is the output of a register inside the module that is waiting for the completeness of q). In cycle ♯1, $b2$ is generated, and q completes ($ended1 = 1$). Since $ended1$ and $ended2\_reg$ are 1 in cycle ♯1, the sequence completes in cycle ♯1 ($ended = 1$).

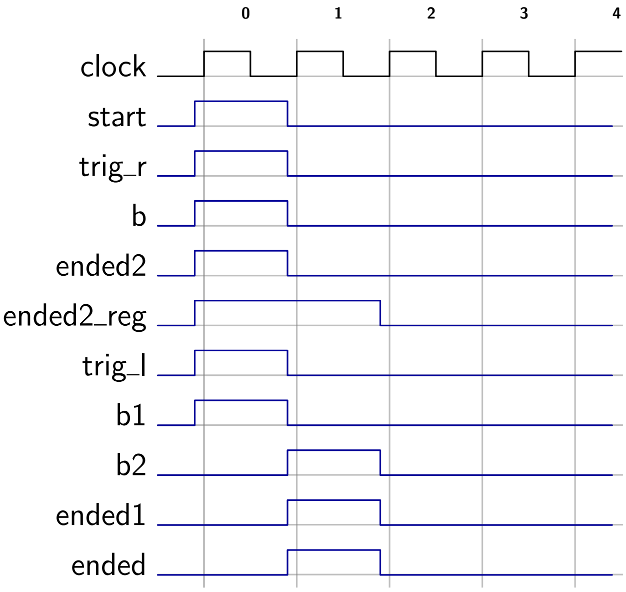


FIGURE 6.16: Timing diagram of $\{\{b1; b2\}\&b\}$

Example 15. Implementation of $\varphi = \{q1\}\&\{q2\}$

If we want to generate φ, we should generate both $q1$ and $q2$. Figure 6.17 shows the implementation. Both circuits of $q1$ and $q2$ start at the same time ($trig\_l = trig\_r = start$). The sequence φ holds if $q1$ and $q2$ hold, and completes when the longer sequence completes. Therefore, we need to wait for the sub-sequences to complete ($ended1 = 1$ and $ended2 = 1$), and specify the value of $ended$ based on $ended1$ and $ended2$.

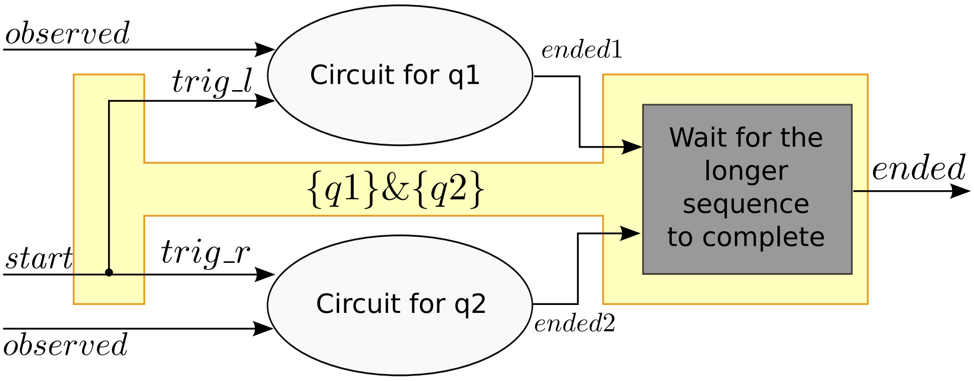


FIGURE 6.17: Implementation of $\{q1\}\&\{q2\}$

In the case of monitoring, we should monitor both $q1$ and $q2$. In this case, the hardware is more complex and is not beneficial, since we need to implement an extra circuit to guarantee if two sub-sequences start at the same time. At this point, we put a limitation that $\varphi = \{q1\}\&\{q2\}$ can be just generated, not monitored. Later in this chapter, we explain how we can overcome this limitation.

6.4.2.3 Unbounded SEREs

Example 16. Implementation of $\varphi = b[+]$

Assume that b is generated. The implementation is shown in Fig. 6.18. In this figure the *trig_l* signal constrains b. The *once_more* internal signal becomes 1, one cycle after each occurrences of b. b should be generated for 1 (when $start = 1$) or more times (when $once\_more = 1$). However, when should the generation of b be terminated? As was discussed in Section 6.2, we put a limitation that each unbounded repetition should be followed by a Boolean, which is the stopping condition.

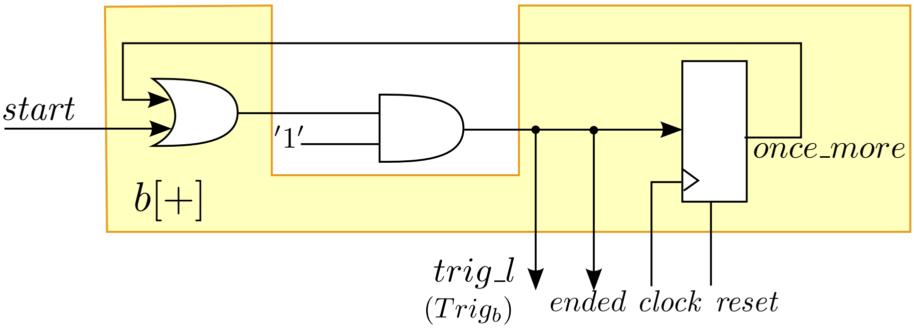


FIGURE 6.18: Implementation of $b[+]$

Example 17. Implementation of $\varphi = b1[*]; b2$

In this this example, we observe $b2$, and stop generating $b1$ as soon as $b2$ becomes 1. Figure 6.19 shows the corresponding circuit. Therefore, in addition to considering *start* and *once_more* for generating $b1$, we should consider $b2$. The shaded AND gate implements the dependency of $b1$ on $b2$: *trig_l*, which constrains $b1$ becomes 1 if $b2$ is 0. As is shown in Fig. 6.19 the completeness of the sequence is detected by the circuit of the concatenation. If $b2$ is 1 in the first cycle, $b1$ does not occur, and the sequence completes.

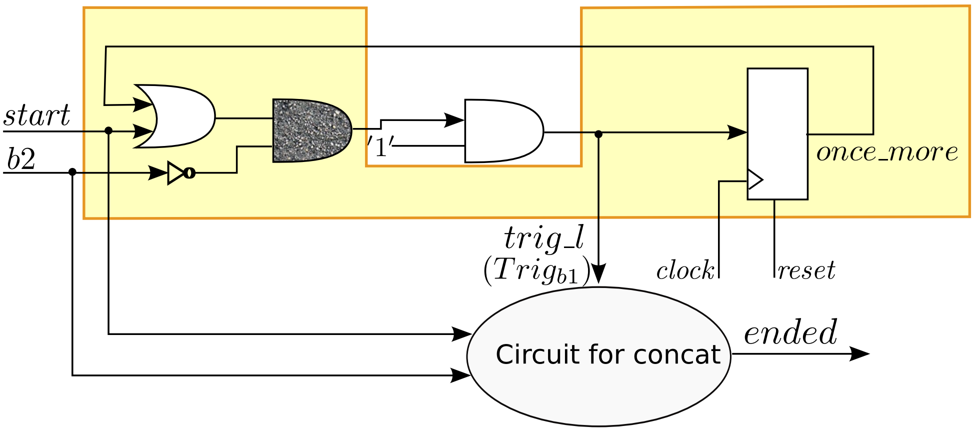


FIGURE 6.19: Implementation of $b1[*]; b2$

Example 18. Implementation of $\varphi = q[*]; b$

Figure 6.20 shows the corresponding circuit. The only difference is that after the first generation of q, if it is possible due the value of b, we should wait for its completeness (wait until *q.ended* becomes 1), and then start generating the next q.

For clarifying the discussion assume that $q = \{b1; b2\}$. The trace is shown in Fig. 6.21. *start* is 1 in cycle ♯0, and b is 0. Therefore, *q.start* becomes 1 to start generating the first

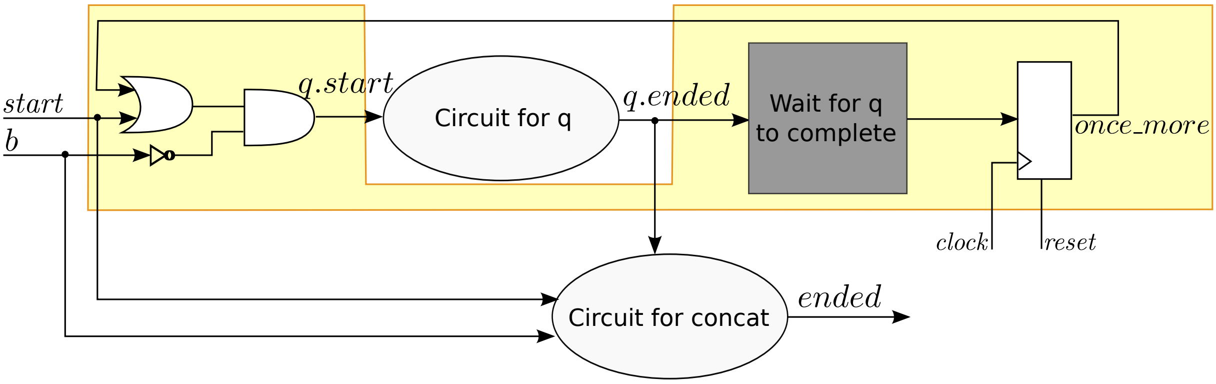


FIGURE 6.20: Implementation of $q[*];b$

occurrence of $\{b1;b2\}$. $\{b1;b2\}$ completes ($q.ended = 1$) in cycle ♯1, *once_more* becomes 1 in the following cycle, and since b is 0, $q.start$ becomes 1 to start the circuit of q. q completes in cycle ♯3. In cycle ♯4, b is 1. Therefore, $q.start = 0$, and the sequence completes ($ended = 1$).

Now, assume that the *start* signal becomes 1 again in cycle ♯6. We generate $\{b1;b2\}$; it completes in cycle ♯7; therefore, *once_more* becomes 1 in cycle ♯8, and since b is 0, $q.start$ becomes 1 to generate $\{b1;b2\}$. However, b becomes 1 in cycle ♯9, when $\{b1;b2\}$ has not been completed yet. At this point, the value of b is not taken into account until cycle ♯10, i.e. the cycle after $\{b1;b2\}$ completes.

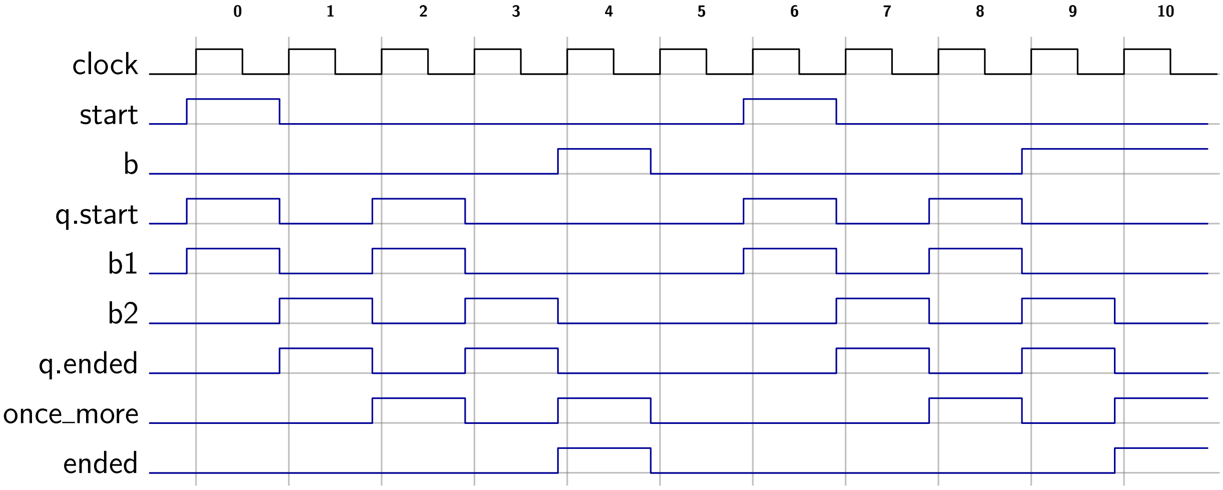


FIGURE 6.21: Timing diagram of $\{\{b1;b2\}[*];b\}$

6.5 Summary

The “synthesizable subset of SEREs” supported in this thesis are the SEREs in the following form:

$$
\begin{aligned}
SERE_{synth} = & BoolExpr.\\
& \mid \{SERE_{synth}\}\\
& \mid SERE|{-}{>}SERE_{synth}\\
& \mid SERE|{=}{>}SERE_{synth}\\
& \mid SERE_{synth}; SERE_{synth}\\
& \mid SERE_{synth} : SERE_{synth}\\
& \mid SERE_{synth} \& SERE_{synth}\\
& \mid \{BoolExpr\} \mid \{BoolExpr\}\\
& \mid SERE_{synth}[*n]\\
& \mid SERE_{synth}[*]; BoolExpr\\
& \mid SERE_{synth}[+]; BoolExpr\\
& \mid [+]; BoolExpr\\
& \mid [*]; BoolExpr
\end{aligned}
$$

Assume that φ is a SERE, A and B (if it exists) are its sub-sequences, and Ω is any SERE operator defined above. Then φ can be synthesized if it meets the following conditions:

1 If $\varphi = A[\Omega]$:

   1 If $\Omega \in \{+, *\}$, then $A[\Omega]$ should be followed by a Boolean expression, which is the stopping condition for evaluating A. Therefore, we should have: $\varphi = \{A[\Omega]; b\}$, where b is a Boolean expression and is observed.

   2 If $\Omega = +$, we assume that b is 0 when the sequence starts (A should occur at least once).

2 If $\varphi = A\Omega B$, and $\Omega \in \{;, :\}$, if the left sub-sequence is an unbounded repetition the right sub-sequence should be a Boolean expression.

3 If $\varphi = A\Omega B$, and $\Omega \in \{\&, |\}$:

   1 The left and right sub-sequences are not unbounded repetition.

   2 If all the signals of both operands are in $\mathbf{P}_{out}$ and $\Omega = |$, both operands should be Boolean expressions.

   3 If $\Omega = \&$, if all the signals of an operand, assume A, are in $\mathbf{P}_{in}$, and some of the signals of the other operand B are in $\mathbf{P}_{out}$, A should be a Boolean expression. Due to the commutativity of &, the roles of A and B can be exchanged.

Chapter 7

Annotation of the signals

Contents

7.1 Introduction

Each property written in a set of circuit specifications defines a piece of the circuit behavior, for which some signals are considered inputs (they are observed or monitored), and some signals are outputs (they are generated).

To synthesize a property, it is essential to specify the direction of the signals involved in the property. We call this process *annotation*. This is the topic of this chapter, using the formalism introduced in Chapters 5 and 6. The annotation of FL properties is illustrated on the receiver side of GenBuf, which we call GenBufRec.

7.2 Problem definition and overall view

If we are interested in monitoring an existing design, the global property outputs *trig* (in the case of FL) or *trig_l* and *trig_r* (in the case of SEREs) are used to *monitor* the expected value of the operand. To do so, the property trigger (*trig*, *trig_l*, or *trig_r*) is connected to the input *trig* of the multiplexer of Fig. 7.1. This constitutes the monitor for the property: all the signals are inputs of the monitor.

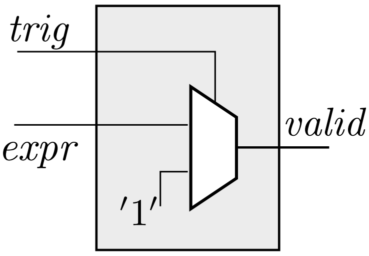


FIGURE 7.1: Monitoring the value of a Boolean expression

If we are interested in synthesizing the design, the global property's trigger output (*trig*, *trig_r* or *trig_l*) is used to *generate* the value of one or more signals, depending on the values of the *monitored* input signals.

Example 1. Property `P3_rec_0`.

Consider property `P3_rec_0` from GenBufRec (see Chapter 4):

```
P3_rec_0:
  assert (always ( rose (BtoR_REQ(0)) -> next! (next_event! (prev(not
     BtoR_REQ(0))) (not BtoR_REQ(0) until_ (BtoR_REQ(1))))));
```

In this property, *BtoR_REQ(0)* is both monitored and generated.

It is thus essential to determine, for each instance of a signal *Sig* in a property, if it is monitored (and we annotate it *Sig_m*) or generated (annotated as *Sig_g*).

7.2.1 Representation of the dependency relation

The dependency $\lfloor A \lhd B \rfloor$ can be represented as is shown in Fig. 7.2. As this figure implies, the value of B should be *monitored*, and A is *generated* based on the value of B.

In particular, assuming that $\varphi = A\Omega B$, and Ω is a binary FL or SERE operator, we can represent the $\lfloor A \lhd B \rfloor_w$ dependency among A and B as is demonstrated in Fig. 7.3(a).

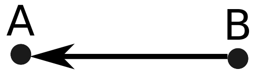
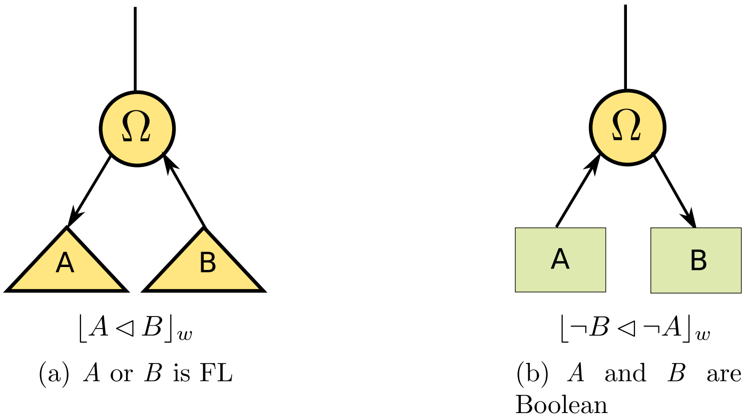


FIGURE 7.2: The representation of $\lfloor A \lhd B \rfloor_w$

In this figure, we represent φ by its Abstract Syntax Tree (AST). The directions obtained from the dependency relation directly implies that we should observe B, and based on its value generate A. If A and B are Boolean, the dependency relation can be reversed (Property 5 in Chapter 5): we observe A and generate B (see Fig. 7.3(b)).



$\lfloor A \lhd B \rfloor_w$

(a) A or B is FL

$\lfloor \neg B \lhd \neg A \rfloor_w$

(b) A and B are Boolean

FIGURE 7.3: The representation of the $\lfloor A \lhd B \rfloor_w$ dependency relation

Representing the dependency relations in this way enables us to specify the signals' directions. Therefore, the idea is considering the properties one-by-one, and constructing the Abstract Syntax Tree of each property. We use the dependency relations of the FL and SERE operators, and interpret these relations into the edge directions of the abstract syntax tree.

The remainder of this chapter explains the principles of annotating reactant operands.

7.3 Construction of the property Abstract Syntax Tree (AST)

The Abstract Syntax Tree (AST) of a property is a classical binary non-directed tree. The leaves are the design signals [1], that may be observed or generated; the other nodes are the temporal and logical operators. We denote AST = (V, E), where:

- V is the set of nodes (or vertices) of the tree. L is the set of leaves (the operands of the property), and $N = V \setminus$ L is the set of internal nodes (the operators).
- $E \subset V \times V$ is the set of edges of AST. $(v_1 - v_2)$ denotes an edge between two nodes v_1 and v_2; v_1 is the parent and v_2 is a child.

Three partial functions are defined on V: $\mathcal{P}(v)$, Lch(v), Rch(v) return the parent, the left child and the right child of node v.

[1] There are some exceptions, e.g. when having bounded repetition, the bounds of the repetition operator are leaves of the tree. For instance, in $[*3\ to\ 6]$, 3 and 6 are the leaves.

Example 2. AST of `P3_rec_0`.

Figure 7.4 illustrates the AST of `P3_rec_0` (see Chapter 4 for details).

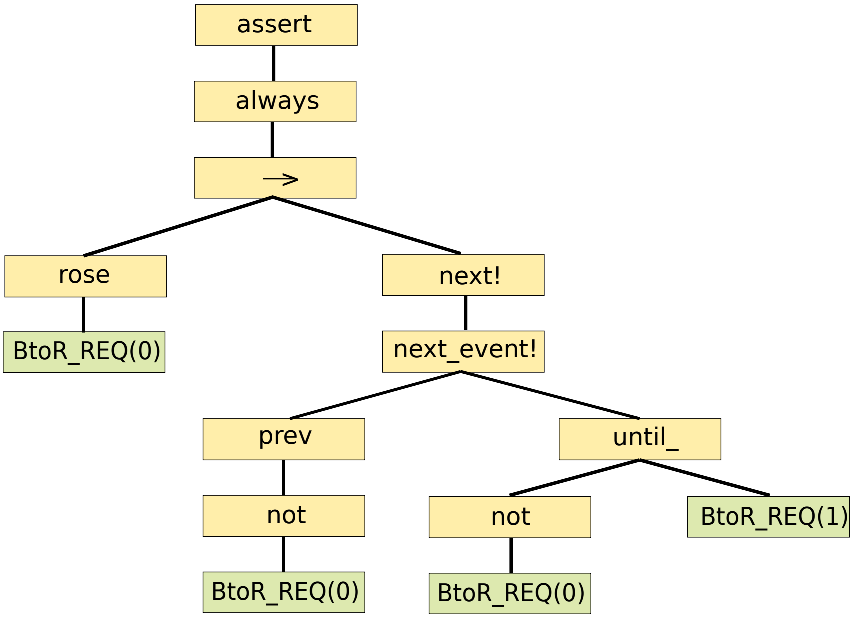


FIGURE 7.4: The abstract syntax tree of `P3_rec_0`

Take as example, $v =$ `next!`. Then we have:

$$\mathcal{P}(v) = \text{–>}$$
$$\text{Lch}(v) = \texttt{next\_event!}$$
$$\text{Rch}(v) = NULL$$

Example 3. AST of `HDLC_240`.

Consider property `HDLC_240` [2].:

```
HDLC_240:
  assert always( {not TxLastBit and not TxDataWr; TxLastBit and not
      TxDataWr}
    |-> { {not TxDout; (TxDout)[*6]; not TxDout};
      {TxDout}[*]; { {TxEnable and TxDataWr} | {TxDout and (not TxEnable
          or not TxDataWr)} } }) abort not reset_n;
```

Fig. 7.5 illustrates part of the AST of `HDLC_240`.
As an example, consider the leftmost repetition, $v =$ `REP`. Then we have:

$$\mathcal{P}(v) = ;$$
$$\text{Lch}(v) = TxDout$$
$$\text{Rch}(v) = *$$

[2] The complete set of properties describe High-level Data Link Controller (HDLC), and is taken from [PPSQ13]

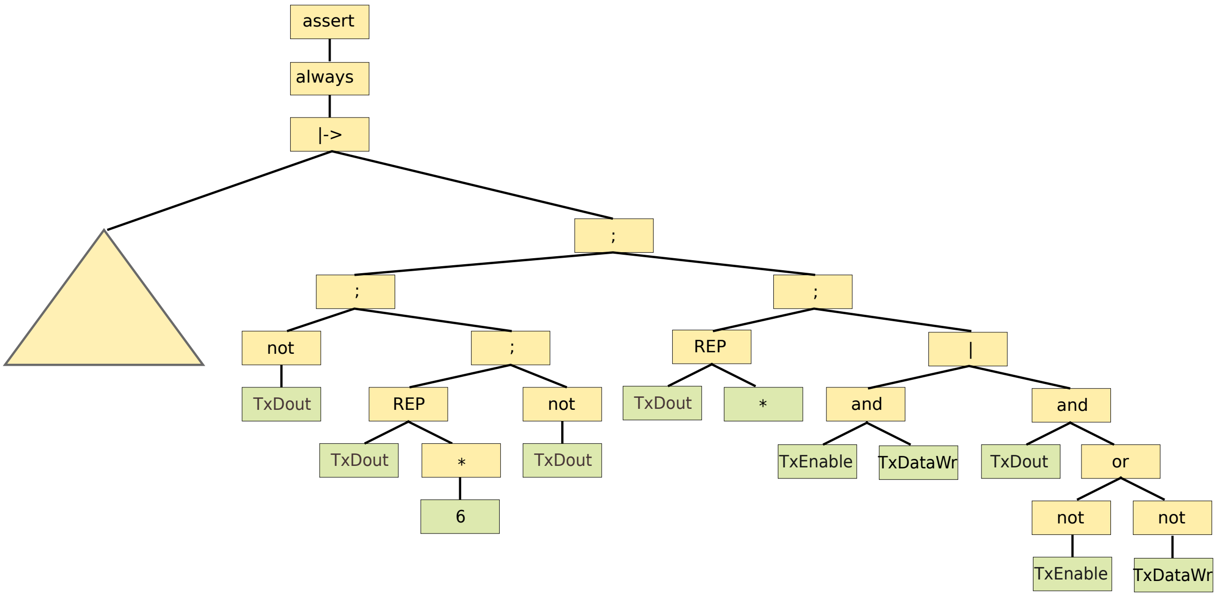


FIGURE 7.5: The abstract syntax tree of `HDLC_240`

7.4 Construction of the Directed Abstract Syntax Tree (DAST)

For each PSL operator, one or more dependency rules have been defined between its operands, in accordance with the formal semantics of the operator (see Chapters 5 and 6).

In order to determine which variables are read by a property, and which are generated, we translate the dependency relations to a directed graph, and build the *Directed Abstract Syntax Tree* (DAST) of each AST. DAST $= (V, E')$ represents the dependency between the signals of the property, going through its operators. It is built using:

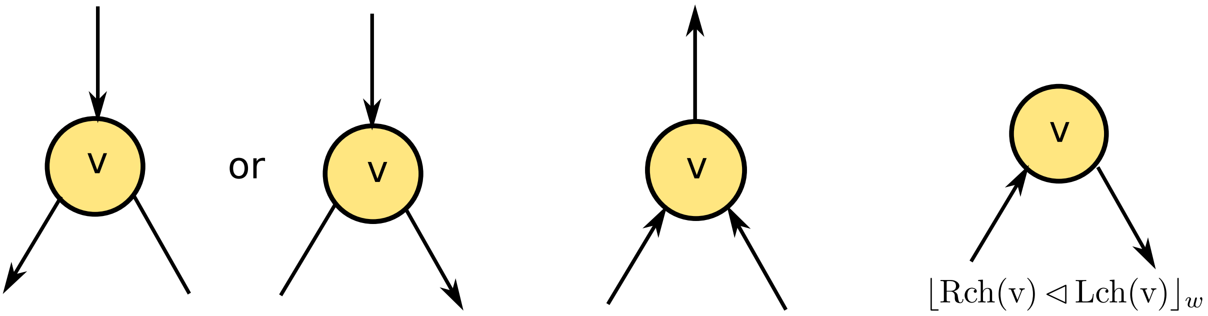
- the input/output direction of the module interface signals,
- the implementation of the dependency rules as a direction between the node for an operator and its children.

Each edge in E' is a directed edge of E. The direction of an edge is seen from the parent node. Let `v` denote the sub-property extracted from node v in DAST and `l` denote the Boolean expression of a leaf l.

- An *outgoing* edge from a parent to its child ($\mathcal{P}(v) \rightarrow v$) means that the value of `v` is constrained to ‘1’; in other words, the signals in sub-property `v` must be constrained to give value ‘1’ to `v`. It represents the dependency relation $\lfloor v \lhd \mathit{true} \rfloor$. If v is a leaf, it will be generated, otherwise this outgoing edge needs to be propagated to at least one child of v (see Fig. 7.6 (a)).
- An *ingoing* edge to a parent from its child ($\mathcal{P}(v) \leftarrow v$) means that the value of `v` is not constrained. If v is a leaf it will be observed, otherwise this ingoing edge needs to be propagated to v from all its children (see Fig. 7.6 (b)).

- If there is a directed path between the two children of a node v, e.g. $\mathrm{Lch}(v) \rightarrow v \rightarrow \mathrm{Rch}(v)$, the value of the left child constrains the value of the right child (see Fig. 7.6 (c)). It represents the relation $\lfloor \mathrm{Rch}(v) \lhd \mathrm{Lch}(v) \rfloor$

- A dependency relation may be reversed if both operands are Boolean (Property 5 of Chapter 5). Otherwise, the dependency relation may not be turned around, as this would introduce a negated FL, which is not allowed in the PSL simple subset. When two directions may hold ($\lfloor \mathrm{Rch}(v) \lhd \mathrm{Lch}(v) \rfloor$ or $\lfloor \neg\mathrm{Lch}(v) \lhd \neg\mathrm{Rch}(v) \rfloor$), edges are directed and marked *unsettled* (denoted with dash arrows). A node is unsettled if the edges to both its children are unsettled. When a node is unsettled, all its sub-trees are unsettled. Conversely, when a node is *settled*, the path from the node to the root is settled.



(a) $(\mathcal{P}(v) \rightarrow v)$ (b) $(\mathcal{P}(v) \leftarrow v)$ (c) $\mathrm{Lch}(v) \rightarrow v \rightarrow \mathrm{Rch}(v)$

FIGURE 7.6: Propagation of ingoing and outgoing edges

For each PSL and SERE operator, we have defined the direction of the parent and children edges according to their dependency relation. Here, we explain how to build the DAST of each category of FL and SERE operators based on their dependency relations. When observing a node v, all its children are observed and the problem is solved. Here, we focus on constraining a node. After constructing the DAST of each operator, we propose an annotation algorithm that utilizes these DASTs for specifying the signal directions in the AST of a property.

In the remainder of this chapter, we shall only consider DASTs in which the root node edges are outgoing to its children.

7.4.1 DAST of simple FL operators

This category contains the `always`, `never`, `eventually!`, and `next!` (also its weak version) operators. All these operators are annotated in the same way. Consider the `next!` operator, and assume $\varphi = \texttt{next!}(A)$; then:

$$\lfloor \varphi \lhd \mathit{true} \rfloor_w \text{ iff } \lfloor A \lhd \mathit{true} \rfloor_{w^{1\ldots}}$$

From the dependency relation $\lfloor A \lhd \mathit{true} \rfloor_{w^{1\ldots}}$ we can deduce that there will be an outgoing edge from `next!` to A (Fig. 7.7).

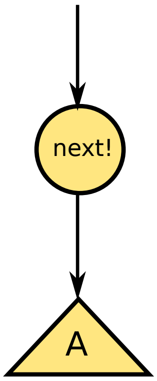


FIGURE 7.7: Edges direction for `next!`

7.4.2 DAST of extended next FL operators

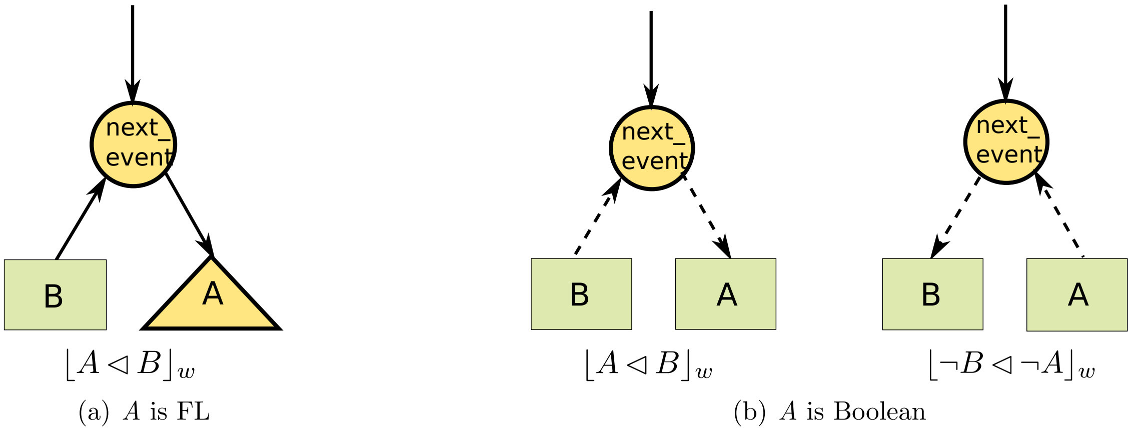
This category contains the `next_a`, `next_e`, `next_event`, `next_event_a`, and `next_event_e` operator families. The annotation of the `next_a` and `next_e` families is the same as the annotation of the `next!` operator. Here, we demonstrate the DAST of `next_event!`. Other operators of the `next_event` family have a similar DAST.

Since `next_event!` is an FL operator, its parent (if any) is an FL operator, and its parent edge direction is outgoing and it is settled.

Dependency Rule 9 (Chapter 5) gives the dependency relation of $\varphi =$ `next_event!`$(B)A$:

$$\lfloor \varphi \lhd \mathit{true} \rfloor_w \text{ iff } \exists k < |w|, \lfloor B \lhd \mathit{true} \rfloor_{w^{k\ldots}} \land \forall i \leq k, \lfloor A \lhd B \rfloor_{w^{i\ldots}}$$

The left operand B needs to be asserted before the end of the simulation. Other properties are expected to provide it. The dependency relation $\lfloor B \lhd \mathit{true} \rfloor$ is thus removed. From the second dependency relation $\lfloor A \lhd B \rfloor$ we can deduce that there will be an outgoing path from B to A (Fig. 7.8 (a)). If A is Boolean, the dependency direction may be reversed, and the path is unsettled (Fig. 7.8 (b)).



$\lfloor A \lhd B \rfloor_w$

(a) A is FL

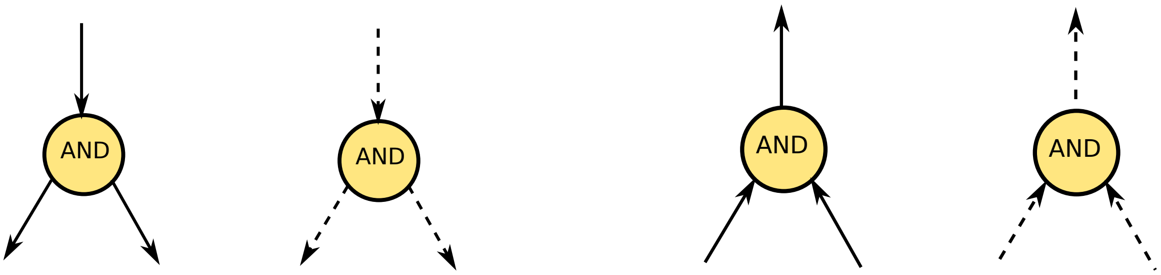
$\lfloor A \lhd B \rfloor_w$ $\lfloor \neg B \lhd \neg A \rfloor_w$

(b) A is Boolean

FIGURE 7.8: Edges direction for `next_event!`

7.4.3 DAST of FL logical operators

For the Boolean operator `and`, Property 3 of Chapter 5 ($\lfloor (A \text{ and } B) \lhd C \rfloor_w \Leftrightarrow \lfloor A \lhd C \rfloor_w \land \lfloor B \lhd C \rfloor_w$) implies that either both operands are generated (Fig. 7.9 (a)) or they are both observed (Fig. 7.9 (b)). The directions may be settled or unsettled based on the edge between "and" and its parent.

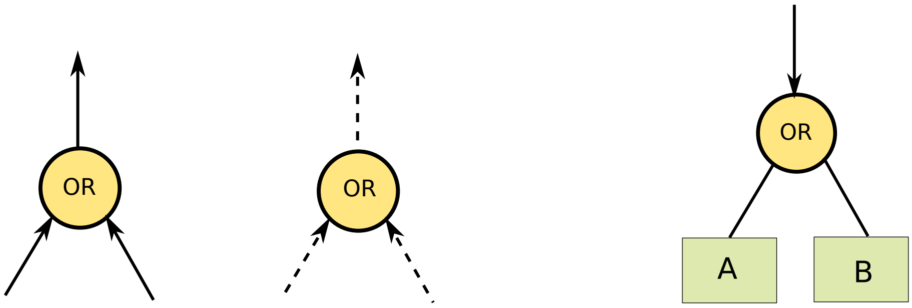


(a) Generation

(b) Observation

FIGURE 7.9: Edges direction for and

For the Boolean operator or, Property 4 of Chapter 5 ($\lfloor (A \,\texttt{or}\, B) \lhd C \rfloor_w \Leftrightarrow \lfloor A \lhd (C \wedge \neg B) \rfloor_w$) implies that either both operands are observed (Fig. 7.10 (a)), or one of the operands is generated based on the value of the other operand. We cannot annotate the signals in the case of generation (Fig. 7.10 (b)), since we may have either dependency $\lfloor A \lhd \neg B \rfloor_w$ or $\lfloor B \lhd \neg A \rfloor_w$. In Chapter 9 we explain how this problem can be solved.



(a) Observation

(b) Generation

FIGURE 7.10: Edges direction for or

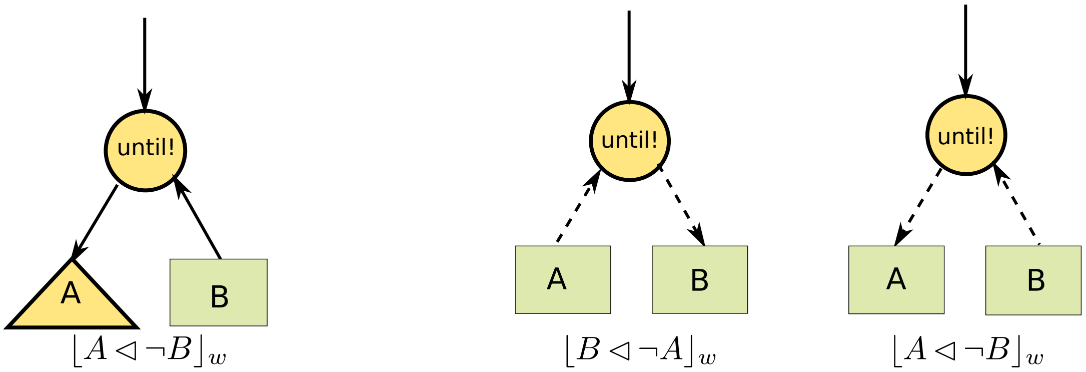
7.4.4 DAST of compound FL operators

This category contains the abort, until, and before families of operators. Here, we consider the annotation of the until! operator. Since until! is an FL operator, its parent (if any) is an FL operator, and its parent edge direction is outgoing and it is settled.

Dependency Rule 6 (see Chapter. 5) gives the dependency relation of $\varphi = A \,\texttt{until!}\; B$:

$$\lfloor \varphi \lhd \mathit{true} \rfloor_w \text{ iff } \exists k < |w|, \lfloor B \lhd \mathit{true} \rfloor_{w^{k\ldots}} \wedge \forall i < k, \lfloor A \lhd \neg B \rfloor_{w^{i\ldots}}$$

The left operand B needs to be asserted before the end of the simulation. Other properties are expected to provide it. The dependency relation $\lfloor B \lhd \mathit{true} \rfloor_{w^{k\ldots}}$ is thus removed. From the second dependency relation $\lfloor A \lhd \neg B \rfloor_{w^{i\ldots}}$ we can deduce that there will be an outgoing path from B to A (Fig. 7.11 (a)). If A is Boolean, the dependency direction may be reversed, and the path is unsettled (Fig. 7.11 (b)).


until!
A
B
$\lfloor A \lhd \neg B \rfloor_w$
until!
A
B
$\lfloor B \lhd \neg A \rfloor_w$
until!
A
B
$\lfloor A \lhd \neg B \rfloor_w$

(a) A is FL

(b) A is Boolean

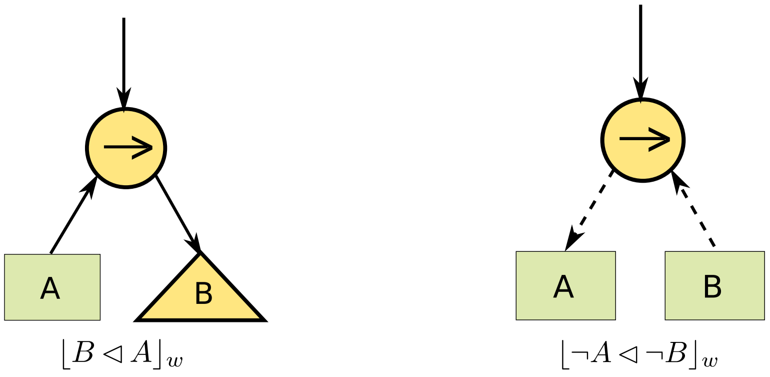
FIGURE 7.11: Edges direction for `until!`

7.4.5 DAST of implication operators

The '–>', '|–>' and '|=>' operators are annotated exactly the same. Here, we consider DAST of the '–>' operator. Since '–>' is an FL operator, its parent (if any) is an FL operator, and its parent edge direction is outgoing and it is settled. Let assume $\varphi = A\text{–>}B$. The dependency relation is given as:

$$\lfloor \varphi \lhd \mathit{true} \rfloor_w \text{ iff } \lfloor B \lhd A \rfloor_w$$

Dependency $\lfloor B \lhd A \rfloor_w$ implies that there will be an outgoing path from A to B (Fig. 7.12 (a)). Since we deal with PSL$_{simple}$ A should be a Boolean. If B is also a Boolean, the dependency direction may be reversed (Fig. 7.12 (b)).


→
A
B
$\lfloor B \lhd A \rfloor_w$
→
A
B
$\lfloor \neg A \lhd \neg B \rfloor_w$

(a) B is FL

(b) B is Boolean

FIGURE 7.12: Edges direction for '–>'

7.4.6 DAST of simple SERE operators

This category contains the ';', ':' and any bounded repetition operator (e.g. $[*n]$). Here, we consider the annotation of the ';' operator as an example.

Since ';' is a SERE operator, its parent (if any) is an FL or SERE operator, and its parent edge direction is outgoing and it is settled.

Dependency Rule 1 (see Chapter. 6) gives the dependency relation for $\varphi = A; B$:

$$\lfloor \varphi \lhd true \rfloor_w \text{ iff } \exists i < |w|, \lfloor Ended_A \lhd true \rfloor_{w^i} \wedge \lfloor B \lhd true \rfloor_{w^{i+1\ldots}}$$

The dependency $\lfloor Ended_A \lhd true \rfloor_{w^i}$ implies that A should have been already generated. Therefore, there will be an outgoing edge from ';' to its left child (see Fig. 7.13). The second relation, $\lfloor B \lhd true \rfloor_{w^{i+1\ldots}}$, implies that there is an outgoing edge from ';' to its right child. These directions are settled.

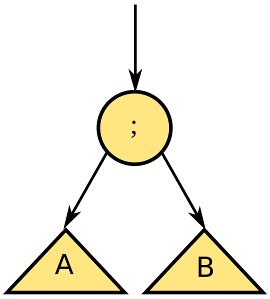


FIGURE 7.13: Edges direction for ';'

It should be mentioned that none of A and B are unbounded repetition. Otherwise, the annotation differs (see Section 7.4.8).

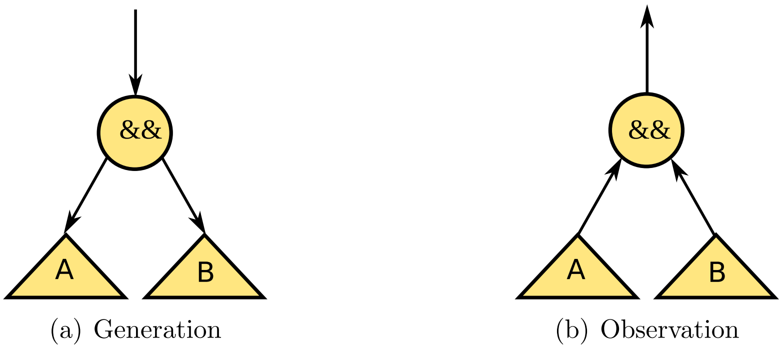
7.4.7 DAST of compound SERE operators

This groups contains the '&&', '&', and '|' operators. Here, we consider the DAST of '&&' and '|'. The DAST of '&' is the same as '&&'.

Assuming that $\varphi = \{A\}\&\&\{B\}$, we have the following dependency relation:

$$\lfloor \varphi \lhd true \rfloor_w \text{ iff } \lfloor A \lhd true \rfloor_w \wedge \lfloor B \lhd true \rfloor_w$$

This dependency relation implies that either both operands are generated or both of them are observed (Fig. 7.14).



(a) Generation

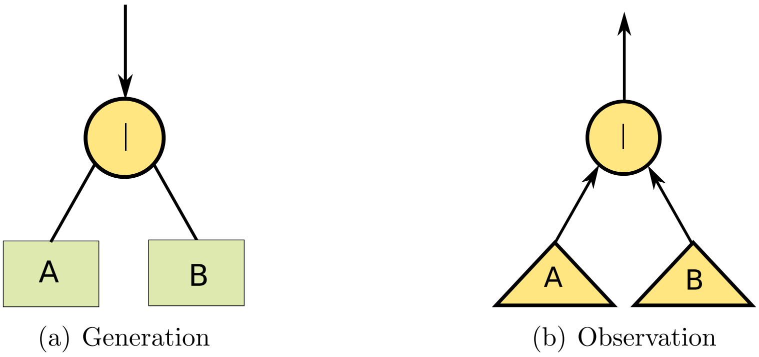
(b) Observation

FIGURE 7.14: Edges direction for '&&'

Assuming that $\varphi = \{A\}|\{B\}$, we have the following dependency relation:

$$\lfloor \varphi \lhd true \rfloor_w \text{ iff } \lfloor A \lhd \neg B \rfloor_w \;\vee\; \lfloor B \lhd \neg A \rfloor_w$$

This dependency relation implies that either both operands are observed, or one of the operands is generated based on the value of the other operand. We cannot annotate the signals in the case of generation (Fig. 7.15), since we may have either dependency $\lfloor A \lhd \neg B \rfloor_w$ or $\lfloor B \lhd \neg A \rfloor_w$. In Chapter 9 we explain how this problem can be solved. However, we can solve the problem only if A and B are Boolean expressions.



(a) Generation (b) Observation

Figure 7.15: Edges direction for '|'

7.4.8 DAST of unbounded SERE operators

This category contains the '$*$' and '$+$' operators.

As was discussed in Chapter 6, each unbounded repetition should be followed by a Boolean expression. Assume that $\varphi = A[*]; B$. We have this dependency relation:

$$\lfloor \varphi \lhd \mathit{true} \rfloor_w \text{ iff } \exists i < |w|, \lfloor B \lhd \mathit{true} \rfloor_{w^{i\ldots}} \wedge \forall k < i, \lfloor A[*] \lhd \neg B \rfloor_{w^{k\ldots}}$$

Since '$*$' is a SERE operator, its parent is a SERE operator, and its parent edge direction is outgoing and it is settled.

The dependency $\lfloor B \lhd \mathit{true} \rfloor_{w^{i\ldots}}$ implies that B finally becomes *true*. From the dependency $\lfloor A[*] \lhd \neg B \rfloor_{w^{k\ldots}}$ we can conclude that there is an outgoing path from B to A (Fig. 7.16).

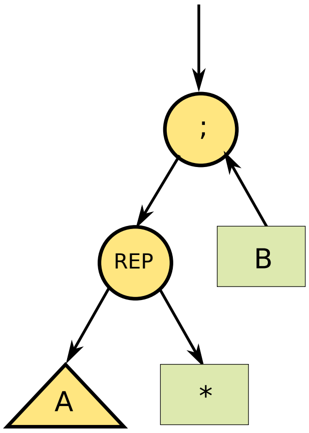


Figure 7.16: Edges direction for '$*$'

7.4.9 DAST of PSL directives and functions

In addition to FL and SERE operators, we annotate the signals for some of the functions of the Boolean and verification layers of PSL. Moreover, we annotate the operands of some the operators of the PSL modeling layer (VHDL flavor).

7.4.9.1 Boolean layer directives

We annotate the functions `rose`, `fell`, and `prev` from the PSL Boolean layer. Let assume $\varphi = \texttt{prev}(A)$, where A is a Boolean expression. A is always annotated as monitored. In addition, the direction is settled because it is based on the previous value of the operand, which may not be changed at the current cycle (see Fig. 7.17).

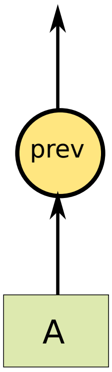


FIGURE 7.17: Edges direction for `prev`

7.4.9.2 Verification layer directives

We annotate the `assume` and `assert` directives. The operand of `assert` is always generated, while the operand of `assume` is always monitored [3].

7.4.9.3 Modeling layer operators

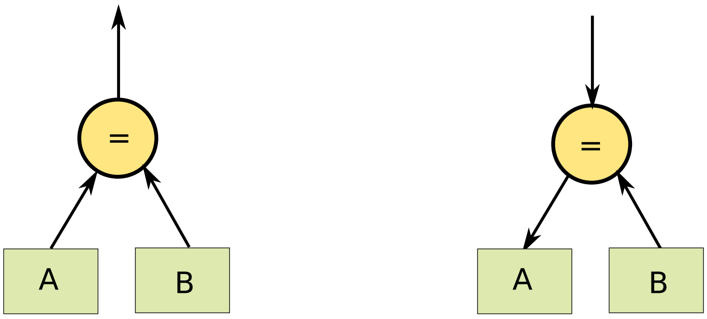
We consider the VHDL operators: Boolean, comparison and arithmetic operators. The annotation of the Boolean operators are the same as FL logical operators. Here, we consider the comparison and arithmetic operators.

Assume that $\varphi = (A = B)$, where A and B are Boolean expressions. The edge directions between '=' and its children depend on the edge direction between '=' and its parent. Figure 7.18 shows two possible directions. If there is an ingoing edge to $\mathcal{P}(=)$ from '=', then it is observed, and both children are observed (Fig. 7.18(a)). Otherwise, if the right child is observed, and A is a Boolean signal, then $A = B$ is interpreted as an assignment of B to A. Then there is an outgoing path from the right child to the left child (Fig. 7.18(b)), and the left child is generated.

In the case of the arithmetic operators, the children are observed.

Example 4.

[3] Industrial users have requested this interpretation of the directives in the context of reactants synthesis.



(a) Observation

(b) Generation

FIGURE 7.18: Edges direction for '='

Consider the following property:

```
P: always (C -> A = prev(B));
```

In this property, A is generated and B is observed, and `prev`(B) is assigned to A.

7.4.10 The annotation algorithm

The annotation process marks each signal instance in each property as either monitored or generated by the property.

Before discussing the annotation algorithm, we introduce some data structures and auxiliary functions that are used in the algorithms. We represent a DAST with the *node* data structure (see Fig. 7.19).

```
//the set of edge directions
enum Directions{none, ingoing, unsettled_ingoing, outgoing,
   unsettled_outgoing};

//the annotation type of a node: m: monitored, g: generated, u: unannotated
enum AnnotType{m, g, u} ;

//the set of all the PSL and SERE operators
enum Types{always, next, *, ...};

struct node{
  int id;                    //unique name of the node in DASTs
  Types type_node;           //the type of the node

  struct node *LCH;          //the left child of the node
  struct node *RCH;          //the right child of the node

  //the direction of edge between node n and its parent and children
  Directions PDir;
  Directions LchDir;
  Directions RchDir;

  AnnotType type_mark;        //the annotation type of a node
};
```

FIGURE 7.19: The necessary data structures for `Annotation`

The following auxiliary functions are called:

- `IsLeaf`(*node* $*a$): returns *true* if a is a leaf,
- `IsInput`(*node* $*a$): returns *true* if the corresponding signal of a is an input signal,
- `SetDir`(*node* $*a$, *Directions LchDir*, *Directions RchDir*, *Directions PDir*): assigns the directions specified in its arguments to the corresponding edges of a; if a is a leaf, this function marks the corresponding signal as 'm' or 'g, based on $PDir$. If $PDir$ = *ingoing* or $PDir$ = *unsettled_ingoing*, the signal is marked as 'm', if $PDir$ = *outgoing* or $PDir$ = *unsettled_outgoing*, the signal is marked as 'g'.
- `SettledEge`(*node* $*a$): This function considers the edge directions (*a–>LchDir*, *a–>RchDir*, and *a–>PDir*) and tries to settle them. Based on the direction of the edges of a and its operator, the new directions of the edges of a are computed, the edge directions are updated by calling `SetDir`, which returns a with the updated direction for its corresponding edges. For each operator, we have defined the set of all the possible directions for its corresponding edges (see Section 7.4). For each possible set of directions, we consider all the possible directions after being settled, and store these directions in a file for each operator.

  As an example, assume a is the `until`! operator, and:

  a–>LchDir = *none*, *a–>RchDir* = *none*, *a–>Pdir* = *settled_outgoing*

  In this case, we can have the following directions:

  - *a–>LchDir* = *settled_outgoing*, *a–>RchDir* = *settled_ingoing*
  - *a–>LchDir* = *unsettled_outgoing*, *a–>RchDir* = *unsettled_ingoing*
  - *a–>LchDir* = *unsettled_ingoing*, *a–>RchDir* = *unsettled_outgoing*

  The last two directions are for the case that the operands are Boolean.

Initially, all the edges are undirected. The annotation process is performed in two steps.

First, we start from the direction of the interface signals, and annotate all the input signals as 'm', and give a direction to its corresponding edge (see recursive function `Annotate_in` in Fig. 7.20). These directions are settled, and cannot be changed later due to the directions of the operator's edges.

```
node * Annotate_in(node *a){
  if(IsLeaf(a)){
    if(IsInput(a))
      SetDir(a, none, none, settled_ingoing);//sets direction (P(a)<-a)
    return a;
  }
  else{
    Annotate_in(a->LCH);      //if(a->LCH != NULL)
    Annotate_in(a->RCH);      //if(a->RCH != NULL)
  }
}
```

FIGURE 7.20: the pseudo code for the `Annotate_in` function

Then, the recursive `Annotate` function (Fig. 7.21) takes as input a partially directed DAST (a); it returns a more settled DAST. It starts from the root of the tree, and based on its operator, gives the direction to the corresponding edges.

```
 1 node* a Annotate(node *a){
 2   if(IsLeaf(a))
 3     return a;
 4   else{
 5     a = SettledEdge(a);
 6     left_edge_dir = a->LchDir;
 7     right_edge_dir = a->RchDir;
 8
 9     Lch(a) = Annotate(Lch(a));
10     Rch(a) = Annotate(Rch(a));
11
12     if(left_edge_dir == a->LchDir){
13       if(right_edge_dir == a->RchDir){
14         return (a);
15       else{
16         a = SettledEdge(a);
17         left_edge_dir = a->LchDir;
18         Lch(a) = Annotate(Lch(a));
19         return a;
20       }
21     }
22     else{
23       if(right_edge_dir == a->RchDir){
24         a = SettledEdge(a);
25         right_edge_dir = a->RchDir;
26         Rch(a) = Annotate(Rch(a));
27         return a;
28       }
29       else{
30         a = SettledEdge(a);
31         left_edge_dir = a->LchDir;
32         right_edge_dir = a->RchDir;
33
34         Lch(a) = Annotate(Lch(a));
35         Rch(a) = Annotae(Rch(a));
36
37         return a;
38       }
39     }
40   }
41 }
```

FIGURE 7.21: the pseudo code for the `Annotate` function

If a is a leaf, it is returned unchanged. Otherwise, we call function `SettledEdge` on a. `SettledEdge` computes a first direction for the left and right children edges ($\sharp 6$). Then, the right and left edge directions are updated with the new edge directions ($\sharp 7, 8$). `Annotate` is then called recursively on both children of a ($\sharp 10, 11$). The new edge directions are compared to the ones obtained from function `SettledEdge`. If none of the children edges changes, a is returned ($\sharp 15$). If one of the children edges changes, the change may impact the direction and type of the two other edges of node a (sibling and root edge). If only the direction of the right edge is changed (♯16), function `SettledEdge` is again called on a (♯17), the direction of the left edge is updated (♯18), and the left child is re-annotated (♯19). Similarly, if the direction of the left edge is changed, the right child should be re-annotated (♯27). If both edges' directions are changed (♯30), both children should be re-annotated (♯35, 36). The algorithm stops when none of the edge directions of the DAST changes after calling the `Annotate` function.

Example 5. DAST **of** `P3_rec_0`**.**

Figure 7.22 illustrates the DAST of property `P3_rec_0`. Next to each edge, the recursion depth is written.

At step 0, the child edge of `assert` is outgoing and settled. At step 1, `always` is outgoing and settled (see 7.4.1). At step 2, since –> has a FL operand, the two edges are settled and there is an ingoing path from left child to right child. At step 3 (left child of '–>') and according to Fig. 7.17, the edge must be settled and ingoing: signal *BtoR_REQ_0* is marked settled monitored. Step 4 (the child of `next!`) is similar to step 1, and the edge is settled outgoing. At Step 5, first, the two edges are unsettled, then both DAST children are annotated. The annotation of the left DAST (steps 6 and 7) returns a settled ingoing edge to `next_event!`. Step 8 is similar to step 5, the two edges are unsettled, but in this case the annotation of their two sub-trees is not able to settle the edges.

Finally, we get that the first two occurrences of *BtoR_REQ_0* are marked settled monitored, the third one is unsettled generated, and the occurrence of *BtoR_REQ_1* is unsettled monitored.

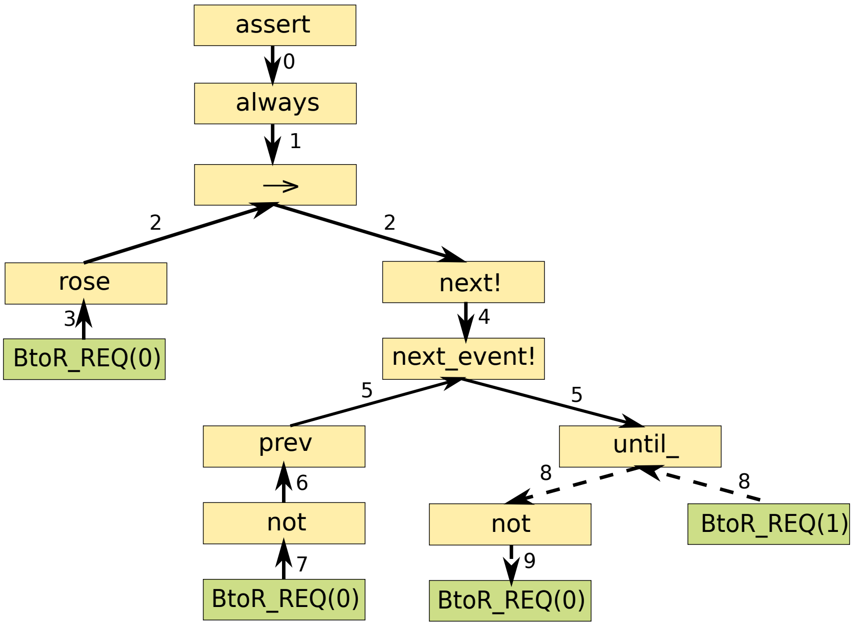


FIGURE 7.22: The directed abstract syntax tree of `P3_rec_0`

Example 6. DAST **of** `HDLC_240`**.**

Figure 7.23 illustrates the DAST of property `HDLC_240`. As is shown in this figure, we have an unbounded repetition: $TxDout[*]$. It is followed by a Boolean expression:

$$\{\{TxEnable\ \texttt{and}\ TxDataWr\} \mid \{TxDout\ \texttt{and}\ (\texttt{not}\ TxEnable\ \texttt{or}\ \texttt{not}\ TxDataWr)\}\}$$

Therefore, this Boolean expression is marked as monitored, and all its signals should be observed.

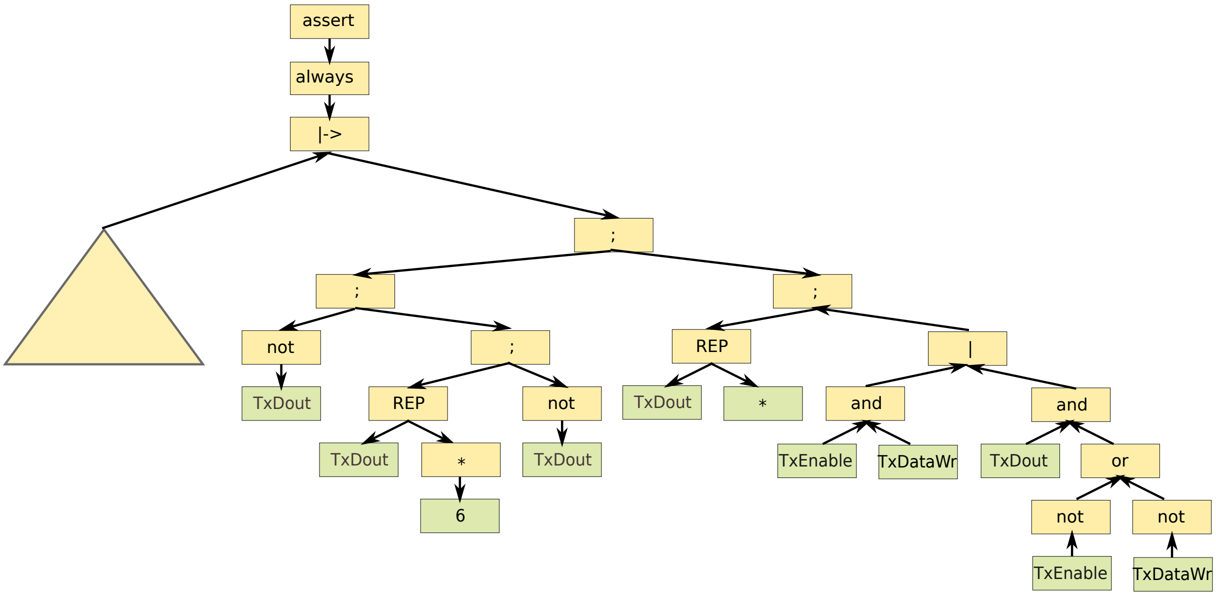


FIGURE 7.23: The directed abstract syntax tree of HDLC_240

Example 7. Annotation of the properties of GenBufRec.

Figure 7.24 shows the properties of GenBufRec after annotation. We can observe that some signals are both observed and generated. Consider *BtoR_REQ(0)*. Property P1_rec constrains this signal to 0. Property P3_rec_0 observes this signal, and also constrains it to 0. In addition, property P4_rec_0 constrains this signal to 1. *BtoR_REQ(0)* is a *duplicated* signal.

Moreover, we can see in Fig. 7.24 that some signals have not been annotated. As an example see signals *BtoR_REQ(0)* and *BtoR_REQ(1)* in property P0_rec. This property states that if GenBuf is not empty, a request should be sent to one of the receivers, but does not state which receiver. This cannot be decided locally based on this property alone; it depends on other properties that constrain *BtoR_REQ(0)* and *BtoR_REQ(1)*. We call such signals *unannotated* signals.

Example 8. Annotation of the properties of High-Level Data Link Controller (HDLC) transmitter.

Figure D.6 of Appendix D shows several annotated SERE properties of the HDLC transmitter[4]. All the signals in the set of properties have been annotated. However, there are several duplicated signals generated by several properties, for instance *TxDout*.

[4] The original properties are taken from [PPSQ13].

```
vunit genbuf_receiver
{
------ receiver side
  P0_rec:
    assert (always(not EMPTY_m -> next!(BtoR_REQ(0) or ( BtoR_REQ(1)))));

  P1_rec:
    assert (always(EMPTY_m -> next!(not BtoR_REQ_g(0) and (not BtoR_REQ_g
        (1))) ));

  P2_rec:
    assert (always (not BtoR_REQ(0) or not BtoR_REQ(1)));

  P3_rec_0:
    assert (always ( rose (BtoR_REQ_m(0)) -> next! (next_event! (prev(not
        BtoR_REQ_m(0))) (not BtoR_REQ_g(0) until_ (BtoR_REQ_m(1))))));

  P3_rec_1:
    assert (always ( rose (BtoR_REQ_m(1)) -> next! (next_event! (prev(not
        BtoR_REQ_m(1))) (not BtoR_REQ_g(1) until_ (BtoR_REQ_m(0))))));

  P4_rec_0:
    assert (always ((BtoR_REQ_m(0)) and (not RtoB_ACK_m(0))-> next! (
        BtoR_REQ_g(0))));

  P4_rec_1:
    assert (always ((BtoR_REQ_m(1)) and (not RtoB_ACK_m(1))-> next! (
        BtoR_REQ_g(1))));

  P5_rec_0:
    assert (always ( (RtoB_ACK_m(0)) -> (next! (not BtoR_REQ_g(0)))));

  P5_rec_1:
    assert (always ( (RtoB_ACK_m(1)) -> (next! (not BtoR_REQ_g(1)))));

---------- FIFO interface
  P6_FIFO_rec:
    assert (always (( fell(RtoB_ACK_m(0)) or (fell(RtoB_ACK_m(1))) and not
        EMPTY_m) -> (DEQ_g)));

  P7_FIFO_rec:
    assert (always ( not fell(RtoB_ACK_m(0)) and not fell(RtoB_ACK_m(1)) ->
          (not DEQ_g)));
}
```

Figure 7.24: Annotated FL specification of GenBuf communication with receiver in the case of two receivers

7.5 Summary

In this chapter we explained how to decide the direction of each signal involved in a property. We started by representing each property using its Abstract Syntax Tree (AST). We interpreted the dependency relation of each FL and SERE operator into a Directed Abstract Syntax Tree (DAST). The annotation algorithm uses the DAST of each operator, and builds recursively the DAST of the property.

As was shown in the above examples, two issues remain to be solved: duplicated and unannotated signals. In Chapter 8 we explain how to find the duplicated and unannotated signals using DASTs, and in Chapter 9 we explain how to resolve these signals.

Chapter 8

Complex Reactant

Contents

8.1 Introduction

In this chapter we explain how to construct the complex reactant of a property, having the primitive reactants and signal directions. We use the Directed Abstract Syntax Tree (DAST) of each property to interconnect the primitive reactants and construct the complex reactant.

8.2 Intuitive construction of a property reactant

The DAST of each property is either fully directed, or may have some undirected subtrees. The reactant is built for the fully directed sub-tree of the DAST. Each non-terminal node is replaced by an instance of the primitive reactant (i.e. hardware implementation for a temporal operator) or logic gate (for a Boolean operator) interconnected to its children. For a logic gate, the interconnection is obvious. For a primitive reactant, the interconnection principles are discussed based on the operator.

8.2.1 Intuitive construction of an FL reactant

To interconnect the FL primitive reactants corresponding to each node v of a DAST we should consider the direction of the corresponding edges of v.

- If the direction is $(\mathcal{P}(v) \to v)$, the *trig* output of $\mathcal{P}(v)$ is connected to the *start* input of v. If v is a leaf, the DAST whose root is v is assigned to the *trig* output; *trig* constrains `v`.

- If the direction is $(\mathcal{P}(v) \leftarrow v)$, the observed signal (for a leaf) or the *trig* output of v (for an internal node) is connected to the *cond* input of $\mathcal{P}(v)$.

Example 1. Reactant construction for `P5_rec_0`.

Consider the annotated property `P5_rec_0` from GenBufRec:

```
P5_rec_0:
  always ( (RtoB_ACK_m(0)) -> (next! (not BtoR_REQ_g(0))));
```

Figure 8.1 shows the DAST of this property. The DAST is fully directed; therefore, the reactant is built for the DAST.

Consider the connection of the primitive reactant of '–>' to its children. Therefore v in the above discussion is any child of '–>'. On the right-hand side, $v =$ `next!`. We have (–> → `next!`); therefore, the *trig* of '–>' should be connected to the *start* signal of `next!` (Fig. 8.2). For the left child we have (–> ← *RtoB_ACK_m(0)*); therefore, *RtoB_ACK_m(0)* should be connected to the *cond* port of the primitive reactant of '–>'.

Example 2. Reactant construction for `P0_rec`.

Consider the annotated property `P0_rec` from GenBufRec:

```
P0_rec:
  always (not EMPTY_m -> next!(BtoR_REQ(0) or BtoR_REQ(1) ) );
```

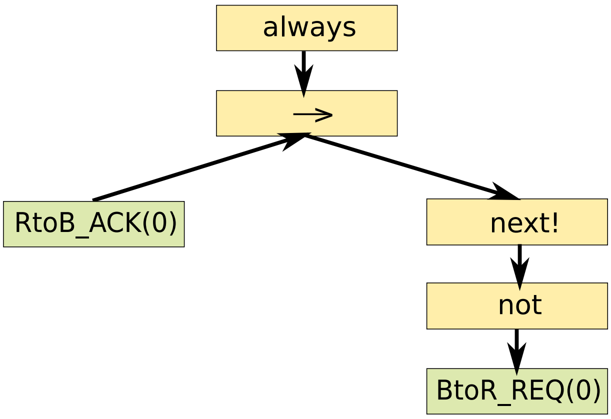


FIGURE 8.1: DAST of `P5_rec`

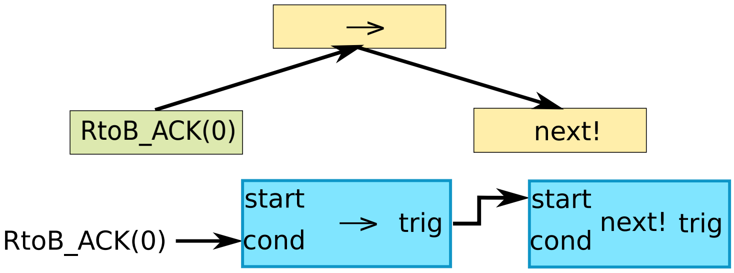


FIGURE 8.2: Interconnection of the '–>' primitive reactant (`P5_rec`)

Figure 8.3(a) shows the DAST of this property. The DAST is not fully directed (see the sub-tree whose root is `or`); signals *BtoR_REQ(0)* and *BtoR_REQ(1)* are unannotated. The reactant is built for the fully annotated sub-tree of this DAST.

The complex reactant for `P0_rec` is shown in Fig. 8.3(b). In this figure, the *trig* output of the `next!` primitive reactant constrains a Boolean expression (*BtoR_REQ(0)* `or` *BtoR_REQ(1)*), instead of constraining a signal value. We represent this expression with *Expr*, and the trigger output of the reactant with *Etrig*. Actually, *Etrig* corresponds to the unannotated sub-tree of the DAST. Component *assert* activates the circuit after reset.

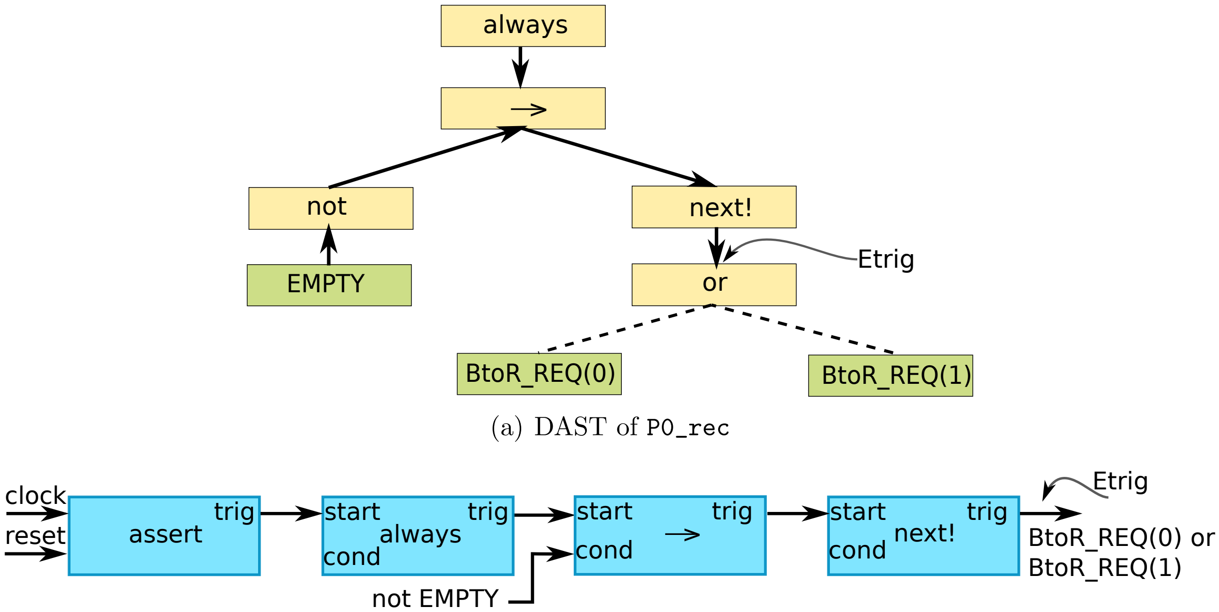
8.2.2 Intuitive construction of a SERE reactant

To interconnect the SERE primitive reactants, we should consider various categories of SERE operators. Remember from Chapter 6 that the primitive reactant of a SERE operator, has *start*, *cond*1, and *cond*2 inputs in addition to the synchronization signals. It has also three outputs: *trig_l*, *trig_r*, and *ended*. The *ended* output becomes 1 when the sequence completes.

8.2.2.1 Simple SERE

In a simple SERE sequence, e.g. $\varphi = A; B$, the primitive reactant of ';' and its left subsequence (A) start at the same time; therefore, they share the same *start* signal. Based on the edge directions between the simple SERE operator and its children we have:

- If v is the left child ($v = \mathrm{Lch}(\mathcal{P}(v))$):



(a) DAST of `P0_rec`

(b) The complex reactant for `P0_rec`

Figure 8.3: Reactant for `P0_rec`

- If v is an internal node:
  - the *start* input of $\mathcal{P}(v)$ is connected to the *start* input of v.
  - the *ended* output of v is connected to the *cond*1 input of $\mathcal{P}(v)$.
- If v is a leaf:
  - If the direction is $(\mathcal{P}(v) \leftarrow v)$, `v` (the signal represented by v) is connected to the *cond*1 input.
  - If the direction is $(\mathcal{P}(v) \rightarrow v)$, *cond*1 is connected to '1', and *trig_l* constrains `v`.

- If v is the right child $(v = \text{Rch}(\mathcal{P}(v)))$, the same rules as for the left child apply, replacing:
  - *cond*1 by *cond*2
  - *trig_l* by *trig_r*

Here, we assumed that none of the left and right sub-sequences are an unbounded repetition.

Example 3. Simple SERE reactant construction.

Consider the annotated property `HDLC_240` from the HDLC transmitter:

```
HDLC_240:
  always( {not TxLastBit_m and not TxDataWr_m; TxLastBit_m and not
      TxDataWr_m}
    |-> { {not TxDout_g; (TxDout_g)[*6]; not TxDout_g};
      {TxDout_g}[*]; { {TxEnable_m and TxDataWr_m} | {TxDout_m and (not
          TxEnable_m or not TxDataWr_m)} } });
```

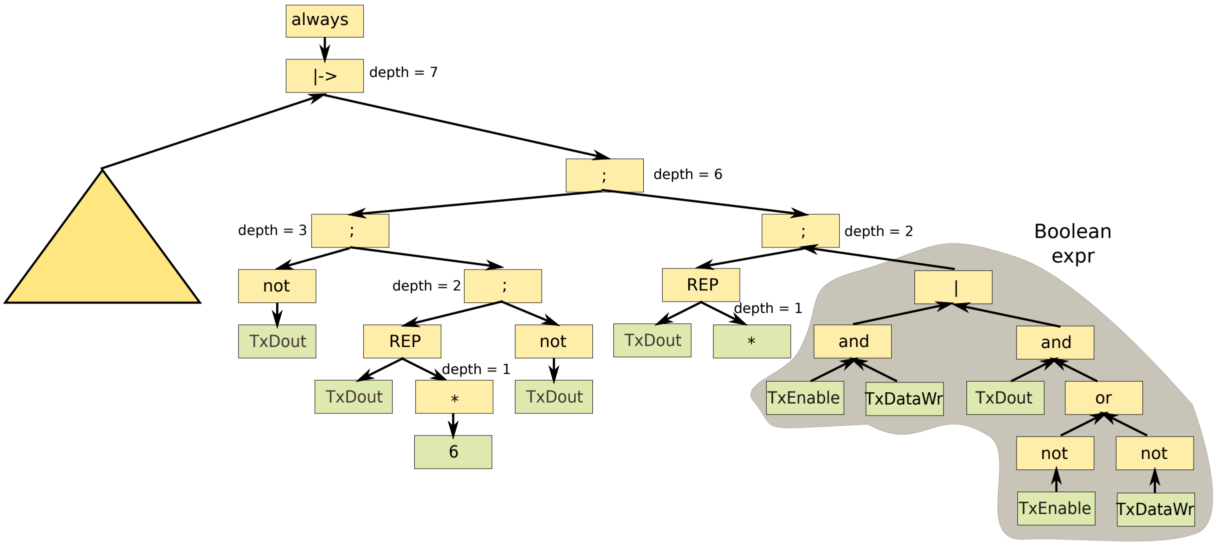
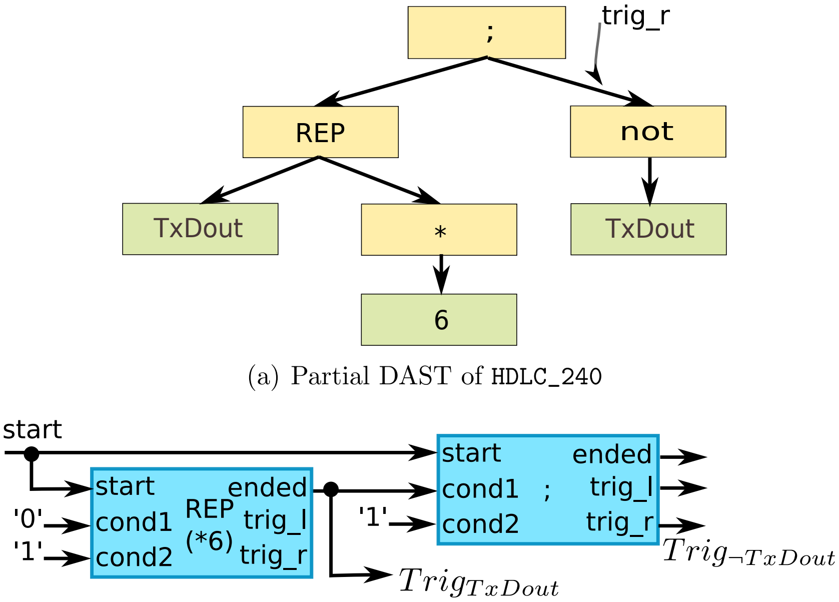


Figure 8.4: The directed abstract syntax tree of HDLC_240

The DAST of this property is shown in Fig. 8.4. In this figure, the depth of each sub-property, defined in Section 8.3.3, is written next to the root of its sub-tree.

Figure 8.5(a) shows a sub-tree of the DAST of property HDLC_240, corresponding to $(TxDout)[*6]; \texttt{not}\, TxDout$. The root of this sub-tree is operator ';'.



(a) Partial DAST of HDLC_240

(b) Interconnection of the ';' primitive reactant (HDLC_240)

Figure 8.5: Simple SERE primitive reactant interconnection

We connect the primitive reactant of ';' to the primitive reactants of its children. Node v in the above discussion is any child of ';'. First assume that $v =$ REP, left child of ';'. We have (; → REP); therefore, the *start* signal of REP is connected to the *start* signal of ';', and the *ended* signal of REP is connected to *cond*1 of ';' (Fig. 8.5(b)).

For the right child we have (; → `not`). The right child is a Boolean expression; therefore, the *cond*2 input is connected to '1' and *trig_r* constrains *TxDout* to 0 (Fig. 8.5(b)). The sub-tree of the partial DAST with root `not` is associated to the *trig_r*. The *ended* output of ';' indicates that the sequence completes, with the emission of *trig_r*.

8.2.2.2 Compound SERE

In a compound SERE sequence, both sub-sequences start at the same time.

- If v is the left child ($v = \mathrm{Lch}(\mathcal{P}(v))$):
  - If v is an internal node:
    - the *trig_l* output of $\mathcal{P}(v)$ is connected to the *start* input of v.
    - the *ended* output of v is connected to the *cond*1 input of $\mathcal{P}(v)$.
  - If v is a leaf:
    - If the direction is ($\mathcal{P}(v) \leftarrow v$), `v` (the signal represented by v) is connected to the *cond*1 input of $\mathcal{P}(v)$.
    - If the direction is ($\mathcal{P}(v) \rightarrow v$), *cond*1 is connected to '1', and *trig_l* constrains `v`.
- If v is the right child ($v = \mathrm{Rch}(\mathcal{P}(v))$), the same rules as for the left child apply, replacing:
  - *cond*1 by *cond*2
  - *trig_l* by *trig_r*

Here, we assumed that none of the left and right sub-sequences are unbounded repetition.

8.2.2.3 Unbounded SERE

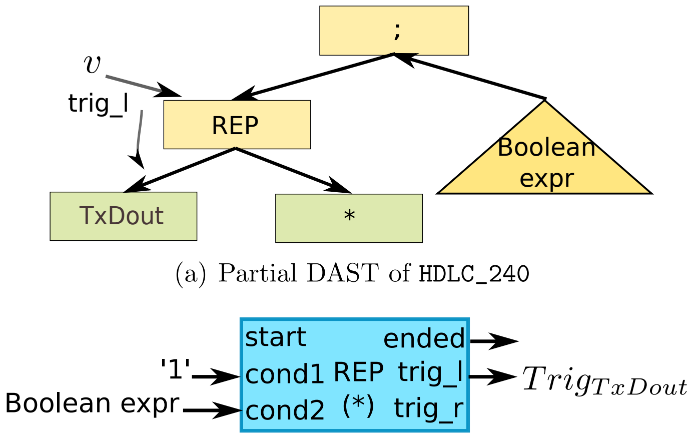
As was mentioned in Chapter 6, an unbounded repetition should be followed by a Boolean expression (see the right-most sub-tree in Fig. 8.4, whose root is ';'). Let v be node `REP`, the root of the unbounded repetition sub-tree.

- If $\mathrm{Lch}(v)$ is not a leaf:
  - The *trig_l* output of the '$*$' primitive reactant is connected to the *start* input of $\mathrm{Lch}(v)$.
  - The *ended* output of $\mathrm{Lch}(v)$ is connected to the *cond*1 input of the '$*$' primitive reactant.
- If $\mathrm{Lch}(v)$ is a leaf:
  - The *trig_l* output of the '$*$' primitive reactant constrains the signal of $\mathrm{Lch}(v)$.
  - The *cond*1 input of the '$*$' primitive reactant is connected to '1'.
- the Boolean expression associated to the sibling of v ($\mathrm{Rch}(\mathcal{P}(v))$) is connected to the *cond*2 input of the '$*$' primitive reactant.

Example 4. Unbounded SERE reactant construction.

Figure 8.6(a) shows the simplified sub-tree of the unbounded repetition in Fig. 8.4.

Node v is `REP`. Its sibling is a Boolean expression that is connected to *cond*2 of the '$*$' primitive reactant. The *trig_l* output of this reactant constrains *TxDout* (Fig. 8.6)(b).



(a) Partial DAST of `HDLC_240`

(b) Interconnection of the '$*$' primitive reactant (`HDLC_240`)

FIGURE 8.6: Unbounded SERE primitive reactant interconnection

8.3 Principles of the recursive construction

In this section, we explain the principles of the recursive construction using the concepts and formalism introduced in Chapters 5 and 6.

Informally, all the operands of φ stand for signals of the reactant, some of which are observed, others are generated (see Chapter 7). All reactants have a *reset* and a *clock* input signal. By default, we consider that all signal values are taken at the rising edge of clock. An input signal *start* is used to set the reactant active, and the activity may take one or more clock cycles.

During its activity, the reactant *observes* input signals and *constrains* the value of one or more signals. We recall that the output of a reactant is *not* the value of a signal, but the *trigger* that will *start* the primitive hardware component in charge of the signal value generation or observation.

For each operator, circuit $\mathcal{C}$ implements its primitive reactant.

A complex reactant is built by interconnecting the primitive reactants; it is done recursively according to the depth (number of nested FL or SERE operators) of property φ, denoted $\mid \varphi \mid$.

We now show how, for all $n \in \mathbb{N}$, for all properties φ_n of depth n, we construct a reactant circuit that implements φ_n.

$$\forall n \in \mathbb{N}, \forall \varphi_n,\ n = \mid \varphi_n \mid,\ \exists \mathcal{C}_n, \mathcal{C}_n \Vvdash \varphi_n$$

8.3.1 The base case

Let $n = 0$ be the depth of property φ_0. In this case, φ_0 is a Boolean expression, which is implemented using a Boolean primitive reactant introduced in Chapter 5. Here, we just bring again Fig. 8.7 as a reminder. The trigger output(s) of a primitive reactant constrains its operand(s).

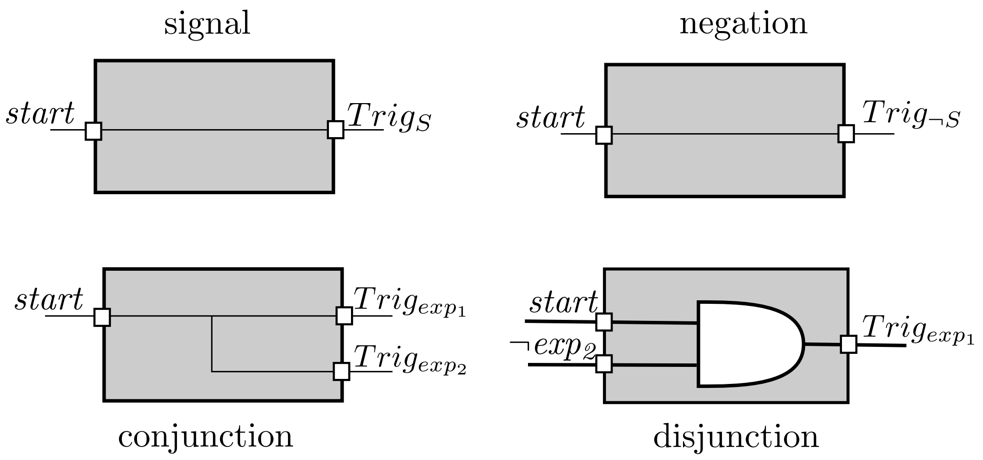


FIGURE 8.7: Base case: Boolean reactants

In the remaining of this chapter, we eliminate all the Boolean reactants from the figures for the sake of simplicity.

8.3.2 FL properties

Let $n > 0$ be the depth of property φ_n. According to the abstract syntax tree of φ_n, there exists a FL operator denoted Ω_n, a subproperty φ_{n-1} and a Boolean operand op_n such that $\varphi_n = \Omega_n(\varphi_{n-1}, op_n)$. The circuit $\mathcal{C}_n$ is the interconnection of the subcircuit $\mathcal{C}_{n-1}$ and a primitive reactant that implements Ω_n (Fig. 8.8).

Circuit $\mathcal{C}_n$ takes the synchronization signal, *start* and some observed signals. The *trig* output of Ω_n connects to the *start* input of circuit $\mathcal{C}_{n-1}$, if φ_{n-1} is not Boolean; otherwise, *trig* constrains φ_{n-1}.

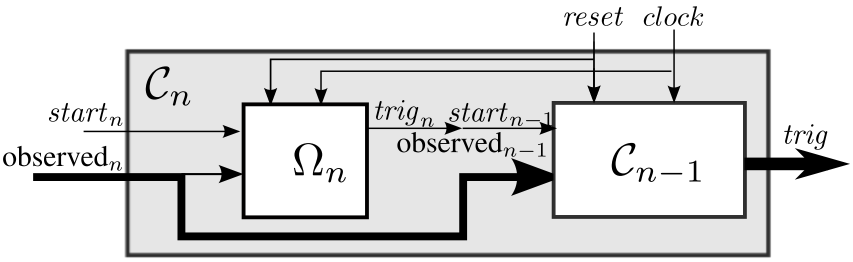


FIGURE 8.8: Recursive construction of circuit $\mathcal{C}_n$

Example 5. *Reactant for property* `P5_rec_0`.

The construction of the reactant for this property follows the principles just explained. Property `P5_rec_0` is a depth 3 property, with Ω_3 = `always`, Ω_2 = –>, and Ω_1 = `next!`. The complex reactant of this property is shown in Fig. 8.9. Signal *BtoR_REQ(0)* is constrained by the *trig* signal of the `next!` operator primitive reactant.

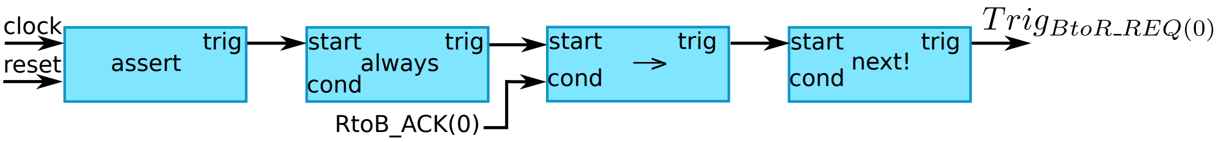


FIGURE 8.9: Implementation of Genbuf property `P5_rec_0`

8.3.3 SERE properties

Here, we show how a complex reactant for a SERE property is constructed recursively, based on the depth of the property, $\mid \varphi \mid$. The depth of a SERE property is defined as $n = \mid \varphi^L \mid + \mid \varphi^R \mid +1$, where $\mid \varphi^L \mid$ and $\mid \varphi^R \mid$ are the total number of nested SERE operators in φ^L and φ^R. As an example, see the DAST of property `HDLC_240`, shown in Fig. 8.4. Here, we show how for each category of SERE operators, $\mathcal{C}_n$ is constructed using the primitive reactant of the operators.

8.3.3.1 Simple SEREs

Let $n > 0$ be the depth of property φ_n, $n^L \geq 0$ and $n^R \geq 0$ the depths of its left and right children, $\varphi^L_{n^L}$ and $\varphi^R_{n^R}$.

We assume that the left sub-sequence is not an unbounded repetition. Then, the circuit $\mathcal{C}_n$ is the interconnection of the sub-circuit $\mathcal{C}^L_{n^L}$, $\mathcal{C}^R_{n^R}$ and a primitive reactant that implements $\Omega_n \in \{;, :\}$.

In the simple SEREs, the $\mathcal{C}_n$ and $\mathcal{C}^L_{n^L}$ sub-circuits start at the same time, after the activation of the start signal ($start^L{}_{n^L} = start_n$). Ω_n generates three signals:

1 The $\Omega_n.trig\_l$ signal is directly connected to $\Omega.cond1$. It constrains the left operand, if it is a Boolean.

2 The $\Omega_n.trig\_r$ signal starts the sub-circuit $\mathcal{C}^R_{n^R}$, if $\varphi^R_{n^R}$ is not Boolean; otherwise, it constrains $\varphi^R_{n^R}$.

3 The $\Omega_n.ended$ output indicates φ completes.

Figure. 8.10 illustrates the recursive construction.

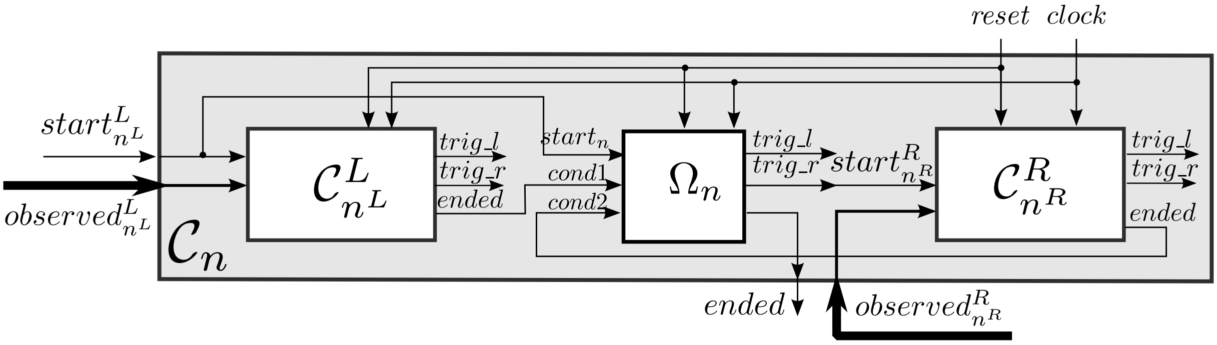


FIGURE 8.10: Recursive construction of circuit $\mathcal{C}_n$ ($\Omega_n \in \{;, :\}$)

8.3.3.2 Compound SEREs

Let $n > 0$ be the depth of property φ_n. According to the abstract syntax tree of φ_n, there exists a SERE operator denoted Ω_n, a left sub-sequence $\varphi^L_{n^L}$ and a right sub-sequence $\varphi^R_{n^R}$.

As the right and left sub-sequences start at the start time of $\mathcal{C}_n$ ($start^L_{n^L} = start^R_{n^R} = start_n$). Ω_n generates these signals:

1 The $\Omega_n.trig\_l$ signal indicates the start of the reactant of $\varphi^L_{n^L}$, if it is not Boolean. Otherwise, it constrains the left operand.

2 The $\Omega_n.trig\_r$ signal indicates the start of the reactant of $\varphi^R_{n^R}$, if it is not Boolean. Otherwise, it constrains the right operand.

3 The $\Omega_n.ended$ signal indicates if φ_n completes.

Figure. 8.11 illustrates the recursive construction.

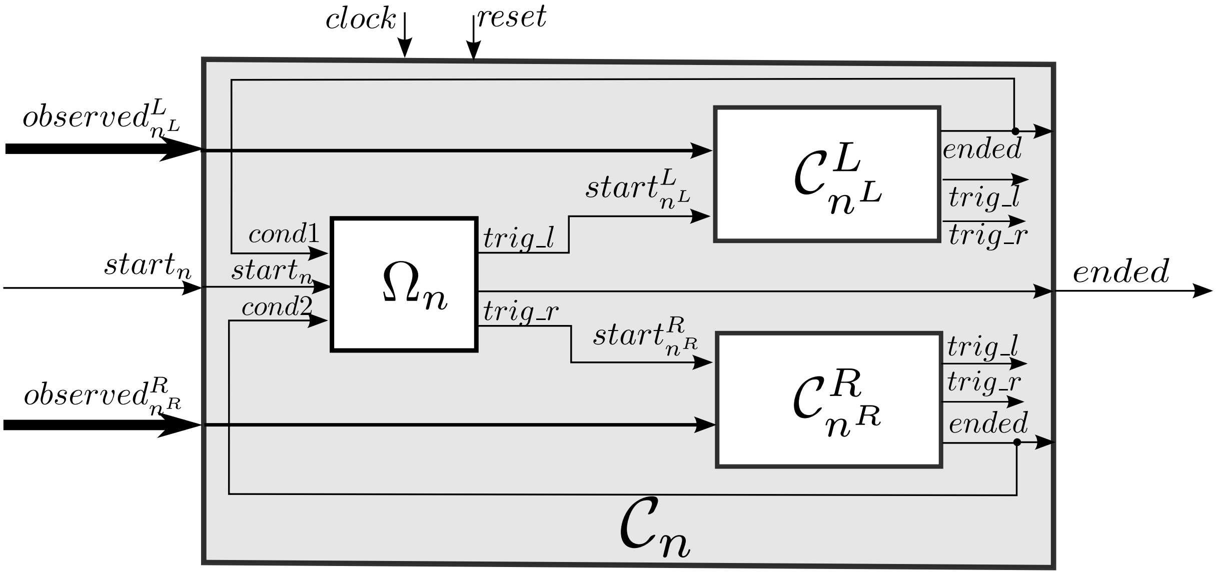


FIGURE 8.11: Recursive construction of circuit $\mathcal{C}_n$ ($\Omega_n \in \{\&, \&\&, |\}$)

8.3.3.3 Unbounded SEREs

Assume that $\varphi_n = \varphi_{n-1}[*]$. Figure. 8.12 illustrates the recursive construction of $\varphi_{n-1}[*]$. As was discussed before, an unbounded repetition is followed by a Boolean expression (op_0). The Boolean expression is connected to the *cond*2 input of the reactant of $\Omega_n = *$. The primitive reactant of Ω_n generates the following signals:

1 The $\Omega_n.trig\_l$ signal that is connected to $\mathcal{C}_{n-1}.start$, if φ_{n-1} is not Boolean. Otherwise, $\Omega_n.trig\_l$ is the output of the reactant and constrains φ_{n-1}.

2 The $\Omega_n.ended$ signal that indicates each time φ_{n-1} occurs.

Example 6. *Reactant for property* `HDLC_240` *(from HDLC transmitter).*

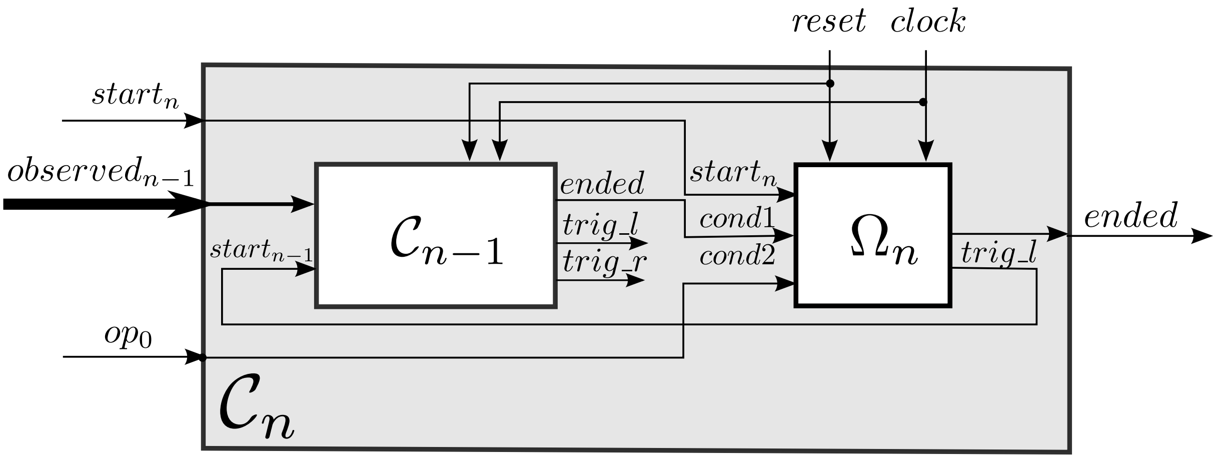


FIGURE 8.12: Recursive construction of circuit $\mathcal{C}_n$ ($\Omega_n \in \{*, +\}$)

The construction of the reactant for this property follows the principles just explained. We show the reactant of the right-hand side of the implication operator. Figure 8.13 shows the complex reactant of this sub-sequence.

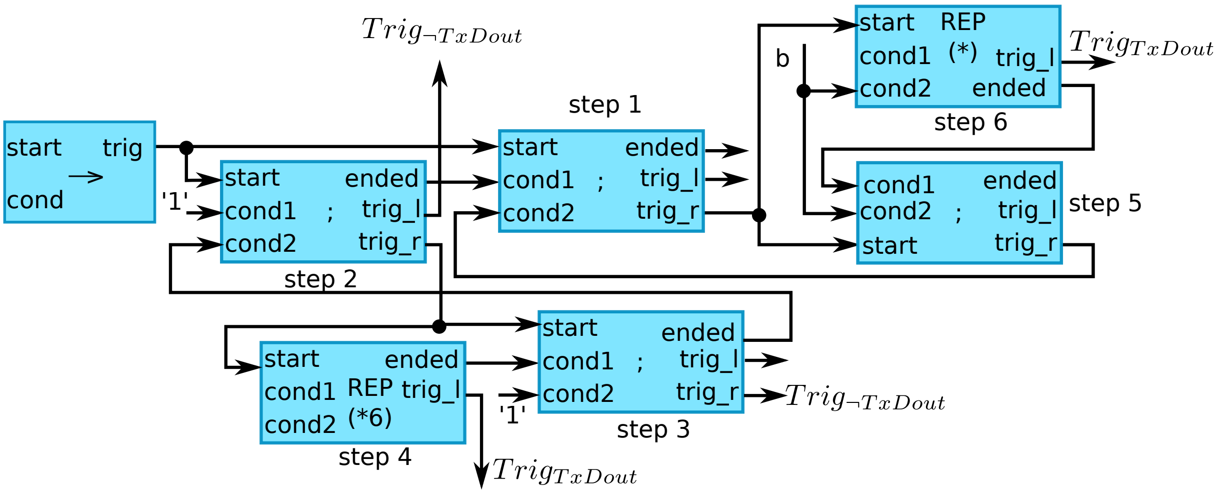


FIGURE 8.13: Implementation of HDLC property `HDLC_240`

The steps for constructing this complex reactant are shown in the following. In each step, the left and right sub-sequences are shown. In the first step, we instantiate the primitive reactant for ';'. Then we connect it to the primitive reactant of φ_3^L and φ_2^R. Therefore, we construct the reactant of these sub-sequences recursively.

$$
\begin{aligned}
\text{step 1 :} & \varphi_6 = \{\texttt{not}\, TxDout; \{TxDout[*6]; \texttt{not}\, TxDout\}\}; \{TxDout[*]; BoolExpr\} \\
& \Omega_6 =; \\
& \varphi_3^L = \texttt{not}\, TxDout; \{TxDout[*6]; \texttt{not}\, TxDout\} \\
& \varphi_2^R = TxDout[*]; BoolExpr \\
\text{step 2 :} & \varphi_3 = \texttt{not}\, TxDout; \{TxDout[*6]; \texttt{not}\, TxDout\} \\
& \Omega_3 =; \\
& \varphi_0^L = \texttt{not}\, TxDout \\
& \varphi_2^R = TxDout[*6]; \texttt{not}\, TxDout \\
\text{step 3 :} & \varphi_2 = TxDout[*6]; \texttt{not}\, TxDout \\
& \Omega_2 =; \\
& \varphi_1^L = TxDout[*6] \\
& \varphi_0^R = \texttt{not}\, TxDout \\
\text{step 4 :} & \varphi_1 = TxDout[*6] \\
& \Omega_1 = [*6] \\
& \varphi_0 = TxDout \\
\text{step 5 :} & \varphi_2 = TxDout[*]; BoolExpr \\
& \Omega_2 =; \\
& \varphi_1^L = TxDout[*] \\
& \varphi_0^R = BoolExpr \\
\text{step 6 :} & \Omega_1 = [*] \\
& \varphi_0 = TxDout
\end{aligned}
$$

As is shown in Fig. 8.13, signal *TxDout* is constrained several times in the property. A signal s is called *duplicated* if it is generated several times in a property, or is generated by several properties.

8.4 Summary

In this chapter we discussed the principles of the recursive construction of a reactant. A reactant is constructed for each property. The advantage of our construction method is having access to the trigger of each primitive reactant module. It increases the observability of the generated circuit, and makes it appropriate for debugging purposes. However, two issues still remain to be solved: *duplicated* signals and *unannotated* signals. In the next chapter, we solve these issues.

Chapter 9

Resolution of the signals

Contents

9.1 Introduction

In this chapter we explain how to resolve the value of the *duplicated* and *unannotated* signals. We express the dependency among all properties using a *Dependency Graph*. To this goal, we partition the Directed Abstract Syntax Trees (DASTs) of the properties into *directed* and *semi-directed*. Then, we consider directed DASTs to extract the dependencies for a duplicated signal, and analyze the semi-directed DASTs to extract the dependencies for unannotated signals.

We construct two kind of solvers: 1) *simple* solvers for resolving the duplicated signals, and 2) *complex* solvers for resolving the unannotated signals.

Additionally, from the dependency graph we extract the information for verifying if the set of properties are complete and consistent.

9.2 Constraints computed from directed DASTs

In a directed DAST all the edges are directed, and hence, all the signals are annotated. We start by an example.

Example 1. Directed DASTs **of GenBufRec.**

As was shown in Chapter 7, several properties of GenBufRec are fully annotated, and there are several properties that constrain a signal. For example, consider the three following properties:

```
P1_rec:
  always(EMPTY_m -> next!(not BtoR_REQ_g(0) and (not BtoR_REQ_g(1))) );

P3_rec_0:
  always( rose (BtoR_REQ_m(0)) -> next!(next_event!(prev(not BtoR_REQ_m
      (0))) (not BtoR_REQ_g(0) until_ (BtoR_REQ_m(1)))));

P4_rec_0:
  always((BtoR_REQ_m(0)) and (not RtoB_ACK_m(0))-> next!(BtoR_REQ_g(0)));
```

The DASTs of these properties are shown in Fig. 9.1, Fig. 9.2, and Fig. 9.3.

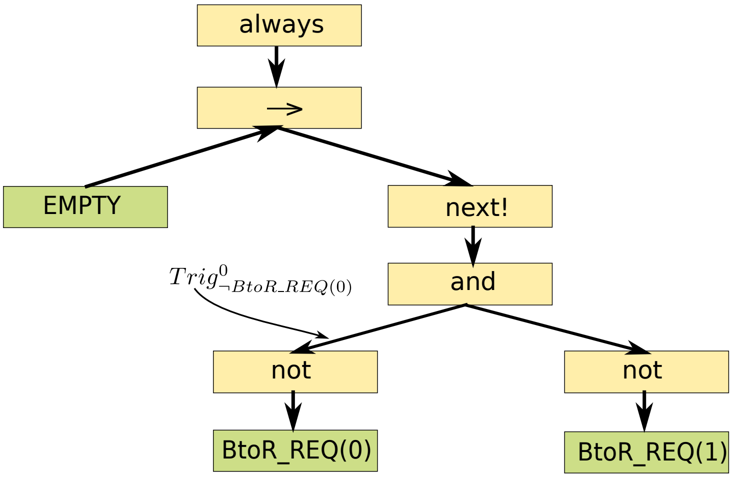


FIGURE 9.1: DAST of `P1_rec`

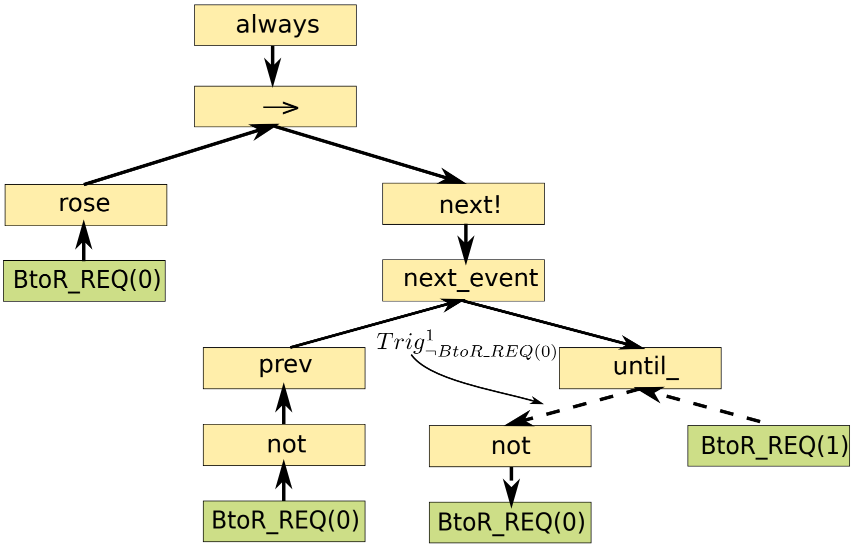


FIGURE 9.2: DAST of `P3_rec`

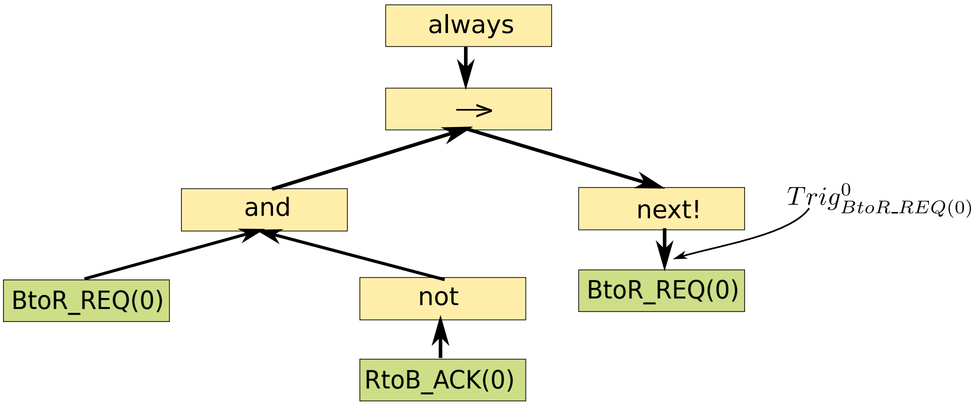


FIGURE 9.3: DAST of `P4_rec`

All these DASTs are fully directed, and all the signals are annotated. As is shown in these DASTs, *BtoR_REQ(0)* is constrained to 0 by properties `P1_rec` and `P3_rec_0`, while it is constrained to 1 by `P4_rec_0`.

Considering all the properties of GenBufRec (see Chapter 7, Fig. 7.24), the set of the directed DASTs is:

DIRECTED = { `P1_rec`, `P3_rec_0`, `P3_rec_1`, `P4_rec_0`, `P4_rec_1`, `P5_rec_0`, `P5_rec_1`, `P6_FIFO_rec`, `P7_FIFO_rec` }

As was discussed in Chapter 8, for each DAST of *DIRECTED*, the reactant is built, and the trigger output, *trig*, constrains a signal to 0 or 1. For each signal z, there may be several trigger signals that constrain z to 0 ($\mathit{Trig}_{\neg z}$) or to 1 (Trig_z). We should find all these trigger signals. Let $\mathit{Trig}^i_{\neg z}, 0 \leq\ i \leq\ nb_0 - 1$ be the nb_0 triggers of the reactants that constrain signal z to 0, and $\mathit{Trig}^j_z, 0 \leq\ j \leq\ nb_1 - 1$ be the nb_1 triggers that constrain z to 1. Two vectors $\mathcal{T}\mathit{0}_z$ and $\mathcal{T}\mathit{1}_z$ are defined as:

$$\mathcal{T}\mathit{0}_z = (\mathit{Trig}^0_{\neg z}, \mathit{Trig}^1_{\neg z}, \ldots, \mathit{Trig}^{nb_0-1}_{\neg z})$$
$$\mathcal{T}\mathit{1}_z = (\mathit{Trig}^0_z, \mathit{Trig}^1_z, \ldots, \mathit{Trig}^{nb_1-1}_z)$$

For each signal z, signals $\mathit{T0}_z$ and $\mathit{T1}_z$ are defined as:

$$\mathit{T0}_z = \bigvee_i \mathit{Trig}^i_{\neg z}, \text{ and } \quad \mathit{T1}_z = \bigvee_j \mathit{Trig}^j_z$$

Back to Example 1, for signal *BtoR_REQ(0)* we have:

$$\mathcal{T}\mathit{0}_{BtoR\_REQ(0)} = (\mathit{Trig}^0_{\neg BtoR\_REQ(0)}, \mathit{Trig}^1_{\neg BtoR\_REQ(0)})$$
$$\mathcal{T}\mathit{1}_{BtoR\_REQ(0)} = (\mathit{Trig}^0_{BtoR\_REQ(0)})$$

where, $\mathit{Trig}^0_{\neg BtoR\_REQ(0)}$ corresponds to property `P1_rec`, $\mathit{Trig}^1_{\neg BtoR\_REQ(0)}$ corresponds to property `P3_rec_0`, and $\mathit{Trig}^0_{BtoR\_REQ(0)}$ corresponds to property `P4_rec_0` (see figures 9.1, 9.2, and 9.3). Consequently,

$$\mathit{T0}_{BtoR\_REQ(0)} = \mathit{Trig}^0_{\neg BtoR\_REQ(0)} \vee \mathit{Trig}^1_{\neg BtoR\_REQ(0)}$$
$$\mathit{T1}_{BtoR\_REQ(0)} = \mathit{Trig}^0_{BtoR\_REQ(0)}$$

9.3 Constraints computed from semi-directed DASTs

As was shown in Chapter 7 several signals of properties remain unannotated. We start by an example.

Example 2. Semi-directed DASTs **of GenBufRec.**

Consider the two following properties of GenBufRec:

```
P0_rec:
  always(not EMPTY_m -> next!(BtoR_REQ(0) or ( BtoR_REQ(1))));

P2_rec:
  always(not BtoR_REQ(0) or not BtoR_REQ(1));
```

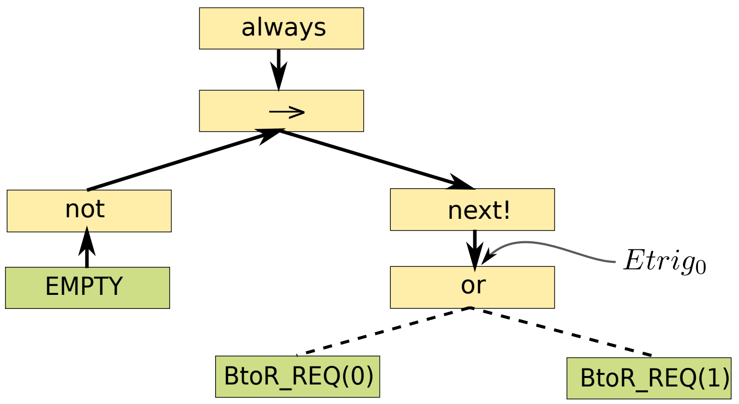


FIGURE 9.4: DAST of `P0_rec`

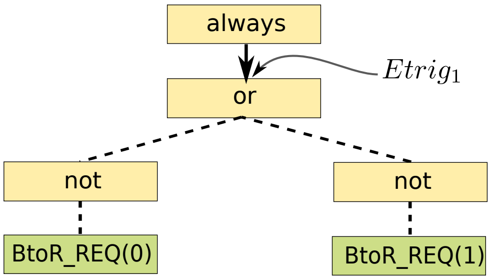


FIGURE 9.5: DAST of `P2_rec`

Figures 9.4 and 9.5 show the DASTs of properties `P0_rec` and `P2_rec`.

Both DASTs are semi-directed, since they have some unannotated signals: *BtoR_REQ(0)* and *BtoR_REQ(1)*. This is due to the fact that several dependency rules apply for operator `or`. Intuitively, property `P0_rec` states that whenever the *EMPTY* signal is 0, there should be a request to a receiver, but is does not say which receiver.

Considering these properties in isolation cannot suffice to decide which signal among *BtoR_REQ(0)* and *BtoR_REQ(1)* must be generated. All the properties that affect these signals must be considered together.

In the case of GenBufRec, the set of semi-directed DASTs is:
SEMIDIRECTED = { `P0_rec`, `P2_rec` }

For each DAST of *SEMIDIRECTED*, the semi-directed sub-trees are pruned away, and the reactant is built for the directed sub-tree with the method explained in Chapter 8. Let $Etrig_j$ be the output signal of such reactant, and $Expr_j$ the Boolean expression for the pruned sub-tree. The expressions $\mathcal{E} = (Expr_0, \ldots, Expr_m)$ are triggered by $Etrig_0, \ldots, Etrig_m$ (see Fig. 9.4 and Fig. 9.5).

9.4 Dependency Graph ($\mathcal{DG}$)

The Dependency Graph $\mathcal{DG}$ of a set of properties ($\mathtt{P}_0$, …, $\mathtt{P}_{k-1}$) is a semi-directed and labeled graph. We denote $\mathcal{DG} = (V, E)$, where:

- $V = V1 \cup V2$ is the set of nodes:
  - $V1 = L_0 \cup \cdots \cup L_{k-1}$, where $L_0, \ldots, L_{k-1}$ are the set of leaves of $\text{DAST}_0, \ldots, \text{DAST}_{k-1}$

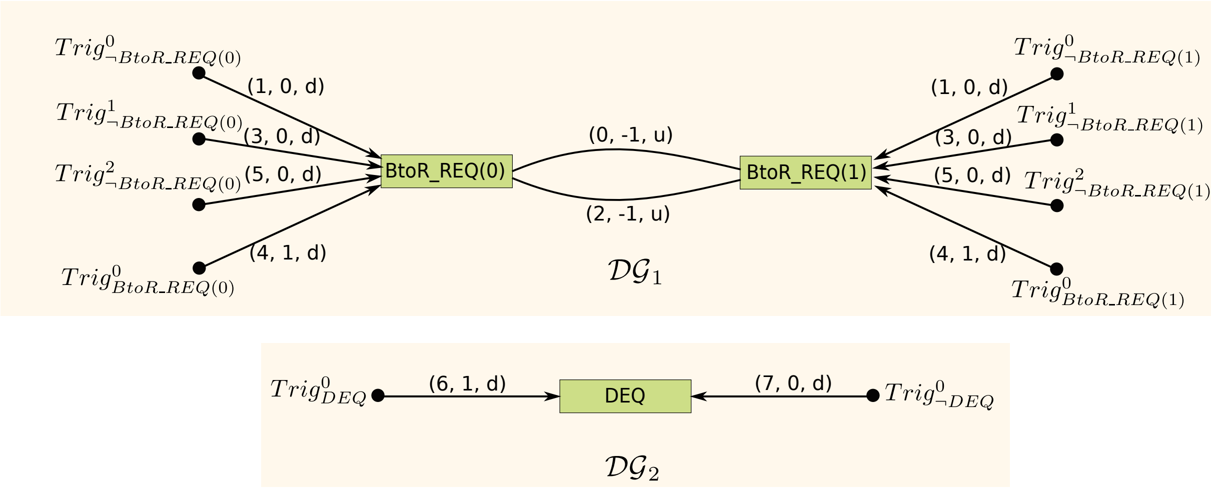
  - $V2$ is the set of all the *trig* outputs of all the properties.

- $E = E1 \cup E2$, where $E1$ is the set of the directed edges, and $E2$ is the set of undirected edges, and:
  - $E1 \subset V2 \times V1$, e.g. $e = (\mathit{Trig}_l \rightarrow l)$
  - $E2 \subset V1 \times V1$, e.g. $e = (l_1 – l_2)$

- Each edge e of the graph has a label $w = (\mathit{id},\ \mathit{val},\ \mathit{type})$, where:
  - *id*: identifies the property that creates edge e; therefore, $0 \leq \mathit{id} \leq k-1$
  - *val*: if e is directed, *val* specifies the value of the destination node, if the value of the source node is 1. If e is undirected, *val* is set to -1. Therefore, $\mathit{val} \in \{0, 1, -1\}$.
  - *type*: specifies if the edge is directed; therefore, $\mathit{type} \in \{d, u\}$, where '$d$' means directed, and '$u$' means undirected.

The dependency graph may have several strongly connected components, each one specifies a set of interdependent generated signals $\mathcal{Z} = \{z_1, \ldots z_n\}$ (see Fig. 9.6).

Example 3. *Dependency graph of GenBufRec*



$\mathit{Trig}^0_{\neg BtoR\_REQ(0)}$
$\mathit{Trig}^1_{\neg BtoR\_REQ(0)}$
$\mathit{Trig}^2_{\neg BtoR\_REQ(0)}$
$\mathit{Trig}^0_{BtoR\_REQ(0)}$
(1, 0, d)
(3, 0, d)
(5, 0, d)
(4, 1, d)
BtoR_REQ(0)
(0, -1, u)
(2, -1, u)
BtoR_REQ(1)
$\mathcal{DG}_1$
$\mathit{Trig}^0_{\neg BtoR\_REQ(1)}$
$\mathit{Trig}^1_{\neg BtoR\_REQ(1)}$
$\mathit{Trig}^2_{\neg BtoR\_REQ(1)}$
$\mathit{Trig}^0_{BtoR\_REQ(1)}$
(1, 0, d)
(3, 0, d)
(5, 0, d)
(4, 1, d)
Trig^0_{DEQ}
(6, 1, d)
DEQ
(7, 0, d)
$\mathit{Trig}^0_{\neg DEQ}$
$\mathcal{DG}_2$

FIGURE 9.6: Dependency graph of GenBufRec

The dependency graph of GenBufRec has 2 strongly connected components. $\mathcal{DG}_1$ consists in nodes *BtoR_REQ(0)* and *BtoR_REQ(1)* and all their corresponding related triggers; $\mathcal{Z}_1 = \{$*BtoR_REQ(0)*, *BtoR_REQ(1)*$\}$. Consider edge (*BtoR_REQ(0)* – *BtoR_REQ(1)*), with label $w = (0, -1, u)$. The label means that the edge is created due to DAST$_0$, `P0_rec`; and the edge is undirected. It means that *BtoR_REQ(0)* and *BtoR_REQ(1)* depend on each other.

Now consider node $v_1 =$ *BtoR_REQ(0)*, and $v_2 = \mathit{Trig}^0_{\neg BtoR\_REQ(0)}$. There is a directed edge $e = (v2 \rightarrow v1)$. The label of this edge is $w = (1, 0, d)$. The first element of the label means that this edge is created due to DAST$_1$, `P1_rec`. The second element means that if the value of $Trig^0_{\neg BtoR\_REQ(0)}$ is 1, then *BtoR_REQ(0)* is constrained to 0. The third element means that this edge is directed.

Component $\mathcal{DG}_2$ consists in signal *DEQ* and its two triggers; $\mathcal{Z}_2 = \{DEQ\}$. Since this component has only one generated signal, the value of *DEQ* is independent from other generated signals, it only depends on the value of its triggers ($Trig^0_{DEQ}$ and $Trig^0_{\neg DEQ}$).

9.5 Dependency Graph construction

The dependency graph is constructed in two steps from the DASTs of the properties.

1 For each directed DAST, DAST$_k$:

- 1-1 For each generated leaf l of DAST$_k$ (i.e. $(\mathcal{P}(l) \rightarrow l)$):
  - 1-1-1 Add l to the nodes of $\mathcal{DG}$ (if it is not in V)
  - 1-1-2 Add the corresponding trigger ($Trig^i_l$ or $Trig^j_{\neg l}$) to V
  - 1-1-3 Create an edge e from the trigger node to the signal node, $e = (Trig^i_l \rightarrow l)$
  - 1-1-4 If the corresponding signal of l is constrained to 0: create the label $w = (k, 0, d)$, otherwise $w = (k, 1, d)$

2 For each semi-directed DAST, DAST$_k$:

- 2-1 Prune away the fully directed sub-tree, and keep the undirected sub-tree
- 2-2 Add all the leaves of the undirected sub-tree into V (if they are not in V)
- 2-3 For each pair of nodes l_1 and l_2 coming from DAST$_k$:
  - 2-3-1 Add an edge between l_1 and l_2
  - 2-3-2 Create the label $w = (k, -1, u)$

Example 4. *Dependency graph construction for GenBufRec*

Using the principles explained above, we show how to construct the dependency graph of GenBufRec (see Fig. 9.6). Here, the result of each construction step is shown for the DAST of properties `P0_rec` and `P1_rec`:

1 For DAST$_1$ (corresponds to `P1_rec`):

- 1-1 For leaf *BtoR_REQ(0)*:
  - 1-1-1 *BtoR_REQ(0)* is added to the nodes of $\mathcal{DG}$
  - 1-1-2 $Trig^0_{\neg BtoR\_REQ(0)}$ is added to V
  - 1-1-3 $e = (Trig^0_{\neg BtoR\_REQ(0)} \rightarrow BtoR\_REQ(0))$
  - 1-1-4 $w = (1, 0, d)$
- 1-2 The above steps are repeated for leaf *BtoR_REQ(1)*

2 For DAST$_0$ (corresponds to `P0_rec`):

- 2-1 keep the undirected sub-tree whose root is `or`
- 2-2 *BtoR_REQ(0)* and *BtoR_REQ(1)* already exist in V
- 2-3 For *BtoR_REQ(0)* and *BtoR_REQ(1)*
  - 2-3-1 $e = (BtoR\_REQ(0) - BtoR\_REQ(1))$ is added to E
  - 2-3-2 $w = (0, -1, u)$

9.6 The resolution function: *solver*

Now we discuss how to use the dependency graph $\mathcal{DG}$ for generating the solvers. We construct two types of solvers: *simple* and *complex*. The first one specifies the value of the *duplicated* signals, while the second one specifies the value of the *unannotated* signals.

To extract the list of the duplicated and unannotated signals, we consider each strongly connected sub-graph $\mathcal{DG}_i$ $(0 \leq i \leq k)$ of $\mathcal{DG}$. For each $\mathcal{DG}_i$:

1 If it contains only one signal, z, the signal is annotated, and it is constrained only by its triggers. If there is more than one trigger, the signal is duplicated. We need a *simple* solver to specify its value based on its triggers.

2 If $\mathcal{DG}_i$ contains a set of signals $\mathcal{Z} = \{z_1, \ldots z_n\}$, it means that the signals are not annotated in all the properties, and their values depend not only on their triggers, but also on the value of other signals in $\mathcal{Z}$. In this case, we need a *complex* solver to identify the signals' values. Here, in addition to $\mathcal{T}1_{z_j}$ and $\mathcal{T}0_{z_j}$ for each signal z_j, we need to find $\mathcal{T}_{\mathcal{Z}} = (Etrig_0, \ldots, Etrig_{m-1})$, where m is the number of the expressions that relate the signals of $\mathcal{Z}$.

9.6.1 Resolving duplicated signals: *simple* solver

In $\mathcal{DG}_i$ we consider each edge $e = (v \rightarrow z)$. If label $w = (i, 0, d)$ we add the trigger that is represented by node v to $\mathcal{T}0_z$; if label $w = (i, 1, d)$ we add the trigger to $\mathcal{T}1_z$. After finding $\mathcal{T}1_z$ and $\mathcal{T}0_z$, the value of z should be calculated.

Signals $T0_z$ and $T1_z$ are the inputs of the solver component. The output will be the final value of signal z (see Fig. 9.7).

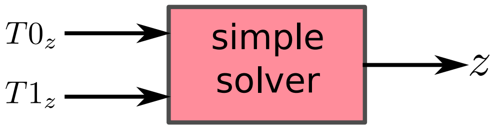


FIGURE 9.7: The interface of simple solver for duplicated signals

In our implementation, if none of $T0_z$ and $T1_z$ is active, the user has the choice to select if signal z keeps its previous value or takes a *default* value. The solver function is one of:

$$z = '0' \text{ when } T0_z =' 1' \text{ else} \quad '1' \text{ when } T1_z =' 1';$$

$$z = '0' \text{ when } T0_z =' 1' \text{ else} \quad '1' \text{ when } T1_z =' 1' \text{ else} \quad default\_value;$$

Example 5. Simple solver for GenBufRec.

For GenBufRec we have $\mathcal{Z}_2 = \{DEQ\}$ (see Example 3). Considering $\mathcal{DG}_2$ of the dependency graph, it is obvious that there are two edges whose destination node is DEQ. Considering the label of each edge, we add $Trig^0_{DEQ}$ to $\mathcal{T}1_{DEQ}$ and $Trig^0_{\neg DEQ}$ to $\mathcal{T}0_{DEQ}$.

Then, a simple solver computes the value of DEQ:

$$\begin{aligned} DEQ = {}&'0' \text{ when } T0_{DEQ} =' 1' \text{ else} \\ &'1' \text{ when } T1_{DEQ} =' 1' \text{ else} \\ &'0'; \end{aligned}$$

Here, the default value is 0.

9.6.2 Resolving unannotated signals: *complex* solver

Assume that $\mathcal{Z} = (z_1, \ldots, z_n)$, then, all the signals z_i of this list are unannotated in at least one property. These signals are interdependent through the list of expressions $\mathcal{E} = (Expr_0, \ldots, Expr_{m-1})$, which are triggered by $\mathcal{T}_\mathcal{Z} = (Etrig_0, \ldots, Etrig_{m-1})$. Generally, we can say that signals z_i, ..., z_n are the operands of the expressions $Expr_0, \ldots, Expr_{m-1}$ triggered by $Etrig_0, \ldots, Etrig_{m-1}$.

At the first step, for each signal z_i we find the list $\mathcal{T}1_{z_i}$ and $\mathcal{T}0_{z_i}$ using the principles explained in Section 9.6.1.

Then, we should find $\mathcal{T}_\mathcal{Z}$. We consider each edge e of $\mathcal{DG}$. If e is undirected, we add the corresponding trigger signal, $Etrig_j$, to $\mathcal{T}_\mathcal{Z}$, and add its corresponding expression $Expr_j$ to $\mathcal{E}$.

9.6.2.1 Complex solver implementation

A complex solver takes $(Etrig_0, \ldots, Etrig_{m-1})$, $(T1_{z_1}, \ldots, T1_{z_n})$, and $(T0_{z_1}, \ldots, T0_{z_n})$ as inputs, and it outputs the values of $(z_1, \ldots, z_n)$.

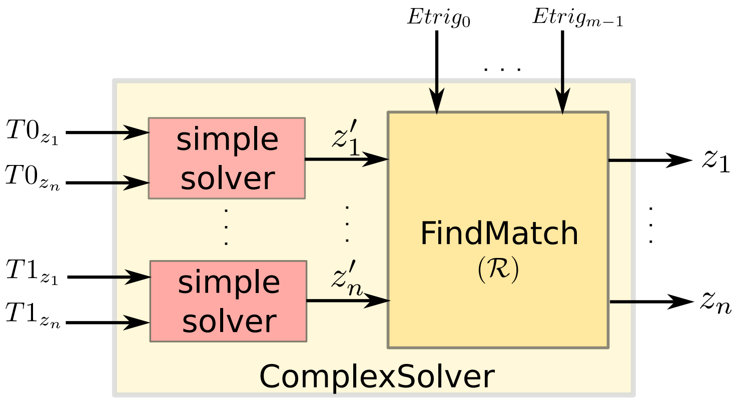


FIGURE 9.8: The interface of complex solver for unannotated signals

The problem is to solve the following set of equations, i.e. the values of $(z_1, \ldots, z_n)$, by considering $(T1_{z_1}, \ldots, T1_{z_n})$ and $(T0_{z_1}, \ldots, T0_{z_n})$.

$$\begin{cases} \vdots \\ Etrig_j \rightarrow Expr_j(z_1, \ldots, z_n) = 1 \\ \vdots \end{cases}$$

The brute force idea is to construct a Look Up Table (LUT) for the `FindMatch` submodule (Fig. 9.8) by enumerating all the values of $\mathcal{Z}$ for each value t of $T_\mathcal{Z}$ (initially, the

LUT has $m+n$ columns, and at most 2^{m+n} rows). Then, we select an appropriate row of this LUT. However, most of these combinations are impossible due to the properties and some valuation of $T_{\mathcal{Z}}$ may never happen.

We consider the 2^m values of vector $T_{\mathcal{Z}} = (\mathit{Etrig}_0, \ldots, \mathit{Etrig}_{m-1})$. Each value t of $T_{\mathcal{Z}}$ corresponds to the set of triggers that are simultaneously active. We associate to this set of active triggers the global Boolean expression that is the "and" of the Expr_j corresponding to $\mathit{Etrig}_j = 1$.

Mathematically, we define a function $\mathcal{F}$ that associates to each set of active triggers its Boolean expression:

$$\begin{array}{rrcc} \mathcal{F}: & 2^m & \rightarrow & \mathit{ExprBool}_n \\ & t & \mapsto & \displaystyle\bigwedge_{\substack{j=1 \\ \mathit{Etrig}_j=1 \ in \ t}}^{m} \mathit{Expr}_j \end{array}$$

$\mathit{ExprBool}_n$ is the set of Boolean functions of n variables.

$\mathcal{F}(t)$ is an expression of $z_1, \ldots, z_n$. Any assignment of $z_1, \ldots, z_n$ satisfying $\mathcal{F}(t)$ (i.e. verifying that $\mathcal{F}(t) = 1$) is a combination of the signal values that is compatible with the value t of $T_{\mathcal{Z}}$.

We denote $\mathcal{S}(t) = \{Z = (z_1, \ldots, z_n) | \mathcal{F}(t)(Z) = 1\}$, and we add these valuations of $T_{\mathcal{Z}}$ and $\mathcal{Z}$ to the LUT (a $m+n$ bit vector, the first m-bits represent value t of $T_{\mathcal{Z}}$, and the next n-bits represent the valuation of $Z \in \mathcal{S}(t)$).

After constructing the LUT, we should find an appropriate row of the LUT based on the value of $T_{\mathcal{Z}}$, $(\mathit{T1}_{z_1}, \ldots, \mathit{T1}_{z_n})$, and $(\mathit{T0}_{z_1}, \ldots, \mathit{T0}_{z_n})$. This is done in two steps:

1 First, we use n instances of the simple solver. For each signal z_i, if either $\mathit{T1}_{z_i} = 1$ or $\mathit{T0}_{z_i} = 1$, the value of z_i is obtained from its triggers: $z_i = \mathit{T1}_{z_i} \vee \neg \mathit{T0}_{z_i}$. Therefore, we first fix the value of such signals (see Fig. 9.8). The value of the other signals is *don't care*. The output of this step is thus the vector as $\mathcal{Z}' = (z'_1, \ldots, z'_n)$, where some signal values have been fixed and the others are don't care and should be obtained from the LUT based on the signals that have fixed values and $T_{\mathcal{Z}}$.

2 We define a compatibility relation $\mathcal{R}$ between the values of $T_{\mathcal{Z}}$ and the values of $\mathcal{Z}'$: $t \ \mathcal{R} \ z'$ means that the particular combination of signal values z' may hold when the triggers have value t.

$$\begin{array}{rrcc} \mathcal{R}: & 2^m \times 2^n & \rightarrow & \mathit{Bool} \\ & t \ \mathcal{R} \ z' & \Leftrightarrow & \mathcal{F}(t)(z') = 1 \\ & & \Leftrightarrow & z' \ \in \mathcal{S}(t) \end{array}$$

In Fig. 9.8, the `FindMatch` circuit implements relation $\mathcal{R}$, and it returns the value of $\mathcal{Z}$, if relation $\mathcal{R}$ holds. This circuit first looks for the rows of the LUT that correspond to value t of $T_{\mathcal{Z}}$. There may be several rows that match the value t. These rows give various valuation of z' ($\mathcal{S}(t)$ has more than one member).

Among these rows, we select a row that matches with the signals that have fixed values, and obtained in step 1. If there is just one such row, the value of the other signals is obtained from this row. Otherwise, we have to choose one row (currently, we take the first one).

Example 6. Complex solver of GenBufRec.

For GenBufRec, $\mathcal{Z}_1$ =(*BtoR_REQ(0)*, *BtoR_REQ(1)*). To construct the LUT, we need $T_{\mathcal{Z}_1} = (\mathit{Etrig}_0, \mathit{Etrig}_1)$ (see Fig. 9.4 and Fig. 9.5). First, we add all the possible valuation of $T_{\mathcal{Z}1}$ into the LUT, and for each valuation enumerate all the values of $\mathcal{Z}_1$.

To this goal, we consider expressions Expr_0 and Expr_1:

$$\mathit{Expr}_0 = \mathit{BtoR\_REQ}(0)\ \texttt{or}\ \mathit{BtoR\_REQ}(1)$$
$$\mathit{Expr}_1 = \texttt{not}\ \mathit{BtoR\_REQ}(0)\ \texttt{or}\ \texttt{not}\ \mathit{BtoR\_REQ}(1)$$

For value “11” of $T_{\mathcal{Z}}$, where Etrig_0 and Etrig_1 are simultaneously active, we have:

$$\mathit{ExprBool}_2 = \mathit{Expr}_0 \wedge \mathit{Expr}_1$$

Based on this expression, if $T_{\mathcal{Z}}$ =“11”, then $\mathcal{S}(11) = \{01, 10\}$. Therefore, we add lines “1101” and “1110” to the LUT. Similarly, if $T_{\mathcal{Z}}$ = “10”, then $\mathcal{S}(10) = \{01, 10\}$, and we add “1001” and “1010” to the LUT. Actually, this combination of Etrig_0 and Etrig_1 never happens, since Etrig_1 corresponds to the child of the `always` operator and it is always 1 (Fig. 9.5). To eliminate this row, we need to do model checking, to identify which combinations of the *Etrig* signals never occur. If $T_{\mathcal{Z}}$ = “01”, $\mathcal{S}(01) = \{00, 01, 10\}$, and “0100”, “0101”, and “0110” are added to the LUT. The LUT is shown in Fig. 9.9.

Etrig_0	Etrig_1	*BtoR_REQ(0)*	*BtoR_REQ(0)*
1	1	0	1
1	1	1	0
1	0	0	1
1	0	1	0
0	1	0	0
0	1	0	1
0	1	1	0

FIGURE 9.9: The LUT of the complex solver of GenBufRec

Now assume that $T_{\mathcal{Z}}$ = “01”, $\mathit{T1}_{\mathit{BtoR\_REQ}(0)} = 1$, $\mathit{T0}_{\mathit{BtoR\_REQ}(0)} = 0$, $\mathit{T1}_{\mathit{BtoR\_REQ}(1)} = 0$, and $\mathit{T0}_{\mathit{BtoR\_REQ}(1)} = 0$. First, we consider the trigger signals. For signal *BtoR_REQ(0)* we have $\mathit{T1}_{\mathit{BtoR\_REQ}(0)} \vee \mathit{T0}_{\mathit{BtoR\_REQ}(0)} = 1$, therefore, the value of *BtoR_REQ(0)* is obtained from its trigger signals and it is 1. For signal *BtoR_REQ(1)* both trigger signals are 0. Therefore, the value of this signal should be obtained from the LUT. $T_{\mathcal{Z}}$ = “01”, therefore, the last three lines of the LUT are selected. The value of *BtoR_REQ(0)* has been already fixed to 1. Consequently, we should select the last line, and the value of *BtoR_REQ(1)* becomes 0.

9.7 The final circuit

The final circuit is the interconnection of the property reactants (see Chapter 8) and solvers.

Example 7. The final circuit of GenBufRec.

Figure 9.10 shows the interconnection of all the property reactants with the solver components for GenBufRec.

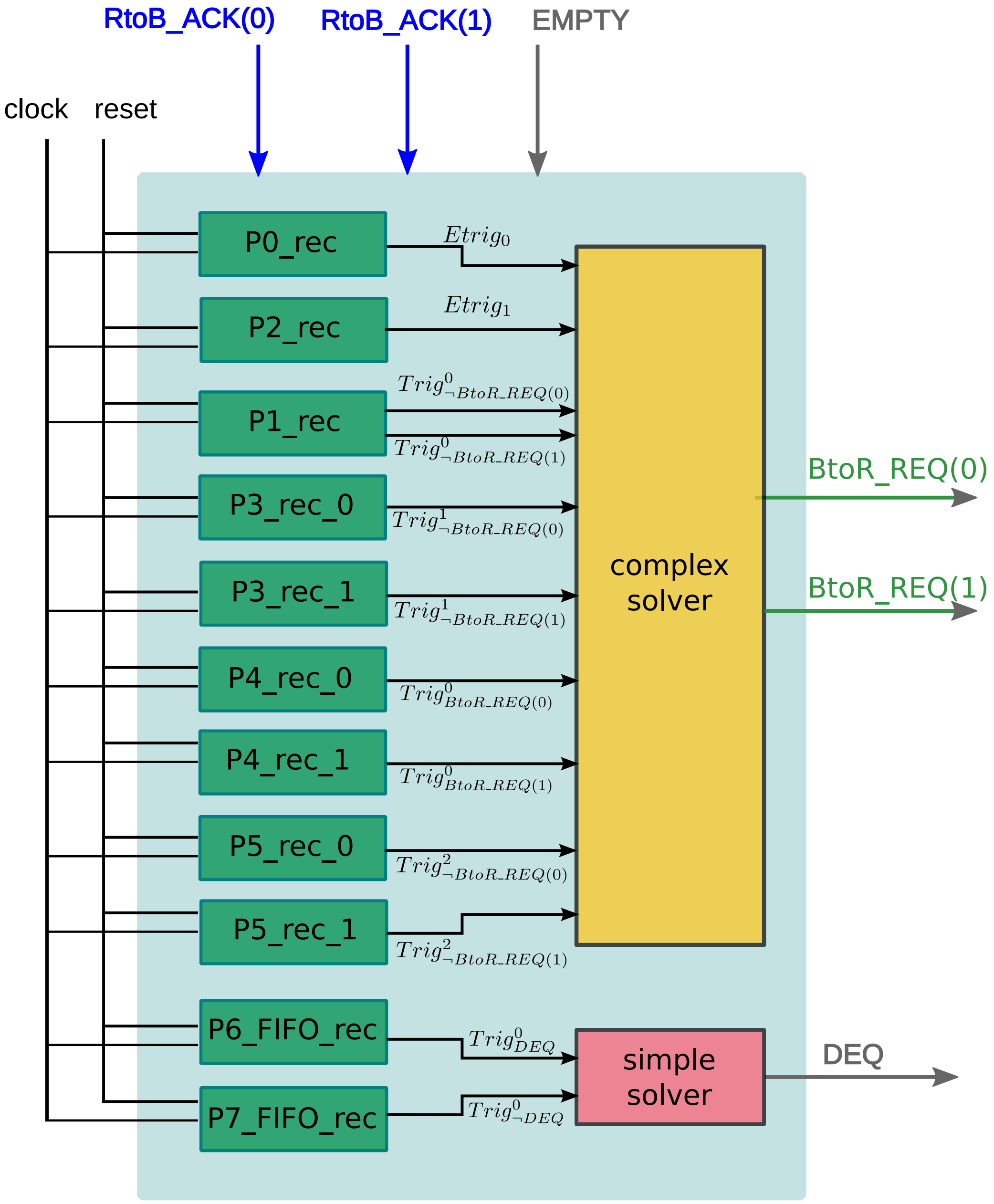


FIGURE 9.10: The final circuit of GenBufRec

9.7.1 Checking the consistency

The definition of z is consistent iff in all the cycles:

$$T1_z \wedge T0_z = 0$$

For each signal we obtained $T1_z$ and $T0_z$ from the dependency graph. Therefore, we can generate complementary properties automatically for verifying the above condition. These properties, together with the generated circuit are the inputs of a formal verification tool to prove the correctness of the generated circuit (see Chapter 4, Fig. 4.1).

Example 8. Consistency checking for the *BtoR_REQ(0)* signal from GenBufRec

Considering all the properties of GenBufRec, for signal *BtoR_REQ(0)* we have:

$$T0_{BtoR\_REQ(0)} = Trig^0_{\neg BtoR\_REQ(0)} \vee Trig^1_{\neg BtoR\_REQ(0)} \vee Trig^2_{\neg BtoR\_REQ(0)}$$
$$T1_{BtoR\_REQ(0)} = Trig^0_{BtoR\_REQ(0)}$$

The corresponding timing diagram is shown in Fig. 9.11.

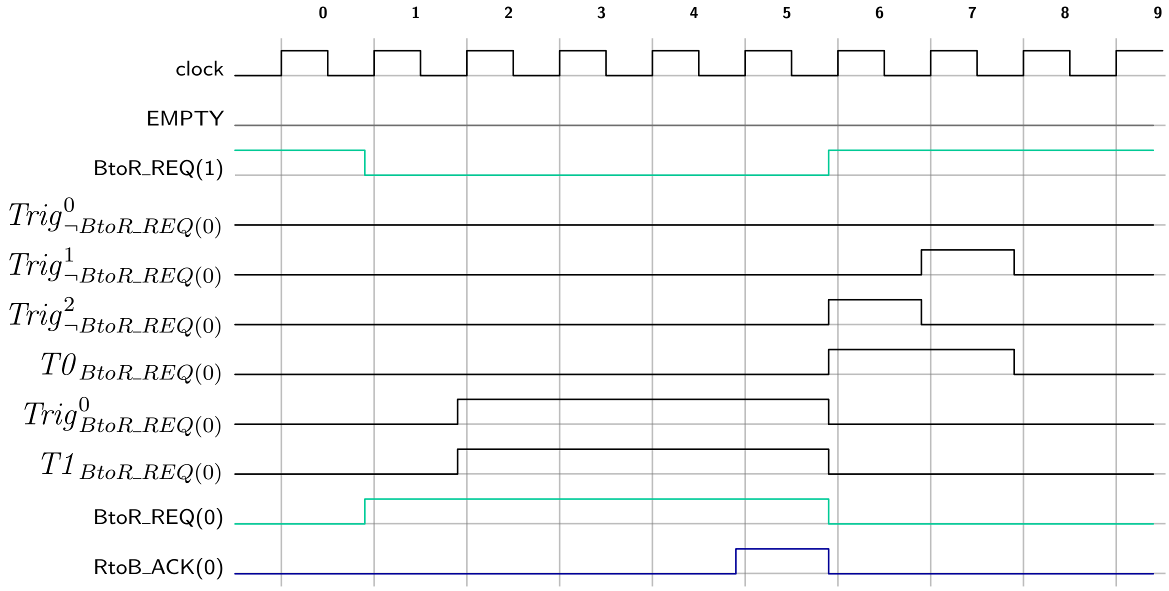


FIGURE 9.11: Timing diagram of *BtoR_REQ(0)* and corresponding trigger signals

The PSL property that verifies if the set of properties constraining *BtoR_REQ(0)* are consistent is:

```
always (not T1_BtoR_REQ_0 or not T0_BtoR_REQ_0 );
```

Now, assume that we change `P5_rec_0` as follows:

```
P5_rec_0_false:
 assert (always ((not RtoB_ACK_m(0))-> next! (not BtoR_REQ_g(0))));
```

The new timing diagram is shown in Fig. 9.12. As is shown, the property fails in cycles ♯2 to ♯5, since in these cycles both $T1_{BtoR\_REQ(0)}$ and $T0_{BtoR\_REQ(0)}$ are active.

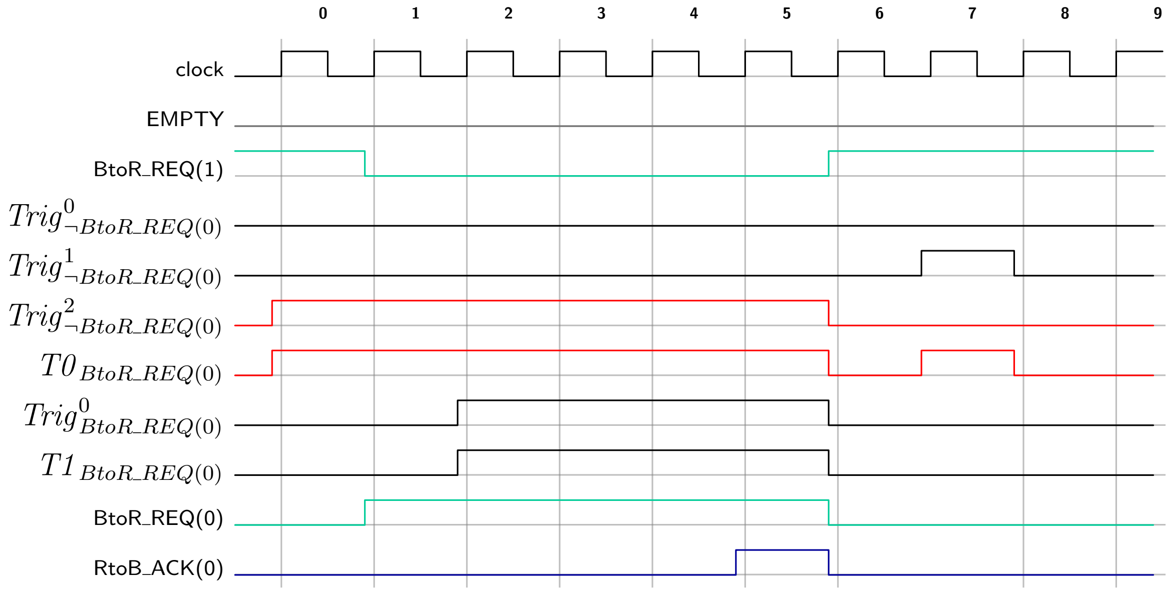


FIGURE 9.12: Timing diagram of *BtoR_REQ(0)* and corresponding trigger signals

9.7.2 Checking the completeness

The definition of z is complete iff in all the cycles:

$$T1_z \vee T0_z = 1$$

Again, we can generate complementary properties automatically for verifying the above condition.

Example 9. Checking the completeness of the properties that constrain the *BtoR_REQ(0)* signal from GenBufRec

Considering $T0_{BtoR\_REQ(0)}$ and $T1_{BtoR\_REQ(0)}$, for signal *BtoR_REQ(0)*. The PSL property that verifies if the set of properties constraining *BtoR_REQ(0)* are complete is:

```
        always (T1_BtoR_REQ_0 or T0_BtoR_REQ_0 ) ;
```

In the timing diagram of Fig. 9.12, in cycles ♯0 and ♯1 none of the $T0_{BtoR\_REQ(0)}$ and $T1_{BtoR\_REQ(0)}$ signals are active; therefore, the annotated properties are not sufficient to specify the *BtoR_REQ(0)* signal completely.

SyntHorus2 generates the PSL properties and VHDL assertions for verifying the consistency and completeness of the set of properties both in formal verification and simulation.

If the definition is consistent but not complete, one or more signal z may not have been annotated and depends on other signals. In this case, its value should be specified by a *complex* solver. Otherwise, the designer may wish to provide a default value for signal z.

9.8 Summary

In this chapter we discussed how to resolve the value of unannotated and duplicated signals. We explained how to express the dependency among all properties by constructing a dependency graph. Using this graph we identified the properties that constrain a signal, and built a simple solver for resolving the value of such signals. Additionally, from the dependency graph we obtained the unannotated signals, and their dependencies. We constructed a complex solver for resolving the value of each set of unannotated signals. The final circuit is the interconnection of properties' reactants and solvers. Moreover, we can generate complementary properties for verifying the consistency and completeness of the specification.

Chapter 10

Practical Experiments and Results

Contents

10.1 Introduction

We applied our synthesis method to several case studies: GenBuf [1](see Chapter 4), AMBA[2] Arbiter (Appendix C), HDLC[3] (Appendix B), CRC[4], and SDRAM[5]. The generated circuits are synthesized both for FPGA[6] and ASIC[7] implementation. The results are compared to the results of another tool, Ratsy. We show how our method considers coherency and completeness of the properties. Finally, some guidelines are provided for writing the properties so that the generated circuits are smaller.

10.2 Hardware prototyping and synthesis results

SyntHorus2 implements the ABS[8] method disclosed in this thesis. It takes as input the *entity* (interface) declaration of the specified module and a set of properties written in the *simple subset* of PSL, and produces a RTL design in the *synthesizable subset* of VHDL (see Appendix E for details about running SyntHorus2).

As discussed in Chapter 3, most of the other works that deal with ABS are automata based, and are based on the “two players game”. Since our approach is so different, it is of interest to evaluate how it compares on a set of benchmarks of increasing complexity.

Acacia [FJR09, ACA] inputs LTL specifications, and outputs a design in *dot* format. We have written a *dot* to VHDL translator to enter the same logic synthesis tool for the last processing phase. Several options may be selected (backward or forward state space traversing; the circuit player or the environment player has the initial move), leading to very different results. Whatever the option, we were not able to process a real example.

Unbeast [EKH12, UNB] inputs LTL specifications in XML syntax and produces an intermediate NuSMV file that is turned to an *aig* format by AIGER. ABC is used to translate *aig* into Verilog.

Ratsy [RAT] inputs *GR(1)* PSL properties through a graphical interface and produces a Verilog design. Also based on game theory, in this system the environment player moves first. Ratsy checks every input, and the properties must be partitioned into a *guarantee* and an *assume* part.

Table 10.1 summarizes the characteristic of these tools: the input and output formats, and the subset of PSL that each supports.

Table 10.1: ABS tools

Tool	Input	Output	FL	SERE
Acacia	LTL	*dot*	✓	no
Unbeast	LTL in XML format	NuSMV	✓	no
Ratsy	LTL	Verilog	*GR(1)* subset of PSL	no
SyntHorus2	PSL	VHDL and PSL properties	PSL_{simple}	partially (see Chapter 6)

We have installed SyntHorus2, Acacia, Unbeast and Ratsy on a workstation with the

[1] IBM Generalized Buffer
[2] ARM Advanced Microcontroller Bus Architecture
[3] High-level Data Link Controller
[4] Cyclic Redundancy Check
[5] Single Data-rate Random Access Memory
[6] Field Programmable Gate Array
[7] Application Specific Integrated Circuit
[8] Assertion Based Synthesis

following characteristics: 64 bit Intel Core 2 Duo CPU E8400, clock rate 3.0GHz, RAM size 2 giga bytes.

For each case study, we took the same specification for all the tools. We had to rewrite our specifications between Ratsy and SyntHorus2, because the *GR(1)* PSL subset does not accept operators `rose`, `fell`, and `next_event` with a temporal expression operand nor any weak PSL operators.

We executed Unbeast, Acacia, Ratsy, and SyntHorus2 for each example. We were able to run Unbeast on the small examples provided with the software distribution, but we timed out on GenBuf even without FIFO. So we shall not enter Unbeast in the comparison. Acacia can also work just on very simple and small examples, e.g. it works for GenBuf, with 2 senders and 2 and 3 receivers, without considering its FIFO; therefore, we excluded the results from the tables.

After execution, the result of all tools has been synthesized with the same synthesis tool to allow a fair comparison in terms of logic gates and area.

Here, we give the synthesis results for the case studied.

10.2.1 IBM Generalized Buffer (GenBuf)

We generate the hardware of GenBuf with multiple senders and 2 receivers, and 2 senders and multiple receivers using SyntHorus2 and Ratsy. In each case, the hardware generation time of the tools are compared, and the generated circuits are synthesized using Quartus II and Design Vision.

Figure 10.1 compares the hardware generation time of SyntHorus2 and Ratsy for GenBuf with multiple senders and 2 receivers.

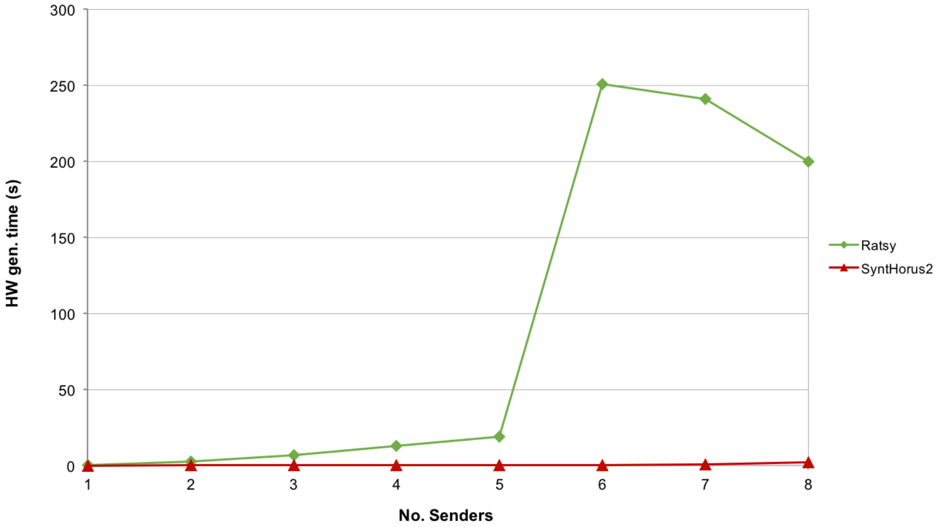


FIGURE 10.1: HW generation time: GenBuf with multiple senders and two receivers

Figure 10.2 compares the hardware generation time of SyntHorus2 and Ratsy for GenBuf with multiple receivers and 2 senders.

The circuit generation time is one to two orders of magnitude smaller for SyntHorus2 depending on the number of senders/receivers. The higher the number, the higher the

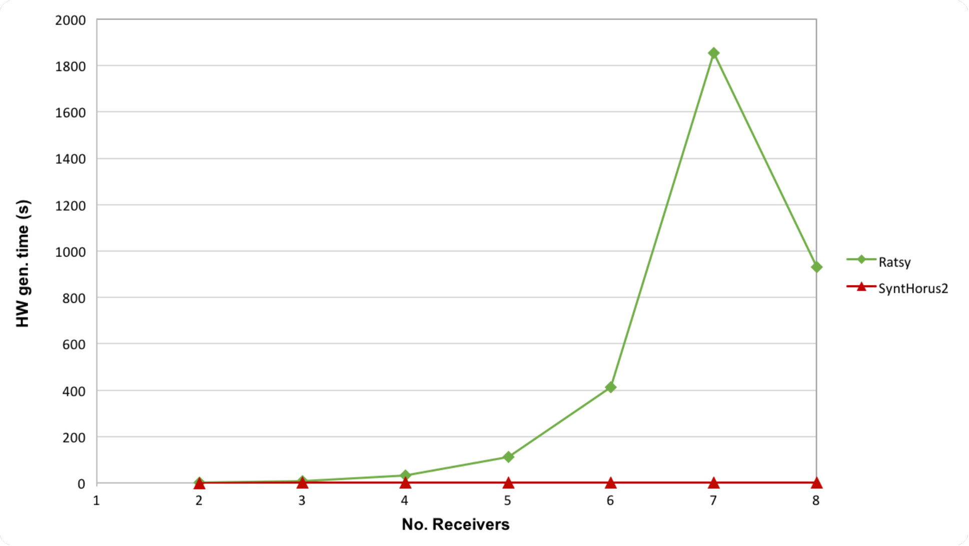


Figure 10.2: HW generation: GenBuf with 2 senders, multiple receivers and with FIFO

order of magnitude. At this point, it is fair to say that SyntHorus2 does not perform verification, while Ratsy has verification embedded in the generation process. Thus, comparing the runtimes of the two tools is not relevant for who wants to perform verification.

Acacia timed out for GenBuf with multiple senders, and multiple receivers. It just worked for GenBuf with 2 senders, and 2 and 3 receivers, without considering FIFO. For other cases, it timed out after 24 hours, most probably due to memory explosion.

10.2.1.1 Synthesis for FPGA implementation

First, we synthesized the generated circuits of SyntHorus2 and Ratsy using Quartus II in order to implement them on a FPGA board (device EP4CE30F23C6 from Cyclone IV device family).

Multiple senders and two receivers

Table 10.2 gives the synthesis results of Quartus II for GenBuf with multiple senders and two receivers running SyntHorus2 and Ratsy. In this table, the number of registers, the total number of LUTs (2-input, 3-input, and 4-input LUTs), and the maximum clock frequency are reported. The reported clock frequency is the maximum potential frequency of the circuits. Based on the selected FPGA device, this frequency may be limited to the maximum clock frequency of the selected FPGA device.

Multiple receivers and two senders, with FIFO

We synthesized GenBuf circuits with multiple receivers and two senders using Quartus II on the same FPGA. Table 10.2 gives the synthesis results for SyntHorus2 and Ratsy.

SyntHorus2 generates faster circuits with less LUTs than Ratsy, but with more registers.

Table 10.2: Quartus II synthesis result for GenBuf controller with multiple senders, and 2 receivers

	SyntHorus2				Ratsy			
# sen.	# prop.	# reg.	# LUTs	F (MHz)	# prop.	# reg.	# LUTs	F (MHz)
1	20	25	57	726.8	41	16	158	251.5
2	25	34	89	701.8	49	21	961	158.2
3	29	33	97	806.4	59	25	2361	133.6
4	33	38	118	766.8	67	29	3417	124.1
5	37	43	132	782.5	76	33	2647	132.6
6	41	46	149	668.0	85	38	6007	104.5
7	45	49	172	659.6	99	42	7160	108.5
8	49	54	188	746.8	103	46	6524	99.9

Table 10.3: Quartus II synthesis result for GenBuf controller with FIFO, multiple receivers, and 2 senders

	SyntHorus2				Ratsy			
# rec	# prop.	# reg.	# LUTs	F (MHz)	# prop.	# reg.	# LUTs	F (MHz)
3	28	37	98	756.4	56	24	2092	130.4
4	31	45	119	781.2	63	27	2587	140.2
5	34	53	146	609.0	70	30	4251	121.6
6	37	61	166	745.7	77	34	10408	92.5
7	40	99	196	616.5	84	36	15191	89.8
8	46	77	215	647.7	91	40	18180	83.6

10.2.1.2 Synthesis for ASIC implementation

We synthesized all the circuits generated by SyntHorus2 and Ratsy with Design Vision under the same conditions:

- considering the typical conditions of the *C35_corelib* library
- setting the clock period to 20 ns, to do static timing analysis, and computing the approximate frequency of the circuit
- setting the "ungroup" compile option
- using the "Exact Map" option with the "medium" effort in mapping, area, and power

For each tool, we provide the number of properties used for the circuit synthesis, the execution time, and the size and timing characteristics of the resulting circuit: number of combinational and sequential cells, total area (including the interconnection area) and approximate clock frequency. The Design Vision execution time is not reported, it is negligible.

Multiple senders and two receivers

Table 10.4 gives the results of our experiments on Genbuf with a FIFO, for 1 to 8 senders and 2 receivers, running SyntHorus2 and Ratsy.

As is shown in this table, SyntHorus2 generates more registers and less combinational cells than Ratsy; and the generated circuits are smaller. For 1 to 5 senders, the clock frequency is less dependent on the number of the inputs. The critical path is determined by the round-robin properties in the receiver side (properties P3_rec_0 and P3_rec_1). By

Table 10.4: Design Vision synthesis result for GenBuf controller with FIFO, multiple senders, and two receivers

	SyntHorus2					Ratsy				
# send	# prop.	# comb. cells	# seq. cells	Total area	F (MHz)	# prop.	#comb. cells	# seq. cells	Total area	F (MHz)
1	20	231	63	33624	571	41	221	16	25034	222
2	25	335	83	46748	521	49	1322	21	127502	127
3	29	360	88	50054	571	59	2541	25	240411	90
4	33	467	101	60934	571	67	3818	29	358160	78
5	37	590	113	72594	521	76	3230	33	303860	77
6	41	643	124	80203	463	85	6954	38	648910	64
7	45	764	134	92016	442	99	8320	42	773090	69
8	49	879	147	105155	418	103	7471	46	695581	64

increasing number of the senders from 5 to 8, a property in the sender side determines the critical path, which depends on the number of senders.

Figure 10.3 compares the total number of the gates for the circuits that are generated by SyntHorus2 and Ratsy. For each circuit, the total number of gates is presented as the number of the 2-input NAND gates, which is computed by dividing the total cell area (not the total area reported in the table) by the area of a 2-input NAND gate obtained from *C35_corelib* library. This number is not accurate, and is just for giving an evaluation of the number of gates.

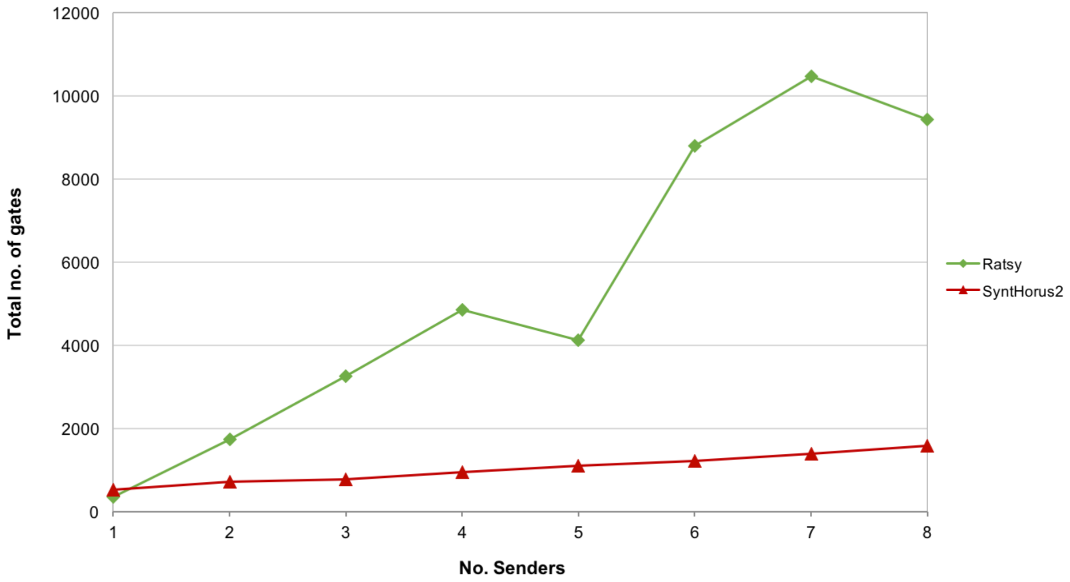


FIGURE 10.3: Total number of gates: GenBuf with multiple senders and 2 receivers

Multiple receivers and two Senders, with FIFO

Table 10.5 gives the results of our experiments on Genbuf with a FIFO, for 3 to 8 receivers and 2 senders, running SyntHorus2 and Ratsy.

As is shown in this table, the clock frequency of the circuits generated by SyntHorus2 is less dependent on the number of receivers, than the circuits generated by Ratsy.

Figure 10.4 compares the total number of gates for the circuits that are generated by SyntHorus2 and Ratsy. Again, the number of gates is the number of 2-input NAND gates

Table 10.5: Design Vision synthesis result for GenBuf controller with FIFO, multiple receivers, and two senders

	SyntHorus2					Ratsy				
# rec	# prop.	# comb. cells	# seq. cells	Total area	F (MHz)	# prop.	#comb. cells	# seq. cells	Total area	F (MHz)
3	28	414	102	57876	568	56	2781	24	259796	96
4	31	467	118	66715	565	63	3285	27	306134	80
5	34	546	134	76961	555	70	5146	30	475880	67
6	37	624	150	87188	555	77	12970	34	1198182	57
7	40	714	175	100559	555	84	17934	36	1647189	56
8	46	860	191	114564	555	91	20828	40	1894378	65

and it is computed approximately.

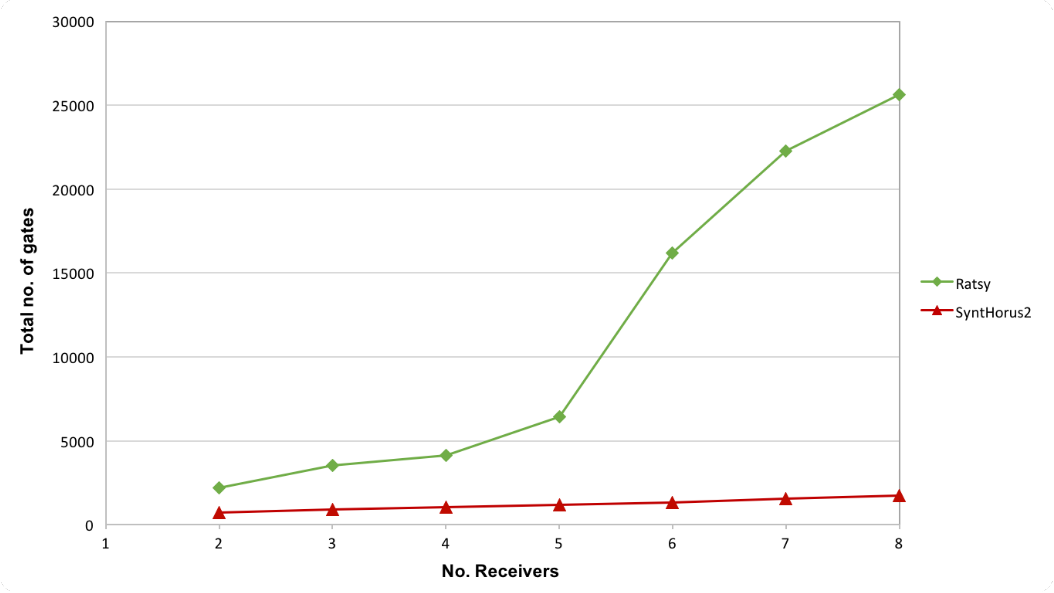


FIGURE 10.4: Total number of gates: GenBuf with multiple receivers and 2 senders

10.2.2 AMBA arbiter

We have performed the same kind of experiments on the AMBA-AHB bus arbiter, a popular benchmark. Figure 10.5 compares the HW generation time of SyntHorus2 and Ratsy.

As is shown in this figure, SyntHorus2 generates the circuits faster than Ratsy. We synthesized AMBA using Quartus II and Design Vision, with the same options as GenBuf.

10.2.2.1 Synthesis for FPGA implementation

Table 10.6 shows the synthesis results of Quartus II for AMBA arbiter.

As is shown in this table, the AMBA arbiter specification holds 2 to 3 times more properties for Ratsy than for SyntHorus2. It should be noted that some complex properties accepted by SyntHorus2 become 14 or 16 simple properties after rewriting to comply with the Ratsy acceptable PSL subset. For example, the following property of AMBA arbiter is rewritten into 14 simpler properties.

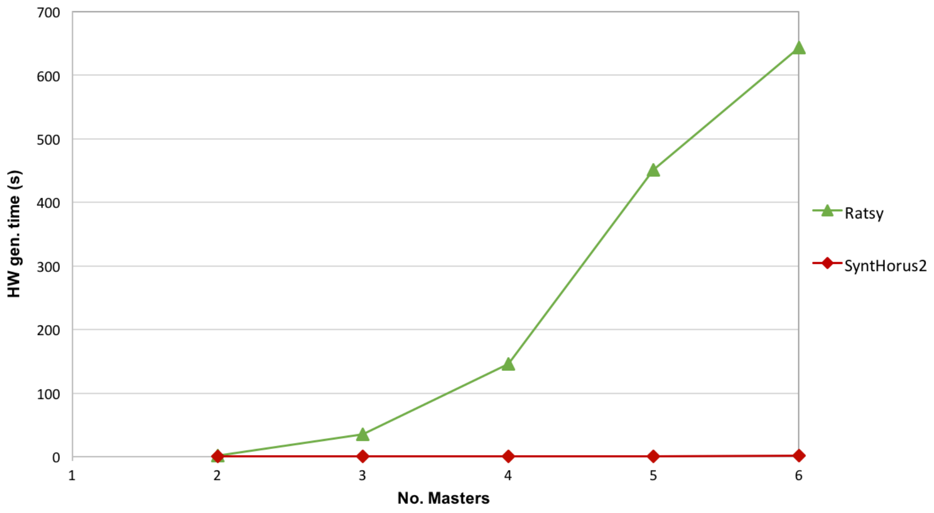


FIGURE 10.5: HW generation time: AMBA arbiter

Table 10.6: Quartus II synthesis result for AMBA arbiter with 2 slaves and multiple masters

	SyntHorus2				Ratsy			
# masters	# prop.	# reg.	# LUTs	F (MHz)	# prop.	# reg.	# LUTs	F (MHz)
2	35	24	59	795.5	77	21	382	199.2
3	46	33	113	852.5	96	29	2941	131.1
4	56	42	119	923.4	114	29	6085	106.9
5	66	51	150	795.5	133	34	3091	130.45
6	77	60	181	758.1	151	37	4355	115.8

```
G3:
  always ((HMASTLOCK and (HBURST = INCR4) and HREADY and (HTRANS = NON–SEQ)
      ) -> next ((HTRANS = SEQ) until_[3] HREADY));
```

In SyntHorus this property can be rewritten as two properties, using the `next_event` operator, which is not supported in Ratsy.

10.2.2.2 Synthesis for ASIC implementation

Table 10.7 gives our results for 2 slaves and different numbers of masters. The columns have the same meaning as in Table 10.5. Again, we find that SyntHorus2 produces smaller and faster circuits, but with more registers; in addition, the speed of the circuit is less sensitive to the number of masters.

Table 10.7: Design Vision synthesis results for AMBA arbiter

	SyntHorus2					Ratsy				
# masters	# prop.	# comb. cells	# seq. cells	Total area	F (MHz)	# prop.	#comb. cells	# seq. cells	Total area	F (MHz)
2	33	303	90	45165	637	77	515	21	51985	164
3	46	513	132	71915	621	96	3867	29	362589	92
4	56	567	159	82522	606	114	7712	29	721363	67
5	66	725	194	102762	629	133	3920	34	370179	82
6	77	660	228	121855	625	151	5855	37	552310	62

Figure 10.6 compares the total number of gates (2-input NAND gates) of the circuits that are generated by SyntHorus2 and Ratsy.

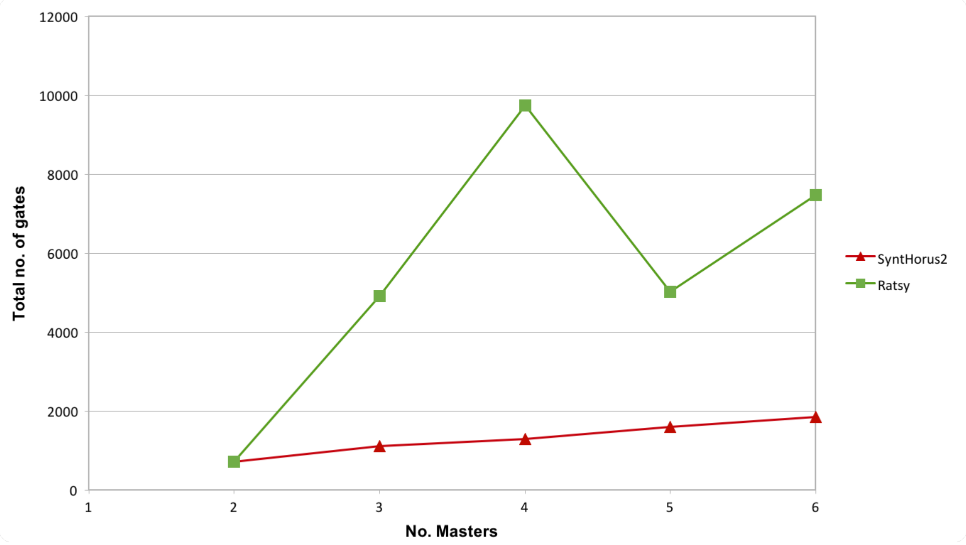


FIGURE 10.6: Total number of gates: AMBA arbiter

The following comments can be made on these experiments (GenBuf and AMBA arbiter):

- The number of properties used *to generate the design* is higher for Ratsy than for SyntHorus2. This is due to the underlying method: game-based methods need to consider both the *guarantee* and the *assume* properties, while the modular method of SyntHorus2 only takes the *guarantee* properties to produce the circuit design.

- In both example, except for GenBuf with 1 sender, the size of the combinational part and the total area of the generated circuit is smaller for SyntHorus2.
- SyntHorus2 generates more registers than Ratsy. This may be in relation with the fact that the maximum clock frequency is higher for the circuits generated by SyntHorus2. The difference is particularly significant for GenBuf with 2 senders and multiple receivers. However, the total circuit size is smaller.

10.2.3 Other examples

Our final three benchmarks are reported in Table 10.8. To our knowledge, they have never been published in the ABS context. The SDRAM controller is one of the test cases of the OneSpin formal verification tools distribution. The CRC is a hardware implementation of the cyclic redundancy check for error detection. The High-level Data Link Controller (HDLC) is an ISO standard for point to point communication at the network data link layer. We have complemented the assertions found in [PPSQ13] to fully specify the HDLC controller. With 120 properties, it is the largest specification processed, and the largest circuit generated. Yet the circuit generation time remains small (1.06 sec), and the clock frequency high (429 MHz).

Table 10.8: Design Vision synthesis results for HDLC, SDRAM, and CRC

			SyntHorus2				
Circuit	# prop.	Hw. gen. time (s)	# comb. cells	# seq. cells	Total area	Total # of gates	F (MHz)
HDLC	120	1.06	2646	1433	600527	9588	429
SDRAM	9	0.2	1045	769	295107	4765	513
CRC	14	0.14	641	293	131122	2401	406

10.2.4 Comparison between FLs and SEREs

To show the applicability of our synthesis method to SEREs, the SERE properties are provided for GenBuf, AMBA arbiter and HDLC, the corresponding VHDL designs are generated using SyntHorus2, and are synthesized using Design Vision. The number of properties, the hardware generation time, the number of combinational and sequential cells, the total area, and the circuit frequency are given. For all benchmarks, the properties processing time by SyntHorus2 is very small, a fraction of a second for the classical GenBuf and AMBA bus, less than two seconds for the more complex HDLC.

10.2.4.1 GenBuf

The FL properties of GenBuf are translated into SEREs.

Multiple senders

Table 10.9 shows the synthesis results for GenBuf with multiple senders and two receivers. The generated circuits have larger number of registers and combinational cells comparing to the circuits generated from the FL properties. However, the total number of gates is very close to the circuits obtained from FLs For all the cases, the clock frequency is 370 MHz, and is independent from the number of senders.

Table 10.9: Design Vision **synthesis results for GenBuf with multiple senders (for SERE properties)**

# senders	# prop.	HW gen. time (s)	# comb. cells	# seq. cells	Total area	Total# of gates
1	20	0.15	319	76	45196	701
2	25	0.16	417	94	56858	879
3	29	0.20	458	100	61644	951
4	33	0.23	550	115	72416	1114
5	37	0.28	634	125	81174	1243
6	41	0.27	721	138	91258	1392
7	45	1.52	836	148	102263	1558
8	49	0.81	941	162	115783	1752

Figure 10.7 compares the total number of gates for the circuits generated from FLs and SEREs. In average, the circuits generated from SEREs have 20% more gates than the circuits generated from FLs.

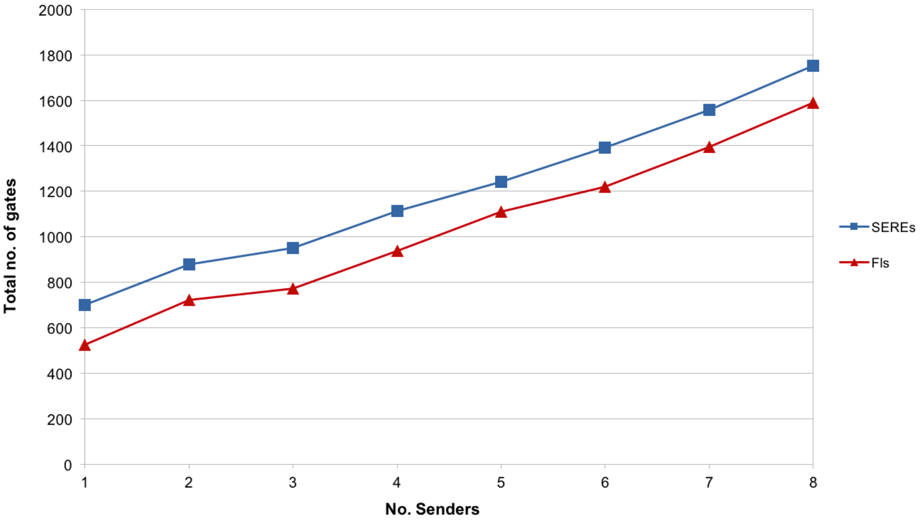


FIGURE 10.7: Total number of gates: GenBuf with multiple senders and 2 receivers (generated from FLs and SEREs)

Multiple receivers

Table 10.10 shows the synthesis results for GenBuf with multiple receivers and two senders. Compared to the circuits generated from FLs, the generated circuits from SEREs have more gates, and are almost 2 times slower .

Figure 10.8 compares the total number of gates for the circuits generated from FLs and SEREs. As is shown in this figure, the difference between the number of gates of the two circuits generated from FLs and SEREs becomes more significant as the number of receivers increases. With multiple senders and 2 receivers, this difference was almost the same for all numbers of senders. It is due to the properties that specify the round-robin policy in the receiver side. When rewriting these properties as SEREs, the property involves the '$*$' unbounded repetition. The primitive reactant of '$*$' is the area and frequency bottleneck. In the case of multiple senders and two receivers, we have two round-robin

Table 10.10: Design Vision **synthesis results for GenBuf with multiple receivers (for SERE properties)**

# receivers	# prop.	HW gen. time (s)	# comb. cells	# seq cells	Total area	Total # of gates	Freq. (MHz)
3	27	0.19	529	119	720701	1123	308
4	30	0.25	651	146	901106	1392	320
5	33	0.35	775	175	108083	1669	290
6	36	0.61	905	206	127124	1963	296
7	42	0.26	1052	248	150525	2325	296
8	45	0.29	1215	287	174253	2691	280

properties (one for each receiver, see properties `P3_sere_rec_0` and `P3_sere_rec_1` in Fig. 4.9) whatever the number of senders. In the case of multiple receivers, the number of round-robin properties increases with the number of receivers, and the area increases more.

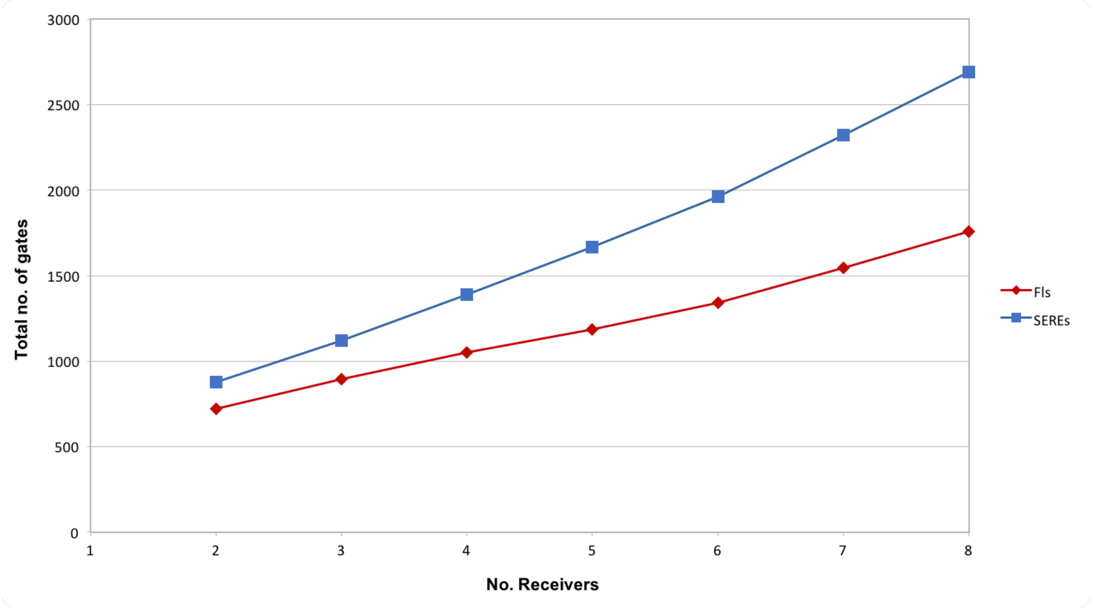


FIGURE 10.8: Total number of gates: GenBuf with multiple receivers and 2 senders (generated from FLs and SEREs)

The interconnection area is larger for the circuits generated from SEREs than the circuits generated from FLs.

10.2.4.2 AMBA Arbiter

For the AMBA bus arbiter, we wrote the SERE specification based on its protocol description in English and the FL properties. Table 10.11 summarizes the synthesis results for 2 slaves and 2-6 masters. Compared to the circuits resulting from FL properties, the circuits generated from SEREs are 20-30% smaller (less combinational and sequential cells), and significantly slower (up to half the speed).

10.2.4.3 HDLC

For HDLC, we provided two set of properties:

Table 10.11: Design Vision synthesis results for AMBA arbiter (for SERE properties)

# masters	# prop.	HW gen. time (s)	# comb. cells	# seq. cells	Total area	Total # of gates	F (MHz)
2	28	0.14	223	62	33614	523	368
3	41	0.24	341	95	50146	777	324
4	52	0.32	429	120	63209	978	307
5	63	0.41	520	145	76471	1181	296
6	74	0.63	628	170	90546	1394	266

- SERE1: all the FL properties are translated to their equivalent SERE properties.
- SERE2: three modules, FlagDetection, ZeroDetection, and ZeroInsertion, are directly expressed using SERE properties (see Appendix B). These SEREs are written based on the protocol, and they have not been obtained by rewriting FLs.

Table 10.12 summarizes the synthesis result. In the SERE1 specification, 120 FL properties are translated to 120 SERE properties. The SERE2 specification has 108 SERE properties.

The circuit obtained from SERE1 has more combinational and sequential cells compared to the circuit generated from FLs, and it is almost 5 times slower.

The circuit obtained from SERE2 has more combinational cells and less sequential cells compared to the circuit generated from FLs, and it is almost 6 times slower.

These results show that translating FLs to SEREs increases the circuit size and decreases the clock frequency; it is the case of GenBuf and HDLC obtained from SERE1. In contrast, expressing the circuit behavior using SEREs may decrease the area, at the cost of speed. It is the case of AMBA and HDLC obtained from SERE2.

Table 10.12: Design Vision synthesis results for HDLC (for SERE properties)

	# prop.	HW gen. time (s)	# comb. cells	# seq. cells	Total area	Total # of gates	F (MHz)
SERE1	120	1.22	3238	1157	614507	9700	82
SERE2	108	1.51	3017	839	516363	8050	59

10.3 Completeness and coherency consideration

We used OneSpin for formally verifying if the generated circuits correspond to the specification.

Then, we verified if the set of specification is complete and consistent. As was discussed in Chapter 9, SyntHorus2 generates complementary properties for checking the completeness and coherency of the properties. For all the circuits, we generated these properties. We used the properties in ModelSim and also OneSpin. For all the case studies, these properties hold both in simulation and formal verification; i.e. the specification is complete and consistent.

Example 1. Checking consistency

Consider the annotated properties that are shown in Fig. 10.9. These properties are taken from GenBuf specification.

```
P0_sender_0:
  always(not BtoS_ACK_m(0) and not StoB_REQ_m(0) ->next!(not BtoS_ACK_g(0))
      );

P1_sender_0:
  always(BtoS_ACK_m(0) and StoB_REQ_m(0) ->next!(BtoS_ACK_g(0)));

P2_sender_0:
  always(rose(StoB_REQ_m(0)) -> not BtoS_ACK_g(0));
```

FIGURE 10.9: Some properties from GenBuf that generate *BtoS_ACK(0)*

The circuit is generated by SyntHorus2 and the complementary properties are generated for checking the consistency. Figure 10.10 shows the VHDL assertion generated by SyntHorus2 and considers the mutual exclusion of triggers that correspond to the *BtoS_ACK(0)* signal.

```
...

process(clk) begin
    if (clk = '0' and clk'event) then
     if(reset_n='1') then

       ASSERT (( (not (trigger_1) ) or (not (trigger_0 or trigger_2) )) =
            '1')
       REPORT "T0 (trigger_0 or trigger_2) and T1 (trigger_1) are not
            mutually exclusive"
       SEVERITY ERROR;
      end if;
  end if;
end process;

...
```

FIGURE 10.10: The assertion for considering the mutual exclusion of *BtoS_ACK(0)* triggers

The circuit is simulated using ModelSim along with the complementary properties. The waveform obtained from ModelSim is shown in Fig. 10.11, in which the assertion passed in all the cycles.

Now, suppose that `P1_sender_0` is written incorrectly, as shown in Fig. 10.12, and the circuit is generated for the incorrect properties.

The waveform is shown in Fig. 10.13, in which the assertion (see Fig. 10.10) fails at $t = 13200\ ns$, and we get the name of the triggers that are not consistent. Therefore, the designer can debug the properties more easily.

10.4 Guidelines for obtaining smaller circuits

Here we give some guidelines for writing the properties in a way that generates the smaller circuits.

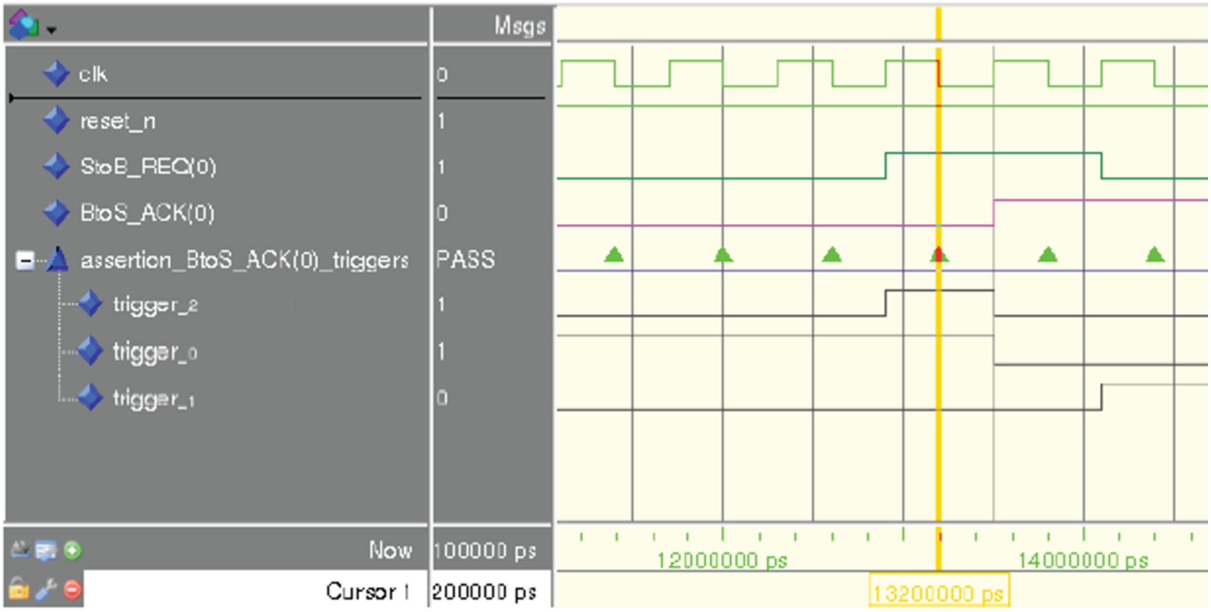


FIGURE 10.11: The wave form, without any failure

```
P1_sender_0_false:
  always(StoB_REQ_m(0) -> BtoS_ACK_g(0));
```

FIGURE 10.12: A modified property of GenBuf that generates *BtoS_ACK(0)*

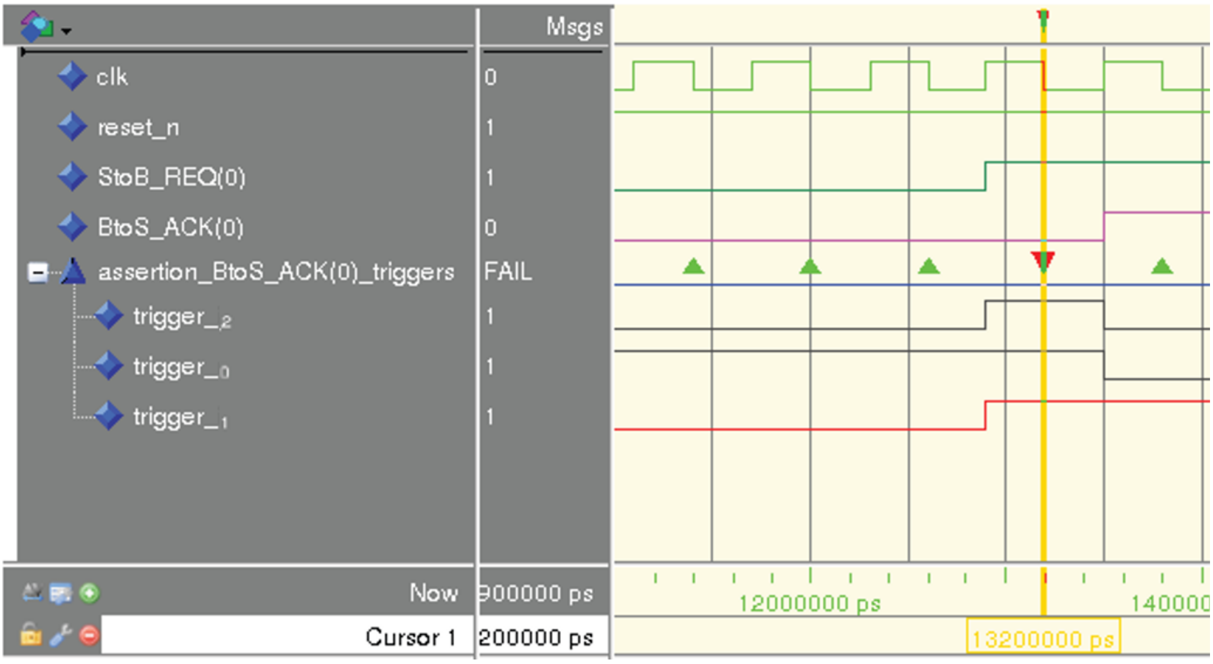


FIGURE 10.13: The wave form, with assertion failure

Since our method is modular, for each property it instantiates all the primitive reactants of a property. Therefore, if we are able to merge several properties into one property, we generate a smaller circuit.

Example 1. Merging properties.

Assume that we have the following properties:

```
P0: always(A -> next!(B));
P1: always(A -> C and next!(D));
```

In this case, we have two instances of the `always`, '–>', and `next!` primitive reactants. Since the left-hand side of both properties are the same, we can rewrite these properties as follow:

```
P: always(A -> C and next!(B and D));
```

In this case, we have one instance of the `always`, '–>', and `next!` primitive reactants.

Another factor that affects the size of the generated circuit significantly is the size of the complex solvers.

Remember from Chapter 9, if we have $\mathcal{Z} = (z_1, \ldots z_n)$ and $T_{\mathcal{Z}} = (Etrig_0, \ldots, Etrig_m)$, then the complex solver has a LUT that has $m + n$ columns and at most 2^{m+n} rows. Either by reducing the number of the dependent signals (n) or by reducing the number of the properties that affect these dependent signals (m) the size of the complex solvers decreases.

Example 2. Reducing the size of the solver: reducing the number of *Etrig* signals.

Assume that we have the following properties:

```
P0: always(A -> next!(B or C));
P1: always(D -> next!(B or C));
```

Here, we have two signals that are dependent: $\mathcal{Z} = (B, C)$, and two trigger signals: $T_{\mathcal{Z}} = (Etrig_0, Etrig_1)$. Therefore, the LUT has at most 16 rows. However, we can rewrite the properties as follow:

```
P: always(A or D -> next!(B or C));
```

In this case, we have one trigger signal; therefore, the LUT has at most 8 rows.

Example 3. Reducing the size of the solver: reducing the number of dependent signals.

As another example assume that we have the following properties, where A is an input.

```
P0: always(A or B or C);
P1: always(D -> next!(B or C));
```

Here, we have three signals that are dependent: $\mathcal{Z} = (A, B, C)$, and two trigger signals: $T_{\mathcal{Z}} = (Etrig_0, Etrig_1)$. Therefore, the LUT has at most 32 rows. However, we can rewrite the properties as follow:

```
P0_modified: always(not A -> (B or C));
P1: always(D -> next!(B or C));
```

In this case, we have two dependent signal and two trigger signals; therefore, the LUT has at most 16 rows.

10.4.1 GenBuf: Multiple senders

For more complex properties, the way in which the properties are written has a significant effect on the size of the generated circuit. To illustrate this fact, the properties of GenBuf for multiple senders are modified: the properties in the form of $A \rightarrow B$, where A and B are Boolean, have been rewritten as `:not` A `or` B.

The generated circuit is synthesized using Design Vision. In the original specification, signals *BtoS_ACK(0)*, *BtoS_ACK(1)*, and *ENQ* are dependent. In the rewritten specification, *BtoS_ACK(0)*, *BtoS_ACK(1)*, *ENQ*, and *DEQ* are dependent; therefore, the area is increased. Figure 10.14 compares the total number of gates for the two circuits generated from the original and the rewritten specifications.

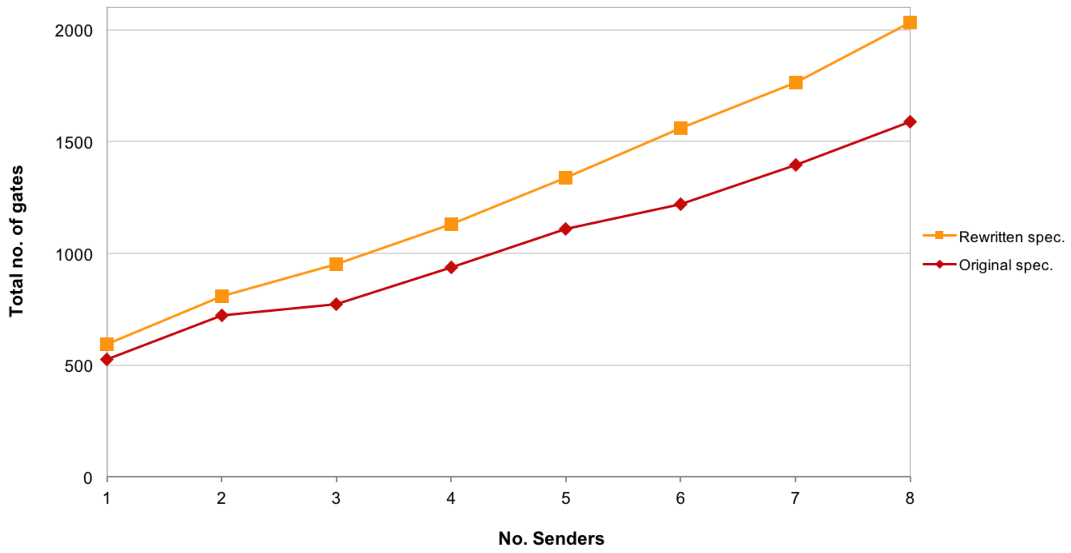


FIGURE 10.14: Total number of gates for GenBuf with multiple senders: original and rewritten specification

10.5 Summary

In this chapter, we applied our method to several case studies, and compared our results with other ABS tools. The experiments show that SyntHorus2 generates smaller and faster circuits.

Chapter 11

Conclusion and future works

Contents

In this thesis, we have presented a modular method to synthesize the controller part of circuits, reactants not monitors, from their temporal properties written in PSL.

11.1 Contributions

Here, the main contributions of this work are summarized.

- Starting from the trace semantics of PSL, we formally defined a *dependency relation* between the operands of the temporal SERE[1] operators. Then, we gave a hardware interpretation of the dependency relations, which constitutes the basis on which the library of primitive reactants is built (see Chapter 6).
- These dependency relations, accompanied by the formal dependency relation of FLs, are the formal model on which the *annotation* algorithm is written (see Chapter 7).
- Considering the dependency among all the properties, *solver* components have been generated to resolve the value of the duplicated and unannotated signals (see Chapter 9).
- We generate some complementary properties to check the coherency and completeness of the set of the properties.
- The prototype tool SyntHorus2 has been implemented, based on the principles described in this thesis. It is being adapted based on the requirements in industry.
- SyntHorus2 was exercised on a set of benchmarks, and also on real size circuits such as the AMBA-AHB bus arbiter and the HDLC controller. Comparing SyntHorus2 with other ABS tools is difficult, since each tool requires its own subset of LTL or PSL that should be adapted for each tool. Comparing to other tools, SyntHorus2 generates smaller and faster designs on the bigger examples.

Intermediate results of SyntHorus2 allow debugging a specification, and verify if it is consistent and complete. Assertions that are generated automatically on the trigger signals may be verified by a simulator, or model checked with a formal verification tool. In addition, SyntHorus2 can provide an environment prototype that complies with the specifications, for testing another circuit module.

11.2 Future works

Now, SyntHorus2 only processes scalar and vector Boolean signals in the properties. Future works include the recognition of more complex data types, such as integers and enumerated types.

SyntHorus2 supports partially the modeling layer of PSL. It supports arithmetic and comparison operators. However, it does not support the definition of local signals. This feature should be added to SyntHorus2.

As was explained in Chapter 6, our synthesizable subset of SEREs has some limitations. For example, we cannot have non consecutive repetition. Therefore, the synthesizable subset of SEREs should be extended and the limitations should be alleviated.

[1] Sequential Extended Regular Expression

To overcome some of the limitations, e.g. observing $\varphi = A\&B$ where A and B are sequences, we can take advantage of the automata-based method. We can combine the automata-based and modular methods, then, use automata for the left-hand side of an implication, which should be observed, and the modular method for the right-hand side of an implication that should be generated.

Moreover, we should optimize the primitive reactants for SEREs. As was shown in Chapter 10, if we rewrite an FL property into its equivalent SERE property, the circuit obtained from the SERE is larger and slower. We should optimize both SERE primitive reactants and their interconnections.

We have provided some guidelines on how to write properties to generate smaller circuits; they should be enhanced. We can provide predefined sets of properties to express some behaviors of the signals, e.g. mutual exclusion, round-robin scheme, 4-phase handshaking protocol, etc.

The implementation of the complex solvers can be enhanced to generate smaller solvers. As was discussed in Chapter 9, some rows of the LUT may not be useful, because some combinations of *Etrig* signals never happen. Such rows can be eliminated by model checking on the properties. This has not been automated.

As was discussed in Chapter 9, to calculate the value of the unannotated signals, there may be several choices obtained from the LUT. Now, we select the first row of the LUT that matches our requirement. Other selection policies can be considered.

At this point, complex solvers are provided for scalar signals. They should be extended to support vectors. In addition, complex solvers cannot be used in the cases that the unannotated signals depend on the modeling layer operators and functions. We should solve this problem.

Our method is modular; for each property it instantiates all the primitive reactants of the property operators. This leads to redundant components. This is similar to the early days of synthesis: each instance of an operator in the RTL design produced a distinct hardware operator. An optimization step is needed to share primitive reactants in the generated circuit.

Appendix **A**

Symbols

Table A.1: Symbols

Symbol	Definition	Reference
$\mathbf{P}$	non-empty set of atomic propositions	Chapter 2
Σ	the set of all possible valuations of $\mathbf{P}$ ($\Sigma = 2^{\mathbf{P}}$)	Chapter 2
ℓ	“letter”: a valuation of all the propositions in $\mathbf{P}$	Chapter 2
w	“word”: in practice, the succession over time of the signal values, i.e. an execution trace	Chapter 2
$\ell \vdash exp$	“exp is true in ℓ”: exp takes value true if all its variables take their value as in ℓ	Chapter 2
$w \models$ `property`	“`property`” is true on word w: the extended semantics by structural induction over FL properties to words	Chapter 5
$\lfloor A \lhd B \rfloor_w$	A depends on B on w	Chapter 5
$Trig_z$	a trigger signal that constrains z to 1	Chapter 5
$Trig_{\neg z}$	a trigger signal that constrains z to 0	Chapter 5
$\mathcal{C} \| \!\vdash \varphi$	$\mathcal{C}$ implements φ	Chapter 5
$w \mathrel{|}\!\equiv$ `property`	“`property`” holds tightly on word w: the extended semantics by structural induction over SERE properties to words	Chapter 6
AST	Abstract Syntax Tree	Chapter 7
DAST	Directed Abstract Syntax Tree	Chapter 7
$\varphi^L_{n^L}$	the left sub-sequence of φ, whose depth is n^L	Chapter 8
$\varphi^R_{n^R}$	the right sub-sequence of φ, whose depth is n^R	Chapter 8
$\mathcal{DG}$	the dependency graph	Chapter 9

Symbol	Definition	Reference
$\mathcal{T}0_z = (Trig^0_{\neg z}, Trig^1_{\neg z}, \ldots, Trig^{nb_0-1}_{\neg z})$	the vector of the trigger signals that constrain z to 0	Chapter 9
$\mathcal{T}1_z = (Trig^0_{z}, Trig^1_{z}, \ldots, Trig^{nb_1-1}_{z})$	the vector of the trigger signals that constrain z to 1	Chapter 9
$T0_z = \bigvee_i Trig^i_{\neg z}$	the disjunction of the trigger signals that constrain z to 0	Chapter 9
$T1_z = \bigvee_j Trig^j_{z}$	the disjunction of the trigger signals that constrain z to 1	Chapter 9
$\mathcal{Z} = (z_1, \ldots, z_n)$	the vector of n dependent (unannotated) signals	Chapter 9
$Expr_j$	the expression that represents the unannotated sub-tree of DAST$_j$	Chapter 9
$\mathcal{E} = (Expr_0, \ldots, Expr_m)$	the vector made of $Expr_j$ signals	Chapter 9
$Etrig_j$	the trigger signal that triggers $Expr_j$	Chapter 9
$\mathcal{T}_{\mathcal{Z}} = (Etrig_0, \ldots, Etrig_m)$	the vector made of $Etrig_j$ signals	Chapter 9

Appendix B

Case study: High-level Data Link Controller

Contents

High-level Data Link Controller (HDLC) permits synchronous or start/stop, code-transparent data transmission. The HDLC controller IP has a transmitter and receiver for transmitting the frames with a specific format (see Fig. B.1). The HDLC controller IP performs serialization/deserialization, CRC generation, transparency and abort generation/detection.

The transmitter receives its input data from an external device, and sends it with a specific frame format to the receiver. Each frame is made of an open flag (the value is "01111110"), the information (address, control, info), the CRC and the closing flag.

As is shown in Fig. B.1, the transmitter and receiver have several components. The objective is generating each component from its properties using SyntHorus2.

First, we tried to generate the HDLC circuit from the properties given in [PPSQ13]. However, these properties are not complete. When we want to generate a hardware from the properties, the set of properties should completely specify all signals behaviors. We provided the FL properties for HDLC, based on its protocol. We used the SERE properties of [PPSQ13] to verify if the circuit obtained from SyntHorus2 works correctly.

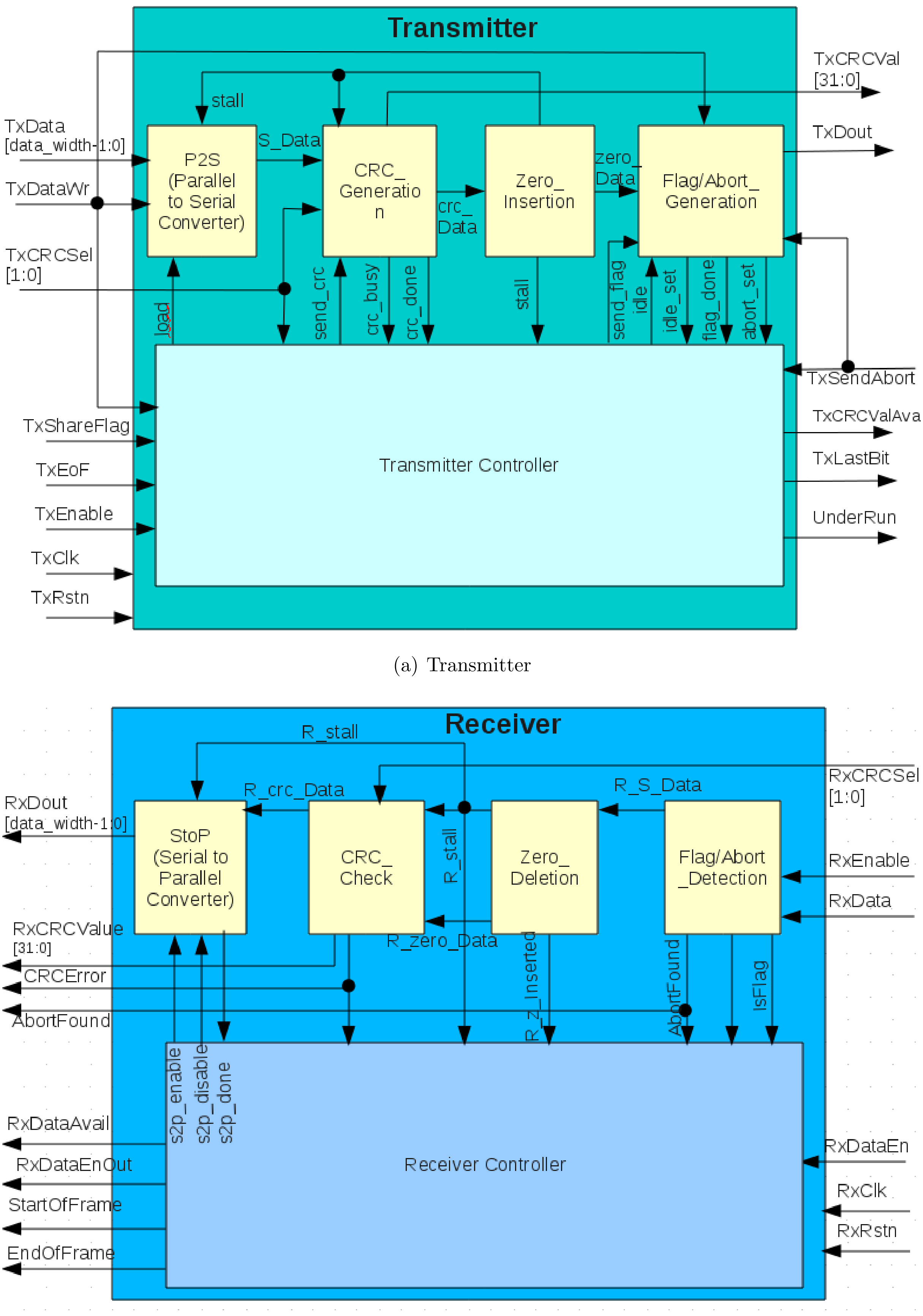
B.1 Transmitter

Figure B.1 (a) shows the block diagram of the HDLC transmitter. Before describing the functional behavior of the system, the transmitter interface signals are introduced:

- Inputs
  - *TxClk*: the actual clock of the transmitter
  - *TxRstn*: the active-high reset signal
  - *TxEnable*: if this signal is 1, the transmitter is enabled
  - *TxData*: the transmitter parallel data bus
  - *TxDataWr*: informs the transmitter that an external packet is ready to send
  - *TxSendAbort*: informs the transmitter that sending the frame should be stopped and an abort sequence should be sent
  - *TxCRCSel*: sets the CRC size (no checksum, 8, 16, or 32 bits)
- Outputs
  - *UnderRun*: indicates the underrun error. It is generated if no new valid data is written within 8 cycles after BuffEmpty goes high.
  - *TxDout*: provides the serial output data
  - *TxCRCValue*: the CRC value
  - *TxLastBit*: indicates the last bit of the last flag of the transmission has been sent out

The transmitter is disable when *TxEnable* is 0. In this mode, idle mode, there is no valid data on *TxData* (*TxDataWr = 0*), and consecutive 1s are transmitted. As soon as the transmitter becomes enable, data transmission starts when the *TxDataWr* signal is asserted. First, the transmitter sends an opening flag (by activating the `FlagAbortGen`

B.1 : Transmitter



(a) Transmitter

(b) Receiver

FIGURE B.1: HDLC controller block diagram

component). After, the data will be sent. The CRC value is sent out after sending the last byte of the data. Then, a closing flag is appended to mark the end of the frame.

The data transmission mechanism is transparent. It means that the transmitter should prevent of occurring the flag pattern in the data (including CRC). Anytime a sequence of five consecutive 1 occurs, a 0 should be inserted. It is done by `ZeroInsertion` component, which is enabled by the transmitter controller.

When the *TxSendAbort* signal is asserted, the transmitter goes into the abort mode, in which the transmitter should send 7 consecutive ones, without inserted zero.

The transmitter controller is responsible for enabling/disabling the `P2S`, `CRCGen`, `ZeroInsertion` and `FlagAbortGen` components.

In the following, each component is explained briefly, and the properties are shown.

B.1.1 Parallel to Serial converter

Figure B.2 shows the properties for `P2S`. In these properties, *TxData*, *stall*, and *load* are inputs, and *S_Data* is the output, and *P_Data* is an internal signal. The `P2S` unit is active if *stall*=0. In this mode, if there is no new data to be loaded (*load*=0), the data is shifted out. If `P2S` is stalled, 0 is shifted out, and the *P_Data* internal signal keeps its previous value. The new data is loaded into *P_Data* when `P2S` is active, and the *load* signal is 1.

```
vunit P2S
{
  P0_init:
    always(not T_frame_valid -> ((P_Data = ”00000000”) and S_Data = ’0’);

  P1_load:
    always(T_frame_valid and not stall and load -> (P_Data = TxData) and (
        S_Data = P_Data(7)));

  P2_stall_data:
    always(T_frame_valid and stall -> S_Data = ’0’);

  P3_not_shift:
    always(T_frame_valid and not stall and not load -> (S_Data = P_Data(7))
         and (P_Data(0) = ’0’) and (P_Data(1 to 7) = prev(P_Data(0 to 6))));

  P4_keep_data:
    always(T_frame_valid and stall and not load ->  (P_Data = prev(P_Data))
        );

}
```

FIGURE B.2: Properties that describe `P2S`

B.1.2 CRC generation

The `CRCGen` component calculates a CRC across the transmitted message whenever the *send_crc* signal is 1. Otherwise, it transfers the input data (*S_Data*) to the output(*crc_data*). Two different polynomials can be selected by specifying the value of *TxCRCSel* in the idle mode of the transmitter. The 16-bit CRC uses the polynomial $x^{16} + x^{12} + x^5 + 1$, and the

32-bit CRC uses the polynomial $x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$. Figure B.3 shows the PSL properties for `CRCGen` (for 16-bit CRC).

In the specifications, *poly16*, *send_crc*, and *S_Data* are the inputs and *crc_busy*, *crc_done*, and *crc_data* are the outputs. *R16* is an internal signal for storing and shifting the input data. *crc_counter* is used to calculate when the transfer is finished.

B.1.3 Zero insertion

Since the data transmission mechanism is transparent, anytime a sequence of five consecutive 1 occurs, a 0 should be inserted. It is done by the `ZeroInsertion` component. Figure B.4 shows the FL properties for `ZeroInsertion`. If the last 5 input data (*crc_data*) are 1, in the next cycle, the *stall* signal becomes 1 that halts the `P2S` and `CRCGen` modules. In the following cycle, 0 is put on the output and the *stall* signal becomes 1, which informs the transmitter controller that a 0 is inserted on the data sequence. If the input data does not contain five consecutive 1s, the *stall* signal becomes 0 and the *zero_Data* output signal takes the value of input *crc_data*.

B.1.4 Flag generation

The `FlagAbortGen` component is enabled by transmitter controller for sending an opening and closing flag whenever the *send_flag* signal is asserted. In addition, this component should send eight consecutive 1s when *TxSendAbort = 1*. Moreover, this component sends consecutive 1s when transmitter is idle (it is also possible to send flag or partial flag in the idle mode, but here, we are sending consecutive 1s).

Figure B.5 shows the properties for `FlagAbortGen`. In these properties, *zero_Data*, *idle*, *send_flag*, *TxSendAbort*, and *TxDataWr* are inputs. *TxDout* is the output signal, and *flag_done*, *abort_set*, and *idle_set* are internal signals.

B.1.5 Transmitter controller

The controller can be in the following states:

1 idle mode: when transmitter is enabled ($TxEnable = 1$), it is in idle mode. In this mode the controller sets the *idle* signal to 1 (to send consecutive 1 to the output). As soon as a valid data is available ($TxDataWr = 1$), *idle* becomes 0. If it is between the frames, in the next cycle an opening flag should be sent ($send\_flag = 1$).

2 sending a new frame: the first frame is sent when transmitter becomes enable, and a valid data is available. The next frame can be sent immediately, or can be sent several cycles after finishing the current frame (in this case, the transmitter is idle between the frames).

i sending opening flag: the transmitter controller sets *send_flag* to 1 to start sending a new frame. It may happen after idle mode of the transmitter controller, or just after the closing flag of the previous frame.

ii sending data: to send a new data, controller should set the *load* signal to 1. If it is the first data of the frame, *load* becomes 1 after specific number of cycles after *send_flag*= 1. Then, *load* becomes 1 after specific number of cycles after the

```
vunit CRCGen
{
  P0_init_data:
    not crc_busy and not crc_done and (R16 = ”0000000000000000”) and (
        crc_counter = ”00000”);

  P1_crc_16_data_2_ns:
    always (not abort_set and not stall and not send_crc and not crc_busy
        and (TxCRCSel = ”01”) and R16(15) -> next!(R16(15 downto 1) = (
        poly16(15 downto 1) XOR (prev(R16(14 downto 0)) ) ) ) and next!(R16
        (0) = (poly16(0) XOR prev(S_Data) )));

  P2_crc_16_data_1_ns:
    always (not abort_set and not stall and not send_crc and not crc_busy
        and (TxCRCSel = ”01”) and not R16(15) -> next!(R16(15 downto 1) =
        prev(R16(14 downto 0))) and next!(R16(0) = prev(S_Data)));

  P3_crc_16_data_1_s:
    always (not abort_set and stall and not send_crc and not crc_busy and (
        TxCRCSel = ”01”) -> next!(R16 = prev(R16)) );

  P4_CRC_16_send_data:
    always (not abort_set and not crc_busy and not send_crc  and (
        crc_counter < ”01111”) and (TxCRCSel = ”01”) -> (crc_data = S_Data))
        ;

  P5_send_crc_16_nstall:
    always (not abort_set and not stall and crc_busy and (TxCRCSel = ”01”)
        and (crc_counter < ”01111”)-> next!(crc_busy) and next!(R16(0) =
        ’0’) and next!(crc_counter = (prev(crc_counter) + ”00001”)) and next
        !(R16(15 downto 1) = prev(R16(14 downto 0))) and (crc_data =  (R16
        (15)) ) );

  P6_send_crc_16_stall:
    always (not abort_set and stall and crc_busy and (TxCRCSel = ”01”) and
        (crc_counter < ”01111”)-> next!(crc_busy ) and next!(crc_counter =
        prev(crc_counter)) and (crc_data = ’0’) and next!(R16 = prev(R16)));

  P7_crc_16:
    always (not abort_set and send_crc and (TxCRCSel = ”01”) -> crc_busy
        and (crc_counter = ”00000”));

  P8_done_crc_16:
    always (not abort_set and crc_busy and (TxCRCSel = ”01”) and (
        crc_counter = ”01111”)-> next!(not crc_busy and crc_counter = (prev(
        crc_counter) + ”00001”)) and next!(R16(15 downto 1) = prev(R16(14
        downto 0))) and next!(R16(0) = ’0’) and (crc_data =  (R16(15))));

  P9_abort:
    always (abort_set -> next!(not crc_busy) and next!(R16(15 downto 1) = ”
        0000000000000000”) and next!(crc_counter = ”00000”));

  P10_crc_done:
    always (fell(crc_busy) -> crc_done and next! (not crc_done) and next!(
        crc_counter = ”00000”));
}
```

FIGURE B.3: Properties that describe `CRCGen` (for 16-bit CRC)

```
vunit ZeroInsertion
{
  P0_init:
    ((stall = '0') and (zero_Inserted = '0') and (zero_Data = '0'));

  P1_Zero:
    always(
      prev(prev(prev(prev(crc_Data)))) and prev(prev(prev(crc_Data))) and
          prev(prev(crc_Data)) and prev(crc_Data) and crc_Data
      ->
      next!(stall and  zero_Data)
      );

  P2_noZero:
    always(
      not prev(prev(prev(prev(crc_Data)))) or not prev(prev(prev(crc_Data))
          ) or not prev(prev(crc_Data)) or not prev(crc_Data) or not
          crc_Data
      ->
      next!(not stall) and next! (not zero_Inserted) and next!((zero_Data =
           prev(crc_Data)))
      );
}
```

Figure B.4: FL properties that describe `ZeroInsertion`

```
vunit FlagAbortGen
{
  P0_T_FA_init:
    flag_done and not abort_set and not TxDout;

  P1_send_flag:
    always(rose(send_flag) -> not flag_done and next_a![1 to 7](not
        flag_done) and not TxDout and next_a![1 to 6](TxDout) and next![7](
        not TxDout) and next![8](not send_flag -> flag_done));

  P2_send_data:
    always(flag_done and not abort_set and not send_flag and not idle-> (
        TxDout = zero_Data));

  p3_idle:
    always(idle and not send_flag and flag_done -> TxDout);

  P4_TxAbort:
    always(TxSendAbort -> (TxDout and abort_set) and next_a![1 to 7](
        abort_set and TxDout) and next![8](not abort_set));
}
```

Figure B.5: Properties that describe `FlagAbortGen`

previous load (the number of the cycles is $8 + no.\ of\ the\ stalls$). Transmitter specifies the last data of a frame by asserting *TxEoF*. After that, no new data is loaded until there is a new frame.

iii sending CRC: after transmitting the last data of a frame, transmitter controller sets the *send_crc* signal to 1. Then, based on the value of *TxCRCSel* signal, transmitter should wait for a specific number of cycles until *crc_done* becomes 1.

iv sending closing flag: when *crc_done* becomes 1, *send_flag* is set to 1 to send the closing flag.

3 abort command: if $TxSendAbort = 1$, data transmission is stopped immediately, and after eight cycles, a closing flag should be sent ($send\_flag = 1$).

Figure B.6 shows the PSL properties for the transmitter controller.

B.2 Receiver

Figure B.1 (b) shows the block diagram of the HDLC receiver.

Before describing the functional behavior of the system, the transmitter interface signals have been introduced:

- Inputs
  - *RxClk*: the actual clock of the receiver
  - *RxRstn*: the active-hight reset signal
  - *RxEnable*: Specifies if the receiver is enable
  - *RxCRCSel*: Specifies if CRC is 8, 16, or 32 bits. The value of this signal is specified by the user when the receiver is not enable.
- Outputs
  - *RxDataAvail*: If this signal is 1 it means that the data of *RxDout* is valid. This signal is set to 1 for a cycle whenever `S2P` finishes paralleling the data.
  - *RxStartOfFrame*: This signal is set to 1 whenever an opening flag is detected, and the first data of the frame is available.
  - *RxEndOfFrame*: This signal specifies the received data is the last data of the current frame.
  - *RxCRCValue*: Shows the CRC value of the received data.
  - *RxCRCError*: This output signal shows if the data has been transmitted correctly.
  - *RxAbortFound*: This signal specifies if an abort sequence has been detected.

The receiver gets the serial HDLC frames continuously through the *RxData* port. When an opening flag is recognized, the receiver indicates the beginning of the frame by asserting *StartOfFrame*. Bytes will be passed to the user until a closing flag or abort sequence is detected. At this point, the last byte will be passed and the *EndOfFrame* signal is asserted. As is shown in Fig. B.1 (b), the receiver has several sub-modules: `FlagAbortDet`, `ZeroDet`, `CRCheck`, `S2P`, and `RController`.

```
vunit T_Controller
{
  P0_T_C_reset:
    not TxLastBit and not load and not send_flag and idle and (load_counter
        = ”0000”) and not eof and not send_crc and (TxCRCValue = ”
       00000000000000000000000000000000”) and not TxCRCValAvail  and (
       data_counter = ”000”) and not (TxUnderRun) and not T_frame_valid;

  P1_T_enable_idle:
    always (rose(TxEnable) and not TxDataWr -> not send_flag and not eof
       and not send_crc);

  P2_frame_after_idle:
    always (idle and TxEnable and rose(TxDataWr) -> next!(not idle and
       send_flag));

  P3_load_first_data:
    always (rose(send_flag) and not prev(crc_done) and not prev(abort_set)
       and TxDataWr and not TxShareFlag -> not load and (load_counter = ”
       0000”) and next![7](load) and next_a![1 to 7](load_counter = ”0000”)
         and next![7](TxEnable -> T_frame_valid));

  P4_T_frame_valid:
    always((T_frame_valid) and not TxSendAbort and not eof and not idle and
        TxEnable-> next!(T_frame_valid));

  P5_disable_mid_fr:
    always(T_frame_valid and not TxSendAbort and not eof and not idle and
       not TxEnable and (load_counter < ”0111”)-> next!(T_frame_valid));

  P6_T_frame_not_valid:
    always(send_crc or TxSendAbort or idle or (not TxEnable and (
       load_counter = ”0111”)) -> next! (not T_frame_valid and not load));

  P7_load_data_nstall:
    always(((load_counter < ”0111”) and (load_counter > ”0000”) and not
       stall and T_frame_valid) and not TxSendAbort-> next!(not TxSendAbort
        -> (load_counter= (prev(load_counter)+”0001”))) and (not load));

  P8_load_data_nstall_2:
    always((load_counter = ”0000”) and not stall and T_frame_valid and not
       TxSendAbort-> next!(load_counter= (prev(load_counter)+”0001”)) and (
       TxDataWr -> load));

  P9_load_data_wstall:
    always (load_counter < ”0111” and (load_counter > ”0000”) and stall and
        not eof and  T_frame_valid-> next!(T_frame_valid -> load_counter =
       (prev(load_counter))) and next!(not load));

  P10_load_data_wstall_2:
    always ((load_counter = ”0000”) and stall and not eof and T_frame_valid
         -> next!(T_frame_valid -> load_counter = (prev(load_counter))) and
        (not load) and next!(T_frame_valid and TxDataWr -> load));
```

```
P11_load_new_data:
  always ((load_counter = ”0111”) and TxDataWr and not eof and
      T_frame_valid -> next!(((load_counter = ”0000”))));

P12_set_counter_no_load:
  always ((load_counter = ”0111”) and not T_frame_valid -> next!((
      load_counter = ”0000”) until! (not rose(send_flag))));

P13_end_crc_close_flag:
  always (rose(crc_done) -> next!(send_flag and TxLastBit) and next_a![1
      to 7]((not T_frame_valid)) );

P14_idle_between_frames:
  always (rose(crc_done) -> next_a![1 to 8](not idle) and next![9]((not
      TxDataWr -> idle)));

P15_new_frame_share_flag:
  always (rose(crc_done) and TxShareFlag -> next![6](TxDataWr ->
      T_frame_valid));

P16_new_frame_open_flag:
  always (rose(crc_done) and not TxShareFlag -> next!(load_counter = ”
      0000”) and next!(TxCRCValue = ”00000000000000000000000000000000”)
      and next![9](next_event(TxDataWr and not idle)(send_flag and not
      load)));

P17_deaasert_send_flag:
  always (send_flag -> next!(not send_flag));

P18_eof:
  always (TxDataWr and TxEoF and load and T_frame_valid -> next![3](eof)
      and next![3](data_counter = ”000”));

P19_keep_eof_no_stall:
  always (eof and not stall and (data_counter < ”101”) and T_frame_valid
      -> next!(eof) and next!((data_counter = (prev(data_counter) + ”001”)
      )) );

P20_keep_eof_with_stall:
  always (eof and stall and (data_counter < ”101”) and T_frame_valid->
      next!((eof = ’1’)) and next!((data_counter = prev(data_counter))) );

P21_eof_start_crc:
  always (eof and (data_counter = ”101”) and T_frame_valid-> next!(not
      eof and (data_counter = ”000”)) and send_crc );

P22_eof_start_crc_nf:
  always (eof and (not T_frame_valid )-> next!(not eof and (data_counter
      = ”000”)) and send_crc );

P23_deaasert_send_crc:
  always (send_crc -> next! (not send_crc and not TxCRCValAvail) and next
      !((load_counter = ”0000”)));
  P24_abort_close_flag:
    always (TxSendAbort -> next![8](send_flag and TxLastBit) and next!(
        load_counter = ”0000”) and next_a![1 to 16](not idle) and next
        ![17]((idle)) and next_a![1 to 17](not T_frame_valid) and next![9](
        next_event(send_flag)[8](T_frame_valid)));

  P25_crc_val_16:
    always(rose(send_crc) and (TxCRCSel = ”01”) -> (TxCRCValue(15 downto 0)
         = R16) and (TxCRCValue(31 downto 16) = ”0000000000000000”) and
        TxCRCValAvail);

  P26_crc_val_32:
    always(rose(send_crc) and (TxCRCSel = ”10”) -> (TxCRCValue = R32) and
        TxCRCValAvail);

  P27_crc_val_init:
    always (rose(crc_done) -> next!(TxCRCValue = ”
        00000000000000000000000000000000”) );

  P28_last_bit:
    always (TxLastBit -> next!(not TxLastBit));

  P29_T_disbale:
    always ( T_frame_valid and ((fell(TxEnable) and (load_counter /= ”0000”
        )) or (fell(TxEnable) and (load_counter = ”0000”) and load )) ->
        next!(next_event(load_counter = ”0111”)(eof)) );

  P30_between_data:
    always ((load_counter > ”0111”) and (load_counter < ”1111”) and not
        TxDataWr and not eof and T_frame_valid and TxEnable-> next!(((
        load_counter = prev(load_counter) + ”0001”))));

  P31_underrun:
    always ((load_counter = ”1111”) and not TxDataWr and not eof and
        T_frame_valid and TxEnable-> next!(load_counter = ”0000”) and next!(
        TxUnderRun));

  P32_new_data:
    always ((load_counter > ”0111”) and (load_counter < ”1111”) and rose(
        TxDataWr) and not eof and T_frame_valid and TxEnable-> next!(
        load_counter = ”0000”));

  P33_not_underrun:
    always(TxUnderRun -> next! (not TxUnderRun));

}
```

FIGURE B.6: Properties that describe transmitter controller

B.2.1 Flag and abort detection

The receiver begins the operation by detecting the opening flag through the `FlagAbortDet` module. Once an opening flag is detected, the receiver begins to receive the incoming frame, while `FlagAbortDet` is monitoring the frame for a closing flag. Figure B.7 shows the FL properties for the `FlagAbortDet` module. In these properties, *RxEnable* and *RxData* are inputs. The serial data is put on the *R_S_Data* output port. The output signals *IsFlag*, *IsAbort*, and *AbortFound* indicate if a flag or abort sequence is detected. *data_seq* is an internal signal that stores 8 consecutive bits of the incoming data.

```
vunit FlagAbortDet
{
  P0_R_FA_init:
    not R_S_Data and (data_seq = ”00000000”) and not AbortFound;

  P1_R_FA_receive_data:
    always (RxEnable -> ( R_S_Data = data_seq(7)));

  P2_R_FA_store_data:
    always (RxEnable -> next!( data_seq(0) = RxData) and next!(data_seq(7
        downto 1) = prev(data_seq(6 downto 0))));

  P3_R_FA_is_abort:
    always ((data_seq = ”11111111”) -> IsAbort and (AbortFound until! (fell
        (IsFlag) )) and (next_event(fell(IsFlag))(not AbortFound)));

  P4_R_FA_is_not_abort:
    always ((data_seq /= ”11111111”) -> not IsAbort);


  P5_R_FA_is_flag:
    always ((data_seq = ”01111110”) -> IsFlag and not IsAbort);

  P6_R_FA_is_not_flag:
    always ((data_seq /= ”01111110”) -> not IsFlag);
}
```

FIGURE B.7: SERE properties that describe `FlagAbortDet`

Figure. B.8 shows the SERE properties for `FlagAbortDet`.

B.2.2 Zero detection

The `ZeroDetection` module checks the incoming data from `FlagAbortDet` to verify if a zero is inserted after five consecutive 1s. In this situation, the inserted 0 is deleted from the incoming frame, and *R_zero_Inserted* is asserted. Figure B.9 shows the FL properties for the `ZeroDetection` module. In this module, *R_S_Data* and *RxEnable* are inputs, and *R_stall*, *R_zero_Inserted*, and *R_zero_Data* are outputs. *data_seq* is an internal signal to buffer the 7 consecutive bits of the incoming data.

Figure B.10 shows the SERE properties for `ZeroDetection`.

```
vunit FlagAbortDet_sere
{
  P0_R_FA_init:
    not R_S_Data and not AbortFound;

  P1_is_flag:
    always ({RxEnable and not R_S_Data; R_S_Data[*6]; not R_S_Data} |-> {
        IsFlag and not IsAbort});

  P2_is_abort:
    always ({RxEnable and not R_S_Data; R_S_Data[*7]} |-> {{IsAbort} & {
        AbortFound[*]; fell(IsFlag) }});
}
```

Figure B.8: SERE properties that describe `FlagAbortDet`

```
vunit ZeroDetection
{
  P0_R_Z_init:
    not R_zero_Data and not R_stall and not R_zero_Inserted and (data_seq =
         ”0000000”);

  P1_R_Z_receive_data: -- P1_send_data:
    always (RxEnable -> ( R_zero_Data = R_S_Data ));

  P2_R_Z_store_data: -- P2_store_data:
    always (RxEnable -> next!( data_seq(0) = R_S_Data) and next!(data_seq(6
         downto 1) = prev(data_seq(5 downto 0))));

  P3_R_Z_zero_inserted:
    always ((data_seq = ”0111110”) -> R_zero_Inserted and R_stall);

  P4_R_Z_zero_not_inserted:
    always ((data_seq /= ”0111110”) -> not R_zero_Inserted and not R_stall)
        ;
}
```

Figure B.9: FL properties that describe `ZeroDetection`

```
vunit ZeroDetection_sere
{
  P0_R_Z_init:
    not R_zero_Data and not R_stall and not R_zero_Inserted;
  P1_zero_detection:
    always ({RxEnable and not R_S_Data; R_S_Data[*5]; not R_S_Data} |->
        R_zero_Inserted and R_stall);
}
```

Figure B.10: SERE roperties that describe `ZeroDetection`

B.2.3 CRC checker

The `CRCCheck` module performs the same generator polynomial division as the transmitter across the entire transmitted message, including the CRC field. The remainder is then compared to the remainder of the `CRCGen` module. Figure B.11 shows the specification of the `CRC_Checker` module (for 16-bit CRC). In these properties, *RxCRCSel*, *poly16*, *R_stall*, *R_zero_Data*, and *R_crc_check* are the inputs of `CRCCheck`. *R_crc_data*, *R_crc_error*, and *R_R16* are the outputs of the module. An internal signal is also defined (*crc_buffer*) for storing the incoming serial data.

B.2.4 Serial to Parallel converter

Figure B.12 shows the PSL properties for the `S2P` module. In this module, *s2p_enable*, *s2p_disable*, *R_stall*, and *R_crc_data* are inputs, and *RxDout* and *s2p_done* are outputs. *R_load_counter*, *s2p_buffer*, and *frame_valid* are the internal signals.

B.2.5 Receiver Controller

Figure B.13 shows the properties for synthesizing receiver controller. When the receiver is enable, the controller waits for receiving one of these signals: *IsAbort* or *IsFlag*.

If an abort sequence is not detected ($IsAbort = 0$), *RxAbortFound* is set to 0. Otherwise, the receiver looks for a closing flag, and sets the *RxAbortFound* signal to 1, and this signal remains 1 until receiving the closing flag.

If a flag is detected ($IsFlag = 1$), the receiver should specify if a detected flag is an opening flag (a flag sequence that is followed by a non-flag sequence), or a closing flag. To detect the opening and closing flags, an internal signal is defined: *R_flag_counter*. Initially, this signal is set to 0. After receiving the first flag, this signal is set to 1, which indicates a flag has already been detected. So, the next detected flag would be the closing flag. When a closing flag is detected, *R_flag_counter* is again set to 0.

When an opening flag is detected, the `CRCCheck` module should become enable after 8 clock cycles. Then, receiver waits for the *R_crc_error* signal, and reads this signal in the cycle in which the *RxEndOfFrame* is 1 to detect if an error exists in the received data. Then, controller enables the `StoP` module, and wait for receiving *s2p_done* that specifies if the `S2P` module has finished paralleling 8 bits of data.

```
vunit CRCCheck
{
  P0_R_crc_init_data:
    always (RxEnable and rose(R_crc_check) -> not R_crc_error and (R_R16 =
        ”0000000000000000”) and (crc_buffer = ”
        00000000000000000000000000000000”));

  P1_R_crc_16_data_2_ns:
    always (RxEnable and not R_stall and R_crc_check and (RxCRCSel = ”01”)
        and R_R16(15) -> next!(R_R16(15 downto 1) = (poly16(15 downto 1) XOR
         (prev(R_R16(14 downto 0)) ) ) ) and next!(R_R16(0) = (poly16(0) XOR
         prev(crc_buffer(15) ) )));

  P2_R_crc_16_data_2_s:
    always (RxEnable and R_stall and R_crc_check and (RxCRCSel = ”01”)  ->
        next!(R_R16 = (prev(R_R16) )));

  P3_R_crc_16_data_1_ns:
    always (RxEnable and not R_stall and R_crc_check and (RxCRCSel = ”01”)
        and not R_R16(15) -> next!(R_R16(15 downto 1) = prev(R_R16(14 downto
         0))) and next!(R_R16(0) = prev(crc_buffer(15) )));

  P4_R_crc_16_shift_ns:
    always (RxEnable and not R_stall and R_crc_check and (RxCRCSel = ”01”)
        -> next!(crc_buffer(31 downto 1) = prev(crc_buffer(30 downto 0)))
        and next!(crc_buffer(0) = prev(R_zero_Data) ) );

  P5_R_crc_16_shift_s:
    always (RxEnable and R_stall and R_crc_check and (RxCRCSel = ”01”) ->
        next!(crc_buffer(31 downto 1) = prev(crc_buffer(31 downto 1))) and
        next!(crc_buffer(0) = prev(crc_buffer(0)) ) );

  P6_R_crc_16_send_data:
    always (RxEnable and R_crc_check and (RxCRCSel = ”01”) -> (R_crc_data =
         R_zero_Data) );

  P7_R_crc_16_error:
    always (RxEnable and R_crc_check and (crc_buffer(15 downto 0) /= R_R16)
         and (RxCRCSel = ”01”) -> R_crc_error);

  P8_R_crc_16_not_crc_err:
    always (RxEnable and R_crc_check and (crc_buffer(15 downto 0) = R_R16)
        and (RxCRCSel = ”01”) -> not R_crc_error);

  P9_R_crc_disable:
    always(not RxEnable or not R_crc_check -> R_crc_data = ’0’);
}
```

FIGURE B.11: Properties that describe `CRCCheck` (for 16-bit CRC)

```
vunit S2P
{
  P0_R_S2P_init:
    not frame_valid and not s2p_done and (R_load_counter = ”0000”) and (
        s2p_buffer = ”00000000” and (RxDout = ”00000000”);

  P1_R_S2P_frame:
    always (rose(s2p_enable) -> (frame_valid until! (s2p_disable)));

  P2_R_S2P_frame:
    always (rose(s2p_disable) -> (not frame_valid) until! (s2p_enable));

  P3_R_S2P_shift:
    always (not R_stall and (frame_valid) and (R_load_counter < ”1000”)->
        next!(s2p_buffer(7) = prev(R_crc_data)) and next!(s2p_buffer(6
        downto 0) = prev(s2p_buffer(7 downto 1)) ) and next!((R_load_counter
         = prev(R_load_counter) + ”0001” ) ) );

  P4_R_S2P_no_valid_frame:
    always (not (frame_valid) -> next!(s2p_buffer = ”00000000”) and next!((
        R_load_counter = ”0000”)) );

  P5_R_S2P_stall:
    always (R_stall and (frame_valid) and (R_load_counter < ”1000”)-> next
        !(R_load_counter = prev(R_load_counter)) and next!(s2p_buffer = prev
        (s2p_buffer)));

  P6_R_S2P_data_ready_ns:
    always (not R_stall and (R_load_counter = ”1000”) -> next!(
        R_load_counter = ”0001”) and (RxDout = s2p_buffer) and s2p_done and
        next!(RxDout = ”00000000”));

  P7_R_S2P_data_ready_s:
    always ((R_stall) and (R_load_counter = ”1000”) -> next!(R_load_counter
         = ”0000”) and (RxDout = s2p_buffer) and s2p_done and next!(
        s2p_buffer = prev(s2p_buffer)) );

  P8_R_S2P_not_done:
    always(rose(s2p_done) -> next!(not s2p_done) );
}
```

FIGURE B.12: Properties that describe S2P

```
vunit RController
{
  P0_R_Ctr_init:
    not R_crc_check and not R_crc_error and not s2p_enable and not
        s2p_disable and not RxDataAvail and (R_flag_counter = ”00”) and not
        RxEndOfFrame and not RxStartOfFrame and not RxAbortFound and (
        RxCRCValue = ”00000000000000000000000000000000”);

  P1_R_Ctr_s2p_en:
    always (rose(R_crc_check) -> (s2p_enable));

  P2_R_Ctr_s2p_not_en:
    always (rose(s2p_enable) -> next! (not s2p_enable));

  P3_R_Ctr_open_flag:
    always (IsFlag and (R_flag_counter = ”00”) -> next! (R_flag_counter = ”
        01”) and next![8](R_crc_check));

  P4_R_Ctr_close_flag:
    always (IsFlag and (R_flag_counter = ”01”) -> next!(s2p_disable and not
         R_crc_check) and next!(R_flag_counter = ”00”));

  P5_R_Ctr_s2p_not_dis:
    always (rose(s2p_disable) -> next! (not s2p_disable));

  P6_R_Ctr_data_rdy:
    always (s2p_done -> RxDataAvail);

  P7_R_Ctr_data_not_rdy:
    always (rose(RxDataAvail) -> next! (not RxDataAvail));

  P8_R_Ctr_SoF:
    always (IsFlag and (R_flag_counter = ”00”) -> next![8](not IsAbort and
        not AbortFound and not IsFlag -> next_event (RxDataAvail)(
        RxStartOfFrame)));

  P9_R_Ctr_not_SoF:
    always (rose(RxStartOfFrame) -> next! (not RxStartOfFrame));

  P10_R_Ctr_EoF_n_abort:
    always (RxStartOfFrame and (R_flag_counter = ”01”) and not RxAbortFound
         -> next_event (RxDataAvail and IsFlag)(RxEndOfFrame));

  P11_R_Ctr_EoF_abort:
    always (IsFlag and R_flag_counter = ”01” and RxAbortFound->RxEndOfFrame
        );

  P12_R_Ctr_not_EoF:
    always (rose(RxEndOfFrame) -> next! (not RxEndOfFrame));

  P13_R_Ctr_CRC_Err:
    always ((rose(RxEndOfFrame) and R_crc_error) -> (RxCRCError) and next!
        (not RxCRCError));
  P14_R_Ctr_not_CRC_Err:
    always (rose(RxStartOfFrame) -> (not RxCRCError) until! (RxEndOfFrame
        and R_crc_error) );

  P15_R_Ctr_CRC_value:
    always (RxEndOfFrame and (RxCRCSel = "01")-> (RxCRCValue(15 downto 0) =
         R_R16) and next!(RxCRCValue = "00000000000000000000000000000000") )
        ;

  P16_R_abort_found:
    always ((R_flag_counter /= "00") -> (next!( RxAbortFound = AbortFound))
        );
}
```

FIGURE B.13: Properties that describe `RController`

Appendix C

Case study: Advanced Microcontroller Bus Architecture

The Advanced Microcontroller Bus Architecture (AMBA) [AMB] is a chip communication standard that has been developed to design high-performance embedded microcontrollers. One category of AMBA bus is Advanced High-Performance Bus (AHB).

The AMBA AHB bus is used to interconnect and communicate high-clock frequency modules, like high- performance processors or on-chip memories. A basic bus is composed of 4 different modules: one or multiple masters, one or multiple slaves, an arbiter and a decoder (see Fig. C.1).

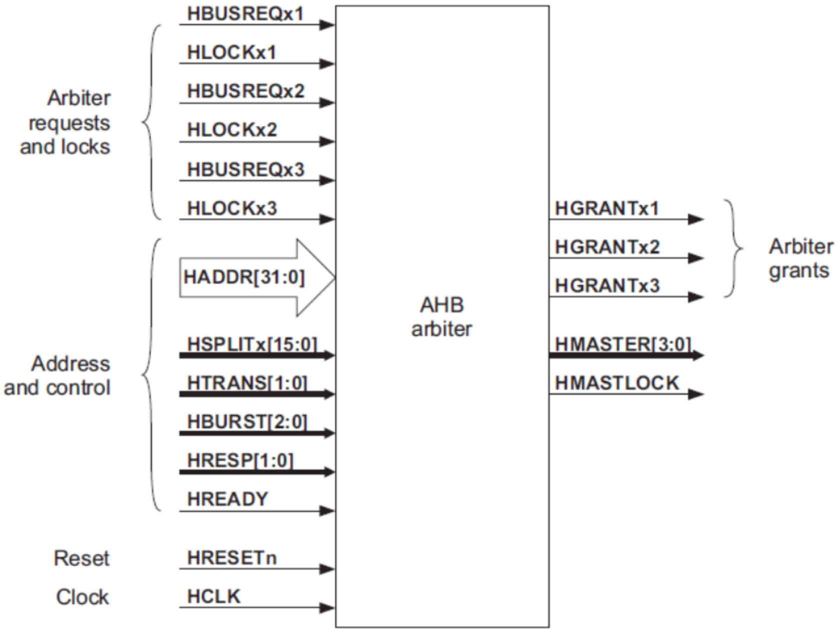


FIGURE C.1: The AMBA arbiter block diagram (the figure is taken from [AMB])

The masters will send an address and some control signals to indicate which slave they want to communicate with, and which type of transfer they are going to do. The master is the element (for example a peripheral or the control unit of a processor) that can take control of the bus and send or request information to other elements connected to the bus. To obtain the control of the bus, the master must send a request to the arbiter and wait until it is responded with a grant signal. The role of the arbiter is to decide which master should take the control of the bus in a specific cycle, and depending on the type of the transfer, arbiter should also decide for how long this master has the control of the bus. The task of the decoder is to analyze the address given by the master and decide which slave should be selected. Each slave has a set of memory spaces available to read or write data, if the address sent by the master corresponds to one of these spaces the slave is selected to make the transfer. The slave should also send an answer to the master whenever a transfer is finished to see if it was successful or not.

Figure C.2 shows the FL properties that describe AMBA arbiter. The original properties are taken from [GCH], and have been rewritten to be used in SyntHorus2. In addition to the input/output signals shown in Fig. C.1, some internal signals have been defined and used in the properties:

- *BUSREQ*: becomes 1 whenever there is a request from one of the masters,
- *DECIDE*: indicates the cycle in which the arbiter decides who is the next master,
- *GRANTED*: is used for deciding the start of the new access to the bus,
- *mast_j*: if this signal is 1, it implies that *HMASTER* = j.

```
vunit arbiter{
  P0_G1_1_m0:
    always(HBUSREQ_0 and mast_0 -> BUSREQ);
  P1_G1_2_m0:
    always(not HBUSREQ_0 and mast_0 -> not BUSREQ);
  P2_G1_1_m1:
    always(HBUSREQ_1 and mast_1 -> BUSREQ);
  P3_G1_2_m1:
    always(not HBUSREQ_1 and mast_1 -> not BUSREQ);
  P4_G4_m0_m1:
    always (DECIDE and (HBUSREQ_0 or HBUSREQ_1) ->next!(GRANTED));
  P5_G5_1:
    always (GRANTED and HREADY -> next!(not GRANTED));
  P6_G5_2:
    always (GRANTED and not HREADY -> next!(GRANTED));
  P7_G6_1_m0:
    always (HREADY and HGRANT_0 -> next!(mast_0));
  P8_G6_2_m0:
    always (HREADY and not HGRANT_0 -> next!(not mast_0));
  P9_G6_1_m1:
    always (HREADY and HGRANT_1-> next!(mast_1));
  P10_G6_2_m1:
    always (HREADY and not HGRANT_1-> next!(not mast_1));
  P11_G7_m0_m1:
    always ((HREADY and ((HLOCK_0 and HGRANT_0) or (HLOCK_1 and HGRANT_1))
        -> next!(HMASTLOCK)));
  P12_G8_1_1_m0:
    always ((not HREADY or not GRANTED) and mast_0_m-> next!(mast_0));
  P13_G8_1_2_m0:
    always ((not HREADY or not GRANTED) and (not mast_0)-> next!(not mast_0
        ));
  P14_G8_1_1_m1:
    always ((not HREADY or not GRANTED) and mast_1-> next!(mast_1));
  P15_G8_1_2_m1:
    always ((not HREADY or not GRANTED) and (not mast_1)-> next!(not mast_1
        ));
  P16_G8_2_1_m0_m1:
    always (not HREADY or not GRANTED and HMASTLOCK-> next!(HMASTLOCK));
  P17_G8_2_2_m0_m1:
    always (not HREADY or not GRANTED and not HMASTLOCK-> next!(not
        HMASTLOCK));
  P18_G9_1_m0:
    always (not DECIDE and HGRANT_0-> next!(HGRANT_0));
  P19_G9_2_m0:
    always (not DECIDE and not HGRANT_0-> next!(not HGRANT_0));
  P20_G9_1_m1:
    always (not DECIDE and HGRANT_1-> next!(HGRANT_1));
  P21_G9_2_m1:
    always (not DECIDE and not HGRANT_1-> next!(not HGRANT_1));
  P22_G10_1_m1:
    always (not HGRANT_1 -> next! ((not HGRANT_1) until! HBUSREQ_1));
  P23_G10_2:
    always (DECIDE and (not HBUSREQ_0) and (not HBUSREQ_1) -> next! (
        HGRANT_0));
  P24_G12:
    DECIDE and HGRANT_0 and ((not HMASTER_0) and (not HMASTER_1) and (not
        HMASTER_2) and (not HMASTER_3)) and not GRANTED and not HMASTLOCK
        and not HGRANT_1;
  P25_gen_decide:
    always (((HBUSREQ_0 or HBUSREQ_1) or (HLOCK_0 or HLOCK_1)) and HREADY
        and (not (GRANTED) and not DECIDE)-> next!(DECIDE));
  P26_gen_not_decide:
    always(DECIDE -> next! (not DECIDE));
  P27_grant0:
    always (((GRANTED) and (not prev(HGRANT_0)) and (not HLOCK_1)) or (
        HLOCK_0 and prev(HGRANT_0))-> HGRANT_0);
  P28_grant1:
    always (((GRANTED) and (not prev(HGRANT_1)) and (not HLOCK_0)) or (
        HLOCK_1 and prev(HGRANT_1))-> HGRANT_1);
  P29_not_HMASTER1:
    always not HMASTER_1 and not HMASTER_2 and not HMASTER_3;
  P30_mast_0:
    always mast_0-> not HMASTER_0;
  P31_mast_1:
    always mast_1-> HMASTER_0;
  P32_one_grant:
    always not HGRANT_0 or not HGRANT_1;
}
```

Figure C.2: Annotated FL specification of AMBA arbiter (for 2 masters and 2 slaves)

Appendix D

The Annotation Results

Contents

D.1 IBM Generalized Buffer

D.1.1 Communication with senders

D.1.1.1 FL properties

```
vunit genbuf_sender
{
  P0_sender_0:
    always ((not BtoS_ACK_m(0)) and (not StoB_REQ_m(0)) -> next! (not
        BtoS_ACK_g(0)));

  P0_sender_1:
    always ((not BtoS_ACK_m(1)) and (not StoB_REQ_m(1)) -> next! (not
        BtoS_ACK_g(1)));

  P1_sender_0:
    always ((BtoS_ACK_m(0) and StoB_REQ_m(0)) -> next! (BtoS_ACK_g(0)));

  P1_sender_1:
    always ((BtoS_ACK_m(1) and StoB_REQ_m(1)) -> next! (BtoS_ACK_g(1)));

  P2_sender_0:
    always (rose(StoB_REQ_m(0)) -> not BtoS_ACK_g(0));

  P2_sender_1:
    always (rose(StoB_REQ_m(1)) -> not BtoS_ACK_g(1));

  P3_sender:
    always ( not BtoS_ACK(0) or not BtoS_ACK(1) );

---------- FIFO interface
  P4_FIFO_sender:
    always (not ENQ or BtoS_ACK(0)  or BtoS_ACK(1));

  P5_sere_FIFO_sender_0:
    always (not BtoS_ACK_m(0)}-> next!(ENQ or not BtoS_ACK(0)));

  P5_sere_FIFO_sender_1:
    always (not BtoS_ACK_m(1)}-> next!(ENQ or not BtoS_ACK(1)));

  P6_FIFO_sender:
    always ( (StoB_REQ_m(0) or StoB_REQ_m(1)) and (not FULL_m) and (not
       ENQ_m) -> next! (ENQ_g) and next![2](not ENQ_g) );

  P7_FIFO_sender_0:
    always (rose(BtoS_ACK_m(0)) -> SLC_g = 0);

  P7_FIFO_sender_1:
    always (rose(BtoS_ACK_m(1)) -> SLC_g = 1);
}
```

Figure D.1: Annotated FL specification of the sender side of GenBuf (2 senders)

D.1.1.2 SERE properties

```
vunit genbuf_sender_sere
{
  P0_sere_sender_0:
    always ({not BtoS_ACK_m(0) and not StoB_REQ_m(0)} |=> {not BtoS_ACK_g
        (0)}!);

  P0_sere_sender_1:
    always ({not BtoS_ACK_m(1) and not StoB_REQ_m(1)} |=> {not BtoS_ACK_g
        (1)}!);

  P1_sere_sender_0:
    always ({(BtoS_ACK_m(0) and StoB_REQ_m(0))} |=> {(BtoS_ACK_g(0))}!);

  P1_sere_sender_1:
    always ({(BtoS_ACK_m(1) and StoB_REQ_m(1))} |=> {(BtoS_ACK_g(1))}!);

  P2_sere_sender_0:
    always ({not StoB_REQ_m(0); StoB_REQ_m(0)} |-> {not BtoS_ACK_g(0)});

  P2_sere_sender_1:
    always ({not StoB_REQ_m(1); StoB_REQ_m(1)} |-> {not BtoS_ACK_g(1)});

  P3_sere_sender:
    always ( not BtoS_ACK(0) or not BtoS_ACK(1) );

—————— FIFO interface
  P4_sere_FIFO_sender:
    always (BtoS_ACK(0)  or BtoS_ACK(1) or not ENQ );

  P5_sere_FIFO_sender_0:
    always ({not BtoS_ACK_m(0)} |=> {ENQ or not BtoS_ACK(0)}!);

  P5_sere_FIFO_sender_1:
    always ({not BtoS_ACK_m(1)} |=> {ENQ or not BtoS_ACK(1)}!);

  P6_sere_FIFO_sender:
    always ({(StoB_REQ_m(0) or StoB_REQ_m(1)) and not FULL_m and not ENQ_m}
         |=> {ENQ_g; not ENQ_g}!);

  P7_sere_FIFO_sender_0:
    always ({not BtoS_ACK_m(0); BtoS_ACK_m(0)} -> SLC_g = 0);

  P7_sere_FIFO_sender_1:
    always ({not BtoS_ACK_m(1); BtoS_ACK_m(1)} -> SLC_g = 1);
}
```

Figure D.2: Annotated SERE specification of the sender side of GenBuf (2 senders)

D.1.2 Communication with receivers

D.1.2.1 FL properties

```
vunit genbuf_receiver
{
------ receiver side
  P0_rec:
    assert (always(not EMPTY_m -> next!(BtoR_REQ(0) or ( BtoR_REQ(1)))));

  P1_rec:
    assert (always(EMPTY_m -> next!(not BtoR_REQ_g(0) and (not BtoR_REQ_g
        (1))) ));

  P2_rec:
    assert (always (not BtoR_REQ(0) or not BtoR_REQ(1)));

  P3_rec_0:
    assert (always ( rose (BtoR_REQ_m(0)) -> next! (next_event! (prev(not
        BtoR_REQ_m(0))) (not BtoR_REQ_g(0) until_ (BtoR_REQ_m(1))))));

  P3_rec_1:
    assert (always ( rose (BtoR_REQ_m(1)) -> next! (next_event! (prev(not
        BtoR_REQ_m(1))) (not BtoR_REQ_g(1) until_ (BtoR_REQ_m(0))))));

  P4_rec_0:
    assert (always ((BtoR_REQ_m(0)) and (not RtoB_ACK_m(0))-> next! (
        BtoR_REQ_g(0))));

  P4_rec_1:
    assert (always ((BtoR_REQ_m(1)) and (not RtoB_ACK_m(1))-> next! (
        BtoR_REQ_g(1))));

  P5_rec_0:
    assert (always ( (RtoB_ACK_m(0)) -> (next! (not BtoR_REQ_g(0)))));

  P5_rec_1:
    assert (always ( (RtoB_ACK_m(1)) -> (next! (not BtoR_REQ_g(1)))));

---------- FIFO interface
  P6_FIFO_rec:
    assert (always (( fell(RtoB_ACK_m(0)) or (fell(RtoB_ACK_m(1))) and not
        EMPTY_m) -> (DEQ_g)));

  P7_FIFO_rec:
    assert (always ( not fell(RtoB_ACK_m(0)) and not fell(RtoB_ACK_m(1)) ->
          (not DEQ_g)));
}
```

FIGURE D.3: Annotated FL specification of the receiver side of GenBuf (2 receivers)

D.1.2.2 SERE properties

```
vunit genbuf_receiver_sere
{
  P0_sere_rec:
    always({not EMPTY_m} |=> {BtoR_REQ(0) or  BtoR_REQ(1)}!);

  P1_sere_rec:
    always({EMPTY_m} |=> {not BtoR_REQ_g(0) and not BtoR_REQ_g(1)}! );

  P2_sere_rec:
    always (not BtoR_REQ(0) or not BtoR_REQ(1));

  P3_sere_rec_0:
    always ( {not BtoR_REQ_m(0); BtoR_REQ_m(0); {BtoR_REQ_m(0)[*]; not
        BtoR_REQ_m(0)}} |=> {(not BtoR_REQ_g(0))[*]; (prev(BtoR_REQ_m(1)))
        }!);

  P3_sere_rec_1:
    always ( {not BtoR_REQ_m(1); BtoR_REQ_m(1); {BtoR_REQ_m(1)[*]; not
        BtoR_REQ_m(1)}} |=> {(not BtoR_REQ_g(1))[*]; (prev(BtoR_REQ_m(0)))
        }!);

  P4_sere_rec_0:
    always ( {BtoR_REQ_m(0) and (not RtoB_ACK_m(0))} |=> {BtoR_REQ_g(0)}!);

  P4_sere_rec_1:
    always ( {BtoR_REQ_m(1) and (not RtoB_ACK_m(1))} |=> {BtoR_REQ_g(1)}!);

  P5_sere_rec_0:
    always ( {RtoB_ACK_m(0)} |=> {not BtoR_REQ_g(0)}!);

  P5_sere_rec_1:
    always ( {RtoB_ACK_m(1)} |=> {not BtoR_REQ_g(1)}!);

  P6_sere_FIFO_rec:
    always ((fell(RtoB_ACK_m(0)) or (fell(RtoB_ACK_m(1))) and not EMPTY_m)
        -> (DEQ_g));

  P7_sere_FIFO_rec:
    always (not fell(RtoB_ACK_m(0)) and not fell(RtoB_ACK_m(1)) -> (not
        DEQ_g));

}
```

FIGURE D.4: Annotated SERE specification of the receiver side of GenBuf (2 receivers)

D.1.3 Communication with FIFO

```
vunit genbuf_FIFO
{
  P0_FIFO:
    always (FULL_m and not DEQ_m -> not ENQ_g);

  P1_FIFO:
    always (EMPTY_m -> not DEQ_g);
}
```

Figure D.5: Annotated FL specification of the FIFO side of GenBuf

D.2 HDLC

D.2.1 Transmitter

Figure. D.6 shows the annotated SERE specification of HDLC transmitter given in [PPSQ13]. We did not use these properties for generating the transmitter of HDLC, since they were not complete.

```
vunit assertions_Transmitter(Transmitter(A_Transmitter)){
  p160_a :
    always ({not T_frame_valid_m; T_frame_valid_m } |=>
      ((TxDout_g = prev(crc_Data_m)) until ((stall_m) or not prev(
          T_frame_valid_m))));
  sequence START_160_m is
    {(not crc_Data_m and T_frame_valid_m) or eof_m};

  sequence HOP_160_m is
    { ((not eof_m) and (TxSendAbort_m or (not T_frame_valid_m)))[*] };
  sequence REC_1_160_m is
    { (not eof_m and ((crc_Data_m and T_frame_valid_m) ))};
  sequence REC_1_END_160_m is
    { crc_Data_m and T_frame_valid_m };
  p160_b_1 :
    always({START_160_m; {HOP_160_m; REC_1_160_m}[*4]; HOP_160_m;
        REC_1_END_160_m } |=> {stall_g;not TxDout_g} );
  p160_b_2 :
    always({START_160_m; {HOP_160_m; REC_1_160_m}[*4]; HOP_160_m;
        REC_1_END_160_m; {HOP_160_m; REC_1_160_m; [*1]; {HOP_160_m;
        REC_1_160_m}[*3]; HOP_160_m; REC_1_END_160_m }[+] } |=> {stall_g;not
         TxDout_g} );
  HDLC_300 :
    always({((load_counter_m = ”1001”) and (not TxDataWr_m and TxEnable_m)
        = ’1’); (not TxDataWr_m and TxEnable_m)[*7]}
                |=>  (TxUnderRun_g) );
  HDLC_250_1:
    always({TxEnable_m; TxEnable_m = ’0’ and TxCRCSel_m = ”00” and
        crc_busy_m = ’0’ and T_frame_valid_m = ’1’}
      |=> {[*0 to 8]; [*8]; not TxDout_g; TxDout_g[*6]; not TxDout_g});
  HDLC_200 :
    always({not TxSendAbort_m and TxEnable_m; TxSendAbort_m and TxEnable_m}
              |-> {TxDout_g[*7]; {not TxDout_g; TxDout_g[*6]; not TxDout_g
                  }; TxDout_g[*]});
  HDLC_240:
    always( {not TxLastBit_m and not TxDataWr_m; TxLastBit_m and not
        TxDataWr_m}
      |-> { {not TxDout_g; (TxDout_g)[*6]; not TxDout_g};
        {TxDout_g}[*]; { {TxEnable_m and TxDataWr_m} | {TxDout_m and (not
            TxEnable_m or not TxDataWr_m)} } });
}
```

FIGURE D.6: Annotated SERE specification of HDLC transmitter

D.2.1.1 P2S

```
vunit P2S
{
  P0_init:
    always (not T_frame_valid_m -> ((P_Data_g = "00000000") and S_Data_g =
        '0');

  P1_load:
    always(T_frame_valid_m and not stall_m and load_m -> (P_Data_g =
        TxData_m) and (S_Data_g = P_Data_m(7)));

  P2_stall_data:
    always (T_frame_valid_m and stall_m -> S_Data_g = '0');

  P3_not_shift:
    always (T_frame_valid_m and not stall_m and not load_m -> (S_Data_g =
        P_Data_m(7)) and (P_Data_g(0) = '0') and (P_Data_g(1 to 7) = prev(
        P_Data_m(0 to 6)) ));

  P4_keep_data:
    always(T_frame_valid_m and stall_m and not load_m ->  (P_Data_g = prev(
        P_Data_m)) );

}
```

FIGURE D.7: Annotated FL specification of P2S

D.2.1.2 CRC generation

```
vunit CRCGen
{
  P0_init_data:
    not crc_busy_g and not crc_done_g and (R16_g = ”0000000000000000”) and
        (crc_counter_g = ”00000”);
  P1_crc_16_data_2_ns:
    always (not abort_set_m and not stall_m and not send_crc_m and not
        crc_busy_m and (TxCRCSel_m = ”01”) and R16_m(15) -> next!(R16_g(15
        downto 1) = (poly16_m(15 downto 1) XOR (prev(R16_m(14 downto 0)) ) )
         ) and next!(R16_g(0) = (poly16_m(0) XOR prev(S_Data_m) )));
  P2_crc_16_data_1_ns:
    always (not abort_set_m and not stall_m and not send_crc_m and not
        crc_busy_m and (TxCRCSel_m = ”01”) and not R16_m(15) -> next!(R16_g
        (15 downto 1) = prev(R16_m(14 downto 0))) and next!(R16_g(0) = prev(
        S_Data_m)));
  P3_crc_16_data_1_s:
    always (not abort_set_m and stall_m and not send_crc_m and not
        crc_busy_m and (TxCRCSel_m = ”01”) -> next!(R16_g = prev(R16_m)) );
  P4_CRC_16_send_data:
    always (not abort_set_m and not crc_busy_m and not send_crc_m  and (
        crc_counter_m < ”01111”) and (TxCRCSel_m = ”01”)  -> (crc_data_g =
        S_Data_m) );
  P5_send_crc_16_nstall:
    always (not abort_set_m and not stall_m and crc_busy_m and (TxCRCSel_m
        = ”01”) and (crc_counter_m < ”01111”)-> next!(crc_busy_g) and next!(
        R16_g(0) = ’0’) and next!(crc_counter_g = (prev(crc_counter_m) + ”
        00001”)) and next!(R16_g(15 downto 1) = prev(R16_m(14 downto 0)))
        and (crc_data_g =  (R16_m(15)) ) );
  P6_send_crc_16_stall:
    always (not abort_set_m and stall_m and crc_busy_m and (TxCRCSel_m = ”
        01”) and (crc_counter_m < ”01111”)-> next!(crc_busy_g ) and next!(
        crc_counter_g = prev(crc_counter_m) ) and (crc_data_g =  ’0’ ) and
        next!(R16_g = prev(R16_m)) );
  P7_crc_16:
    always (not abort_set_m and send_crc_m and (TxCRCSel_m = ”01”) -> (
        crc_busy_g) and (crc_counter_g = ”00000”) );
  P8_done_crc_16:
    always (not abort_set_m and crc_busy_m and (TxCRCSel_m = ”01”) and (
        crc_counter_m = ”01111”)-> next!(not crc_busy_g) and next!(
        crc_counter_g = (prev(crc_counter_m) + ”00001”)) and next!(R16_g(15
        downto 1) = prev(R16_m(14 downto 0))) and next!(R16_g(0) = ’0’) and
        (crc_data_g =  (R16_m(15))));
  P9_abort:
    always (abort_set_m -> next!(not crc_busy_g) and next!(R16_g(15 downto
        1) = ”0000000000000000”) and next!(crc_counter_g = ”00000”));
  P10_crc_done:   always (fell(crc_busy_m) -> crc_done_g and next! (not
      crc_done_g) and next!(crc_counter_g = ”00000”));
}
```

FIGURE D.8: Annotated FL specification of CRCGen

D.2.1.3 Zero insertion

```
vunit ZeroInsertion
{
  P0_init:
    ((stall_g = '0') and (zero_Inserted_g = '0') and (zero_Data_g = '0'));

  P1_Zero:
    always(
      prev(prev(prev(prev(crc_Data_m)))) and prev(prev(prev(crc_Data_m)))
          and prev(prev(crc_Data_m)) and prev(crc_Data_m) and crc_Data_m
      ->
      next!(stall_g and  zero_Data_g)
      );

  P2_noZero:
    always(
      not prev(prev(prev(prev(crc_Data_m)))) or not prev(prev(prev(
          crc_Data_m))) or not prev(prev(crc_Data_m)) or not prev(crc_Data_m
          ) or not crc_Data_m
      ->
      next!(not stall_g) and next! (not zero_Inserted_g) and next!((
          zero_Data_g = prev(crc_Data_m)))
      );
}
```

FIGURE D.9: Annotated FL specification of `ZeroInsertion`

D.2.1.4 Flag/Abort generation

```
vunit FlagAbortGen
{
  P0_T_FA_init:
    flag_done_g and not abort_set_g and not TxDout_g;

  P1_send_flag:
    always(rose(send_flag_m) -> not flag_done_g and next_a![1 to 7](not
        flag_done_g) and not TxDout_g and next_a![1 to 6](TxDout_g) and next
        ![7](not TxDout_g) and next![8](not send_flag_m -> flag_done_g));

  P2_send_data:
    always(flag_done_m and not abort_set_m and not send_flag_m and not
        idle_m-> (TxDout_g = zero_Data_m));

  p3_idle:
    always(idle_m and not send_flag_m and flag_done_m -> TxDout_g);

  P4_TxAbort:
    always(TxAbort_m -> (TxDout_g and abort_set_g) and next_a![1 to 7](
        abort_set_g and TxDout_g) and next![8](not abort_set_g));
}
```

FIGURE D.10: Annotated FL specification of `FlagAbortGen`

D.2.1.5 Transmitter controller

```
vunit T_Controller
{
  P0_T_C_reset:
    not TxLastBit_g and not load_g and not send_flag_g and idle_g and (
        load_counter_g = ”0000”) and not eof_g and not send_crc_g and (
        TxCRCValue_g = ”00000000000000000000000000000000”) and not
        TxCRCValAvail_g and (data_counter_g = ”000”) and not (TxUnderRun_g)
        and not T_frame_valid_g;

  P1_T_enable_idle:
    always (rose(TxEnable_m) and not TxDataWr_m -> not send_flag_g and not
        eof_g and not send_crc_g);

  P2_frame_after_idle:
    always (idle_m and TxEnable_m and rose(TxDataWr_m) -> next!(not idle_g
        and send_flag_g));

  P3_load_first_data:
    always (rose(send_flag_m) and not prev(crc_done_m) and not prev(
        abort_set_m) and TxDataWr_m and not TxShareFlag_m -> not load and (
        load_counter = ”0000”) and next![7](load) and next_a![1 to 7](
        load_counter = ”0000”)  and next![7](TxEnable_m -> T_frame_valid_g))
        ;

  P4_T_frame_valid:
    always((T_frame_valid_m) and not TxSendAbort_m and not eof_m and not
        idle_m and TxEnable_m-> next!(T_frame_valid_g));

  P5_disable_mid_fr:
    always(T_frame_valid_m and not TxSendAbort_m and not eof_m and not
        idle_m and not TxEnable_m and (load_counter_m < ”0111”)-> next!(
        T_frame_valid_g));

  P6_T_frame_not_valid:
    always(send_crc_m or TxSendAbort_m or idle_m or (not TxEnable_m and (
        load_counter_m = ”0111”)) -> next! (not T_frame_valid_g and not
        load_g));

  P7_load_data_nstall:
    always(((load_counter_m < ”0111”) and (load_counter_m > ”0000”) and not
         stall_m and T_frame_valid_m) and not TxSendAbort_m-> next!(not
        TxSendAbort_m -> (load_counter_g= (prev(load_counter_m)+”0001”)))
        and (not load_g));

  P8_load_data_nstall_2:
    always((load_counter_m = ”0000”) and not stall_m and T_frame_valid_m
        and not TxSendAbort_m -> next!(load_counter_g= (prev(load_counter_m)
        +”0001”)) and (TxDataWr_m -> load_g));

  P9_load_data_wstall:
    always (load_counter_m < ”0111” and (load_counter_m > ”0000”) and
        stall_m and not eof_m and  T_frame_valid_m-> next!(T_frame_valid_m
        -> load_counter_g = (prev(load_counter_m))) and next!(not load_g));
```

```
P10_load_data_wstall_2:
  always ((load_counter_m = ”0000”) and stall_m and not eof_m and
      T_frame_valid_m  -> next!(T_frame_valid_m -> load_counter_g = (prev(
      load_counter_m))) and (not load_g) and next!(T_frame_valid_m and
      TxDataWr_m -> load_g));

P11_load_new_data:
  always ((load_counter_m = ”0111”) and TxDataWr_m and not eof_m and
      T_frame_valid_m -> next!(load_counter_g = ”0000”));

P12_set_counter_no_load:
  always ((load_counter_m = ”0111”) and not T_frame_valid_m -> next!((
      load_counter_g = ”0000”) until! (not rose(send_flag_m))));

P13_end_crc_close_flag:
  always (rose(crc_done_m) -> next!(send_flag_g and TxLastBit_g) and
      next_a![1 to 7]((not T_frame_valid_g)) );

P14_idle_between_frames:
  always (rose(crc_done_m) -> next_a![1 to 8](not idle_g) and next![9]((
      not TxDataWr_m -> idle_g)));

P15_new_frame_share_flag:
  always (rose(crc_done_m) and TxShareFlag_m -> next![6](TxDataWr_m ->
      T_frame_valid_g));

P16_new_frame_open_flag:
  always (rose(crc_done_m) and not TxShareFlag_m -> next!(load_counter_g
      = ”0000”) and next!(TxCRCValue_g = ”00000000000000000000000000000000
      ”) and next![9](next_event(TxDataWr_m and not idle_m)(send_flag_g
      and not load_g)));

P17_deaasert_send_flag:
  always (send_flag_m -> next!(not send_flag));

P18_eof:
  always (TxDataWr_m and TxEoF_m and load_m and T_frame_valid_m -> next
      ![3](eof_g) and next![3](data_counter_g = ”000”));

P19_keep_eof_no_stall:
  always (eof_m and not stall_m and (data_counter_m < ”101”) and
      T_frame_valid_m-> next!(eof_g) and next!((data_counter_g = (prev(
      data_counter_m) + ”001”))) );

P20_keep_eof_with_stall:
  always (eof_m and stall_m and (data_counter_m < ”101”) and
      T_frame_valid_m-> next!((eof_g = ’1’)) and next!((data_counter_g =
      prev(data_counter_m))) );

P21_eof_start_crc:
  always (eof_m and (data_counter_m = ”101”) and T_frame_valid_m-> next!(
      not eof_g and (data_counter_g = ”000”)) and send_crc_g );
  P22_eof_start_crc_nf:
    always (eof_m and (not T_frame_valid_m )-> next!(not eof_g and (
        data_counter_g = ”000”)) and send_crc_g );

  P23_deaasert_send_crc:
    always (send_crc_m -> next! (not send_crc_g and not TxCRCValAvail_g)
        and next!((load_counter_g = ”0000”)));

  P24_abort_close_flag:
    always (TxSendAbort_m -> next![8](send_flag_g and TxLastBit_g) and next
        !(load_counter_g = ”0000”) and next_a![1 to 16](not idle_g) and next
        ![17](idle_g) and next_a![1 to 17](not T_frame_valid_g) and next
        ![9](next_event(send_flag_m)[8](T_frame_valid_g)));

  P25_crc_val_16:
    always(rose(send_crc_m) and (TxCRCSel_m = ”01”) -> (TxCRCValue_g(15
        downto 0) = R16_m) and (TxCRCValue_g(31 downto 16) = ”
        0000000000000000”) and TxCRCValAvail_g);

  P26_crc_val_32:
    always(rose(send_crc_m) and (TxCRCSel_m = ”10”) -> (TxCRCValue_g =
        R32_m) and TxCRCValAvail_g);

  P27_crc_val_init:
    always (rose(crc_done_m) -> next!(TxCRCValue_g = ”
        00000000000000000000000000000000”) );

  P28_last_bit:
    always (TxLastBit_m -> next!(not TxLastBit_g));

  P29_T_disbale:
    always ( T_frame_valid_m and ((fell(TxEnable_m) and (load_counter_m /=
        ”0000”)) or (fell(TxEnable_m) and (load_counter_m = ”0000”) and
        load_m )) -> next!(next_event(load_counter_m = ”0111”)(eof_g)) );

  P30_between_data:
    always ((load_counter_m > ”0111”) and (load_counter_m < ”1111”) and not
         TxDataWr_m and not eof_m and T_frame_valid_m and TxEnable_m-> next
        !(((load_counter_g = prev(load_counter_m) + ”0001”))));

  P31_underrun:
    always ((load_counter_m = ”1111”) and not TxDataWr_m and not eof_m and
        T_frame_valid_m and TxEnable_m-> next!(load_counter_g = ”0000”) and
        next!(TxUnderRun_g));

  P32_new_data:
    always ((load_counter_m > ”0111”) and (load_counter_m < ”1111”) and
        rose(TxDataWr_m) and not eof_m and T_frame_valid_m and TxEnable_m->
        next!(load_counter_g = ”0000”));

  P33_not_underrun:
    always(TxUnderRun_m -> next! (not TxUnderRun_g));
}
```

Figure D.11: Annotated FL specification of transmitter controller

D.2.2 Receiver

D.2.2.1 Flag/Abort detection

FL properties

```
vunit FlagAbortDet
{
  P0_R_FA_init:
    not R_S_Data_g and (data_seq_g = ”00000000”) and not AbortFound_g;

  P1_R_FA_receive_data:
    always (RxEnable_m -> ( R_S_Data_g = data_seq_m(7)));

  P2_R_FA_store_data:
    always (RxEnable_m -> next!( data_seq_g(0) = RxData_m) and next!(
        data_seq_g(7 downto 1) = prev(data_seq_m(6 downto 0))));

  P3_R_FA_is_abort:
    always ((data_seq_m = ”11111111”) -> IsAbort_g and (AbortFound_g until!
         (fell(IsFlag_m) )) and (next_event(fell(IsFlag_m))(not AbortFound_g
        )));

  P4_R_FA_is_not_abort:
    always ((data_seq_m /= ”11111111”) -> not IsAbort_g);


  P5_R_FA_is_flag:
    always ((data_seq_m = ”01111110”) -> IsFlag_g and not IsAbort_g);

  P6_R_FA_is_not_flag:
    always ((data_seq_m /= ”01111110”) -> not IsFlag_g);
}
```

FIGURE D.12: Annotated FL specification of `FlagAbortDet`

SERE properties

```
vunit FlagAbortDet_sere
{
  P0_R_FA_init:
    not R_S_Data_g and not AbortFound_g;

  P1_is_flag:
    always ({RxEnable_m and not R_S_Data_m; R_S_Data_m[*6]; not R_S_Data_m}
         |-> {IsFlag_g and not IsAbort_g});

  P2_is_abort:
    always ({RxEnable_m and not R_S_Data_m; R_S_Data_m[*7]} |-> {{IsAbort_g
        } & {AbortFound_g[*]; fell(IsFlag_m) }});
}
```

FIGURE D.13: Annotated SERE specification of `FlagAbortDet`

D.2.2.2 Zero detection

FL properties

```
vunit ZeroDetection
{
  P0_R_Z_init:
    not R_zero_Data_g and not R_stall_g and not R_zero_Inserted_g and (
        data_seq_g = ”0000000”);

  P1_R_Z_receive_data:
    always (RxEnable_m -> ( R_zero_Data_g = R_S_Data_m ));

  P2_R_Z_store_data:
    always (RxEnable_m -> next!( data_seq_g(0) = R_S_Data_m) and next!(
        data_seq_g(6 downto 1) = prev(data_seq_m(5 downto 0))));

  P3_R_Z_zero_inserted:
    always ((data_seq_m = ”0111110”) -> R_zero_Inserted_g and R_stall_g);

  P4_R_Z_zero_not_inserted:
    always ((data_seq_m /= ”0111110”) -> not R_zero_Inserted_g and not
        R_stall_g);
}
```

Figure D.14: Annotated FL specification of `ZeroDetection`

SERE properties

```
vunit ZeroDetection_sere
{
  P0_R_Z_init:
    not R_zero_Data_g and not R_stall_g and not R_zero_Inserted_g;
  P1_zero_detection:
    always ({RxEnable_m and not R_S_Data_m; R_S_Data_m[*5]; not R_S_Data_m}
         |-> {R_zero_Inserted_g and R_stall_g});
}
```

Figure D.15: Annotated SERE specification of `ZeroDetection`

D.2.2.3 CRC checker

```
vunit CRCCheck
{
  P0_R_crc_init_data:
    always (RxEnable_m and rose(R_crc_check_m) -> not R_crc_error_g and (
        R_R16_g = "0000000000000000") and (crc_buffer_g = "
        00000000000000000000000000000000"));

  P1_R_crc_16_data_2_ns:
    always (RxEnable_m and not R_stall_m and R_crc_check_m and (RxCRCSel_m
        = "01") and R_R16_m(15) -> next!(R_R16_g(15 downto 1) = (poly16_m(15
         downto 1) XOR (prev(R_R16_m(14 downto 0)) ) ) ) and next!(R_R16_g
        (0) = (poly16_m(0) XOR prev(crc_buffer_m(15) ) )));

  P2_R_crc_16_data_2_s:
    always (RxEnable_m and R_stall_m and R_crc_check_m and (RxCRCSel_m = "
        01")  -> next!(R_R16_g = (prev(R_R16_m) )));

  P3_R_crc_16_data_1_ns:
    always (RxEnable_m and not R_stall_m and R_crc_check_m and (RxCRCSel_m
        = "01") and not R_R16_m(15) -> next!(R_R16_g(15 downto 1) = prev(
        R_R16_m(14 downto 0))) and next!(R_R16_g(0) = prev(crc_buffer_m(15)
        )));

  P4_R_crc_16_shift_ns:
    always (RxEnable_m and not R_stall_m and R_crc_check_m and (RxCRCSel_m
        = "01") -> next!(crc_buffer_g(31 downto 1) = prev(crc_buffer_m(30
        downto 0))) and next!(crc_buffer_g(0) = prev(R_zero_Data_m) ) );

  P5_R_crc_16_shift_s:
    always (RxEnable_m and R_stall_m and R_crc_check_m and (RxCRCSel_m = "
        01") -> next!(crc_buffer_g(31 downto 1) = prev(crc_buffer_m(31
        downto 1))) and next!(crc_buffer_g(0) = prev(crc_buffer_m(0)) ) );

  P6_R_crc_16_send_data:
    always (RxEnable_m and R_crc_check_m and (RxCRCSel_m = "01") -> (
        R_crc_data_g = R_zero_Data_m) );

  P7_R_crc_16_error:
    always (RxEnable_m and R_crc_check_m and (crc_buffer_m(15 downto 0) /=
        R_R16_m) and (RxCRCSel_m = "01") -> R_crc_error_g);

  P8_R_crc_16_not_crc_err:
    always (RxEnable_m and R_crc_check_m and (crc_buffer_m(15 downto 0) =
        R_R16_m) and (RxCRCSel_m = "01") -> not R_crc_error_g);

  P9_R_crc_disable:
    always(not RxEnable_m or not R_crc_check_m -> R_crc_data_g = '0');
}
```

FIGURE D.16: Annotated FL specification of `CRCCheck`

D.2.2.4 S2P

```
vunit S2P
{
  P0_R_S2P_init:
    not frame_valid_g and not s2p_done_g and (R_load_counter_g = ”0000”)
        and (s2p_buffer_g = ”00000000” and (RxDout_g = ”00000000”);

  P1_R_S2P_frame:
    always (rose(s2p_enable_m) -> (frame_valid_g until! (s2p_disable_m)));

  P2_R_S2P_frame:
    always (rose(s2p_disable_m) -> (not frame_valid_g) until! (s2p_enable_m
        ));

  P3_R_S2P_shift:
    always (not R_stall_m and (frame_valid_m) and (R_load_counter_m < ”1000
        ”)-> next!(s2p_buffer(7) = prev(R_crc_data_m)) and next!(
        s2p_buffer_g(6 downto 0) = prev(s2p_buffer_m(7 downto 1)) ) and next
        !((R_load_counter_g = prev_m(R_load_counter) + ”0001” ) ) );

  P4_R_S2P_no_valid_frame:
    always (not (frame_valid_m) -> next!(s2p_buffer_g = ”00000000”) and
        next!((R_load_counter_g = ”0000”)) );

  P5_R_S2P_stall:
    always (R_stall_m and (frame_valid_m) and (R_load_counter_m < ”1000”)->
         next!(R_load_counter_g = prev(R_load_counter_m)) and next!(
        s2p_buffer_g = prev(s2p_buffer_m)));

  P6_R_S2P_data_ready_ns:
    always (not R_stall_m and (R_load_counter_m = ”1000”) -> next!(
        R_load_counter_g = ”0001”) and (RxDout_g = s2p_buffer_m) and
        s2p_done_g and next!(RxDout_g = ”00000000”));

  P7_R_S2P_data_ready_s:
    always ((R_stall_m) and (R_load_counter_m = ”1000”) -> next!(
        R_load_counter_g = ”0000”) and (RxDout_g = s2p_buffer_m) and
        s2p_done_g and next!(s2p_buffer_g = prev(s2p_buffer_m)) );

  P8_R_S2P_not_done:
    always(rose(s2p_done_m) -> next!(not s2p_done_g) );
}
```

FIGURE D.17: Annotated FL specification of *S2P*

D.2.2.5 Receiver controller

```
vunit RController
{
  P0_R_Ctr_init:
    not R_crc_check_g and not R_crc_error_g and not s2p_enable_g and not
        s2p_disable_g and not RxDataAvail_g and (R_flag_counter_g = "00")
        and not RxEndOfFrame_g and not RxStartOfFrame_g and not
        RxAbortFound_g and (RxCRCValue_g = "00000000000000000000000000000000
        ");

  P1_R_Ctr_s2p_en:
    always (rose(R_crc_check_m) -> (s2p_enable_g));

  P2_R_Ctr_s2p_not_en:
    always (rose(s2p_enable_m) -> next! (not s2p_enable_g));

  P3_R_Ctr_open_flag:
    always (IsFlag_m and (R_flag_counter_m = "00") -> next! (
        R_flag_counter_g = "01") and next![8](R_crc_check_g));

  P4_R_Ctr_close_flag:
    always (IsFlag_m and (R_flag_counter_m = "01") -> next!(s2p_disable_g
        and not R_crc_check_g) and next!(R_flag_counter_g = "00"));

  P5_R_Ctr_s2p_not_dis:
    always (rose(s2p_disable_m) -> next! (not s2p_disable_g));

  P6_R_Ctr_data_rdy:
    always (s2p_done_m -> RxDataAvail_g);

  P7_R_Ctr_data_not_rdy:
    always (rose(RxDataAvail_m) -> next! (not RxDataAvail_g));

  P8_R_Ctr_SoF:
    always (IsFlag_m and (R_flag_counter_m = "00") -> next![8](not
        IsAbort_m and not AbortFound_m and not IsFlag_m -> next_event!(
        RxDataAvail)(RxStartOfFrame_g)));

  P9_R_Ctr_not_SoF:
    always (rose(RxStartOfFrame_m) -> next! (not RxStartOfFrame_g));

  P10_R_Ctr_EoF_n_abort:
    always (RxStartOfFrame_m and (R_flag_counter_m = "01") and not
        RxAbortFound_m -> next_event!(RxDataAvail_m and IsFlag_m)(
        RxEndOfFrame_g) );

  P11_R_Ctr_EoF_abort:
    always (IsFlag_m and (R_flag_counter_m = "01") and RxAbortFound_m->
        RxEndOfFrame_g);

  P12_R_Ctr_not_EoF:
    always (rose(RxEndOfFrame_m) -> next! (not RxEndOfFrame_g));
  P13_R_Ctr_CRC_Err:
    always ((rose(RxEndOfFrame_m) and R_crc_error_m) -> (RxCRCError_g) and
        next! (not RxCRCError_g));

  P14_R_Ctr_not_CRC_Err:
    always (rose(RxStartOfFrame_m) -> (not RxCRCError_g) until! (
        RxEndOfFrame_m and R_crc_error_m) );

  P15_R_Ctr_CRC_value:
    always (RxEndOfFrame_m and (RxCRCSel_m = "01")-> (RxCRCValue_g(15
        downto 0) = R_R16_m) and next!(RxCRCValue_g = "
        00000000000000000000000000000000"));

  P16_R_abort_found:
    always ((R_flag_counter_m /= "00") -> (next!( RxAbortFound_g =
        AbortFound_m)));
}
```

Figure D.18: Annotated FL specification of receiver controller

D.3 AMBA arbiter

```
vunit arbiter{
  P0_G1_1_m0:
    always(HBUSREQ_0_m and mast_0_m -> BUSREQ_g);
  P1_G1_2_m0:
    always(not HBUSREQ_0_m and mast_0_m -> not BUSREQ_g);
  P2_G1_1_m1:
    always(HBUSREQ_1_m and mast_1_m -> BUSREQ_g);
  P3_G1_2_m1:
    always(not HBUSREQ_1_m and mast_1_m -> not BUSREQ_g);
  P4_G4_m0_m1:
    always (DECIDE_m and (HBUSREQ_0_m or HBUSREQ_1_m) ->next!(GRANTED_g));
  P5_G5_1:
    always (GRANTED_m and HREADY_m -> next!(not GRANTED_g));
  P6_G5_2:
    always (GRANTED_m and not HREADY_m -> next!(GRANTED_g));
  P7_G6_1_m0:
    always (HREADY_m and HGRANT_0_m -> next!(mast_0_g));
  P8_G6_2_m0:
    always (HREADY_m and not HGRANT_0_m -> next!(not mast_0_g));
  P9_G6_1_m1:
    always (HREADY_m and HGRANT_1_m -> next!(mast_1_g));
  P10_G6_2_m1:
    always (HREADY_m and not HGRANT_1_m -> next!(not mast_1_g));
  P11_G7_m0_m1:
    always ((HREADY_m and ((HLOCK_0_m and HGRANT_0_m) or (HLOCK_1_m and
        HGRANT_1_m)) -> next!(HMASTLOCK_g)));
  P12_G8_1_1_m0:
    always ((not HREADY_m or not GRANTED_m) and mast_0_m-> next!(mast_0_g))
        ;
  P13_G8_1_2_m0:
    always ((not HREADY_m or not GRANTED_m) and (not mast_0_m)-> next!(not
        mast_0_g));
  P14_G8_1_1_m1:
    always ((not HREADY_m or not GRANTED_m) and mast_1_m -> next!(mast_1_g)
        );
  P15_G8_1_2_m1:
    always ((not HREADY_m or not GRANTED_m) and (not mast_1_m)-> next!(not
        mast_1_g));
  P16_G8_2_1_m0_m1:
    always (not HREADY_m or not GRANTED_m and HMASTLOCK_m -> next!(
        HMASTLOCK_g));
  P17_G8_2_2_m0_m1:
    always (not HREADY_m or not GRANTED_m and not HMASTLOCK_m -> next!(not
        HMASTLOCK_g));
  P18_G9_1_m0:
    always (not DECIDE_m and HGRANT_0_m -> next!(HGRANT_0_g));
  P19_G9_2_m0:
    always (not DECIDE_m and not HGRANT_0_m -> next!(not HGRANT_0_g));
  P20_G9_1_m1:
    always (not DECIDE_m and HGRANT_1_m -> next!(HGRANT_1_g));
  P21_G9_2_m1:
    always (not DECIDE_m and not HGRANT_1_m -> next!(not HGRANT_1_g));
  P22_G10_1_m1:
    always (not HGRANT_1_m -> next! ((not HGRANT_1_g) until! HBUSREQ_1_m));
  P23_G10_2:
    always (DECIDE_m and (not HBUSREQ_0_m) and (not HBUSREQ_1_m) -> next! (
        HGRANT_0_g));
  P24_G12:
    DECIDE_g and HGRANT_0_g and ((not HMASTER_0_g) and (not HMASTER_1_g)
        and (not HMASTER_2_g) and (not HMASTER_3_g)) and not GRANTED_g and
        not HMASTLOCK_g and not HGRANT_1_g;
  P25_gen_decide:
    always (((HBUSREQ_0_m or HBUSREQ_1_m) or (HLOCK_0_m or HLOCK_1_m)) and
        HREADY_m and (not (GRANTED_m) and not DECIDE_m)-> next!(DECIDE_g));
  P26_gen_not_decide:
    always(DECIDE_m -> next! (not DECIDE_g));
  P27_grant0:
    always (((GRANTED_m) and (not prev(HGRANT_0_m)) and (not HLOCK_1_m)) or
         (HLOCK_0 and prev(HGRANT_0_m))-> HGRANT_0_g);
  P28_grant1:
    always (((GRANTED_m) and (not prev(HGRANT_1_m)) and (not HLOCK_0_m)) or
         (HLOCK_1 and prev(HGRANT_1_m))-> HGRANT_1_g);
  P29_not_HMASTER1:
    always not HMASTER_1_g and not HMASTER_2_g and not HMASTER_3_g;
  P30_mast_0:
    always mast_0_m -> not HMASTER_0_g;
  P31_mast_1:
    always mast_1_m -> HMASTER_0_g;
  P32_one_grant:
    always not HGRANT_0 or not HGRANT_1;
}
```

FIGURE D.19: Annotated FL specification of AMBA arbiter (with 2 masters and 2 slaves)

Appendix E

SyntHorus2

Contents

In this appendix, our prototype tool, SyntHorus2 is introduced.

E.1 Installation

The script "install" is provided for installing SyntHorus2. It creates all the necessary folders (for writing the intermediate and final results), compiles the source files, and creates the executable file in "../test" directory. To run this script, the following command should be executed in the root directory:

```
source install
```

For a basch terminal, the following command should be executed:

```
chmod +x install
source install
```

E.2 Execution

SyntHorus2 takes PSL properties and the interface definition of the circuit as its inputs, and generates the VHDL files of the circuit by executing the following command (the command should be executed in the "../test" directory):

```
./Synthorus -f <spec_file> -t <type_file> [options]
```

In the above command, "-f <spec_file>" specifies the specification file, "-t <type_file>" specifies the type file in which the interface signals have been defined. Then, based on the requirements, options should be specified.

E.2.1 Specification file

The specification file has the following format:

```
vunit <entity_name>{
  [-- pragma_SYNTHORUS : <family_name> [<option>:<name>]]
  <property_name>: <property>
  ...
}
```

E.2.2 Type file

This file contains the definition of input and output signals. Now, the internal signals are defined as outputs. However, we should enhance the tool to accept internal signals. For each signal we should specify the followings:

- the signal name, which should be at least 2 characters
- the signal direction
- the signal type: std_logic or std_logic_vector. It is specified by providing the direction ("to", "downto", or "nothing")

- the signal range
- the default value: it can be 0, 1, or m (for memorizing)

For each signal, these information should be provided in the following format:

```
<I|O>:<name>:<type>:<low_range>:<high_range>:<default_value>
```

In the above format, <I|O> is replaced by I for an input signal, and by O for an internal or output signal. <type> is replaced by 0 for std_logic, 1 for std_logic_vector with increasing index, and 2 for std_logic_vector with decreasing index. If type is 0, <low_range> and <high_range> are replaced with 0. Otherwise, <low_range> specifies the low index and <high_range> specifies the high index of the signal. <default_value> is replaced by 0, 1, or m.

E.3 Options

Two groups of options have been provided for SyntHorus2: 1) the options that are used in the command line, while executing SyntHorus2, 2) the "pragma" options, that are added to the input file of SyntHorus2, where the properties are given.

E.3.1 Command line options

Here are some of the most useful options of SyntHorus2, which are specified in the command line:

- "-synth": is used to synthesize the properties to reactants,
- "-mon": is used to synthesize the properties to monitors,
- "-gen": is used to synthesize the properties to reactants,
- "-l lib_name": specifies the name of the library of primitive reactants. We use "Horus" as the library name.
- "-cons": is used to write the complementary properties for consistency checking in file "consistency.psl" (for use with the verification tools),
- "-comp": is used to write the complementary properties for checking the completeness in a file "completeness.psl" (for use with the verification tools),
- "-c level": this option specifies if signals are sensitive to the pose-edge ("level = 1") or to the neg-edge of the clock ("level = 0"),
- "-r level": this option specifies if the reset signal is active high ("level = 1") or active low ("level = 0"),
- "-a option": this option specifies if the reset signal is asynchronous ("option = 1") or synchronous ("option = 0"),

- "-v option": this option specifies how to deal with vectors. If "option = 1", the vectors are decomposed to bits. This is useful, when we are generating various ranges of vector A in several properties, where these ranges overlap. As an example, assume that A is a 8-bit vector, and $A(0\ to\ 5)$ is generated by property `P1`, and $A(3\ to\ 7)$ is generated by `P2`. $A(3\ to\ 5)$ is generated by two properties. Therefore, we need to decompose vector A to its bits.

- "-o <intermediate_result_file>": this option specifies the file in which the intermediate results are written. These intermediate results include AST, DAST, $\mathcal{DG}$, and the trigger signals.

E.3.2 Pragma options

It is possible to define some characteristics of the circuit, e.g. the name of clock, reset, entity, architecture, and also enabling clock gating. It is done by using the options "Pragma_SYNTHORUS" in the <spec_file> as follows[1]

```
-- pragma_SYNTHORUS : <family_name> [<option>:<name>]
```

<family_name> corresponds to the type of the selection option. Now, it can be either <DESIGN_NAME> or <GATED_CLOCK>.

1 <GATED_CLOCK>: SyntHorus2 generates the primitive reactants that have a port for the clock enable input.

- "clken_signal_name:<clken_name>": allows changing the name of the clock enable port. By default, this port is "clk_en".
- "clk_signal_name:<clk_name>": allows changing the name of the clock port. By default, this port is "clk".

2 <DESIGN_NAME>: several options may exist:

- "top_name:<new_entity_name>": by default, the name of the top entity is specified in <spec_file>, in front of keyword "vunit". Using this option allows changing the name of the top entity to "new_entity_name".
- "architecture:<arch_name>": this option specifies the name of the architecture. Without this option, the name of the architecture is "top" by default.
- "reset_name:<reset_name>": this option specifies the name of the reset signal. Without this option, the name of the reset signal is "reset" by default.

For example, consider the following pragma option:

```
-- pragma_SYNTHORUS : DESIGN_NAME top_name:GenBuf reset_name:rstn
```

It means that the entity name is "GenBuf", and the name of the reset signal is "rstn".

[1] These options have been provided based on the industrial demand.

E.4 Output

The outputs are generated, and placed in "../test/results" directory. The top module is put in "../test/results/Top", the properties are put in "../test/results/Property", and the solvers (if they exist) are put in "../test/results/Solver". All the library elements (synchronous, asynchronous, and with clock enable) are available at the root directory.

References

[AAH$^+$03] M.S. Abadir, K.L. Albin, J. Havlicek, N. Krishnamurthy, and A.K. Martin. Formal verification successes at Motorola. *Formal Methods in System Design*, 22(2):117–123, March 2003.

[ABBSV00] A. Aziz, F. Balarin, R.K. Brayton, and A.L. Sangiovanni-Vincentelli. Sequential synthesis using S1S. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 19(10):1149–1162, 2000.

[ABC] ABC: A System for Sequential Synthesis and Verification, http://www.eecs.berkeley.edu/˜alanmi/abc/, accessed 2015.

[ABG$^+$00] Y. Abarbanel, I. Beer, L. Gluhovsky, S. Keidar, and Y. Wolfsthal. FoCs–automatic generation of simulation checkers from formal specifications. In *Computer Aided Verification*, pages 538–542. Springer, 2000.

[ACA] Acacia, http://lit2.ulb.ac.be/acaciaplus/, accessed 2015.

[AFF$^+$02] R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-haim, E. Singerman, A. Tiemeyer, M.Y. Vardi, and Y. Zbar. The ForSpec Temporal Logic: A New Temporal Property-Specification Language. In *Tools and Algorithms for Construction and Analysis of Systems*, 2002.

[AMB] AMBA Specification Rev 2.0 (1999), http://www-micro.deis.unibo.it/ maggagni/amba99.pdf, accessed 2015.

[BBDE$^+$01] I. Beer, S. Ben-David, C. Eisner, D. Fisman, A. Gringauze, Y. Rodeh, and Yoav. The temporal logic Sugar. In *Computer Aided Verification*, pages 363–367. Springer, 2001.

[BCC$^+$99] A. Biere, A. Cimatti, E.M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proc. of the 36th Annual ACM/IEEE Design Automation Conference*, (DAC'99), pages 317–320, New York, NY, USA, 1999. ACM.

[BCE$^+$04] R. Bloem, R. Cavada, C. Eisner, I. Pill, M. Roveri, and S. Semprini. Manual for property simulation and assurance tool (deliverable 1.2/4-5). Technical report, PROSYD Project, January 2004.

[BCG$^+$10] R. Bloem, A. Cimatti, K. Greimel, R. Koenighofer, M. Roveri, V. Schuppan, and R. Seeber. RATSY - a new requirements analysis tool with synthesis. In *Proc. of the 22nd International Conference on Computer Aided Verification (CAV'2010)*, pages 425–429. Springer-Verlag, July 15-19 2010.

[BdFR04] S. Ben-david, D. Fisman, and S. Ruah. Automata construction for regular expressions in model checking. In *IBM research report H-0229*, 2004.

[BEK$^+$14] R. Bloem, U. Egly, P. Klampfl, R. Könighofer, and F. Lonsing. SAT-based methods for circuit synthesis. In *Proc. of the 14th Conference on Formal Methods in Computer-Aided Design*, pages 31–34. FMCAD Inc, 2014.

[BFH05] D. Bustan, D. Fisman, and J. Havlicek. Automata construction for PSL. In *https://www.research.ibm.com/haifa/projects/verification/RB_Homepage/ps/automta_construction_TR.pdf*, 2005.

[BGJ$^+$07a] R. Bloem, S. Galler, B. Jobstman, N. Piterman, A. Pnueli, and M. Weiglhofer. Specify, compile, run: Hardware from PSL. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 190:3–16, 2007.

[BGJ$^+$07b] R. Bloem, S. Galler, B. Jobstmann, N. Piterman, A. Pnueli, and M. Weiglhofer. Automatic hardware synthesis from specifications: a case study. In *Proc. of the conference on Design, Automation and Test in Europe (DATE'2007)*, pages 1188–1193, April 16-20 2007.

[BJP$^+$12] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Saar. Synthesis of reactive(1) designs. *Journal of Computer and System Sciences*, 78(3):911–938, May 2012.

[BL69] J.R. Büchi and L.H. Landweber. Solving sequential conditions by finite-state strategies. *Trans. of the American Mathematical Society*, 138:295–311, 1969.

[BL88] G. M. Brown and M. E. Leeser. Synthesizing correct sequential circuits. In *Proc. of the International Conference on Systolic Arrays*, 1988.

[Bor97] Arne Borälv. The industrial success of verification tools based on Stalmarck's method. In Orna Grumberg, editor, *Computer Aided Verification*, volume 1254 of *Lecture Notes in Computer Science*, pages 7–10. Springer Berlin Heidelberg, 1997.

[BP63] J.A. Brzozowski and J.F. Poage. On the construction of sequential machines from regular expressions. *IEEE Trans. on Electronic Computers*, EC-12(4):402–403, August 1963.

[Brz64] J.A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11:481–494, 1964.

[Brz65] J.A. Brzozowski. Regular expressions for linear sequential circuits. *IEEE Trans. on Electronic Computers*, EC-14(2):148–156, April 1965.

[BUG] BugScope, http://www.atrenta.com/pg/9/, accessed 2015.

[BY96] R.S. Boyer and Y. Yu. Automated proofs of object code for a widely used microprocessor. *Journal of the ACM*, 43(1):166–192, January 1996.

[BZ05] M. Boulé and Z. Zilic. Incorporating efficient assertion checkers into hardware emulation. In *Proc. of IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD'2005)*, pages 221–228, October 2005.

[BZ07] M. Boulé and Z. Zilic. Efficient automata-based assertion-checker synthesis of SEREs for hardware emulation. In *Asia and South Pacific Design Automation Conference (ASP-DAC'2007)*, pages 324–329, January 2007.

[BZ08a] M. Boulé and Z. Zilic. Assertion checkers - enablers of quality design. In *1st Microsystems and Nanoelectronics Research Conference (MNRC'2008)*, pages 97–100, October 2008.

[BZ08b] M. Boulé and Z. Zilic. Automata-based assertion-checker synthesis of PSL properties. *ACM Trans. on Design Automation of Electronic Systems (ACM-TODAES)*, 13(1):Article 4, January 2008.

[BZ08c] M. Boulé and Z. Zilic. *Generating Hardware Assertion Checkers.* Springer, 2008.

[CAD] Incisive Formal Verifier,
http://www.cadence.com/products/fv/formal_verifier/Pages/default.aspx,
accessed 2015.

[CBE$^+$92] L. Claesen, D. Borrione, H. Eveking, G. Milne, J.L. Paillet, and P. Prinetto. Charme: towards formal design and verification for provably correct VLSI hardware. In *Correct-Hardware-Design-Methodologies.-Proc.-of-the-Advanced-Research-Workshop.*, pages 3–25. North-Holland, Amsterdam, Netherlands, 1992.

[CE82] E.M. Clarke and E.A. Emerson. *Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic.* Springer, 1982.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

[CGZ99] E.M. Clarke, S.M. German, and X. Zhao. Verifying the SRT division algorithm using theorem proving techniques. *Formal Methods in System Design*, 14(1):7–44, January 1999.

[Chu57] A. Church. Applications of recursive arithmetic to the problem of circuit synthesis. *Summaries of the Summer Institute of Symbolic Logic*, 1:3–50, 1957.

[Chu62] A. Church. Logic, arithmetic and automata. In *Proc. of International Congress of Mathematicians*, pages 23–25, 1962.

[Cim08] A. Cimatti. Beyond Boolean SAT: Satisfiability modulo theories. In *Proc. of the 9th International Workshop on Discrete Event Systems (WODES'2008)*, pages 68–73, May 2008.

[CRST06] A. Cimatti, M. Roveri, S. Semprini, and S. Tonetta. From PSL to NBA: a modular symbolic encoding. In *Formal Methods in Computer Aided Design (FMCAD'2006)*, pages 125–133, November 2006.

[Cur68] H.A. Curtis. Polylinear sequential circuit realizations of finite automata. *IEEE Trans. on Computers*, C-17(3):251–259, March 1968.

[DB95] D. Déharbe and D. Borrione. Symbolic model checking with past and future temporal modalities: Fundamentals and algorithms. In Bergé, Jean-Michel, Levia, Oz, and Jacques Rouillard, editors, *Model Generation in Electronic Design*, volume 1 of *Current Issues in Electronic Modeling*, pages 105–126. Springer US, 1995.

[EBM] EBMC, http://www.cprover.org/ebmc/, accessed 2015.

[EF06] C. Eisner and D. Fishman. *A Practical Introduction to* PSL. Springer, 2006.

[EFP09] F. Eibensteiner, R. Findenig, and M. Pfaff. SynPSL: Behavioral synthesis of PSL assertions. In R. Moreno-Diaz, F. Pichler, and A. Quesada-Arencibia, editors, *Computer Aided Systems Theory - EUROCAST 2009*, volume 5717 of *Lecture Notes in Computer Science*, pages 69–74. Springer Berlin Heidelberg, 2009.

[Ehl11] R. Ehlers. Unbeast: Symbolic bounded synthesis. In P.A. Abdulla and K.R. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6605 of *Lecture Notes in Computer Science*, pages 272–275. Springer Berlin Heidelberg, 2011.

[EKH12] R. Ehlers, R. Könighofer, and G. Hofferek. Symbolically synthesizing small circuits. In *Proc. of Formal Methods in Computer Aided Design (FMCAD'2012)*, pages 91–100, October 22-25 2012.

[FG05] H. Foster and Working Group. *IEEE standard for property specification language (PSL).* pub-IEEE-STD, 2005.

[FJR09] E. Filiot, N. Jin, and J.F. Raskin. An antichain algorithm for LTL realizability. In *Proc. of the 21st International Conference on Computer Aided Verification: (CAV'2009)*, pages 263–277, July 2009.

[FJR11] E. Filiot, N. Jin, and J.F. Raskin. Antichains and compositional algorithms for LTL synthesis. *Form. Methods Syst. Des.*, 39(3):261–296, December 2011.

[FKL03] H. Foster, A. Krolnik, and D. Lacey. *Assertion-Based Design.* Kluwer Academic Publishers, June 2003.

[FKTMo86] M. Fujita, S. Kono, H. Tanaka, and T. Moto-oka. Tokio: Logic programming language based on temporal logic and its compilation to Prolog. In *Proc. of the 3rd International Conference on Logic Programming*, volume Lecture Notes in Computer Science 225, pages 695–709. Springer, 1986.

[Fos08] H. Foster. Applied assertion-based verification: An industry perspective. *Foundations and Trends in Electronic Design Automation*, 3(1):1–95, 2008.

[Fos15] H.D. Foster. Trends in functional verification: A 2014 industry study. In *Proc. of the 52nd Annual Design Automation Conference (DAC'2015)*, pages 1–6, New York, NY, USA, 2015. ACM.

[FU82] R.W. Floyd and J. D. Ullman. The compilation of regular expressions into integrated circuits. *Journal of the ACM*, 29(3):603–622, 1982.

[Gas05] E. Gascard. From sequential extended regular expressions to deterministic finite automata. In *Enabling Technologies for the New Knowledge Society: ITI 3rd International Conference on Information and Communications Technology*, pages 145–157, December 2005.

[GCH] Y. Godhal, K. Chatterjee, and T.A. Henzinger. Synthesis of AMBA AHB from formal specification: a case study. *International Journal on Software Tools for Technology Transfer.*

[GG05] S.V Gheorghita and R. Grigore. Constructing checkers from PSL properties. In *Proc. of the 15th International Conference on Control Systems and Computer Science (CSCS'2015)*, volume 2, pages 757–762, 2005.

[GHS03] M. Gordon, J. Hurd, and K. Slind. Executing the formal semantics of the Accellera property specification language by mechanised theorem proving. In E. Tronci D. Geist, editor, *Correct Hardware Design and Verification Methods*, volume 2860 of *Lecture Notes in Computer Science*, pages 200–215. Springer Berlin Heidelberg, 2003.

[Gor88] M.J.C Gordon. HOL: A proof generating system for higher-order logic. In G. Birtwistle and P.A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, volume 35 of *The Kluwer International Series in Engineering and Computer Science*, pages 73–128. Springer US, 1988.

[Gre04] D. Greaves. Automated hardware synthesis from formal specification using SAT solvers. In *Proc. of the 15th IEEE International Workshop on Rapid System Prototyping (RSP'2004)*, pages 15–20, 2004.

[Gup92] A. Gupta. Formal hardware verification methods: A survey. *Formal Methods in System Design*, 1(2-3):151–238, October 1992.

[HIL04] Haifa-IBM-Laboratories. *RuleBase Parallel Edition.* IBM, November 2004.

[HNV05] S. Heymans, D.V. Nieuwenborgh, and D. Vermeir. Synthesis from temporal specifications using preferred answer set programming. 3701:280–294, 2005.

[IBM] IBM Generalized Buffer, https://www.research.ibm.com/haifa/projects/verification/RB_Homepage/tutorial3/GenBuf_english_spec.htm, accessed 2015.

[JB06] B. Jobstman and R. Bloem. Optimizations for LTL synthesis. In *Formal Methods in Computer Aided Design (FMCAD'2006)*, pages 117–124, November 2006.

[KAB06] K.Morin-Allory and D. Borrione. Proven correct monitors from PSL specifications. In *Proc. of conference on Design, Automation and Test in Europe (DATE'2006)*, pages 1–6, March 2006.

[KM96] M. Kaufmann and J.S. Moore. Acl2: an industrial strength version of Nqthm. In *Proc. of the 11th Annual Conference on Computer Assurance, Systems Integrity, Software Safety, Process Security (COMPASS'1996)*, pages 23–34, June 1996.

[KMB97] M. Kaufmann, J. S. Moore, and O. Boyer. An industrial strength theorem prover for a logic based on common Lisp. *IEEE Trans. on Software Engineering*, 23:203–213, 1997.

[Kro99] T. Kropf. *Introduction to Formal Hardware Verification.* Springer, 1999.

[KS00] J.H. Kukula and T.R. Shiple. Building circuits from relations. In E. Allen Emerson and A. Prasad Sistla, editors, *CAV*, volume 1855 of *Lecture Notes in Computer Science*, pages 113–123. Springer, 2000.

[LJ88] W. Luk and G. Jones. The derivation of regular synchronous circuits. In *Proc. of the International Conference on Systolic Arrays*, pages 305–314, May 1988.

[LT10] L. Li and M.A. Thornton. *Digital System Verification: A Combined Formal Methods and Simulation Framework.* Morgan and Claypool, 2010.

[MABBZ08] K. Morin-Allory, M. Boulé, D. Borrione, and Z. Zilic. Proving and disproving assertion rewrite rules by automated theorem proving. In *IEEE International High Level Design Validation and Test Workshop (HLDVT'2008)*, pages 56–63, November 2008.

[MAGB07] K. Morin-Allory, E. Gascard, and D. Borrione. Synthesis of property monitors for online fault detection. *Journal of Circuits, Systems and Computers*, 16(06):943–960, 2007.

[MAJB15] K. Morin-Allory, F.N. Javaheri, and D. Borrione. Efficient and correct by construction assertion-based synthesis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, DOI. 10.1109/TVLSI.2014.2386212:1–12, 2015.

[MB08] L. De Moura and N. Bjørner. *Z3: An Efficient SMT Solver*, volume 4963/2008 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin, April 2008.

[McC03] W. McCune. OTTER 3.3 reference manual. In *http://www.cs.unm.edu/ mccune/otter/Otter33.pdf*, 2003.

[McM93] K.L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[Mil72] R. Milner. Logic for computable functions: Description of a machine implementation. Technical report, Stanford, CA, USA, 1972.

[Mil94] G. Milne. *Formal Specification and Verification of Digital Systems.* McGrow-Hill, 1994.

[MIN] MiniSAT, http://minisat.se/Papers.html, accessed 2015.

[MJML14] Ruben Martins, Saurabh Joshi, Vasco Manquinho, and Inês Lynce. Incremental cardinality constraints for MaxSAT. In Barry O'Sullivan, editor, *Principles and Practice of Constraint Programming*, volume 8656 of *Lecture Notes in Computer Science*, pages 531–548. Springer International Publishing, 2014.

[MS95] Joao Marques-Silva. *Search algorithms for satisfiability problems in combinational switching circuits.* PhD thesis, University of Michigan, 1995.

[MY60] R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. In *IEEE Trans Comput*, volume C9, pages 39–47, March 1960.

[Öbe99] J. Öberg. *ProGram : A Grammar-Based Method for Specification and Hardware Synthesis of Communication Protocols.* PhD thesis, KTH, Sweden, 1999.

[Odd09] Y. Oddos. *Verification semi-formelle et synthèse automatique de circuits a partir de specifications temporelles ecrites en PSL.* PhD thesis, Univ. of Grenoble, Nov 2009.

[OH02] M.T. Oliveira and A.J. Hu. High-level specification and automatic generation of IP interface monitors. In *Proc. of the 39th Design Automation Conference (DAC'2002)*, pages 129–134, 2002.

[OLP85] Orna O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proc. of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'1985)*, pages 97–107, New York, NY, USA, 1985. ACM.

[OMAB07] Y. Oddos, K. Morin-Allory, and D. Borrione. Prototyping generators for on-line test vector generation based on PSL properties. In *Design and Diagnostic of Electronic Circuits and Systems (DDECS'2007)*, pages 1–6, April 2007.

[OMAB08] Y. Oddos, K. Morin-Allory, and D. Borrione. Assertion-based design with Horus. In *6th ACM-IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'2008)*, pages 75–76, June 2008.

[OMAB09] Y. Oddos, K. Morin-Allory, and D. Borrione. SyntHorus: Highly efficient automatic synthesis from PSL to HDL. In *Proc. of the 17th IFIP International Conference on Very Large Scale Integration (VLSI-SoC'2009)*, pages 83–88, October 2009.

[ONE] OneSpin360DV, http://www.onespin-solutions.com/index.php/assertion-synthesis.html, accessed 2015.

[ORS92] S. Owre, J.M. Rushby, and N. Shankar. PVS: A prototype verification system. In *Automated Deduction—CADE-11*, pages 748–752. Springer, 1992.

[OVA] OpenVera Assertion, http://www.synopsys.com/Tools/Verification/Documents/ova_wp.pdf, March 2003.

[OVL] Accellera Standard OVL V2, https://eda-playground.readthedocs.org/en/latest/_downloads/ovl_lrm.pdf, March 2014.

[PH09] D.K. Pradhan and I.G. Harris. *Practical Design Verification.* Cambridge University Press, 2009.

[PLN05] M. Pellauer, M. Lis, and R. Nikhil. Synthesis of synchronous assertions with guarded atomic actions. In *Proc. of the 3rd ACM and IEEE International Conference on Formal Methods and Models for Co-Design, (MEMOCODE'2005)*, pages 15–24, July 2005.

[PPS06] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. In E.A. Emerson and K. Namjoshi, editors, *Verification, Model Checking, and Abstract Interpretation*, volume 3855 of *Lecture Notes in Computer Science*, pages 364–380. Springer Berlin Heidelberg, 2006.

[PPSQ13] L. Pierre, F. Pancher, R. Suescun, and J. Quevremont. On the effectiveness of assertion-based verification in an industrial context. In C. Pecheur and M. Dierkes, editors, *Formal Methods for Industrial Critical Systems*, volume 8187 of *Lecture Notes in Computer Science*, pages 78–93. Springer Berlin Heidelberg, 2013.

[PR89] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'1989)*, pages 179–190, New York, NY, USA, 1989. ACM.

[Rab72] M.S. Rabin. *Automata on Infinite Objects and Church's Problem.* American Mathematical Society, Boston, MA, USA, 1972.

[RAT] Rat, http://rat.fbk.eu/ratsy/, accessed 2015.

[Ray96] P. Raymond. Recognizing regular expressions by means of dataflow networks. In *Proc. of the 23rd International Colloquium on Automata, Languages, and Programming, (ICALP'1996)*, pages 336–347. Springer Verlag, 1996.

[SAT] SATRennesPA, http://satrennespa.irisa.fr/WebContent/, accessed 2015.

[SB94] A. Seawright and F. Brewer. Clairvoyant: A synthesis system for production-based specification. *IEEE TVLSI*, pages 172–185, June 1994.

[SDG] SLED SDG (Synthesizable Detector Generator), http://www.dolphin.fr/index.php/eda_solutions/applications/assertion_based_verification, accessed 2015.

[Sif82] J. Sifakis. A unified approach for studying the properties of transition systems. *Theoretical Computer Science*, 18(3):227–258, 1982.

[SM02] R. Siegmund and D. Müller. Automatic synthesis of communication controller hardware from protocol specifications. *IEEE Design & Test of Computers*, 19(4):84–95, 2002.

[SMB+05] J. Srouji, S. Mehta, D. Brophy, K. Pieper, S. Sutherland, and Work Group. *IEEE Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language.* pub-IEEE-STD, November 2005.

[SMDC06] S.Das, R. Mohanty, P. Dasgupta, and P.P Chakrabarti. Synthesis of system verilog assertions. In *Proc. of the conferance on Design, Automation and Test in Europe (DATE'2006)*, volume 2, pages 1–6, March 2006.

[SNBE07] M. Schickel, V. Nimbler, M. Braun, and H. Eveking. An efficient synthesis method for property based design in formal verification. In *Advances in Design and Specification Languages for Embedded Systems (Selected Contributions from FDL'2006)*, pages 163–181. Kluwer, 2007.

[SP01] R. Sidhu and V.K. Prasanna. Fast regular expression matching using FPGAs. In *Proc. of the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, FCCM'2001, pages 227–238, Washington, DC, USA, 2001. IEEE Computer Society.

[Tho68] K. Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968.

[TRA] Transistor count, http://en.wikipedia.org/wiki/Transistor_count, accessed 2015.

[UNB] Unbeast, http://www.react.uni-saarland.de/tools/unbeast/, accessed 2015.

[WVS83] P. Wolper, M.Y. Vardi, and A.P Sistla. Reasoning about infinite computation paths. In *Proc. of the 24th Annual Symposium on Foundations of Computer Science*, pages 185–194, November 1983.

[YAP10] J. Yuan, A. Aziz, and C. Pixley. *Constraint-Based Verification.* Springer, 2010.

[Zha97] Hantao Zhang. SATO: An efficient prepositional prover. In William McCune, editor, *Automated Deduction—CADE-14*, volume 1249 of *Lecture Notes in Computer Science*, pages 272–275. Springer Berlin Heidelberg, 1997.

Acronym

A

ABA	*Alternating Büchi Automaton*
ABV	*Assertion Based Verification*
ABS	*Assertion Based Design*
AMBA	*Advanced Microcontroller Bus Architecture*
AST	*Abstract Syntax Tree*
ASIC	*Application Specific Integrated Circuit*
ASP	*Answer Set Programming*
AP	*Atomic Proposition*
AIGER	

B

BDD	*Binary Decision Diagram*
BMC	*Bounded Model Checking*
BNF	*Backus-Naur Form*

C

CNF	*Conjunctive Normal Form*
CRC	*Cyclic Redundancy Check*
CTL	*Computational Tree Logic*

D

DAG	*Directed Acyclic Graph*
DAST	*Directed Abstract Syntax Tree*
DES	*Data Encryption Standard*
DFA	*Deterministic Finite Automaton*
$\mathcal{DG}$	*Dependency Graph*
DI	*Delay Insensitive*
DIMS	*Delay Insensitive Min-term Synthesis*
DTS	*Discrete Transition System*
DUV	*Design Under Verification*

F

FIFO	*First In First Out*
FPGA	*Field Programmable Gate Array*
FSM	*Finite State Machine*

FL *Foundation Language*

FBDD *Free Binary Decision Diagram*

FPA *Fast Prototyping From Assertions*

G

GDL *General Description Language*

GenBuf *Generalized Buffer*

H

HDL *Hardware Description Language*

HDLC *High-level Data Link Controller*

HLS *High Level Synthesis*

HOL *Higher Order Logic*

I

IEEE *Institute of Electrical and Electronics Engineers*

IP *Intellectual Property*

L

LCF *Logic for Computable Functions*

LTL *Linear Temporal Logic*

LUT *Look Up Table*

N

NFA *Non-deterministic Finite Automaton*

O

OBE *Optional Branching Extension*

OCP *Open Core Protocol*

OVA *Open Vera Assertion*

OVL *Open Verification Library*

P

PFG *Protocol Flow Graph*

PLA *Programmable Logic Array*

PSL *Property Specification Language*

PSL_{simple} *PSL Simple Subset*

PVS *Prototype Verification System*

R

RAT *Requirement Analysis Tool*

RTL *Register Transfer Level*

RE *Regular Expression*

S

SERE *Sequential Extended Regular Expression*

SoC *System on Chip*

SVA *SystemVerilog Assertions*

SAT *boolean SATisfiability problem*

SRGA *Self Reconfigurable Gate Array*

SDG *Synthesizable Detector Generator*

V

VHDL *Very high speed integrated circuit Hardware Description Language*

Publications

Journal paper

- K. Morin-Allory, F.N. Javaheri, and D. Borrione, Efficient and correct by construction assertion-based synthesis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, PP(99):1–1, 2015.

Publications in Refereed International Conference Proceedings

- K. Morin-Allory, F.N. Javaheri, and D. Borrione, Design Understanding with Fast Prototyping from Assertions. *1st Workshop on Design Automation for Understanding Hardware Designs (DUHDe 2014, Friday workshop at DATE 2014)*, Dresden, Germany, Mar 2014.

- K. Morin-Allory, F.N. Javaheri, and D. Borrione, Fast Prototyping from Assertions: a Pragmatic Approach. *Proceeding of the 11th ACM-IEEE International Conference on Formal Methods and Models for Codesign (Memocode'13)*, US, Oct 2013.

- K. Morin-Allory, F. Javaheri, and D. Borrione, SyntHorus-2: Automatic Prototyping from PSL. *Proceeding of the 21st IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC'13)*, Istanbul, Turkey, Oct 2013.

Posters

- F. Javaheri, K. Morin-Allory, and D. Borrione, Revisiting Regular Expressions in SyntHorus2: from PSL SEREs to Hardware. *submitted as work in progress in Forum on specification & Design Languages (FDL)*, 2015.

- F. Javaheri, K. Morin-Allory, and D. Borrione, SyntHorus2: A Tool for Assertion-based Synthesis. *presented at the PhD Forum of 18th Design, Automation and Test in Europe Conference (DATE'15)*, Grenoble, France, March 2015.

- F. Javaheri, K. Morin-Allory, and D. Borrione, Designing from Assertions: from PSL Properties to a Compliant Hardware Prototype. *presented at the PhD Forum* *of 17th Design, Automation and Test in Europe Conference (DATE'14)*, Dresden, Germany, March 2014.

- F. Javaheri, K. Morin-Allory, A. Porcher, and D. Borrione, Automatic Prototyping of declarative properties on FPGA. *presented by D. Borrione at the Electronic System Level Synthesis Conference as invited presentation*, Austin, US, June, 2013.

- F. Javaheri, K. Morin-Allory, A. Porcher, and D. Borrione, Synthorus-2: Automatic Prototyping on FPGA from PSL. *presented at the University Booth of 16th Design, Automation and Test in Europe Conference*, Grenoble, France, 2013.

Abstract– The work presented in this thesis aims at automatically prototype communication and control designs from declarative temporal specifications. From a set of PSL[2] properties, we produce a synthesizable RTL design automatically. The proposed method is modular, in contrast to previously published methods that were based on automata theory. From each property, we produce a component that observes some operands and generates waveforms for the other operands: the *reactant*.

First, a library of primitive reactants has been provided for FL[3] and SERE[4] operators. To this goal, a dependency relation is defined for each operator that expresses the dependency among its operands using the operator's semantics. Then, the dependency relation of each operator is interpreted as a hardware component that implements the operator: the operator's primitive reactant.

Using this formalization, a method is proposed to automatically decide which signals of a property are observed and which are generated. In the cases when specifying the signal direction is not possible, a solver is implemented to identify the signal value. In addition, the way of identifying the value of the signal that is generated in several properties is addressed.

The final circuit is the interconnection of the properties' reactants and solvers.

A prototype tool SyntHorus2, which is an extension to HORUS, has been developed. It takes PSL properties as its inputs, and generates the synthesizable VHDL code of the circuit. In addition, it generates some complementary properties to verify if the set of specification is coherent and complete.

The method is efficient, and synthesizes control circuits in a few seconds. Results obtained on classical benchmarks show that our technique compiles properties more efficiently than previous prototype tools.

Keywords. PSL, assertion-based design, reactant, automatic synthesis, dependency graph, annotation, resolution, solver.

[2] Property Specification Language

[3] Foundation Language

[4] Sequential Extended Regular Expression

Résumé– Les travaux présentés dans cette thèse visent à produire automatiquement des prototypes de circuits de communication et de contrôle à partir de spécifications temporelles déclaratives. Partant d'un ensemble de propriétés écrites en langage PSL, nous produisons un modèle RTL synthétisable automatiquement. La méthode proposée est modulaire, contrairement aux méthodes publiées antérieurement qui étaient fondées sur la théorie des automates. Pour chaque propriété, nous produisons un composant qui observe certains opérandes et génère des chronogrammes pour les autres opérandes : le module réactif.

Tout d'abord, une bibliothèque des modules réactifs primitifs a été développée pour les opérateurs FL et SERE. Pour ce faire, une relation de dépendance a été définie pour chaque opérateur : fondée sur la sémantique de l'opérateur, elle exprime la dépendance entre ses opérandes. Ensuite, la relation de dépendance de chaque opérateur est interprétée comme un composant matériel qui met en œuvre l'opérateur : c'est le module réactif primitif de l'opérateur.

À l'aide de cette formalisation, nous proposons une méthode pour déterminer automatiquement quels signaux d'une propriété sont observés et lesquels sont générés. Dans le cas où il n'est pas possible de déterminer le sens du signal, un solveur est ajouté pour identifier la valeur du signal. Le solveur sert aussi à déterminer la valeur d'un signal généré par plusieurs propriétés. Le circuit final est l'interconnexion des modules réactifs et des solveurs pour l'ensemble des propriétés.

Un outil prototype, SyntHorus2, qui est une extension d'HORUS, a été mis développé. Il prend les propriétés PSL comme entrées et génère le code VHDL synthétisable du circuit. En outre, il génère des propriétés complémentaires pour vérifier si l'ensemble des spécifications est cohérent et complet.

La méthode est efficace et synthétise des circuits de commande en quelques secondes. Les résultats que nous avons obtenus sur des jeux d'essais classiques montrent que notre technique compile les propriétés plus efficacement que les outils prototypes qui l'ont précédée.

Mots-clés. PSL, conception basée sur les assertions, module réactif, synthèse automatique, graphe de dépendance, annotation, résolution, solveur.

UNIVERSITÉ GRENOBLE ALPES

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES
Spécialité : **Nanoélectronique et Nanotechnologies**

Arrêté ministériel : 7 août 2006

Présentée par

Fatemeh (Negin) JAVAHERI

Thèse dirigée par **Mme. Dominique BORRIONE**
et codirigée par **Mme. Katell MORIN-ALLORY**

Préparée au sein du **Laboratoire TIMA**
Dans **l'École Doctorale Electronique, Electrotechnique, Automatique** & **Traitement du Signal (E.E.A.T.S)**

Synthèse automatique de circuits numériques à partir de spécifications temporelles

Thèse soutenue publiquement le **1 octobre 2015**,
devant le jury composé de :

M. Philippe COUSSY
Professeur, Université de Bretagne Sud, Président
M. Paolo PRINETTO
Professeur, Politecnico di Torino, Rapporteur
M. Rolf DRECHSLER
Professeur, Universität Bremen, Rapporteur
Mme. Dominique BORRIONE
Professeur, Université Joseph Fourier, Directrice de thèse
Mme. Katell MORIN-ALLORY
Maître de Conférences, Grenoble INP, Co-Directrice de thèse

Table des matières

Chapitre 1

Introduction

Jour après jour, l'influence des systèmes électroniques sur nos vie augmente. Toutefois, construire un circuit numérique correcte du premier coup est un objectif difficile lorsque l'on considère les architectures actuelles.

Le travail présenté dans cette thèse propose une méthode et un outil prototype pour aider à la vérification des protocoles de contrôle et de communication entre les modules.

Un inconvénient des méthodes de vérification formelle est qu'elles ne peuvent être utilisées qu'après la conception du système. L'objectif principal de ce travail est de proposer des méthodes pour générer un système à partir de ses spécifications et de vérifier son exactitude au cours du processus de production de matériel*correct par construction*.

La figure 1.1 montre une vue très abstraite du flot de conception traditionnelle.

Un processus de conception typique commence par considérer le comportement informel du système. Ensuite, les équipes de conception développent une première implémentation. Lors de l'étape suivante, l'implémentation doit être vérifiée pour considérer si elle est conforme aux spécifications. Après vérification, plusieurs améliorations du circuit peuvent être nécessaires en raison des erreurs détectées.

Une quantité importante de temps au cours du processus de conception est dépensée pour découvrir des erreurs, généralement par simulation ou émulation.

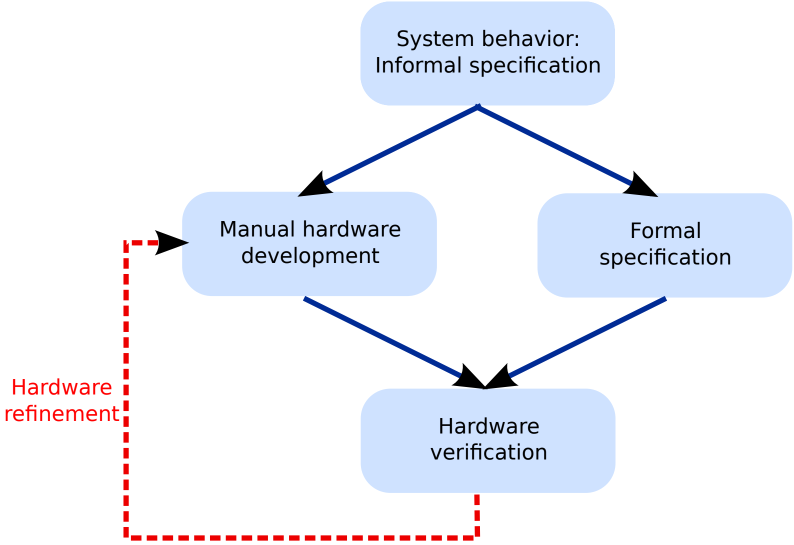


FIGURE 1.1 – Flot de conception classique

1.1 Le flot de conception proposée

Les difficultés du flot de conception classique nous ont amenés vers la synthèse à partir d'assertions (*Assertion Based Synthèse*, ABS) : la production directe de modules conformes (contrôle et communication) à un ensemble d'assertions. Chaque propriété est considérée comme la spécification d'un module à concevoir. L'objectif est alors de concevoir directement un code RTL synthétisable à partir de ses assertions.

En ABS, les propriétés sur le comportement d'un composant (*assertions*) ou de son environnement (*hypothèses*) spécifient les caractéristiques fonctionnelles d'entrée-sortie des modules et les communications entre les parties du système.

Dans le flot de conception proposée nous commençons la conception à partir d'un niveau plus abstrait, et intégrons la vérification dans le processus de conception (voir fig. 1.2). Dans cette méthode, les tâches de conception et de vérification ont été unifiées ; un circuit *correct par construction* est généré directement à partir des spécifications formelles.

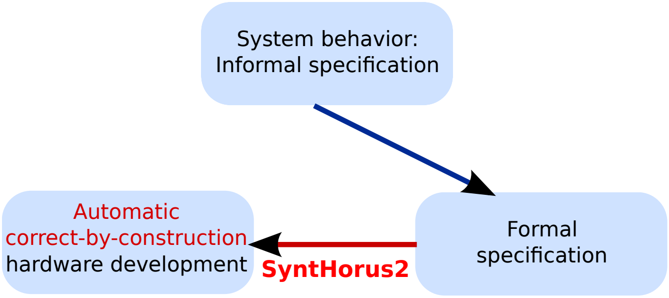


FIGURE 1.2 – Le flot de conception proposé

Le circuit généré est appelé *composant reactif* : il *réagit* aux stimuli sur ses entrées et produit des stimuli sur ses sorties, conformes aux assertions.

Chapitre 2

La vérification à base d'assertions

2.1 Introduction

Généralement, une *assertion* (propriété) sur le circuit exprime le comportement du circuit, et se réfère à des propriétés qui doivent être vérifiées.

Les assertions peuvent être vérifiées à la fois en simulation et en vérification formelle. Les assertions peuvent être exprimées en utilisant un langage de propriétés temporelles. Dans les sections suivantes, nous examinons les techniques de vérification.

2.2 Techniques de vérification

2.2.1 Vérification par simulation

La vérification par simulation est appliquée à un sous-ensemble représentatif des valeurs de signaux et des comportements d'un circuit, pour vérifier si l'implémentation se comporte correctement par rapport à sa spécification [Kro99].

En utilisant ce procédé, des erreurs peuvent être masquées en raison des stimuli ; elles peuvent apparaître en utilisant un autre stimulus, ou en exécutant la simulation pendant plus de cycles.

2.2.2 La vérification formelle

L'objectif de la vérification formelle est de considérer formellement si une *implémentation* satisfait la *spécification*.

Dans la vérification formelle, à la fois les spécifications et les implémentations sont converties en modèles mathématiques.

En utilisant le modèle mathématique du circuit et son comportement, la vérification formelle doit prouver mathématiquement que la conception satisfait la spécification de son comportement, ou qu'une relation existe entre les deux. Si il existe un bug de conception, les techniques de vérification formelle produisent un contre-exemple pour faciliter le processus de débogage.

2.2.2.1 Model checking

La vérification de modèle (*model checking*) a été développée à l'origine en 1981 par Clarke, Emerson et Sifakis [CE82, CES86, Sif82]. Le model checking est une technique automatique pour la vérification des systèmes réactifs à états finis, tels que les circuits séquentiels et les protocoles de communication. Le model checking consiste en un modèle du système, une logique temporelle, et un algorithme de vérification.

Le model checking peut être *explicite* (tout l'espace d'état est énuméré) ou *implicite* (l'espace d'état est modélisé avec des structures de données symboliques telles que BDD).Le model checking implicite (symbolique) est généralement plus puissante.

Bounded model checking

Le model checking borné a été introduit par Biere *et al.* dans [BCC$^+$99]. Il est basé sur les méthodes de satisfiabilité (SAT) . L'idée essentielle pour vérifier une propriété sur un système de transition fini est de rechercher des contre-exemples dans l'espace de toutes les exécutions de longueur k du système.

2.2.2.2 Equivalence checking

La vérification d'équivalence (Equivalence checking) est un procédé basé sur un modèle qui vérifie si deux descriptions d'une conception spécifient le même comportement, ce qui signifie qu'ils produisent des séquences de sorties identiques pour toutes les séquences d'entrée valides. Ces descriptions peuvent être dans différents niveaux d'abstraction.

2.3 Langages d'Assertions

Une spécification formelle est une description concise et abstraite du comportement et des propriétés d'un système. Cette description est écrite dans un langage mathématique et indique ce qu'un système est censé faire.

2.3.1 Language de Spécification de Propriétés (PSL)

Le language de spécification de propriétés PSL (Property Specification Language) [FG05] est la normalisation par Accelera, puis par l'IEEE, du language “Sugar” développé par IBM [BBDE$^+$01].

Une propriété PSL se compose de quatre couches : *Booléenne*, *temporelle*, *vérification* et *modélisation* (voir fig. 2.1).

Nous nous concentrons sur la couche temporelle PSL.

2.3.1.1 Couche temporelle de PSL

La couche temporelle est utilisée pour définir les propriétés qui décrivent le comportement de la conception ou de l'environnement au cours du temps. Elle est utilisée pour décrire les comportements temporels construits à partir d'expressions booléennes et d'opérateurs temporels.

La figure 2.2 montre les quatre couches d'une propriété PSL.

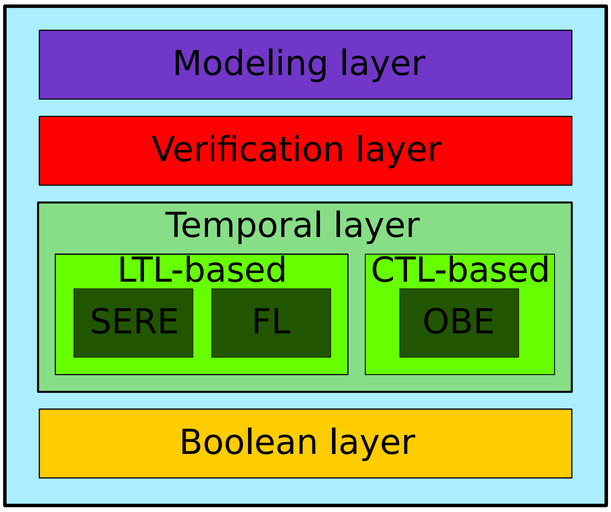
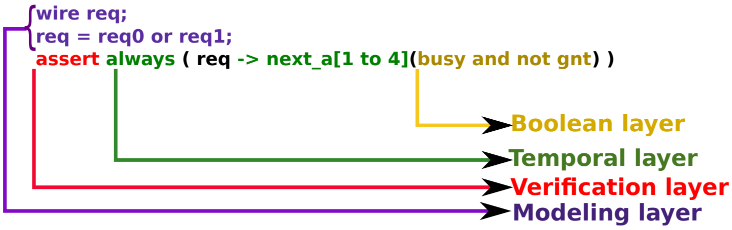


FIGURE 2.1 – Les couches PSL



```
wire req;
req = req0 or req1;
assert always ( req -> next_a[1 to 4](busy and not gnt) )
```

FIGURE 2.2 – Différentes couches d'une propriété PSL

2.3.1.2 Sous-ensembles simple de PSL (PSL$_{simple}$)

Le sous-ensemble simple de PSL, PSL$_{simple}$, est un sous-ensemble qui est conforme à la notion de progression monotone de temps, de gauche à droite à travers la propriété. Les propriétés situées dans le sous-ensemble peuvent être simulées facilement.

2.3.2 System Verilog Assertion (SVA)

Les assertions SystemVerilog [SMB$^+$05] sont intégrées dans SystemVerilg, elles peuvent être utilisées avec d'autres structures de langage. SVA a été défini en même temps que PSL, également à partir de "Sugar", mais est limité aux expressions régulières Seres. Il partage la plupart des primitives de base de PSL, mais sa syntaxe est différente.

Chapitre 3

État de l'art

3.1 Synthèse de propriété sous forme de moniteurs

Un moniteur observe les signaux qui sont des opérandes d'une propriété, et sort le statut de la propriété. Par conséquent, tous les opérandes sont des entrées du moniteur.

Il existe deux méthodes pour synthétiser des moniteurs : soit à partir d'automates, soit une méthode modulaire. Les deux méthodes prennent en charge les FLs et des expressions régulières.

3.1.1 L'approche basée sur l'automate

Les moniteurs sont des machines à états finis qui acceptent ou rejettent certaines traces de simulation.

Traduire des expressions régulières (RE) en automates a été fait dans [MY60] et [Tho68]. La traduction de LTL vers les automates a été prise en compte dans [WVS83].

Sidhu et Prasanna ont présenté une implémentation matérielle d'adaptateurs de RE pour les FPGA [SP01]. Cette approche utilise la méthode de McNaughton-Yamada [MY60] pour construire des NFAs à partir de REs.

Gascard dans [Gas05] propose une méthode pour transformer les SEREs en DFA. Le travail est basé sur les dérivées d'expressions régulières introduites par Brzozowski dans [Brz64].

Le travail présenté dans [GG05] traite de la traduction d'un sous-ensemble de PSL SEREs en moniteurs. Pour chaque opérateur de ce sous-ensemble, une fonction est implémentée qui construit l'automate non-déterministe correspondante.

A notre connaissance, l'approche la plus efficace dans la synthèse de moniteurs de PSL SEREs se fait par Boulé et Zilic [BZ07, BZ08b, BZ08c].

3.1.2 L'approche modulaire

L'implémentation PSL des Seres en utilisant l'approche modulaire a été effectuée par Morin-Allory *et al.* dans [MAGB07] pour la détection de fautes en ligne.

L'approche modulaire introduite par Morin-Allory et Borrione dans [KAB06] génère des moniteurs pour les propriétés temporelles PSL. Dans ce procédé, le sous-ensemble simple de PSL est considéré. Chaque opérateur de PSL dans ce sous-ensemble est implémenté en tant que module VHDL synthétisable, avec une interface générique. Une propriété PSL est généré par l'interconnexion des sous-modules des opérateurs en suivant l'arbre syntaxique abstrait de la propriété. Un prototype d'outil, HORUS, a été développé pour la construction automatique d'un environnement de test pour la conception [OMAB08, Odd09].

3.2 Synthèse de propriété comme circuits corrects par construction

Dans cette section, une propriété est considérée comme la spécification du module à concevoir. L'objectif est alors de produire la conception RTL synthétisable de ses assertions directement.

3.2.1 L'approche basée sur l'automate

Le problème a d'abord été traité par Büchi [BL69], puis par Rabin [Rab72]. Ces approches construisent les automates des propriétés, et réduisent le problème de la synthèse au problème de la vacuité des automates. Si un automate non-vide peut être trouvé pour les spécifications, son circuit correspondant est produit.

Bien sûr la synthèse automatique à partir des spécifications n'est pas une nouveauté, elle a récemment été appliquée à des circuits réels, à travers le développement d'outils de prototypes [PPS06, BGJ$^+$07a, RAT, FJR11, EKH12].

Bloem *et al.* définissent un sous-ensemble de LTL ("Generalized Reactivity(1)") dont les propriétés sont converties en automates [BGJ$^+$07a, BGJ$^+$07b]. Ils utilisent la méthode de jeu à deux joueurs. Les algorithmes de la théorie des jeux calculent tous les comportements corrects du circuit pour toutes les interactions possibles avec l'environnement. Le travail est plus tard étendu et amélioré dans [BJP$^+$12], pour obtenir de plus petits circuits.

Ces méthodes ont été mises en œuvre et des outils de prototypes ont été fournis pour la synthèse de la propriété. Brièvement, Lily et Anzu ont été mis en œuvre sur la base des recherches dans [JB06, PPS06], puis amélioré pour ratsy.

3.2.2 L'approche modulaire

Le sujet a été examiné par Oddos *et al.* dans [OMAB09], et une solution provisoire a été proposée pour synthétiser les circuits de contrôle de propriétés temporelles PSL [Odd09]. La méthode est modulaire, chaque propriété est transformée en un composant combinant les caractéristiques des moniteurs et générateurs : le générateur étendu. Un sous-module VHDL synthétisable est prévu pour chaque opérateur dans PSL$_{simple}$. Chaque propriété est l'interconnexion des sous-modules de ses opérateurs. Le conception finale est l'interconnexion des modules de propriété, et il est correct par construction.

3.3 Outils existants

Il y a une grande variété d'outils de vérification formelle. Selon l'outil utilisé, les propriétés peuvent être exprimées en PSL, LTL, ou CTL. OneSpin [one], Mentor Graphics 0-In, Cadence Incisive [cad] et RuleBased d'IBM [HIL04] sont parmi les outils les plus connus qui peuvent être utilisés pour la vérification formelle.

Pour la compilation des assertions en moniteurs, des outils industriels existent : IBM FOCS [ABG$^+$00], BugScope [Bug] développé par Atrenta, SLED SDG (synthesizable Detector Generator) développé par Dolphin Intégration. En plus des outils industriels mentionnés, il existe des outils académiques pour compiler des assertions en moniteurs : MBAC développé par Boulé [BZ08a] à l'université McGill, Horus [OMAB08] développé dans le groupe VDS de TIMA Lab.

Dans le cadre dela synthèse correcte par construction, il y a peu d'outils.

Acacia$^+$ [ACA] est basé sur les travaux dans [FJR09, FJR11]. Il saisit les spécifications de LTL, et émet un design dans le format *dot* .

Unbeast [UNB] est basé sur les travaux dans [EKH12]. Il saisit les spécifications de LTL dans une syntaxe XML et produit un fichier intermédiaire NuSMV qui est transformé en un format *AIG* par AIGER.

Ratsy (Requirements Analysis Tool with Synthesis) [RAT, BCG$^+$10] est une mise à jour de Rat qui est développé par Bloem *et al.* à l'Université de Gratz. Ratsy entre les propriétés en *GR (1)* PSL$_{simple}$. Les propriétés doivent être partitionnées en un ensemble d'assertions (*garantee*) et un ensemble d'hypothèses (*assume*). Il produit une conception Verilog.

SyntHorus est la version étendue de Horus qui est développée dans le laboratoire TIMA[Odd09]. Au contraire d'autres méthodes existantes, l'outil est basé sur l'approche modulaire, et pourrait synthétiser des propriétés FL PSL$_{simple}$en VHDL.

Dans cette thèse, SyntHorus a été amélioré pour SyntHorus2. Cette nouvelle version prend des propriétés PSL et génère automatiquement le circuit de VHDL synthétisable. Il supporte les propriétés FLs, et prend également en charge partiellement les SEREs.

Chapitre 4

Prototypage rapide d'assertions : le flot global de synthèse

Dans ce chapitre, notre flot de synthèse global est expliqué, et un exemple de fonctionnement est introduit.

4.1 Synthèse de composants réactifs

Dans ce travail, une méthode correcte par construction est proposée pour produire directement une conception RTL synthétisable à partir de ses assertions. La méthode est modulaire ; i.e. le composant réactif de chaque propriété est l'interconnexion des modules de ses opérateurs. Par conséquent, les modules des opérateurs sont appelés *composants réactifs primitifs.*

Ensuite, le circuit final est l'interconnexion des composants réactifs des propriétés.

La fig. 4.1 montre le flot de synthèse globale qui produit un circuit à partir d'un ensemble de propriétés.

Nous avons fourni un prototype d'outil, SyntHorus2 qui met en œuvre le processus de synthèse ci-dessus : il faut un ensemble de propriétés PSL en entrée, et il génère le circuit final en VHDL. Il génère également des propriétés complémentaires afin de vérifier si l'ensemble des propriétés est complet et cohérent (voir fig. 4.1). La méthode proposée est applicable aux parties contrôles d'une conception.

4.2 Exemple d'Exécution : le Generalized Buffer

Ici, nous introduisons le Generalized Buffer d'IBM [IBM] (*GenBuf*).

4.2.1 Présentation

GenBuf est un arbitre qui séquentialise les demandes provenant de *nbsend* émetteurs, et les transmet à un moment à *nbrec* récepteurs (*nbsend* et *nbrec* sont des paramètres génériques). Chaque émetteur a son propre bus, et les récepteurs partagent le même bus. Une FIFO (de profondeur 4 sur 32 bits de données) stocke les données entrantes en attente d'envoi vers les récepteurs.

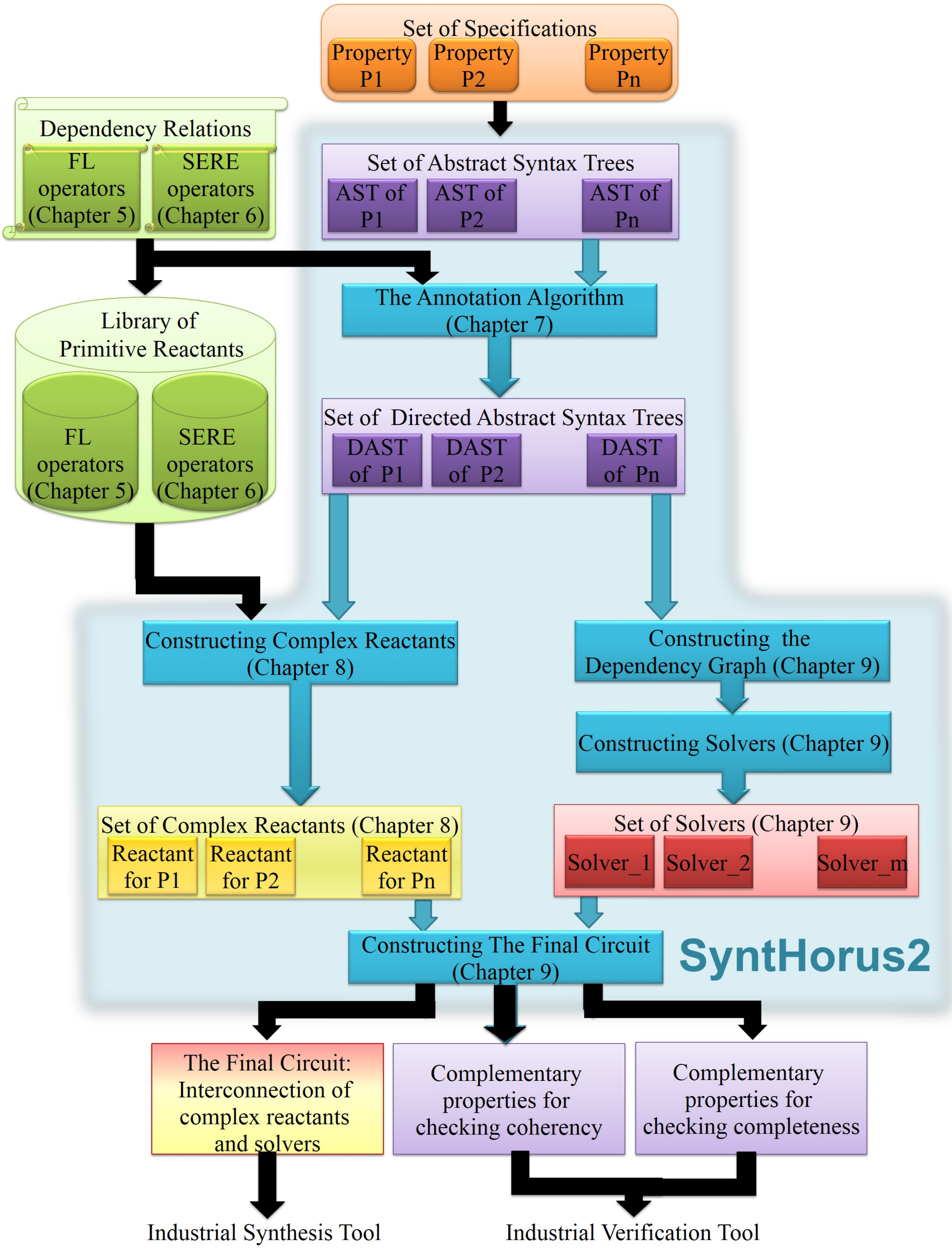

Set of Specifications
Property P1
Property P2
Property Pn
Dependency Relations
FL operators (Chapter 5)
SERE operators (Chapter 6)
Set of Abstract Syntax Trees
AST of P1
AST of P2
AST of Pn
Library of Primitive Reactants
FL operators (Chapter 5)
SERE operators (Chapter 6)
The Annotation Algorithm (Chapter 7)
Set of Directed Abstract Syntax Trees
DAST of P1
DAST of P2
DAST of Pn
Constructing Complex Reactants (Chapter 8)
Constructing the Dependency Graph (Chapter 9)
Constructing Solvers (Chapter 9)
Set of Complex Reactants (Chapter 8)
Reactant for P1
Reactant for P2
Reactant for Pn
Set of Solvers (Chapter 9)
Solver_1
Solver_2
Solver_m
Constructing The Final Circuit (Chapter 9)
SyntHorus2
The Final Circuit: Interconnection of complex reactants and solvers
Complementary properties for checking coherency
Complementary properties for checking completeness
Industrial Synthesis Tool
Industrial Verification Tool

FIGURE 4.1 – Flot global de synthèse

Un contrôleur communique avec tous les modules et la FIFO : il applique une politique de sélection par tourniquet du côté des récepteurs, il bloque les émetteurs lorsque la FIFO est pleine, et bloque les récepteurs lorsque la FIFO est vide. La figure 4.2 affiche l'architecture du système et les signaux de commande d'interface qui sont utilisés pour la communication.

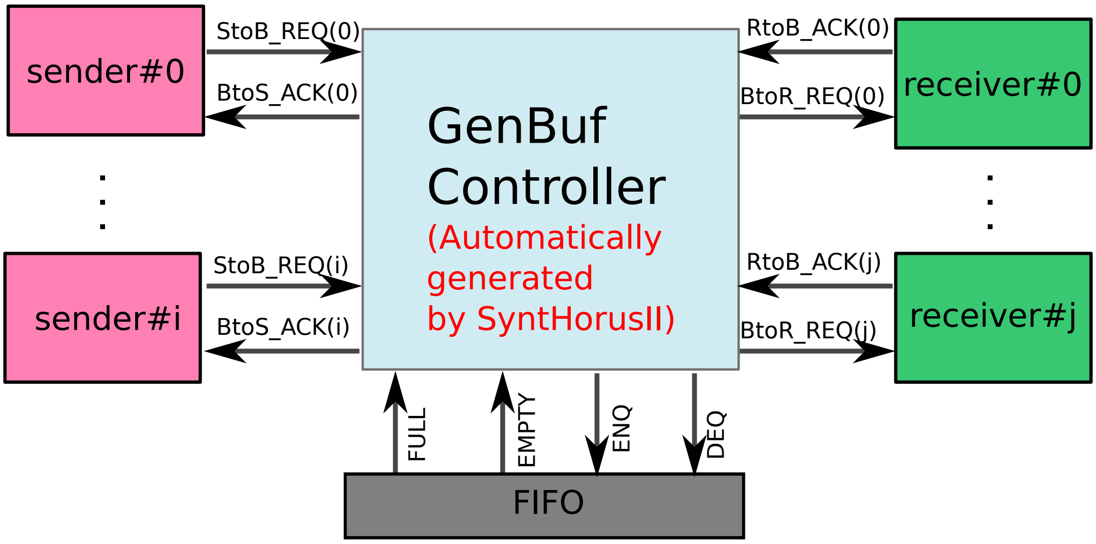


FIGURE 4.2 – Interface de circuit GenBuf

4.2.2 Communication avec les récepteurs

Le GenBuf interagit avec les récepteurs à travers un protocole 4 phases. Le mécanisme d'arbitrage qui est utilisé par GenBuf est le tourniquet : GenBuf ne demande pas le même récepteur deux fois consécutives. La figure 4.3 montre un chronograme par poignée de main entre le récepteur et le GenBuf.

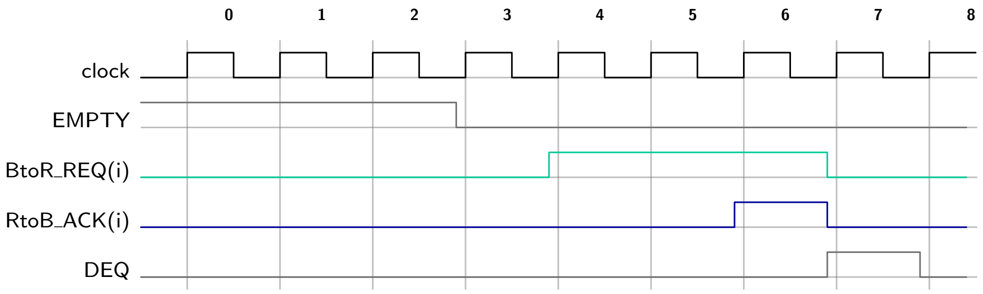


FIGURE 4.3 – Un exemple de chronogramme pour le récepteur

4.2.2.1 Spécification formelle FL

L'ensemble des propriétés FL qui spécifient la communication entre le contrôleur GenBuf, les récepteurs et la FIFO sont présentées figure 4.4 (pour 2 récepteurs).

```
vunit genbuf_receiver
{
------ receiver side
  P0_rec:
    always(not EMPTY -> next!(BtoR_REQ(0) or ( BtoR_REQ(1))));

  P1_rec:
    always(EMPTY -> next!(not BtoR_REQ(0) and (not BtoR_REQ(1))) );

  P2_rec:
    always (not BtoR_REQ(0) or not BtoR_REQ(1));

  P3_rec_0:
    always ( rose (BtoR_REQ(0)) -> next! (next_event! (prev(not BtoR_REQ(0)
        )) (not BtoR_REQ(0) until_ (BtoR_REQ(1)))));

  P3_rec_1:
    always ( rose (BtoR_REQ(1)) -> next! (next_event! (prev(not BtoR_REQ(1)
        )) (not BtoR_REQ(1) until_ (BtoR_REQ(0)))));

  P4_rec_0:
    always ((BtoR_REQ(0)) and (not RtoB_ACK(0))-> next! (BtoR_REQ(0)));

  P4_rec_1:
    always ((BtoR_REQ(1)) and (not RtoB_ACK(1))-> next! (BtoR_REQ(1)));

  P5_rec_0:
    always ( (RtoB_ACK(0)) -> (next! (not BtoR_REQ(0))));

  P5_rec_1:
    always ( (RtoB_ACK(1)) -> (next! (not BtoR_REQ(1))));

---------- FIFO interface
  P6_FIFO_rec:
    always (( fell(RtoB_ACK(0)) or (fell(RtoB_ACK(1))) and not EMPTY) -> (
        DEQ));

  P7_FIFO_rec:
    always (not fell(RtoB_ACK(0)) and not fell(RtoB_ACK(1)) -> (not DEQ));
}
```

FIGURE 4.4 – Spécification FL de la communication GenBuf avec des récepteurs, dans le cas de deux récepteurs

4.2.2.2 Spécification formelle SERE

L'ensemble des propriétés en expressions régulières qui spécifient la communication entre le contrôleur GenBuf, les récepteurs et la FIFO sont présentées dans la fig. 4.5.

```
vunit genbuf_receiver_sere
{
  P0_sere_rec:
    always({not EMPTY} |=> {BtoR_REQ(0) or  BtoR_REQ(1)}!);

  P1_sere_rec:
    always({EMPTY} |=> {not BtoR_REQ(0) and not BtoR_REQ(1)}! );

  P2_sere_rec:
    always (not BtoR_REQ(0) or not BtoR_REQ(1));

  P3_sere_rec_0:
    always ( {not BtoR_REQ(0); BtoR_REQ(0); {BtoR_REQ(0)[*]; not BtoR_REQ
        (0)}} |=> {(not BtoR_REQ(0))[*]; (prev(BtoR_REQ(1)))}!);

  P3_sere_rec_1:
    always ( {not BtoR_REQ(1); BtoR_REQ(1); {BtoR_REQ(1)[*]; not BtoR_REQ
        (1)}} |=> {(not BtoR_REQ(1))[*]; (prev(BtoR_REQ(0)))}!);


  P4_sere_rec_0:
    always ( {BtoR_REQ(0) and (not RtoB_ACK(0))} |=> {BtoR_REQ(0)}!);

  P4_sere_rec_1:
    always ( {BtoR_REQ(1) and (not RtoB_ACK(1))} |=> {BtoR_REQ(1)}!);

  P5_sere_rec_0:
    always ( {RtoB_ACK(0)} |=> {not BtoR_REQ(0)}!);

  P5_sere_rec_1:
    always ( {RtoB_ACK(1)} |=> {not BtoR_REQ(1)}!);

  P6_sere_FIFO_rec:
    always ((fell(RtoB_ACK(0)) or (fell(RtoB_ACK(1))) and not EMPTY) -> (
        DEQ));

  P7_sere_FIFO_rec:
    always (not fell(RtoB_ACK(0)) and not fell(RtoB_ACK(1)) -> (not DEQ));

}
```

FIGURE 4.5 – Spécification SERE de la communication GenBuf avec des récepteurs, dans le cas de deux récepteurs

Chapitre 5

Synthèse FLs

Dans ce chapitre, nous expliquons comment synthétiser un opérateur temporel FL, et nous fournissons une bibliothèque de composants réactifs primitives pour les opérateurs FL.

5.1 Formalisation de l'annotation

5.1.1 Relation de dépendance : définition et notations

Pour prouver les relations de dépendance, nous utilisons les définitions sémantiques de PSL dans l'annexe B de la norme IEEE [FG05].

— $w \models$ `property` ("`property`" est vrai sur le mot w) : la sémantique des propriétés FL est définie par induction structurelle.

Définition 1. *Soit w une trace, A et B deux formules FL. La relation de dépendance entre A et B est définie comme suit :*

$$\lfloor A \lhd B \rfloor_w \iff w \models B \Rightarrow w \models A$$

Lorsque $\forall w, \lfloor A \lhd B \rfloor_w$ on peut écrire : $A \lhd B$.

Propriété 1. $\lfloor A \lhd B \rfloor_w \wedge \lfloor A \lhd C \rfloor_w \Leftrightarrow \lfloor A \lhd (B \,\texttt{or}\, C) \rfloor_w$

Propriété 2. $\lfloor A \lhd B \rfloor_w \vee \lfloor A \lhd C \rfloor_w \Leftrightarrow \lfloor A \lhd (B \,\texttt{and}\, C) \rfloor_w$

Propriété 3. $\lfloor (A \,\texttt{and}\, B) \lhd C \rfloor_w \Leftrightarrow \lfloor A \lhd C \rfloor_w \wedge \lfloor B \lhd C \rfloor_w$

Propriété 4. $\lfloor (A \,\texttt{or}\, B) \lhd C \rfloor_w \Leftrightarrow \lfloor A \lhd (C \wedge \neg B) \rfloor_w$

Propriété 5. $\lfloor A \lhd B \rfloor_w \Leftrightarrow \lfloor \neg B \lhd \neg A \rfloor_w$

Définition 2. *Soit φ une formule FL. A et B sont deux opérandes de φ. Soit w une trace. A dépend de B dans φ si : $\forall w, \lfloor \varphi \lhd true \rfloor_w \Leftrightarrow \lfloor A \lhd B \rfloor_w$.*

5.1.2 Relation de dépendance entre les opérandes de opérateurs FL

5.1.2.1 Always

Règle de dépendance 1. Always
$\varphi =$ `always` A, puis
$\lfloor \varphi \lhd \mathit{true} \rfloor_w$ *iff* $\forall i < |w|, \lfloor A \lhd \mathit{true} \rfloor_{w^{i\ldots}}$

5.1.2.2 Famille Next

Règle de dépendance 2. Next ![k]
$\varphi =$ `next!`$[k]A$, puis
$\lfloor \varphi \lhd \mathit{true} \rfloor_w$ *iff* $\lfloor A \lhd \mathit{true} \rfloor_{w^{k\ldots}}$

Règle de dépendance 3. Next_a !
$\varphi =$ `next_a!`$[i$ to $j]A$, puis
$\lfloor \varphi \lhd \mathit{true} \rfloor_w$ *iff* $\forall k \in [i..j], \lfloor A \lhd \mathit{true} \rfloor_{w^{k\ldots}}$

5.1.2.3 Famille Until

Règle de dépendance 4. Until !
$\varphi = A$ `until!` B, puis
$\lfloor \varphi \lhd \mathit{true} \rfloor_w$ *iff* $\exists k < |w|, \lfloor B \lhd \mathit{true} \rfloor_{w^{k\ldots}} \wedge \forall i < k, \lfloor A \lhd \neg B \rfloor_{w^{i\ldots}}$

Dans cette relation de dépendance, si A et B sont booléens, la relation de dépendance $\lfloor A \lhd \neg B \rfloor_{w^{i\ldots}}$ peut être inversée (voir la propriété 5).

Règle de dépendance 5. Until
$\varphi = A$ `until` B, puis

$$\forall w, \begin{cases} \exists k < |w|, \lfloor B \lhd \mathit{true} \rfloor_{w^{k\ldots}} \wedge \forall i < k, \lfloor A \lhd \neg B \rfloor_{w^{i\ldots}} \\ \text{or} \\ \forall i < |w|, \lfloor A \lhd \mathit{true} \rfloor_{w^{i\ldots}} \end{cases}$$

5.2 La synthèse de la relation de dépendance

Nous donnons une interprétation matérielle de la relation de dépendance $\lfloor \varphi \lhd \mathit{true} \rfloor_w$, où φ signifie un appel à l'un des opérateurs de FL, et Ω représente l'opérateur temporel FL.

5.2.1 Principes de construction d'un composant réactif primitif

Les composant réactifs primitifs ont une interface générique : ils prennent *clock* et *reset* comme signaux de synchronisation. Chaque composant réactif primitif a un signal de *start* pour son activation (voir fig. 5.1).

La sortie d'un composant réactif n'est *pas* la valeur d'un signal, mais le *trigger* qui sert à *commencer* le composant matériel primitif en charge de la génération ou de l'observation de la valeur du signal (voir fig. 5.1).

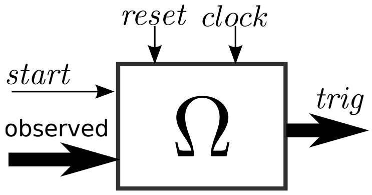


Figure 5.1 – Interface générique d'un composant réactif primitif

Le circuit $\mathcal{C}$ est le circuit qui met en oeuvre un composant réactif primitif. Dans ce chapitre, nous examinons comment synthétiser $\mathcal{C}$ pour des composant réactifs primitifs FL.

5.2.1.1 Composant Réactif Booléen

La figure 5.2 montre les quatre mises en oeuvre différentes pour un composant réactif booléen.

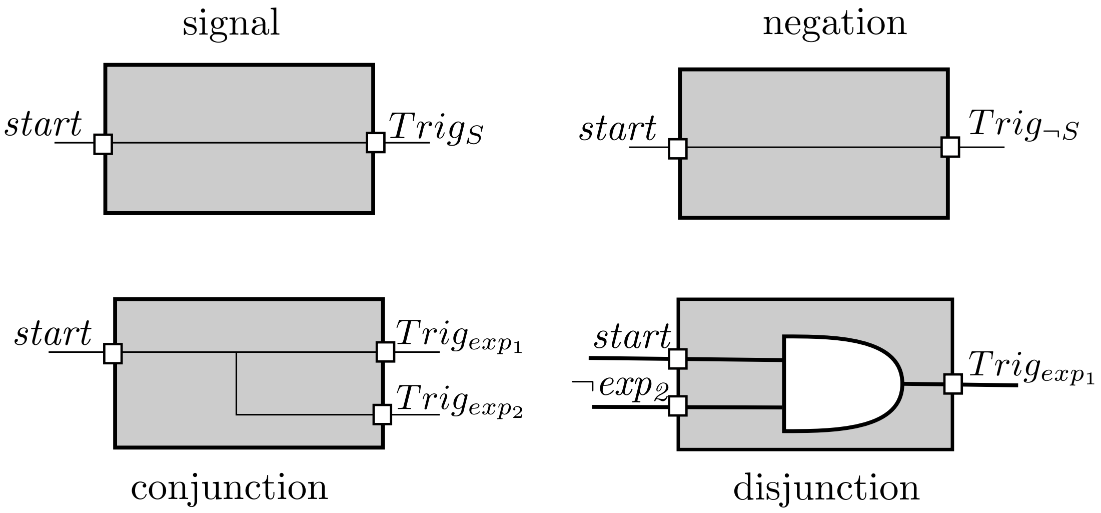


Figure 5.2 – Composant Réactif Booléen

5.2.2 Format générique d'un opérateur FL

Nous avons proposé un format générique pour tous les opérateurs de FL [MAJB15]. Ce format est basé sur la définition de la sémantique de l'opérateur.

Chaque dépendance est un cas particulier de l'un des deux expressions suivantes généralisées :

1 la famille *"forall"* comprend : : `always`, `until`, `next!`, `next_a`, `next_event`, `next_event_a`.

2 la famille *"exists"* comprend : : `eventually!`, `before`, `next_e`, `next_event_e`.

Les expressions "forall" et "exists" généralisées ont le format suivant :

$$\forall i \in [k_{min}, k_{max}], \lfloor exp \lhd cond \rfloor_{w^{k_i}} \tag{5.1}$$

$$\exists i \in [k_{min}, k_{max}], \lfloor exp \lhd cond \rfloor_{w^{k_i}} \tag{5.2}$$

Dans les formules ci-dessus, exp et $cond$ sont deux booléens, et min et max sont deux naturels tels que $max \geq min$. K_{min} et K_{max} sont calculés en utilisant une fonction de comptage, I_{th}. Le fonction I_{th} renvoie le nombre de fois que son opérande, une formule F calculée sur la trace w, a été $true$ sur $w^{0..k}$.

$$I_{th}(\lfloor F \lhd true \rfloor_{w^{k_i}}) = i \wedge \lfloor F \lhd true \rfloor_{w^{k_i}}, \forall i \in \mathbb{N}$$

La séquence $\{k_0, k_1, ..., k_i ...\}$ est l'ensemble de ces points dans le temps (voir fig. 5.3).

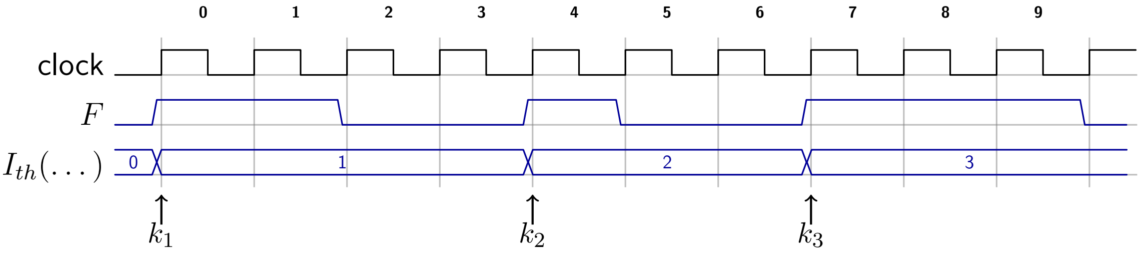


FIGURE 5.3 – Illustration de la fonction $I_{th}(\lfloor F \lhd true \rfloor_{w^{k_i}})$

Les valeurs de $min, max, exp, cond$ et F dépendent de l'opérateur temporel. Le tableau 5.1 donne leur valeur pour chaque opérateur PSL FL.

Temporal Operator	F	min	max	$cond$	exp	opt. _	opt. !
$\texttt{always}\, A$	$true$	0	$\mid w \mid$	$true$	A	no	no
$A\,\texttt{until}\,B$ (1)	B	0	1	$\neg B$	A	yes	yes
$A\,\texttt{before}\,B$ (1)	$\neg B$	0	1	$\neg A$	$\neg B$	yes	yes
$\texttt{next!}[i]A$	$true$	i	i	$true$	A	no	yes
$\texttt{next\_a}[i\ to\ j]A$	$true$	i	j	$true$	A	no	yes
$\texttt{next\_event}[i](B)A$	B	i	i	B	A	no	yes
$\texttt{next\_event\_a}[i\ to\ j](B)A$	B	i	j	B	A	no	yes
$\texttt{eventually!}\, A$	A	0	1	$true$	A	no	no
$A\,\texttt{until}\,B$ (2)	B	0	1	$\neg A$	B	no	yes
$A\,\texttt{before\_}\,B$ (2)	$\neg B$	0	1	B	A	no	yes
$\texttt{next\_e}[i\ to\ j]A$	$true$	i	j	$true$	A	no	yes
$\texttt{next\_event\_e}[i\ to\ j](B)A$	B	i	j	$true$	A	no	yes

TABLE 5.1 – Valeurs des paramètres pour forall (en haut) et existe (en bas) expressions

5.2.2.1 Mise en œuvre d'un opérateur du groupe "forall"

La figure 5.4 illustre la mise en oeuvre de l'expression de "forall".

- Le composant Dep (pour la dépendance $\lhd$) est une simple porte "AND" qui implémente l'expression $\lfloor exp \lhd cond \rfloor_{W^k}$. Il déclenche l'évaluation de exp en fonction de la valeur de $cond$.
- Le composant ForAll implémente l'expression $\forall i \in \lfloor K_{min}, K_{max} \rfloor$. Le signal ForAll.$Trig$ est actif en tout temps entre lb et ub. Selon que l'opérateur chevauche ou non, deux versions sont utilisées (voir fig. 5.5).
- Le composant Min(Max) prend $start$ et $cond$ en entrée. Le signaux de $start$ lance le comptage des occurrences de F sur son entrée Min.$cond$ (Max.$cond$).

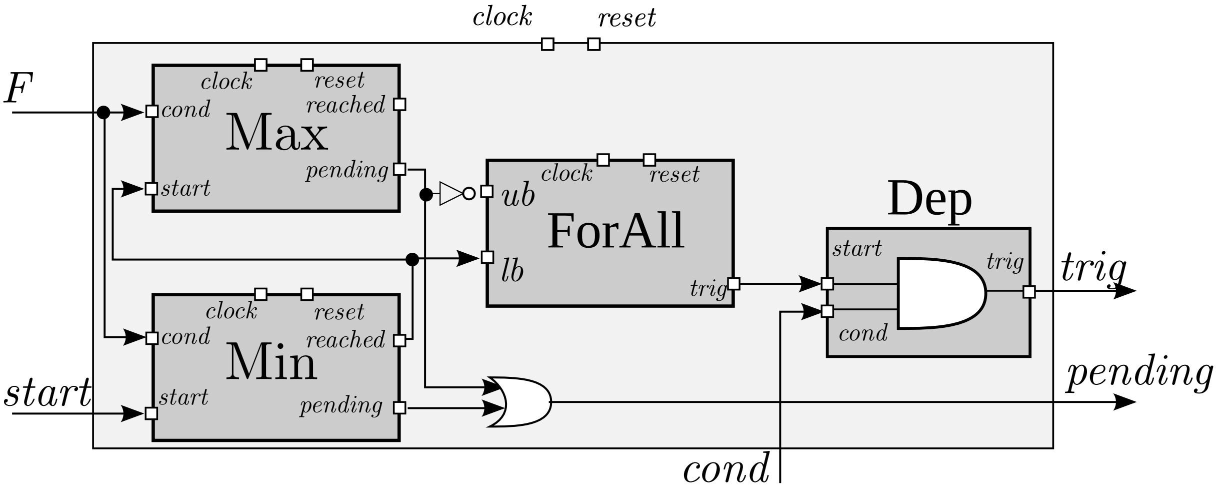
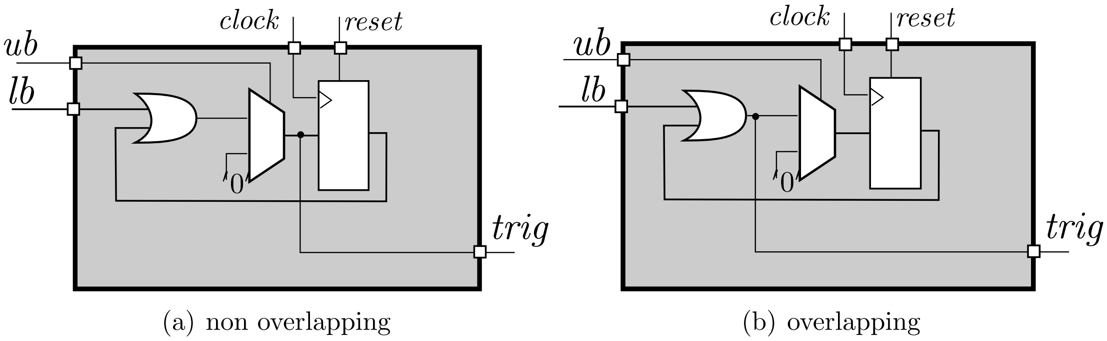


Figure 5.4 – La mise en oeuvre de l'expression de "forall"



(a) non overlapping

(b) overlapping

Figure 5.5 – La mise en oeuvre de $\forall i \in [lb, ub]$

5.2.2.2 Mise en œuvre d'un opérateur du groupe "exist"

La figure 5.6 illustre la mise en oeuvre de l'expression de "exist". Les composants Min et Max observent la formule F, et comptent le nombre d'occcurence dans l'interval $[K_{min}, K_{max}]$.

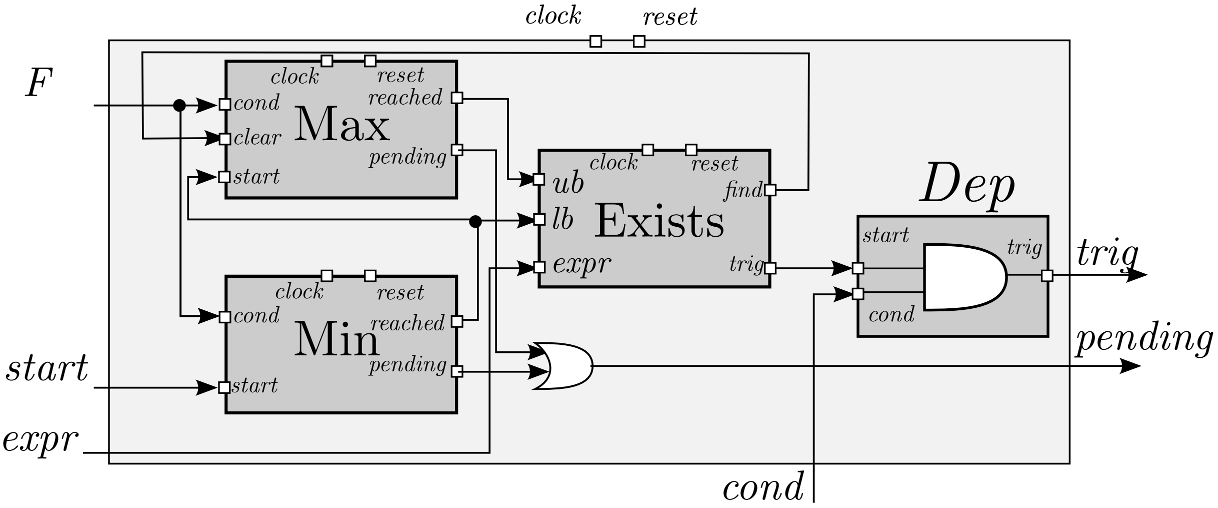


FIGURE 5.6 – La mise en oeuvre de l'expression de "exist"

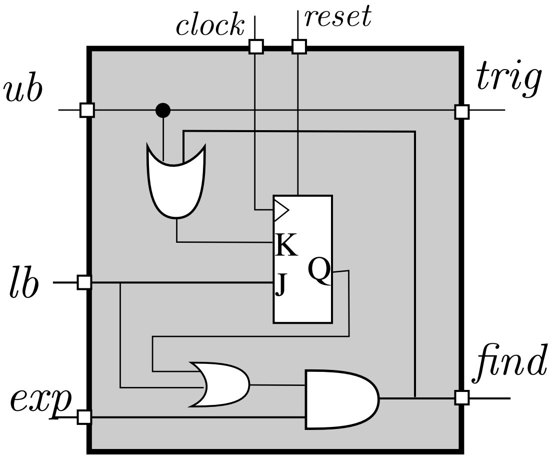


FIGURE 5.7 – La mise en oeuvre de $\exists i \in [lb, ub]$

Chapitre 6

Synthèse des SEREs

6.1 Introduction

Dans ce chapitre, nous examinons les principes de synthèse de SEREs. Les SEREs sont très similaires aux séquences dans SVA. Les SEREs sont une façon commode d'exprimer les formes d'ondes de signaux, en écrivant de simples propriétés de la forme :

$$\{observe\}\ |{=}{>}\ \{observe\}$$

$$\text{ou}$$

$$\{observe\}\ |{=}{>}\ \{generate\}$$

Ces propriétés peuvent représenter le comportement de l'environnement ou d'un protocole de communication.

6.2 Défis et motivations

Les SEREs ne peuvent pas toujours être traduites en FLs. En outre, certains comportements ou specifications en langue naturelle peuvent être exprimés plus facilement en utilisant des SEREs, et de manière plus compacte.

Exemple 1.

Considérez la propriété `P1`, où a, b, et c sont booléens :

```
P1: always {a} |=> {b[*]; c}
```

Supposons que a est observé et nous générons b et c. Alors, la question est : “quand devrions-nous arrêter de contraindre b à 1, et commencer à contraindre c à 1 ”? Si nous voulons générer c, la propriété n'est pas déterministe puisque c peut être contraint à 1 dans un cycle après $a = 1$. Si nous observons b et générons c, quand c doit il être contraint à 1 ? Cela peut dépendre d'autres propriétés.

Si nous observons c et générons b, b n'est plus contraint dès que c devient 1.

6.3 Formalisation de l'annotation

Deux relations de dépendance sont introduites pour chaque opérateur de SERE : une relation de dépendance pour exprimer quand une séquence est active, et une relation de dépendance pour exprimer lorsque la séquence est satisfaite.

6.3.1 Relation de dépendance : définition et notations

Définition 1. *Soit φ une SERE, et $Ended_\varphi$ un booléen qui devient 1 lorsque φ est satisfaite. w est une trace, et ℓ est la j^{th} lettre de w ($\ell = w^j$), telle que $\ell \vdash Ended_\varphi$. Ensuite, pour chaque séquence φ, nous pouvons dire :*

$$\forall w, \lfloor \varphi \lhd true \rfloor_{w^{i...j}} \Leftrightarrow \lfloor Ended_\varphi \lhd true \rfloor_{w^j}$$

La relation $\lfloor Ended_\varphi \lhd true \rfloor_{w^j}$ est ce qu'on appelle $E_\varphi Relation$.

6.3.2 Relation de dépendance entre les opérandes des opérateurs de SERE

Dans cette section, deux relations de dépendance sont données pour chaque opérateur SERE. La première relation de dépendance est appelée "*φRelation*", et exprime la dépendance entre sous-séquences de φ afin que φ soit satisfaite. La deuxième relation de dépendance est appelée "*$E_\varphi Relation$*" et exprime la dépendance entre les sous-séquences de φ au cycle qui φ complète. Cette dépendance est définie en utilisant $\lfloor \lhd \rfloor_{w^i}$ (voir la définition 1).

6.3.2.1 Les cas de base

exp est une expression booléenne, et A est une SERE :

$$\begin{aligned} &\lfloor exp \lhd true \rfloor_w \Leftrightarrow \lfloor exp \lhd true \rfloor_{w^0} \wedge \lfloor Ended_{exp} \lhd true \rfloor_{w^0} \\ &\lfloor \{A\} \lhd true \rfloor_w \Leftrightarrow \lfloor A \lhd true \rfloor_w \end{aligned}$$

6.3.2.2 Concaténation

Règle de dépendance 1. *φRelation pour la concaténation*
$\varphi = A; B$, puis :
$\lfloor \varphi \lhd true \rfloor_w$ *iff* $\exists i < |w|, \lfloor Ended_A \lhd true \rfloor_{w^i} \wedge \lfloor B \lhd true \rfloor_{w^{i+1...}}$

Règle de dépendance 2. *$E_\varphi Relation$ pour la concaténation*
$\varphi = A; B$, puis :
$\exists j < |w|, \lfloor Ended_\varphi \lhd true \rfloor_{w^j}$ *iff* $\exists k < j, \lfloor Ended_A \lhd true \rfloor_{w^k} \wedge \lfloor Ended_B \lhd true \rfloor_{w^j}$

Dans le cas particulier où B est un booléen, alors $j = k + 1$.

6.3.2.3 conjonction de deux séquences de même longueur

Règle de dépendance 3. *φRelation pour la conjonction de même longueur*
$\varphi = A \;\&\&\; B$, *puis*
$\lfloor \varphi \lhd true \rfloor_w$ *iff* $\lfloor A \lhd true \rfloor_w \wedge \lfloor B \lhd true \rfloor_w$

Règle de dépendance 4. *E_φRelation pour la conjonction de même longueur*
$\varphi = A \;\&\&\; B$, *puis*
$\exists j < |w|, \lfloor Ended_\varphi \lhd true \rfloor_{w^j}$ *iff* $\lfloor Ended_A \wedge Ended_B \lhd true \rfloor_{w^j}$

6.3.2.4 Clôture de Kleene

Règle de dépendance 5. *φRelation pour l'étoile*
$\varphi = A[*0]$, *puis*
$\lfloor \varphi \lhd true \rfloor_w$ *iff* $|w| = 0$
$\varphi = A[*]$, *puis*
$\lfloor \varphi \lhd true \rfloor_w$ *iff* $|w| = 0 \vee \exists i < |w|, \lfloor Ended_A \lhd true \rfloor_{w^i} \wedge \lfloor \varphi \lhd true \rfloor_{w^{i+1 \ldots}}$

6.3.2.5 Plus

Règle de dépendance 6. *φRelation pour le plus*
$\varphi = A[+]$, *puis*
$\lfloor \varphi \lhd true \rfloor_w$ *iff* $\exists i < |w|, \lfloor Ended_A \lhd true \rfloor_{w^i} \wedge \lfloor A[*] \lhd true \rfloor_{w^{i+1 \ldots}}$

Règle de dépendance 7. *E_φRelation pour le plus*
$\varphi = A[+]$, *puis*
$\exists j < |w|, \lfloor Ended_\varphi \lhd true \rfloor_{w^j}$ *iff* $\lfloor Ended_A \lhd true \rfloor_{w^j}$

6.4 La synthèse de la relation de dépendance

Afin de construire une bibliothèque de composants réactifs primitifs pour les opérateurs SERE, nous donnons une interprétation matérielle des deux relations de dépendance *φRelation* ($\lfloor \varphi \lhd true \rfloor_w$), et *$E_\varphi$Relation* ($\lfloor Ended_\varphi \lhd true \rfloor_{w^i}$).

6.4.1 Principes de la construction des composants réactifs primitifs

Les composants réactifs primitifs ont une interface générique. Le circuit correspondant pour chaque opérateur de SERE est l'interconnexion des circuits des relations *φRelation* et *E_φRelation* dépendance. Cette interconnexion est représentée sur la fig. 6.1.

Le circuit à gauche sur la fig. 6.1, $\mathcal{C}1$, implémente la *φRelation*.

Ensuite, le circuit droit $\mathcal{C}2$, qui met en œuvre *E_φRelation*, génère un signal qui indique si φ est satisfaite.

Le circuit $\mathcal{C}$ est le circuit qui met en oeuvre un composant réactif primitif, et l'interconnexion de $\mathcal{C}1$ et $\mathcal{C}2$.

Ici, nous expliquons chaque catégorie de SERE brièvement, puis nous discutons de la façon de construire leur matériel correspondant intuitivement.

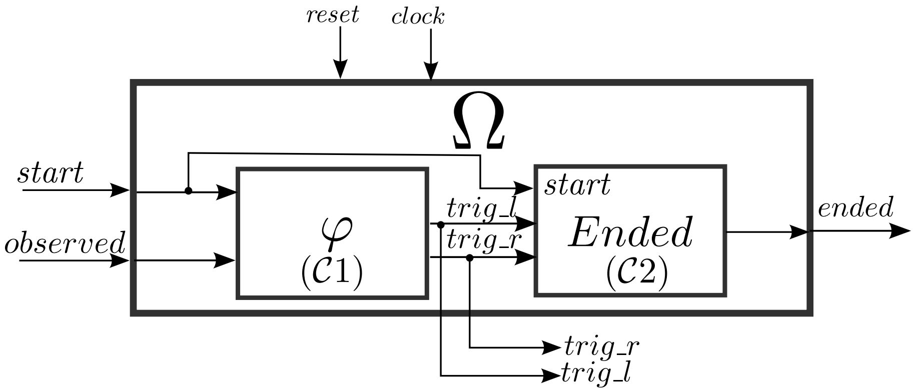


FIGURE 6.1 – Interface générique d'un opérateur de SERE

6.4.1.1 SEREs simple

L'ensemble des opérateurs de SERE simples est noté $\mathit{SimSERE} = \{;,:\}$. Pour un opérateur de SERE simple, nous avons les relations de dépendances suivantes :

$$\exists i < |w|, \lfloor \mathit{Ended}_A \lhd \mathit{true} \rfloor_{w^i} \wedge \lfloor B \lhd \mathit{true} \rfloor_{w^{i+1\ldots}} \text{ for ` ;'}$$
$$\exists i < |w|, \lfloor \mathit{Ended}_A \lhd \mathit{true} \rfloor_{w^i} \wedge \lfloor B \lhd \mathit{true} \rfloor_{w^{i\ldots}} \text{ for ` :'}$$

6.4.1.2 SEREs composées

Un opérateur composé SERE est un opérateur binaire, dont les sous-séquences gauches et droites démarrent en même temps. L'intégralité de la séquence dépend de l'opérateur. L'ensemble des opérateurs de SERE composées est noté $\mathit{CompSERE} = \{\&\&, \&, |\}$. Pour les opérateurs de SERE composées, nous avons les relations de dépendance suivantes :

$$\lfloor A \lhd \mathit{true} \rfloor_w \wedge \lfloor B \lhd \mathit{true} \rfloor_w \text{ for `\&\&'}$$
$$\exists i < |w|, (\lfloor A \lhd \mathit{true} \rfloor_w \wedge \lfloor B \lhd \mathit{true} \rfloor_{w^{0\ldots i}}) \vee (\lfloor A \lhd \mathit{true} \rfloor_{w^{0\ldots i}} \wedge \lfloor B \lhd \mathit{true} \rfloor_w) \text{ for `\&'}$$
$$\lfloor A \lhd \neg B \rfloor_w \vee \lfloor B \lhd \neg A \rfloor_w \text{ for `|'}$$

6.4.1.3 SEREs illimitées

Un opérateur de SERE illimitée est un opérateur unaire. L'ensemble des Seres illimitées est défini comme $\mathit{UnbSERE} = \{*, +\}$.

6.4.2 Mise en œuvre des composants réactifs primitifs des opérateurs de SERE

Dans cette section, nous expliquons comment un composant réactif SERE primitif peut être mis en œuvre de manière intuitive à l'aide d'un exemple simple pour chaque catégorie SERE.

6.4.2.1 SEREs simples

Exemple 2. Implémentation de $\varphi = \{b1; b2\}$

Si φ est généré (voir fig. 6.2), *trig_l* contraint $b1$ ($Trig_{b1} = trig\_l$), et dans le cycle suivant, *trig_r* constraint $b2$ ($Trig_{b2} = trig\_r$), et la séquence se termine.

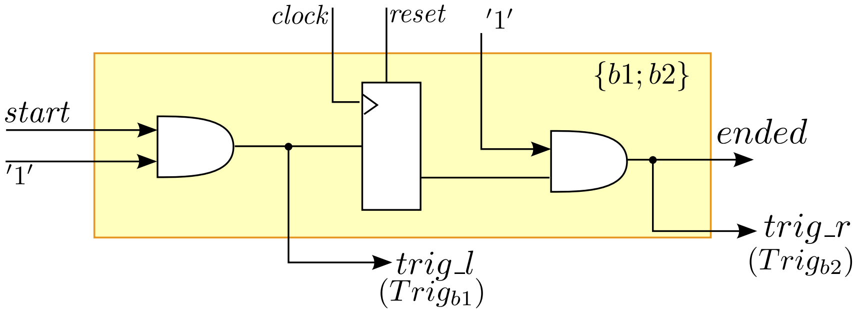


FIGURE 6.2 – Implémentation de $\{b1; b2\}$ ($b1$ et $b2$ sont générés)

Exemple 3. Implémentation de $\varphi = \{q; b\}$

Supposons que nous voulons générer φ ; par conséquent, nous générons q et b.

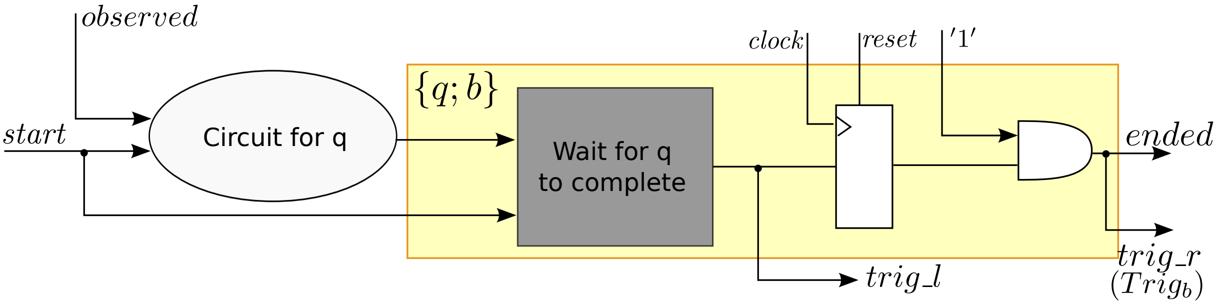


FIGURE 6.3 – Implémentation de $\{q; b\}$ (q et b sont générés)

On suppose que $q = \{b1; b2\}$. La figure 6.4 montre la trace correspondante.

6.4.2.2 SEREs composées

Exemple 4. Implémentation de $\varphi = \{q\}\&\{b\}$

Nous voulons générer φ, par conséquent, les deux sous-séquences devraient être générées, et elles devraient commencer en même temps ($trig\_l = trig\_r = start$). La figure 6.5 montre l'implémentation.

Exemple 5. Implémentation de $\varphi = \{q1\}\&\{q2\}$

Si on souhaite générer φ, nous devrions générer à la fois $q1$ et $q2$. La figure 6.6 montre l'implémentation.

6.4.2.3 SEREs non bornées

Exemple 6. Implémentation de $\varphi = b[+]$

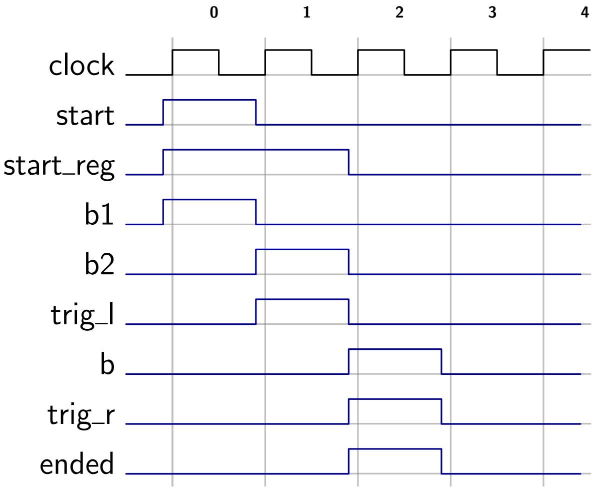


FIGURE 6.4 – Chronogramme de $\{\{b1; b2\}; b\}$

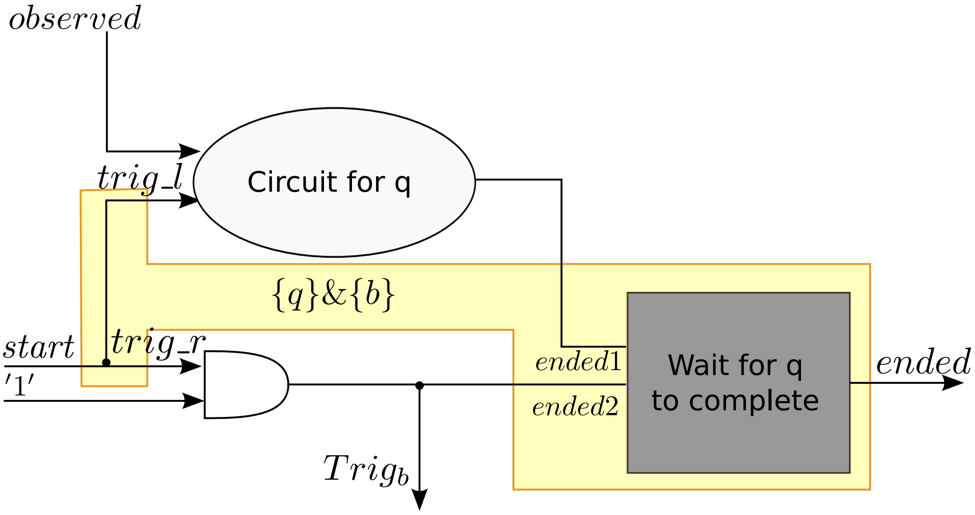


FIGURE 6.5 – Implémentation de $\{q\}\&\{b\}$ (q et b sont générés)

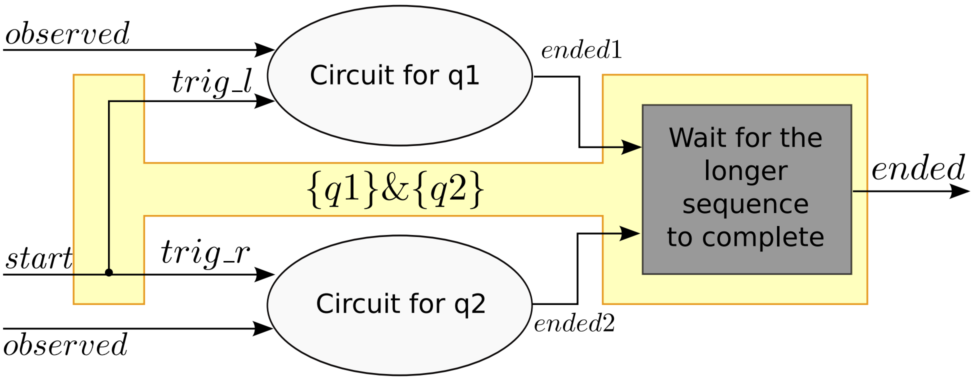


FIGURE 6.6 – Implémentation de $\{q1\}\&\{q2\}$

On suppose que b est généré. L'implémentation est représentée dans la fig. 6.7. Dans cette figure, le *trig_l* contraint b. Le signal interne *once_more* devient 1, un cycle après chaque occurrences de b. b doit être généré une (quand *start* $= 1$) ou plusieurs fois (quand *once_more* $= 1$).

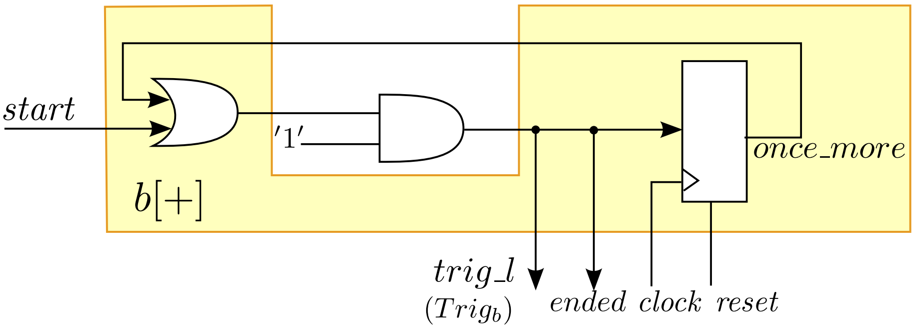


FIGURE 6.7 – Implémentation de $b[+]$

Exemple 7. Implémentation de $\varphi = b1[*]; b2$

Dans cet exemple, nous observons $b2$, et arrêtons de générer $b1$ lorsque $b2$ devient 1. La figure 6.8 montre le circuit correspondant.

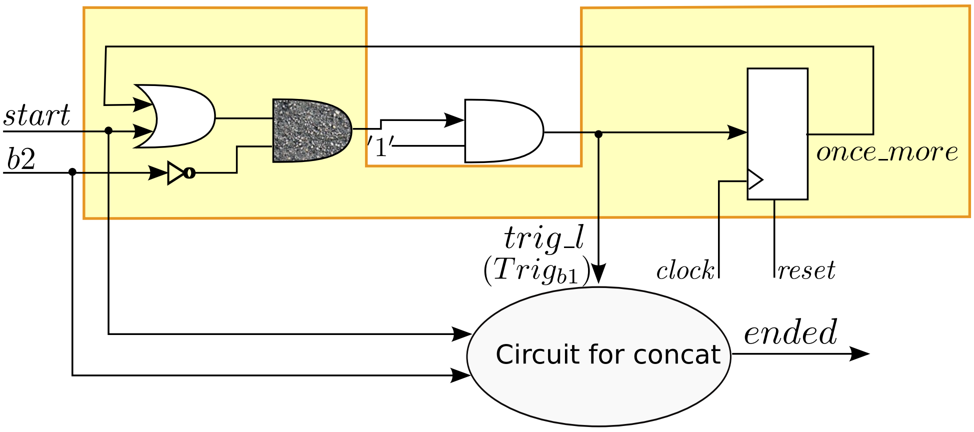


FIGURE 6.8 – Implémentation de $b1[*]; b2$

Exemple 8. Implémentation de $\varphi = q[*]; b$

La figure 6.9 montre le circuit correspondant.

6.5 Sous-ensemble synthétisable de SEREs

Le "sous-ensemble synthétisable de SEREs" pris en charge dans cette thèse sont les SEREs sous la forme suivante :

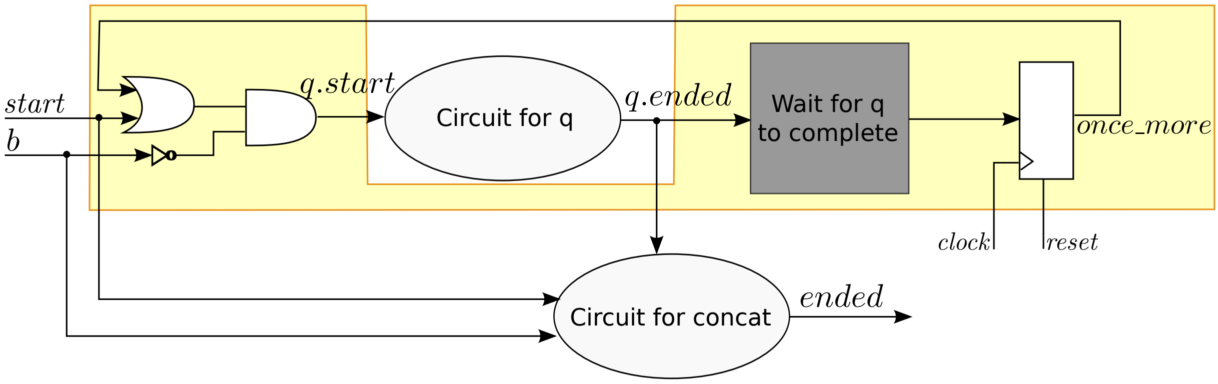


FIGURE 6.9 – Implémentation de $q[*]; b$

$$
\begin{aligned}
SERE_{synth} = & BoolExpr. \\
& \mid \{SERE_{synth}\} \\
& \mid SERE|\text{->}SERE_{synth} \\
& \mid SERE|\text{=>}SERE_{synth} \\
& \mid SERE_{synth}; SERE_{synth} \\
& \mid SERE_{synth} : SERE_{synth} \\
& \mid SERE_{synth} \& SERE_{synth} \\
& \mid \{BoolExpr\} \mid \{BoolExpr\} \\
& \mid SERE_{synth}[*n] \\
& \mid SERE_{synth}[*]; BoolExpr \\
& \mid SERE_{synth}[+]; BoolExpr \\
& \mid [+]; BoolExpr \\
& \mid [*]; BoolExpr
\end{aligned}
$$

Chapitre 7

Annotation des signaux

7.1 Construction de l'arbre syntaxique abstrait de la propriété (AST)

Pour synthétiser une propriété, il est essentiel de préciser le sens des signaux impliqués dans la propriété. Ce processus est appelé *annotation*.

La dépendance $\lfloor A \lhd B \rfloor$ peut être représentée comme le montre la fig. 7.1. Comme le montre cette figure, la valeur de B devrait être *observée* et A est *généré* basé en fonction de la valeur de B.

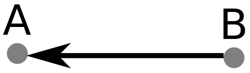


Figure 7.1 – La représentation de $\lfloor A \lhd B \rfloor_w$

L'arbre syntaxique abstrait (AST) d'une propriété est un arbre binaire non orienté classique. Les feuilles sont les signaux de conception qui peuvent être observés ou générés ; les autres nœuds sont les opérateurs temporels et logiques. On note AST $= (V, E)$, où :

— V est l'ensemble des nœuds (ou sommets) de l'arbre. L est l'ensemble des feuilles (les opérandes de la propriété), et $N = V \setminus \mathrm{L}$ est l'ensemble des nœuds internes (les opérateurs).

— $E \subset V \times V$ est l'ensemble des arrêtes de AST. $(v_1 - v_2)$ représente une arrête entre deux nœuds v_1 et v_2 ; v_1 est le parent et v_2 est un enfant.

Trois fonctions partielles sont définies sur V : $\mathcal{P}(v)$, $\mathrm{Lch}(v)$, $\mathrm{Rch}(v)$ retournent le parent, l'enfant gauche et l'enfant droit du nœud v.

7.2 Construction de l'arbre syntaxique abstrait orienté (DAST)

Pour chaque opérateur PSL, une ou plusieurs règles de dépendance ont été définies entre ses opérandes, conformément à la sémantique formelle de l'opérateur (voir les chapitres 5 et 6).

Afin de déterminer les variables qui sont lues par une propriété, et qui sont générées, nous traduisons les relations de dépendance en un graphe orienté, et construisons l' *arbre* *syntaxique abstrait orienté* (DAST) de chaque AST. DAST $= (V, E')$ représente la dépendance entre les signaux de la propriété, en passant par ses opérateurs. Il est construit en utilisant :

— la direction d'entrée/sortie des signaux de l'interface du module,

— l'implémentation des règles de dépendance comme une direction entre le nœud d'un opérateur et de ses enfants.

Chaque arc dans E' est une arrête orientée de E. La direction d'une arrête est considérée à partir du nœud parent, elle est dite entrante ou sortante. Pour chaque opérateur PSL et SERE, nous avons défini la direction des arrêtes du parent et les enfants basée sur leur relation de dépendance. Ici, nous montrons que deux exemples.

7.2.1 DAST des opérateurs FL simples

Cette catégorie contient les opérateurs `always`, `never`, `eventually!`, et `next!` (ainsi que leurs versions faibles). Tous ces opérateurs sont annotés de la même manière. Considérez l'opérateur `next!`, et supposons que $\varphi =$ `next!`(A) ; puis :

$$\lfloor \varphi \lhd \mathit{true} \rfloor_w \text{ iff } \lfloor A \lhd \mathit{true} \rfloor_{w^{1\ldots}}$$

De la relation de dépendance $\lfloor A \lhd \mathit{true} \rfloor_{w^{1\ldots}}$, nous pouvons en déduire qu'il y aura une arrête sortante de `next!` à A (fig. 7.2).

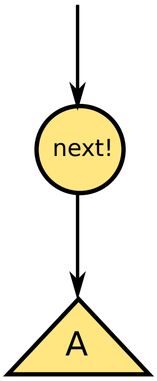


FIGURE 7.2 – La direction des arrêtes pour `next!`

7.2.2 DAST des opérateurs de SERE non bornés

Cette catégorie contient les opérateurs '$*$' et '$+$'.

Comme on l'a vu au chapitre 6, chaque répétition non bornée devrait être suivie par une expression booléenne. On suppose que $\varphi = A[*]; B$. Nous avons cette relation de dépendance :

$$\lfloor \varphi \lhd \mathit{true} \rfloor_w \text{ iff } \exists i < |w|, \lfloor B \lhd \mathit{true} \rfloor_{w^{i\ldots}} \wedge \forall k < i, \lfloor A[*] \lhd \neg B \rfloor_{w^{k\ldots}}$$

La dépendance $\lfloor B \lhd \mathit{true} \rfloor_{w^{i\ldots}}$ implique que B devient enfin *true*. De la dépendance $\lfloor A[*] \lhd \neg B \rfloor_{w^{k\ldots}}$, nous pouvons conclure qu'il y a un chemin sortant à partir de B à A (fig. 7.3).

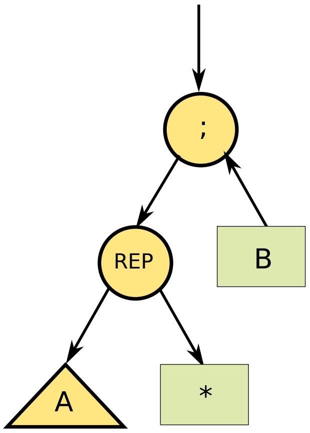


FIGURE 7.3 – La direction des arrêtes pour '$*$'

7.2.3 DAST **de directives et de fonctions PSL**

Outre les opérateurs FL et SERE, nous annotons les signaux de certaines des fonctions de l'opérateur booléen et des couches de vérification de PSL. En outre, nous annotons les opérandes de certains opérateurs de la couche de modélisation de PSL (VHDL).

7.2.4 L'algorithme d'annotation

Le processus d'annotation marque chaque instance de signal dans chaque propriété en observé ou généré.

Initialement, tous les arrêtes ne sont pas orientées. Le processus d'annotation est effectué en deux étapes.

Tout d'abord, nous partons de la direction des signaux d'interface, et annotons tous les signaux d'entrée 'm', et donnons une direction à son arrête correspondante.

Ensuite, le fonction récursive `Annotation` prend en entrée un DAST partiellement orienté ; il retourne un DAST avec plus d'arrêtes orientées. Il commence à partir de la racine de l'arbre, et basé sur son opérateur, donne la direction aux arrêtes correspondantes.

Chapitre 8

Réactif Complexe

8.1 Introduction

Dans ce chapitre, nous expliquons comment construire le composant réactif complexe d'une propriété,en ayant les composants réactifs primitifs et les directions des signaux. Nous utilisons l'arbre syntaxique abstrait (DAST) de chaque propriété pour interconnecter les composants réactifs primitifs et construire le composant réactif complexe.

8.2 Construction intuitive d'un composant réactif de la propriété

Le DAST de chaque propriété est soit entièrement orienté, soit peut avoir quelques sous-arbres non orientés. Le composant réactif est construit pour le sous-arbre entièrement orienté du DAST. Chaque nœud non terminal est remplacé par une instance du composant primitif réactif (i.e. mise en œuvre du matériel pour un opérateur temporel) ou une porte logique (pour un opérateur booléen) interconnectée à ses enfants. Pour une porte logique, l'interconnexion est évidente. Pour un composant réactif primitif, les principes d'interconnexion sont examinés selon l'opérateur.

8.2.1 Construction intuitive d'un réactif FL

Pour interconnecter les composants réactifs primitifs FL correspondant à chaque nœud v d'un DAST nous devrions considérer la direction des arrêtes correspondantes de v.

— Si la direction est $(\mathcal{P}(v) \rightarrow v)$, la sortie *trig* de $\mathcal{P}(v)$ est reliée à l'entrée *start* de v. Si v est une feuille, le DAST dont la racine est v est affecté à la sortie *trig* ; *trig* contraint v.

— Si la direction est $(\mathcal{P}(v) \leftarrow v)$, le signal observé (pour une feuille) ou le sortie *trig* de v (pour un nœud interne) est connecté à l'entrée *cond* de $\mathcal{P}(v)$.

8.2.2 Construction intuitive d'un réactif SERE

Pour interconnecter les composants SERE réactifs primitifs, nous devrions envisager différentes catégories d'opérateurs de SERE. D'après le chapitre 6, le composant réactif primitif d'un opérateur de SERE a les entrées *start*, *cond*1, et *cond*2 en plus des signaux de synchronisation. Il dispose également de trois sorties : *trig_l*, *trig_r*, et *ended*. La sortie de *ended* devient 1 lorsque la séquence est satisfaite.

8.2.2.1 SERE simple

Dans une séquence de SERE simple, par exemple $\varphi = A; B$, le composant réactif primitif de '$;$' et sa sous-séquence gauche (A) démarre en même temps; par conséquent, ils partagent le même signal de *start*. Basé sur les directions des arrêtes entre l'opérateur de SERE simple et ses enfants, nous avons :

- Si v est l'enfant de gauche ($v = \mathrm{Lch}(\mathcal{P}(v))$) :
  - Si v est un nœud interne :
    - l'entrée *start* de $\mathcal{P}(v)$ est reliée à l'entrée *start* de v.
    - la sortie *ended* de v est reliée à l'entrée *cond*1 de $\mathcal{P}(v)$.
  - Si v est une feuille :
    - Si la direction est ($\mathcal{P}(v) \leftarrow v$), `v` est relié à l'entrée *cond*1.
    - Si la direction est ($\mathcal{P}(v) \rightarrow v$), *cond*1 est relié '1', et *trig_l* contraint `v`.
- Si v est l'enfant droit ($v = \mathrm{Rch}(\mathcal{P}(v))$), les mêmes règles que pour l'enfant de gauche s'appliquent, en remplaçant :
  - *cond*1 par *cond*2
  - *trig_l* par *trig_r*

8.2.2.2 SEREs composées

Dans une séquence composée, les deux sous-séquences commencent en même temps.

- Si v est l'enfant de gauche ($v = \mathrm{Lch}(\mathcal{P}(v))$) :
  - Si v est un nœud interne :
    - la sortie *ended* de v est reliée à l'entrée *cond*1 de $\mathcal{P}(v)$.
    - la sortie *trig_l* de $\mathcal{P}(v)$ est reliée à l'entrée *start* de v.
  - Si v est une feuille :
    - Si la direction est ($\mathcal{P}(v) \leftarrow v$), `v` est relié à l'entrée *cond*1 de $\mathcal{P}(v)$.
    - Si la direction est ($\mathcal{P}(v) \rightarrow v$), *cond*1 est relié '1', et *trig_l* contraint `v`.
- Si v est l'enfant droit ($v = \mathrm{Rch}(\mathcal{P}(v))$), les mêmes règles que pour l'enfant de gauche s'appliquent, en remplaçant :
  - *cond*1 par *cond*2
  - *trig_l* par *trig_r*

8.2.2.3 Unbounded SERE

Comme il a été mentionné dans le chapitre 6, une répétition illimitée devrait être suivie par une expression booléenne. Soit v un nœud et `REP` la racine du sous-arbre de répétition non-bornée.

- Si $\mathrm{Lch}(v)$ n'est pas une feuille :
  - La sortie *trig_l* du composant réactif primitif de '$*$' est connectée à l'entrée *start* de $\mathrm{Lch}(v)$.
  - La sortie *ended* de $\mathrm{Lch}(v)$ est connectée à l'entrée *cond*1 du composant réactif primitif de '$*$'.
- Si $\mathrm{Lch}(v)$ est une feuille :

  - La sortie *trig_l* de '$*$' contraint le signal de Lch(v).
  - L'entrée *cond*1 de '$*$' est connectée à '1'.
- l'expression booléenne associée au frère de v (Rch($\mathcal{P}(v)$)) est connectée à l'entrée *cond*2 de '$*$'.

Chapitre 9

Résolution des signaux

9.1 Introduction

Dans ce chapitre, nous expliquons comment résoudre la valeur des signaux *dupliqués* et *non annotés*. Nous exprimons la dépendance parmi toutes les propriétés en utilisant un *Graphe de Dépendance*. Pour cela, nous partionons le DAST des propriétés en sous arbre complètement annoté*orienté* et non annoté *semi-orienté*. Ensuite, nous considérons le DAST orienté pour extraire les dépendances pour un signal dupliqué, et nous analysons les DAST semi-orientés pour extraire les dépendances pour les signaux non annotés.

9.2 Contraintes calculées à partir de DASTs annotés

Soient $Trig^i_{\neg z}, 0 \leq \ i \leq \ nb_0 - 1$ les nb_0 signaux triggers des composants réactifs qui contraignent le signal z à 0, et soient $Trig^j_z, 0 \leq \ j \leq \ nb_1 - 1$ les nb_1 signaux triggers qui contraignent z à 1. Alors :

$$\begin{aligned} \mathcal{T}0_z &= (Trig^0_{\neg z}, Trig^1_{\neg z}, \ldots, Trig^{nb_0-1}_{\neg z}) \\ \mathcal{T}1_z &= (Trig^0_z, Trig^1_z, \ldots, Trig^{nb_1-1}_z) \end{aligned}$$

Pour chaque signal z, les signaux $\mathit{T0}_z$ et $\mathit{T1}_z$ sont définis par :

$$\mathit{T0}_z = \bigvee_i Trig^i_{\neg z}, \text{ and } \mathit{T1}_z = \bigvee_j Trig^j_z$$

9.3 Constraintes calculées à partir de DASTs partiellement annotés

Comme vu dans le chapitre 7, de nombreux signaux de l'arbre sont non annotés. Commençons par un exemple :

Pour chaque DAST de l'ensemble *SEMIDIRECTED*, les sous-arbres semi-orientés sont élagués, et un composant réactif est construit pour les sous-arbres orientés en utilisant la méthode vue chapitre 8.

Soit $Etrig_j$ le signal de sortie d'un tel composant réactif, et $Expr_j$ les expressions booléenes élaguées. Les expressions $\mathcal{E} = (Expr_0, \ldots, Expr_m)$ sont conditionnées par $Etrig_0, \ldots, Etrig_m$.

9.4 Graphe de dépendance ($\mathcal{DG}$)

Le graphe de dépendance $\mathcal{DG}$ d'un ensemble de propriétés $(\mathtt{P}_0, \ldots, \mathtt{P}_{k-1})$ est un graphe étiqueté semi-orienté. Notons $\mathcal{DG} = (V, E)$, avec :

- $V = V1 \cup V2$ est l'ensembre des nœuds :
  - $V1 = L_0 \cup \cdots \cup L_{k-1}$, où $L_0, \ldots, L_{k-1}$ sont les ensembles de feuilles de $\text{DAST}_0, \ldots, \text{DAST}_{k-1}$
  - $V2$ est l'ensemble de toutes les sorties *trig* de toutes les propriétés.
- $E = E1 \cup E2$, où $E1$ est l'ensemble des arrêtes orientées, et $E2$ est l'ensemble des arrêtes non orientées, et :
  - $E1 \subset V2 \times V1$, e.g. $e = (Trig_l \to l)$
  - $E2 \subset V1 \times V1$, e.g. $e = (l_1 – l_2)$
- Chaque arrête e du graphe a une arrête $w = (id,\ val,\ type)$, où :
  - id identifie la propriété qui crée l'arrête e ; ainsi $0 \leq id \leq k - 1$
  - val : si e est orienté, val spécifie la valeur du nœud de destination, si la valeur du nœud source est 1. Si e n'est pas orientée, val vaut -1. Ainsi, $val \in \{0, 1, -1\}$.
  - $type$ spécifie si une arrête est orientée ; ainsi, $type \in \{d, u\}$, où'd' signifie orienté, et 'u' non-orienté.

Le graphe de dépendance peut avoir plusieurs composantes fortement connexes, chacun représentant un ensemble de signaux générés interdépendants $\mathcal{Z} = \{z_1, \ldots z_n\}$ (voir fig. 9.1).

Exemple 1. *Graphe de dépendance du GenBufRec*

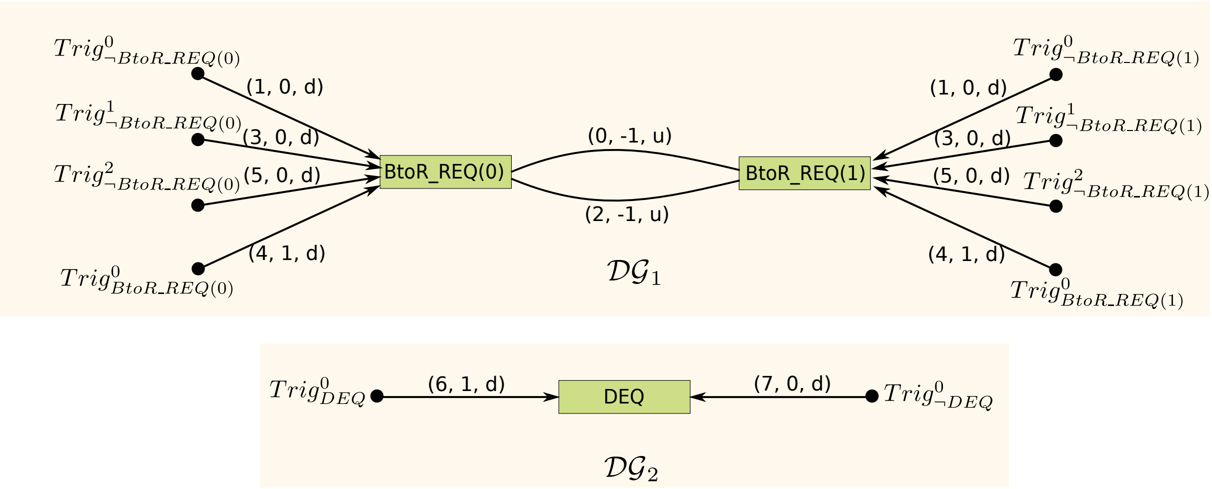


FIGURE 9.1 – Graphe de dépendance de GenBufRec

9.5 Construction du graphe de dépendance

Le graphe de dépendance se construit en deux étapes à partir des DASTs des propriétés.

```
1 Pour chaque DAST orienté, DAST$_k$ :
  1-1 Pour chaque feuille générée $l$ of DAST$_k$ (i.e. $(\mathcal{P}(l) \to l)$) :
      1-1-1 Ajouter $l$ aux nœuds de $\mathcal{DG}$ (si elle n'est pas dans $V$)
      1-1-2 Ajouter le signal trigger correspondant ($Trig_l^i$ ou $Trig_{\neg l}^j$) à $V$
      1-1-3 Créer une arrête $e$ du nœud trigger node jusqu'au nœud signal, $e = (Trig_l^i \to l)$
      1-1-4 Si le signal correspondant à $l$ est contraint à 0 : ajouter l'étiquette $w = (k, 0, d)$, sinon $w = (k, 1, d)$
2 Pour chaque DAST semi-orienté, DAST$_k$ :
  2-1 Elaguer le sous-arbre complètement orienté, et garder uniquement les sous-arbres non-orientés.
  2-2 Ajouter toutes les feuilles des sous-arbres non orientés dans $V$ ( si elles ne sont pas dans $V$)
  2-3 Pour chaque pair de nœuds $l_1$ et $l_2$ partant de DAST$_k$ :
      2-3-1 Ajouter une arrête entre $l_1$ et $l_2$
      2-3-2 Créér l'étiquette $w = (k, -1, u)$
```

9.6 Fonction de résolution : le composant de*résolution*

Maintenant nous discutons comment utiliser le graphe de dépendance $\mathcal{DG}$ pour générer des composants de résolution. Nous construisons deux types de composants de résolution : les composants *simples* et les *complexes*. Le premier spécifie la valeur des signaux *dupliqués*, le second spécifie la valeur des signaux *non annotés*.

9.6.1 Résolution des signaux dupliqués : composant *simple*

Dans $\mathcal{DG}_i$, nous considérons chaque arrête $e = (v \to z)$. Si l'étiquette $w = (i, 0, d)$, nous ajoutons le signal trigger qui est représenté par le nœud v à $\mathcal{T}0_z$; si l'étiquette $w = (i, 1, d)$, nous ajoutons le signal trigger à $\mathcal{T}1_z$. Après avoir trouvé $\mathcal{T}1_z$ et $\mathcal{T}0_z$, la valeur de z doit être calculé.

Les signaux $T0_z$ et $T1_z$ sont les entrées du composant de résolution. La sortie sera la valeur finale de z (voir fig. 9.2).

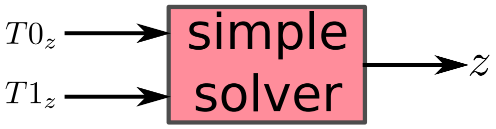


FIGURE 9.2 – Interface d'un composant de résolution simple pour des signaux dupliqués

Dans notre implémentation, si ni $T0_z$ ni $T1_z$ ne sont actifs, l'utilisateur peut choisir de décider si z garde sa valeur précédente ou prend une valeur par *défaut.* La fonction de résolution est l'une des :

$$\begin{aligned} z &= '0' \text{ when } \mathit{T0}_z =' 1' \text{ else} \\ &\ '1' \text{ when } \mathit{T1}_z =' 1'; \end{aligned} \qquad \begin{aligned} z &= '0' \text{ when } \mathit{T0}_z =' 1' \text{ else} \\ &\ '1' \text{ when } \mathit{T1}_z =' 1' \text{ else} \\ &\ default\_value; \end{aligned}$$

9.6.2 Résolution de sigaux non annotés : les composants *complexes*

Supposons que $\mathcal{Z} = (z_1, \ldots, z_n)$, alors tous les signaux z_i sont non annotés dans au moins une propriété. Nous pouvons dire que les signaux z_i, …, z_n sont les opérandes des expressions $\mathit{Expr}_0, \ldots, \mathit{Expr}_{m-1}$ activées par $\mathit{Etrig}_0, \ldots, \mathit{Etrig}_{m-1}$.

9.6.2.1 Implémentation des composants de résolution complexes

Un composant complexe prend $(\mathit{Etrig}_0, \ldots, \mathit{Etrig}_{m-1})$, $(\mathit{T1}_{z_1}, \ldots, \mathit{T1}_{z_n})$, et $(\mathit{T0}_{z_1}, \ldots, \mathit{T0}_{z_n})$ comme entrées, et il ressort les valeurs de $(z_1, \ldots, z_n)$.

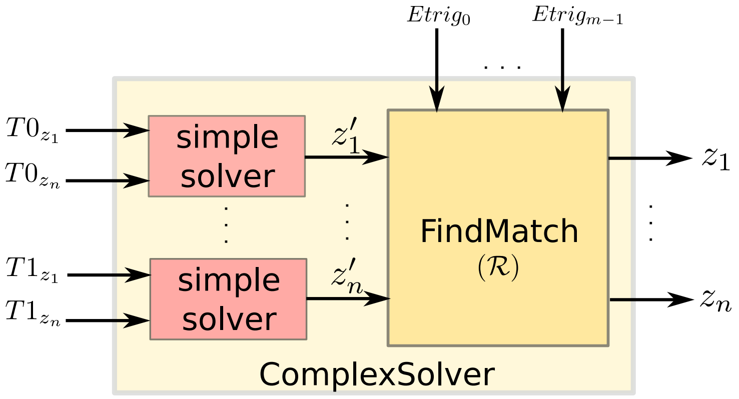


Figure 9.3 – Interface des composants de résolution complexes pour des signaux non annotés

Le problème est de résoudre l'ensembre des équations suivantes :

$$\begin{cases} \vdots \\ \mathit{Etrig}_j \to \mathit{Expr}_j(z_1, \ldots, z_n) = 1 \\ \vdots \end{cases}$$

L'idée brutale est de construire une Look Up Table (LUT) pour le sous-module `FindMatch` (fig. 9.3) en énumérant toutes les valeurs de $\mathcal{Z}$ pour chaque valeur t de $T_{\mathcal{Z}}$. Alors, nous sélectionnons la ligne appropriée de cette LUT.

Nous considérons les 2^m valeurs de ce vecteur $T_{\mathcal{Z}} = (\mathit{Etrig}_0, \ldots, \mathit{Etrig}_{m-1})$. Chaque valeur t of $T_{\mathcal{Z}}$ correspond à l'ensembre des triggers qui sont actifs simultanément. Nous associons à cet ensemble de signaux trigger actifs, l'expression booléene qui est la conjonction des Expr_j correspondant aux $\mathit{Etrig}_j = 1$.

9.7 Le circuit final

Le circuit final est l'interconnexion des composants réactifs des propriétés (voir Chapitre 8) et des composants de résolution.

Exemple 2. Le circuit final de GenBufRec.

La figure 9.4 montre l'interconnexion de tous les composants réactifs avec les composants de résolution GenBufRec.

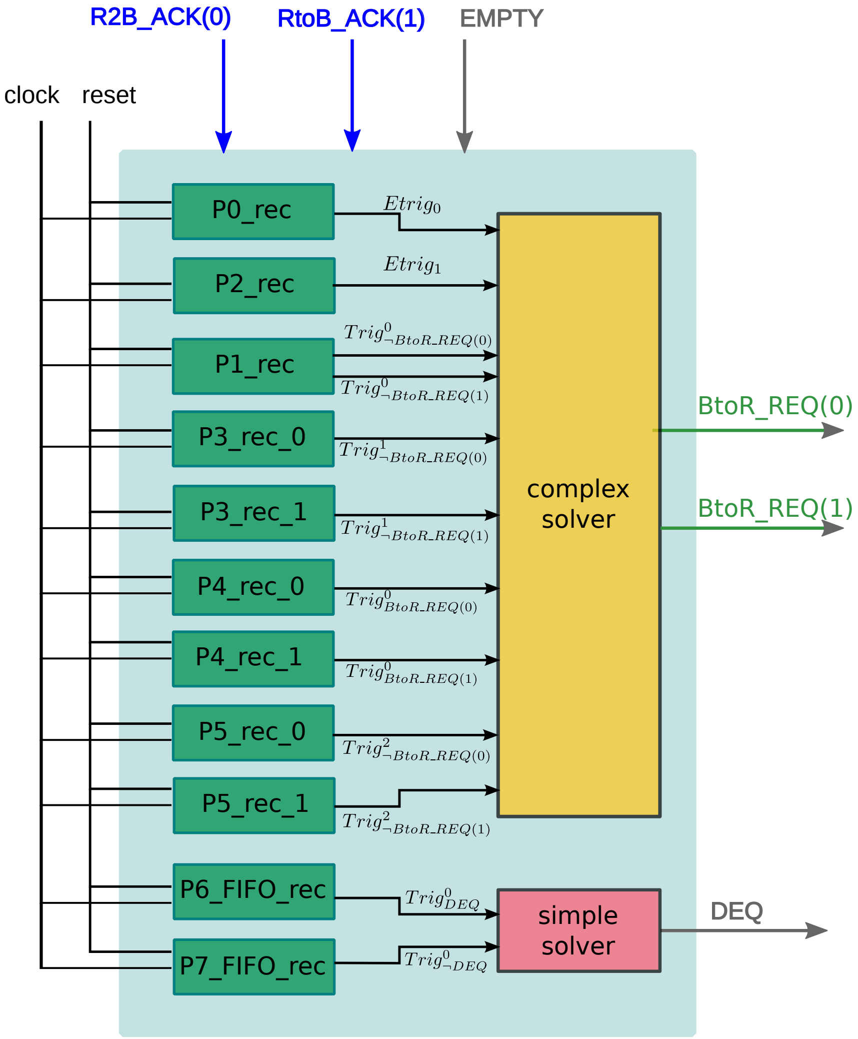


FIGURE 9.4 – Circuit final de GenBufRec

9.7.1 Vérification de la cohérence

La définition de z est cohérente ssi dans tous les cycles :

$$T1_z \wedge T0_z = 0$$

9.7.2 Vérification de la complétude

La définitionde z est complète ssi dans tous les cycles :

$$\mathit{T1}_z \vee \mathit{T0}_z = 1$$

Nous pouvons bien sûr générer toutes ses propriétés complémentaires automatiquement pour vérifier les conditions ci dessus.

Chapitre 10

Les expériences et les résultats pratiques

10.1 Introduction

Nous avons appliqué notre méthode de synthèse à plusieurs études de cas : GenBuf[1], AMBA[2] Arbiter, HDLC[3], CRC[4], et SDRAM[5]. Les circuits générés sont synthétisés à la fois pour un circuit FPGA[6] et un circuit ASIC[7]. Les résultats sont comparés aux résultats d'un autre outil, Ratsy.

10.2 Prototypage du matériel et les résultats de synthèse

La table 10.1 résume les caractéristiques des outils de ABS existants : les formats d'entrée et de sortie, et le sous-ensemble de PSL que chacun d'eux accepte.

TABLE 10.1 – Outils d'ABS

Tool	Input	Output	FL	SERE
Acacia	LTL	*dot*	✓	no
Unbeast	LTL in XML format	NuSMV	✓	no
Ratsy	LTL	Verilog	*GR(1)* subset of PSL	no
SyntHorus2	PSL	VHDL and PSL properties	PSL_{simple}	partially (see Chapter 6)

Unbeast et Acacia peuvent travailler seulement sur des exemples très simples et petits ; Par conséquent, nous avons exclu les résultats des tableaux.

Ici, nous donnons les résultats de synthèse pour le cas étudié.

10.2.1 Generalized Buffer (GenBuf)d'IBM

La figure 10.1 compare le temps de génération de matériel pour SyntHorus2 et pour Ratsy pour le GenBuf (avec plusieurs récepteurs et 2 émetteurs).

[1] IBM Generalized Buffer

[2] ARM Advanced Microcontroller Bus Architecture

[3] High-level Data Link Controller

[4] Cyclic Redundancy Check

[5] Single Data-rate Random Access Memory

[6] Field Programmable Gate Array

[7] Application Specific Integrated Circuit

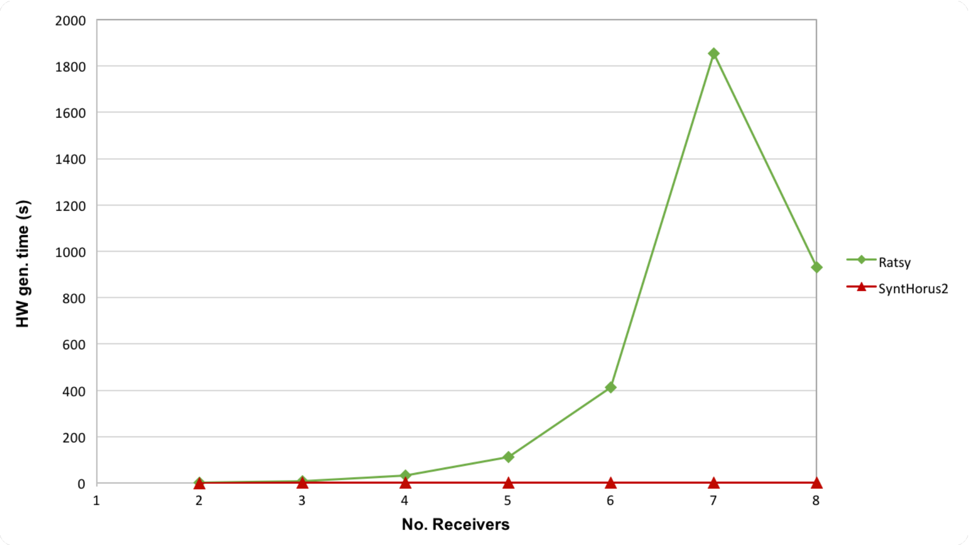


Figure 10.1 – Le temps de génération HW : GenBuf avec 2 émetteurs, plusieurs récepteurs et FIFO

Il est important de préciser que SyntHorus2 n'effectue pas la vérification, tandis que Ratsy intègre la vérification dans le processus de génération. Ainsi, la comparaison des temps d'exécution des deux outils n'est pas pertinente si l'on souhaite effectuer une vérification.

10.2.1.1 Synthèse de circuits sur FPGA

Tout d'abord, nous avons synthétisé les circuits générés par SyntHorus2 et par Ratsy en utilisant Quartus II afin de les mettre en œuvre sur une carte FPGA (EP4CE30F23C6 de l'appareil de la famille de l'appareil Cyclone IV).

Plusieurs récepteurs et deux émetteurs, avec FIFO

La table 10.2 donne les résultats de synthèse pour SyntHorus2 et Ratsy.

Table 10.2 – Quartus II : résultat de synthèse pour le contrôleur GenBuf avec FIFO, plusieurs récepteurs, et 2 émetteurs

	SyntHorus2				Ratsy			
# rec	# prop.	# reg.	# LUTs	F (MHz)	# prop.	# reg.	# LUTs	F (MHz)
3	28	37	98	756.4	56	24	2092	130.4
4	31	45	119	781.2	63	27	2587	140.2
5	34	53	146	609.0	70	30	4251	121.6
6	37	61	166	745.7	77	34	10408	92.5
7	40	99	196	616.5	84	36	15191	89.8
8	46	77	215	647.7	91	40	18180	83.6

SyntHorus2 génère des circuits plus rapides avec moins de LUT que Ratsy, mais avec plusieurs registres.

10.2.1.2 Synthèse pour la mise en oeuvre sur ASIC

Nous avons synthétisé l'ensemble des circuits générés par SyntHorus2 et Ratsy avec Design Vision selon les mêmes conditions.

Plusieurs récepteurs et deux émetteurs, avec FIFO

Table 10.3 donne les résultats de nos expériences sur Genbuf avec une FIFO, pour 3 à 8 récepteurs et 2 expéditeurs, exécutant SyntHorus2 et Ratsy.

TABLE 10.3 – Design Vision résultat de synthèse pour le contrôleur GenBuf avec FIFO, plusieurs récepteurs, et deux expéditeurs

	SyntHorus2					Ratsy				
# rec	# prop.	# comb. cells	# seq. cells	Total area	F (MHz)	# prop.	#comb. cells	# seq. cells	Total area	F (MHz)
3	28	414	102	57876	568	56	2781	24	259796	96
4	31	467	118	66715	565	63	3285	27	306134	80
5	34	546	134	76961	555	70	5146	30	475880	67
6	37	624	150	87188	555	77	12970	34	1198182	57
7	40	714	175	100559	555	84	17934	36	1647189	56
8	46	860	191	114564	555	91	20828	40	1894378	65

Figure 10.2 compare le nombre total de portes pour les circuits qui sont générés par SyntHorus2 et Ratsy.

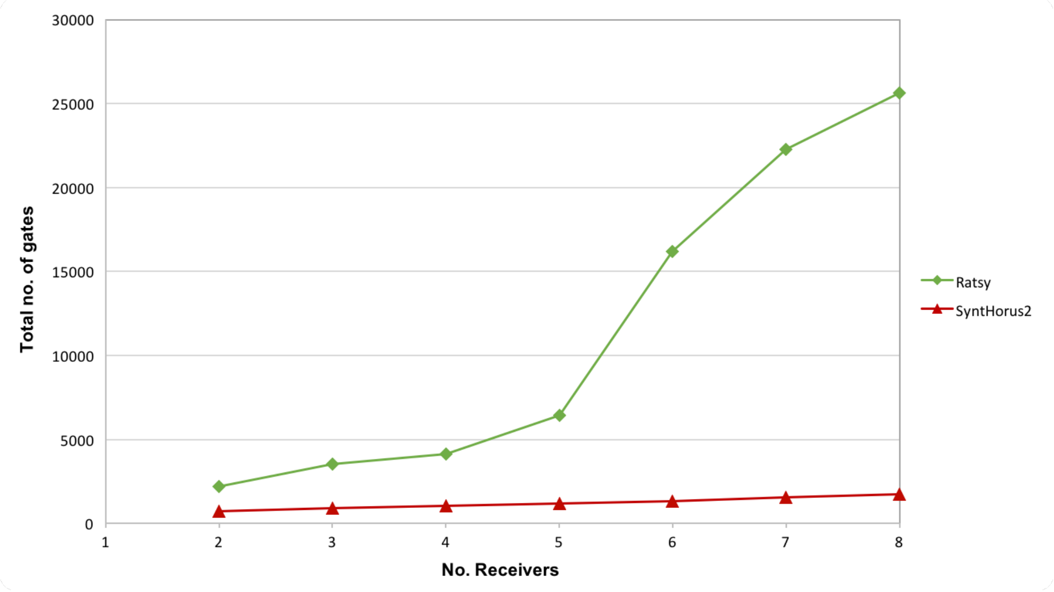


FIGURE 10.2 – Le nombre total de portes : GenBuf avec plusieurs récepteurs et 2 expéditeurs

10.2.2 AMBA arbiter

La figure 10.3 compare le temps de génération de matériel par SyntHorus2 et par Ratsy.

10.2.2.1 Synthèse de mise en œuvre sur FPGA

La table 10.4 montre les résultats de synthèse par Quartus II pour AMBA arbitre.

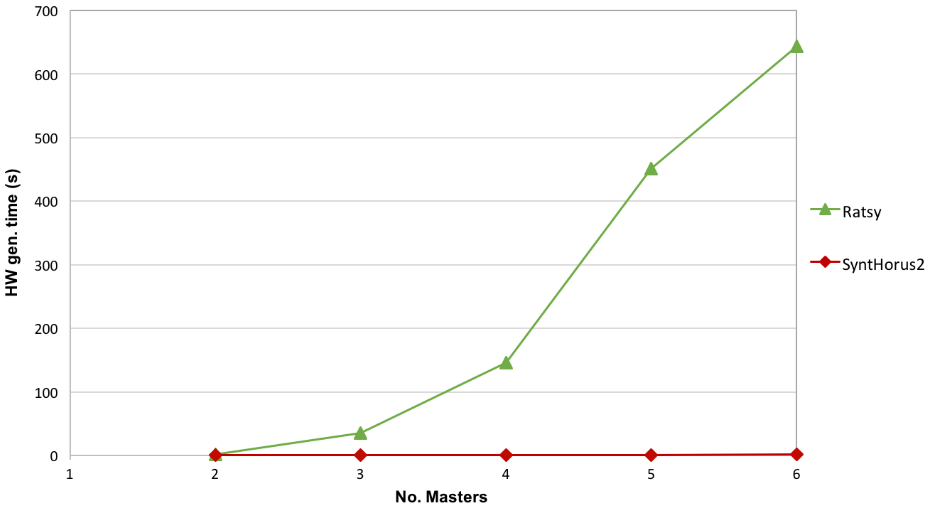


Figure 10.3 – Temps de génération HW : AMBA arbitre

Table 10.4 – Quartus II résultat de synthèse pour AMBA arbitre avec 2 esclaves et plusieurs maîtres

	SyntHorus2				Ratsy			
# masters	# prop.	# reg.	# LUTs	F (MHz)	# prop.	# reg.	# LUTs	F (MHz)
2	35	24	59	795.5	77	21	382	199.2
3	46	33	113	852.5	96	29	2941	131.1
4	56	42	119	923.4	114	29	6085	106.9
5	66	51	150	795.5	133	34	3091	130.45
6	77	60	181	758.1	151	37	4355	115.8

10.2.2.2 Synthèse pour la mise en oeuvre ASIC

La table 10.5 donne nos résultats pour 2 esclaves et des nombres différents de maîtres.

TABLE 10.5 – Design Vision résultats de la synthèse pour AMBA arbitre

	SyntHorus2					Ratsy				
# masters	# prop.	# comb. cells	# seq. cells	Total area	F (MHz)	# prop.	#comb. cells	# seq. cells	Total area	F (MHz)
2	33	303	90	45165	637	77	515	21	51985	164
3	46	513	132	71915	621	96	3867	29	362589	92
4	56	567	159	82522	606	114	7712	29	721363	67
5	66	725	194	102762	629	133	3920	34	370179	82
6	77	660	228	121855	625	151	5855	37	552310	62

Figure 10.4 compare le nombre total de portes des circuits qui sont générés par SyntHorus2 et Ratsy.

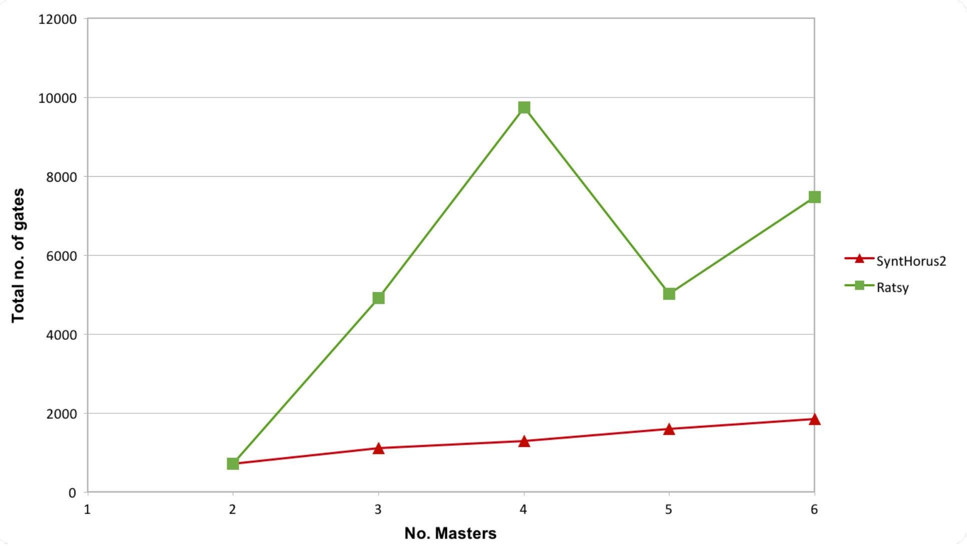


FIGURE 10.4 – Le nombre total de portes : AMBA arbitre

10.2.3 D'autres exemples

Nos trois derniers exemples sont présentés dans la table 10.6.

TABLE 10.6 – Design Vision les résultats de la synthèse pour HDLC, SDRAM, et CRC

			SyntHorus2				
Circuit	# prop.	Hw. gen. time (s)	# comb. cells	# seq. cells	Total area	Total # of gates	F (MHz)
HDLC	120	1.06	2646	1433	600527	9588	429
SDRAM	9	0.2	1045	769	295107	4765	513
CRC	14	0.14	641	293	131122	2401	406

10.2.4 Comparaison entre FLs et SEREs

Pour montrer l'applicabilité de notre méthode de synthèse en Seres, les propriétés SERE sont fournies pour GenBuf, l' arbitre AMBA et le HDLC, les conceptions VHDL correspondantes sont générées en utilisant SyntHorus2, et sont synthétisées en utilisant Design Vision.

10.2.4.1 GenBuf : plusieurs récepteurs

Les propriétés FLs de GenBuf sont converties en SEREs. La table 10.7 montre les résultats de synthèse pour le GenBuf avec plusieurs récepteurs et deux émetteurs.

TABLE 10.7 – Design Vision : les résultats de la synthèse pour GenBuf avec multiples récepteurs (pour les propriétés de SERE)

# receivers	# prop.	HW gen. time (s)	# comb. cells	# seq cells	Total area	Total # of gates	Freq. (MHz)
3	27	0.19	529	119	720701	1123	308
4	30	0.25	651	146	901106	1392	320
5	33	0.35	775	175	108083	1669	290
6	36	0.61	905	206	127124	1963	296
7	42	0.26	1052	248	150525	2325	296
8	45	0.29	1215	287	174253	2691	280

La figure 10.5 compare le nombre total de portes pour les circuits générés à partir de FLs et SEREs.

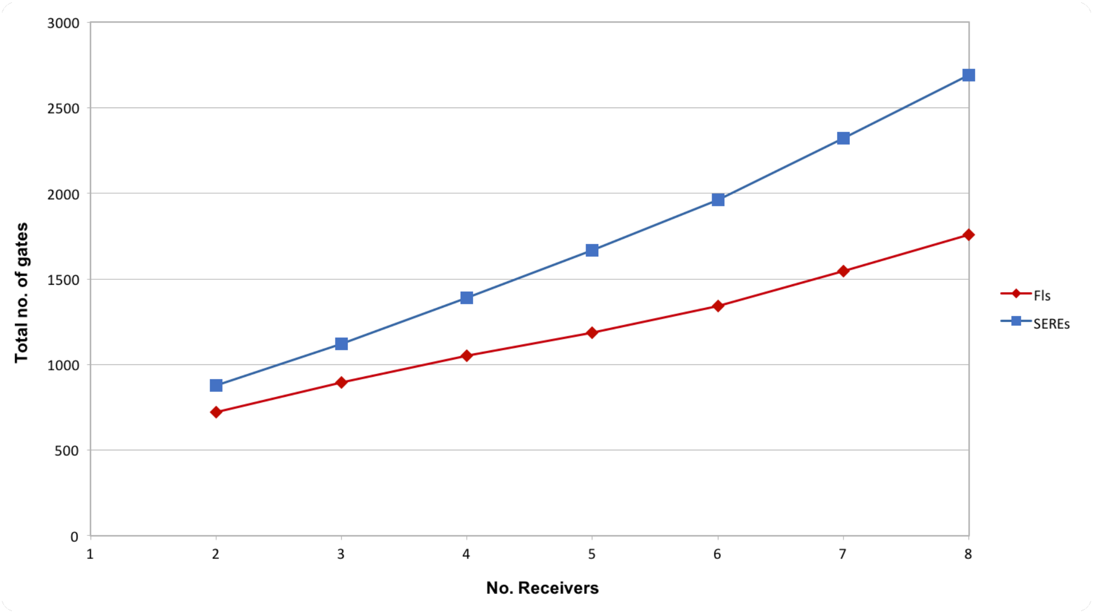


FIGURE 10.5 – Le nombre total de portes : GenBuf avec plusieurs récepteurs et 2 émetteurs (générés à partir de FLs et SEREs)

10.2.4.2 AMBA Arbiter

Pour l'arbitre de bus AMBA, nous avons écrit la spécification en SERE basée sur sa description de protocole en anglais et les propriétés FL. La table 10.8 résume les résultats de synthèse pour 2 esclaves et de 2 à 6 maîtres.

Table 10.8 – Design Vision résultats de la synthèse pour AMBA arbitre (pour les propriétés SERE)

# masters	# prop.	HW gen. time (s)	# comb. cells	# seq. cells	Total area	Total # of gates	F (MHz)
2	28	0.14	223	62	33614	523	368
3	41	0.24	341	95	50146	777	324
4	52	0.32	429	120	63209	978	307
5	63	0.41	520	145	76471	1181	296
6	74	0.63	628	170	90546	1394	266

10.2.4.3 HDLC

Pour le HDLC, nous avons fourni deux ensemble de propriétés :

- `SERE1` : toutes les propriétés FL sont converties sous forme de propriétés SERE équivalentes.
- `SERE2` : trois modules, `FlagDetection`, `ZeroDetection` et `ZeroInsertion`, sont directement exprimés en utilisant les propriétés SERE. Ces SEREs sont écrites depuis le protocole, elles n'ont pas été obtenues par la réécriture FLs.

La table 10.9 résume le résultat de la synthèse.

Table 10.9 – Design Vision résultats de la synthèse pour HDLC (pour les propriétés SERE)

	# prop.	HW gen. time (s)	# comb. cells	# seq. cells	Total area	Total # of gates	F (MHz)
`SERE1`	120	1.22	3238	1157	614507	9700	82
`SERE2`	108	1.51	3017	839	516363	8050	59

Chapitre 11

Conclusion et travaux à venir

Dans cette thèse, nous avons présenté une méthode modulaire pour synthétiser la partie contrôleur des circuits, en composants réactifs et non en moniteurs, à partir de leurs propriétés temporelles écrites en PSL.

11.1 Contributions

Les principales contributions de ce travail sont résumées ci-après :

— A partir de la sémantique de traces de PSL, nous avons défini formellement une relation de dépendance entre les opérandes des opérateurs temporels de SERE [1]. Ensuite, nous avons donné une interprétation matérielle des relations de dépendance, ce qui constitue la base sur laquelle est construite la bibliothèque des composants réactifs primitifs (voir chapitre 6).

— Ces relations de dépendance, accompagnées de la relation de dépendance formelle des FLs, sont le modèle formel sur lequel l'algorithme d'annotation est écrit(voir chapitre 7).

— Si l'on considère la dépendance entre toutes les propriétés, les composants de résolution ont été générés pour résoudre la valeur des signaux dupliqués et non-annotés (voir chapitre 9).

— Nous générons des propriétés complémentaires pour vérifier la cohérence et la complétude de l'ensemble des propriétés.

— L'outil prototype SyntHorus2 a été mis en œuvre, basé sur les principes décrits dans cette thèse. Il est en cours d'adaptation, sur la base des exigences de l'industrie.

— SyntHorus2 a été testé sur un ensemble de jeux d'essai, et également sur des circuits grandeur nature comme l'arbitre de bus AMBA-AHB et le contrôleur HDLC. Comparer SyntHorus2 avec d'autres outils ABS est difficile, car chaque outil requiert son propre sous-ensemble de LTL ou de PSL qui doit être adapté pour chaque outil. Comparé à d'autres outils, SyntHorus2 génère des conceptions plus petites et plus rapides sur les exemples les plus gros.

Les résultats intermédiaires de SyntHorus2 permettent de déboguer une spécification, et de vérifier si elle est cohérente et complète. Les assertions générées automatiquement sur les signaux de “trigger” peuvent être vérifiées par un simulateur, ou par model checking

[1] Sequential Extended Regular Expression

à l'aide d'un outil de vérification formelle. De plus, SyntHorus2 peut fournir un prototype d'environnement conforme aux spécifications, afin de tester un autre module de circuit.

11.2 Travaux à venir

A présent, SyntHorus2 ne traite que des signaux scalaires et des vecteurs booléens dans les propriétés. Les travaux à venir incluent la reconnaissance de types de données plus complexes, comme les types énumérés et entiers.

SyntHorus2 supporte partiellement la couche modélisation de PSL. Il supporte les opérateurs de comparaison et arithmétiques. Cependant, il ne supporte pas la définition de signaux locaux. Cette capacité devrait être ajoutée à SyntHorus2.

Comme expliqué au Chapitre 6, notre sous-ensemble synthétisable de SEREs connaît des limitations. Par exemple, nous ne pouvons pas avoir de répétition non-consécutive. Par conséquent, le sous-ensemble synthétisable de SEREs devrait être étendu et les limitations devraient être allégées. Pour surmonter quelques une des limitations, par exemple en observant $\varphi = A\&B$ où A et B sont des séquences, nous pouvons profiter des méthodes basées sur les automates. Nous pouvons combiner les méthodes basées sur les automates et les modulaires, puis utiliser la méthode basée sur les automates pour la partie gauche d'une implication, qui devrait être observée, et la méthode modulaire pour la partie droite d'une implication qui devrait être générée.

De plus, nous devrions optimiser les modules réactifs primitifs pour les SEREs. Comme démontré dans le Chapitre 10, si nous réécrivons une propriété FL dans son équivalent de propriété SERE, le circuit obtenu depuis une SERE est plus gros et plus lent. Nous devrions optimiser à la fois les modules réactifs des SEREs et leurs interconnections.

Nous avons fourni des lignes directrices sur comment écrire des propriétés afin de générer des circuits plus petits : ils devraient être améliorés. Nous pouvons fournir des sous-ensembles prédéfinis de propriétés pour exprimer certains comportements des signaux, par exemple l'exclusion mutuelle, le tourniquet, le protocole poignée de main à 4 phases, etc...

La mise en œuvre des composants de résolutions complexes peut être améliorée pour générer des composants plus petits. Comme discuté au Chapitre 9, certaines lignes de LUT ne devraient pas être utiles, car certaines combinaisons de signaux Etrig n'apparaissent jamais. Ces lignes peuvent être éliminées par la vérification de modèle sur les propriétés. Cela n'a pas été automatisé.

Comme discuté au Chapitre 9 , afin de calculer la valeur des signaux non-annotés, il pourrait y avoir plusieurs choix obtenus à partir de LUT. A présent, nous sélectionnons la première ligne de LUT qui satisfait notre exigence. D'autres politiques de sélection peuvent être envisagées.

A ce stade, des composants de résolutions complexes sont fournis pour les signaux scalaires. Ils devraient être étendus aux vecteurs. De plus, des composants complexes ne peuvent pas être utilisés dans les cas où les signaux non-annotés dépendent d'opérateurs et fonctions du niveau modélisation. Nous devrions pouvoir résoudre ce problème.

Notre méthode est modulaire ; pour chaque propriété, elle instancie tous les modules réactifs primitifs des opérateurs de la propriété. Cela conduit à des composants redondants. Cela ressemble aux débuts de la synthèse : chaque instance d'un opérateur dans le design RTL produisait un opérateur matériel distinct. Une étape d'optimisation est nécessaire pour partager des modules réactifs primitifs dans le circuit généré.

Résumé– Les travaux présentés dans cette thèse visent à produire automatiquement des prototypes de circuits de communication et de contrôle à partir de spécifications temporelles déclaratives. Partant d'un ensemble de propriétés écrites en langage PSL, nous produisons un modèle RTL synthétisable automatiquement. La méthode proposée est modulaire, contrairement aux méthodes publiées antérieurement qui étaient fondées sur la théorie des automates. Pour chaque propriété, nous produisons un composant qui observe certains opérandes et génère des chronogrammes pour les autres opérandes : le module réactif.

Tout d'abord, une bibliothèque des modules réactifs primitifs a été développée pour les opérateurs FL et SERE. Pour ce faire, une relation de dépendance a été définie pour chaque opérateur : fondée sur la sémantique de l'opérateur, elle exprime la dépendance entre ses opérandes. Ensuite, la relation de dépendance de chaque opérateur est interprétée comme un composant matériel qui met en œuvre l'opérateur : c'est le module réactif primitif de l'opérateur.

À l'aide de cette formalisation, nous proposons une méthode pour déterminer automatiquement quels signaux d'une propriété sont observés et lesquels sont générés. Dans le cas où il n'est pas possible de déterminer le sens du signal, un solveur est ajouté pour identifier la valeur du signal. Le solveur sert aussi à déterminer la valeur d'un signal généré par plusieurs propriétés. Le circuit final est l'interconnexion des modules réactifs et des solveurs pour l'ensemble des propriétés.

Un outil prototype, SyntHorus2, qui est une extension d'HORUS, a été mis développé. Il prend les propriétés PSL comme entrées et génère le code VHDL synthétisable du circuit. En outre, il génère des propriétés complémentaires pour vérifier si l'ensemble des spécifications est cohérent et complet.

La méthode est efficace et synthétise des circuits de commande en quelques secondes. Les résultats que nous avons obtenus sur des jeux d'essais classiques montrent que notre technique compile les propriétés plus efficacement que les outils prototypes qui l'ont précédée.

Mots-clés. PSL, conception basée sur les assertions, module réactif, synthèse automatique, graphe de dépendance, annotation, résolution, solveur.