Safe design method of embedded systems based on COTS

Salam Hajjar

THÈSE

prsentée devant
L'INSTITUT NATIONAL DES SCIENCES APPLIQUÉES DE LYON
pour obtenir le grade de

DOCTEUR DE L'INSA DE LYON

Spécialité : Automatique

par

SALAM HAJJAR

École Doctorale : Electronique Electrotechnique et Automatique (EEA)
Équipe d'accueil : Équipe FDS (Fiabilité, Diagnostique, Supervision) - Laboratoire Ampère

Titre

CONCEPTION SÛRE DE SYSTÈMES EMBARQUÉS À BASE DE COTS

Soutenance prévue le 16 Juillet 2013 devant le jury composé de

M. Martin FABIAN	Université de technologie de Chalmers	Rapporteur
M. Jean François PÉTIN	CRAN - Université de Lorraine	Rapporteur
M. Hassane ALLA	GIPSA - Université de Grenoble	Examinateur
M. Armand TOGUYENI	LAGIS - École Centrale de Lille	Examinateur
M. Éric NIEL	AMPÈRE - INSA de Lyon	Directeur de thèse
M. Emil DUMITRESCU	AMPÈRE - INSA de Lyon	Co-encadrant

A SAFE COTS-BASED DESIGN FLOW OF EMBEDDED SYSTEMS

Abstract

This PhD dissertation contributes to the safe design of COTS-based control-command embedded systems. Due to design constraints bounding delays, costs and engineering resources, component re-usability has become a key issue in embedded design.

The major difficulty in designing these systems is the high number of COTS components, which usually are separately built. The design process amounts to assembling these elementary components; this often establishes a certain amount of interaction between sets components which were not initially intended to interact with each others. Thus, unwanted behaviors may occur, although each component taken separately is considered free of local errors. The challenge that the designer faces is to ensure a safe behavior of the system which is built over the COTS components.

Our proposal is a design method which ensures correction of COTS-based designs. This method uses in synergy a number of design techniques and tools. It starts from modeling of the COTS components which are stored in a generic COTS library, and ends with a design of the global control-command system, verified to be free of errors and ready to be implemented over a hardware chip such as an ASIC or an FPGA "Field Programmable Gate Array".

The designer starts by modeling the temporal and logical local preconditions and post-conditions of each COTS component, then the global pre/post conditions of the assembly which are not necessary a simple combination of local properties. He models also a list of properties that must be satisfied by the assembly. Any violation of these properties is defined as a design error. Then, by using the model checking approach the model of the assembly is verified against the predefined local and global properties. Some design errors can be corrected automatically through the Discrete Controller Synthesis method *(DCS)*, others however must be manually corrected. After the correction step, the controlled control-command system is verified. Finally a global simulation step is proposed in order to perform a system-level verification beyond the capabilities of available formal tools.

A human intellectual intervention in this design method appears is the intermediate step between detecting the errors and correcting them automatically. The model checking technique can only discover the errors and provide a counterexample which indicates where and how a property was violated, however, it leaves the correction task to the designer. On the other hand, DCS can correct errors by generating a “correct-by-construction” patch which controls the bugged component by assigning a subset of its inputs, designated as “controllable”. Despite its obvious advantages, the brute force application of this operation is completely unnatural

to embedded designers. We propose to use the model-checking counterexample as a hint for guiding the application of DCS.

Thus, our study combines three design techniques: the formal verification, the discrete controller synthesis and simulation, in order to provide a system safe by construction with the minimum manual interaction, to avoid making human mistakes in the design. We mention the advantages of each technique and argue its disadvantages and explain how each one is necessary for the others to provide an integrated work. We apply the method on two different systems, one concerns transferring data from senders to receivers through FIFO unit, the other is control-command system of a train passengers' access.

Résumé

Le travail présenté dans ce mémoire concerne une méthode de conception sûre de systèmes de contrôle-commande matériels embarqués constitués à base des composants sur étager (COTS). Un COTS est un composant matériel ou logiciel générique qui est naturellement conçu pour être réutilisable et cela se traduit par une forme de flexibilité dans la mise en oeuve de sa fonctionnalité : en clair, une même fonction peut être réalisée par un ensemble (potentiellement infini) de scénarios différents, tous réalisables par le COTS.

Par ailleurs, comme les COTS sont souvent conçus séparément, leur interconnexion peut engendrer, de par cette flexibilité inhérente, des situations non désirées, liées à la nouvelle exigence fonctionnelle ciblée. Ces situations se traduisent par des états " à éviter " sous peine d'effectuer des actions de contrôle commande incorrectes voire dangereuses. L'intégration des COTS dans le processus de conception des systèmes matériels réduit le temps de conception et permet d'utiliser des composants génériques existant sur le marché. On assemble des COTS pour obtenir des nouvelles fonctions, plus complexes. La démarche d'assemblage de composants se heurte cependant à un dilemme entre généricité, en vue d'une réutilisation la plus large possible, et spécialisation en lien avec un besoin de contrôle commande particulier.

La complexité grandissante des fonctions implémentées fait que ces situations sont très difficiles à anticiper d'une part, et encore plus difficiles à éviter par un codage correct. Réaliser manuellement une fonction composite correcte sur un système de taille industrielle, s'avère être très coûteuse. Elle nécessite une connaissance approfondie du comportement des COTS assemblés. Or cette connaissance est souvent manquante, vu qu'il s'agit de composants acquis, ou développés par un tiers, et dont la documentation porte sur la description de leur fonction et non sur sa mise en IJuvre. Par ailleurs, il arrive souvent que la correction manuelle d'une faute engendre une ou plusieurs autres fautes, provoquant un cercle vicieux difficile à maîtriser. En plus, le fait de modifier le code d'un composant diminue l'avantage lié à sa réutilisation.

C'est dans ce contexte que nous proposons l'utilisation de la technique de synthèse du contrôleur discret (SCD) pour générer automatiquement du code de contrôle commande correct par construction. Cette technique produit des composants, nommés contrôleurs, qui agissent en contraignant le comportement d'un (ou d'un assemblage de) COTS afin de garantir si possible la satisfaction d'une exigence fonctionnelle. La méthode que nous proposons possède plusieurs étapes de conception.

La première étape concerne la formalisation des COTS et des propriété de sûreté et de vivacité (P) en modèles automate à états et/ou en logique temporelle. L'étape suivante concerne

la vérification formelle du modèle d'un(des) COTS pour l'ensemble des propriétés (P). Cette étape découvrir les états de violation des propriétés (P) appelés états d'erreur. La troisième étape concerne la correction automatique des erreurs détectées en utilisant la technique SCD. Dans cette étape génère on génère un composant correcteur qui sera assemblé au(x) COTS original(aux) pour que leur comportement général respecte les propriétés souhaitées. L'étape suivante concerne la vérification du système contrôlé pour un ensemble de propriétés de vivacité pour assurer la passivité du contrôleur et la vivacité du système. En fin, une étape de simulation est proposée pour observer le comportement du système pour quelque scénarios intéressent par rapport à son implémentation finale.

Pour montrer l'applicabilité de la méthode proposée, et sa faculté à être utilisée en milieu industriel, nous l'utilisons sur un exemple de système de contrôle-commande de train.

Contents

List of figures

List of tables

Introduction

Context and Motivation

This thesis has an industrial context: the FERROCOTS project [1]; it focuses on the embedded systems' design, with applications to the train control systems at Bombardier, one of the industrial partners of this work. The results presented in this document rely on a series of choices: design method and underlying techniques, design constraints, as well as physical platform constraints, for the final implementation. This section presents the motivations of this work, and explains the design choices that were made.

Hardware embedded systems

The act of defining the notion of an embedded system is a considerable challenge. It can at least be said that there are many different points of view sometimes partially overlapping. A (quite old) perception of an *embedded computer system* defines it as a computer subsystem, that is a part of a larger system and performs some of the requirements of that system (IEEE,1992). This definition focuses on the meaning of "embedding" i.e. including a functionality within a bigger, more complex one. In the literature, many definitions of embedded systems exist. They have actually evolved through time, over a quite short period. In 1999 David E. Simon in [5] cites that "People use the term embedded system to mean any computer system hidden inside products such as VCRs, digital watches." This gives a very general characterization, emphasizing electronic devices encapsulated within a bigger systems. A more recent point of view, closer to the context of our work, states that embedded systems range from very small systems present inside a digital watch, an MP3 player, a personal computer PC, a microwave, a vacuum cleaner, a car global position system GPS, a car autopilot to large industrial systems, like a computer system used in an aircraft or rapid transit system [6]. This perception narrows

[1] FerroCOTS is railway industrial project managed by BOMBARDIER transport, it focuses on the use of reusable components (COTS) to reduce design costs and provide design flexibility. It seeks to evolve the technology process control electrical relays to FPGAs

down the characteristics of an embedded system, saying that they have dedicated functions, implemented as pieces of integrated electronics or sometimes as computing systems and that they evolve inside an environment having specific physical constraints [7]. In [8], Todd D. Morton defines embedded systems as electronic systems that contain a microprocessor or a micro-controller. Tim Wilmshurst, in 2006 [9] provides a more precise definition. He keeps the notion of hidden system, and adds characteristics of the embedded system functionality, considering the embedded system as a controller part of the larger system. His exact words mention “a system whose principal function is not computational, but which is controlled by a computer embedded within it. The computer is likely to be microprocessor or micro-controller. The word embedded implies it lies over a larger system hidden from view.". In the same year, *Henzinger et al.* in [10] refined the notion of embedded systems: they perform a computation task, they are intended to work within a physical environment with specific constraints, and they must “execute” on a physical platform *also subject to physical constraints*. In 2008, Wayne Wolf in [11] defines an embedded computer system by "any device that includes a programmable computer, but is not itself intended to be a general purpose computer". Regarding his opinion, a personal computer (PC) is not an embedded computing system whereas, a fax machine is. From his definition, it can also be understood that an embedded system has a precise function. It can be integrated into an existing system to add extra features to this latter. In [12], *interactive embedded systems* are characterized by their real-time evolution with respect to environment actions. In the interactive embedded systems, the system responds with respect to the data sent to it by the environments, while the environment monitors the system reaction to calculate or provide new data. Whereas, in *reactive embedded systems*, the system must be able to keep up with the rythm of its environment, which cannot “wait”.

Some recent (and mature) works defining the concept of embedded systems, emphasize the following: reactivity requirements related to operation within a physical environment, safety critical features, performance and application-specific implementation constraints [13], [14], [7]. These aspects are very close to the framework of the FerroCOTS project. Indeed, train automation systems are subject to important safety requirements, related to environment, having safety-critical constraints, and requiring safe, reliable, control automation. On the other hand, for both safety and maintainability reasons the physical technology chosen for implementing control automation is the FPGA technology. For these reasons, the formal models chosen in this work are specific to the hardware design context: the systems we consider are modeled by reactive and intercommunicating concurrent processes. Their underlying semantics is given by the synchronous communicating Finite State Machines model. This formal model is equally convenient for design, verification, and FPGA hardware synthesis.

We designate the *control-command* part of an embedded system as a program, implementing a reactivity infinite loop: receive environment inputs, compute and produce a reaction (outputs)

and possibly update its internal state. In this work, we focus of the safe design of control-command systems.

Embedded design

The embedded system design has become an extremely wide area, encompassing a large number of highly skilled specialties. Thus, high-performance computing meets electronic design, physical integrated design, (micro)-mechanical engineering, or even bio-engineering, in order to achieve specialized, sophisticated functions, subject to various constraints. According to [10], these constraints come from the interaction between computational processes and the physical world. Two kinds of interactions are considered: the reaction to physical, environment stimuli, and the execution on a physical dedicated platform. When related to environment reaction, such constraints express functional requirements and/or deadlines. On the other hand, constraints related to the execution on a physical platform express technological aspects such as processor speeds, memory available or hardware architecture. Such constraints have a strong influence on the design process of the target computational function.

Nowadays techniques are extremely mature in providing efficient design tools allowing embedded design engineers to focus on the desired behavior, and abstract away as much as possible the physical design constraints. Besides, according to Moore's law, the electronic integration capacities doubles every 1.5 to 2 years, for the same production cost. This evolution implies both growing density of electronic chips, but most of all growing speed, allowing realization of more and more complex computations.

Such a growing design complexity calls for appropriate design methods and techniques. A panel of solutions has actually been developed during the past decades, relying on the use of formal or semi-formal techniques for modeling, verification, optimization, code generation, etc. Some among these techniques have become quite mature. They have been embedded inside commercial design tools, and they are nowadays successfully used within industrial embedded design projects. A well-known example is the symbolic model checking technique [15], which has been developed during the early nineties, and whose potential was so important, that huge research and development efforts have been made to enhance even more its performance. Many commercial design tools have currently been made available by most vendors[2], featuring different commercial presentations of the same technique.

Unfortunately, these formal tools seem to have met their limits; their performance does not scale up to follow the growing complexity of the designs. This phenomenon is often due to a bad (exponential) asymptotic complexity, or to a need of a high level of expertise, in order to avoid

[2]Cadence, Mentor Graphics, Synopsis, IBM, ...

undecidability issues. In such a situation, a cost-effective, and thus satisfactory workaround for this situation advocates component reuse.

The design of embedded systems based on Commercial Off The Shelf (COTS) components has made a remarkable evolution in the design of hardware systems. COTS are regarded as "mature" components, and thus worthy of trust for being reused, or even sold. The obvious and undeniable advantage of COTS reuse is the dramatic reduction of design and coding costs. However new problems arise within a COTS-based design process. These are mainly caused by an antagonism between the needs of genericity and specificity. Reusability relies on component genericity, and this is effectively achieved through data type and functional abstraction as well as object-oriented modeling mechanisms. These techniques currently support the design of both efficient and reusable *software*. Unfortunately, such mechanisms are much less mature for integration inside an embedded design project, possibly producing important amounts *hardware*. On the one hand, their efficient compilation into hardware is tricky, and sometimes needs restrictions on the input language constructs, often eliminating most support for genericity. On the other hand, the most powerful among these mechanisms supporting genericity are somehow out of the reach for the average designer because of their lack of intuitiveness.

Thus, a "generic" COTS often achieves *specific* functional requirements, thoroughly verified (through simulation and/or formal verification) and ready to be implemented into hardware. In this context, the genericity can be found in the following points:

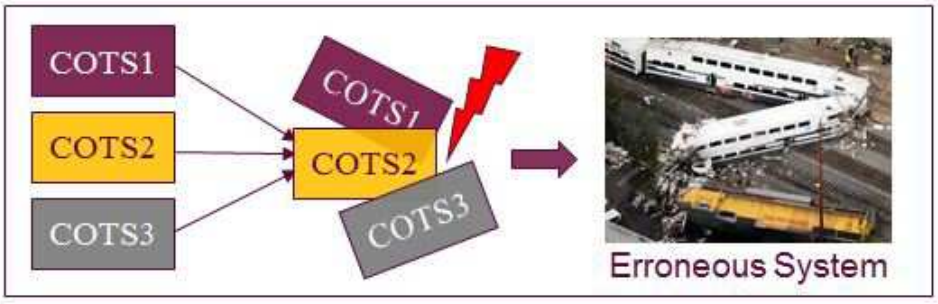
- the use of generic parameters, allowing the dimensioning of a component : for instance a bit-level adder can be described generically as a function of N, the total number of bits. The instantiation of the adder requires to assign an actual, constant value to N. The same mechanism can express alternative behaviors, to be chosen at instantiation time;
- the *interface* (input/output set) of the COTS, featuring a "standard" , known a priori, structure and behavior. For instance, an AHB bus is generic, as it offers a standard interface and exchange protocol;
- the internal behavior of the COTS, through well-documented and/or formally expressed functions.

Thus, an ideal COTS-based design process does not require writing code but only reusing existing components, either locally developed, or purchased. Here, the design task amounts to picking the right components in a library and interconnect them when needed. This approach is quite realistic and undeniably cost-effective: it allows the rapid construction of medium or large systems, quickly able to operate. However, the correction of such systems is extremely difficult to assess; while most common requirements are verified by simulation, *corner case* unwanted

scenarios, corresponding to possibly critical design errors are very likely to remain. Such errors have two possible origins:

- remaining design errors (bugs) within some COTS;
- new design errors *created through COTS interconnection*, as symbolized in Figure 1. Such a situation occurs because a COTS is developed to be generic, not specifically ready to fully co-operate with another COTS in order to deliver a new requirement.

FIGURE 1: Erroneous COTS-based system



Contributions

In this work we advocate the synergy between several design tools within a novel design method, in order to achieve time and cost-effective COTS-based design.

Firstly, formal verification is used to discover design errors. It is usually up to the designer to manually correct these errors. However, manual correction is a tough and error-prone mission, almost unfeasible in industrial sized systems.

Hence, secondly, the discrete controller synthesis (DCS) method is still used to automatically enforce some desired behaviors on a give system. More precisely, we apply the DCS either to automatically correct some of the previously discovered design errors, or to implement a new requirement for a given COTS interconnection. However, the DCS technique was not initially developed for hardware design, so its brute force application is impossible in this context. Indeed, this technique is intended for automatic generation of event-driven control programs able to receive events or values from sensors, and driving a series of *physical actuators*. In embedded hardware design, this paradigm is not directly applicable. We propose a framework allowing design engineers to use DCS as an automatic design error correction tool.

The third technique advocated in our method is simulation. We argue that a fully automatic detection and correction or design errors is not enough to validate the design of a critical system, where any mistake can cost the life of a human-being or in the best cases a huge amount

of money to correct such error. Simulation is involved as a complementary tool, providing additional support for visualizing the behavior of the system before its implementation.

Another important achievement of this work is the application of DCS to an industrial case study for hardware design, obtaining a system ready for FPGA implementation.

The thesis is structured as follows. Chapter 1 assembles the state of art of the hardware embedded systems and the modeling of such systems, as well as the methods used to errors discovery in a design. Then it recall the discrete controller synthesis technique and the principles of its use. Both notations and some arguments used in chapter 1 are a fruit of joint work developed in the same research team and presented on 2011 [16] at INSA de Lyon. Chapter 2 presents our contribution which is a safe design method for COTS-based control-command embedded systems. The method is illustrated on a realistic a hardware system which is a Generalized Buffer consists of a series of pre-built components. Chapter 3 is an application of the method over a real case of a train control subsystem. We finally conclude our work, we discuss the benefits of our method, its shortcomings and we mention some perspectives which can improve our work.

Chapter 1

Safe design of hardware embedded systems based on COTS : State of the art

1.1 Introduction

This chapter assembles the background for the main axes of our work. It recalls the models and tools for the safe design of finite-state Discrete Event Systems (*DES*). Then, we sneak inside the box of a hardware embedded system and explain the discrete time behavior of such systems and the functional requirements needed to be taken in consideration during the design process. Then, we recall three among the most widespread design tools used for verifying the correctness of a design: (1) the theorem proving, (2) the guided simulation and (3) the model checking technique. The later is explicitly used in our contribution. Then we announce how our contribution can fill the gaps of those methods. After that, we tackle the issue of enforcing by Discrete controller synthesis approach (DCS) functional requirements. Then, we present the state of the art for the component-based method for hardware design.

1.2 Modeling hardware systems

The embedded systems' design relies on the formal modeling of the dynamic behaviors. Several modeling techniques exist, and they differ according to their underlying formal model and their implementation technique. The *formal language* theory is the underlying framework formalizing *event-driven* systems. In this framework, sequences of events are fundamental in modeling the dynamic behavior of the system and its functional requirements. On the other hand, *sample-driven* systems are inspired from electronic design techniques. Their behavior is expressed as

sequences of system's states, and functional requirements always refer to subsets of states, either desired or forbidden.

There exists a conceptual difference between the behavioral dynamics of event-driven on the one hand and sample-driven models on the other hand. Both are able to model reactivity. However, event-driven models react upon an event, whenever it occurs, which is also known as an "event-driven" reaction. Sample-driven modeling assumes the existence of one special kind of events, generated by a *clock*. Whenever a clock event occurs, the inputs and the current state are *sampled* before reacting. Such systems run by the events generated by an external clock are also known as Sample-driven systems.

Such a variety of design models and techniques calls for a preliminary choice. In this work, DCS application is intended for hardware design. Besides, hardware systems are designed using Boolean synchronous finite state machines, which are sample-driven models. Thus, the most adequate modeling paradigm is the best suited for this work. This choice is detailed in the sequel.

Even though specific formal techniques have been developed on top of both event-driven and sample-driven models, we wish to highlight the fact that our results can be applied to both sample and event-driven models, through a simple transformation of event-driven into sample-driven models.

1.2.1 Event-driven modeling

Event-driven models consider the behavior of a discrete event system as a set of (possibly infinite) sequences of events. The system has an initial state and evolves according to the events that arrive. An event σ can occur and cause a system pass from its current state to the next state. Only one event can occur at a time. The event-driven modeling of discrete event system is based on the theory of formal languages developed in [17] and presented extensively in [18].

1.2.1.1 Formal language

Definition 1.1 (Formal language)**.** Consider the following items:

- An *alphabet* is a set of possible events of a DES, denoted as Σ;
- A *word* is a sequence of events from Σ;
- An empty word is denoted ε;

- $|\Sigma|$ denotes the cardinality of Σ;
- Σ^* denotes all the possible finite sequences of events from Σ;

A *formal language* defined over an event set Σ is a subset of Σ^*.

Example 1.1. *Given the alphabet Σ = {a,b,c}.*

- *a, ab, abc, abb, bcc are words over the alphabet;*
- *$L = \{\varepsilon,\ a,\ ab,\ abb,\ bcc\}$ is a language over the alphabet.*

Regular languages are an interesting particular class of formal languages. They are able to represent dynamic behaviors as sets of words where some subsets of these words can be expressed as *regular expressions*. Regular expressions over a finite alphabet Σ describe sets of strings by using particular operations such as: grouping, alternate expression, symbol concatenation and cardinality. A relation of bijection exist between *regular languages* and *event-driven finite state machines*: i,e. every regular language could be *generated* by a finite state machine and vice versa. This fact allows the designers to model the dynamics of a system following two distinct ways.

Definition 1.2 (Event-driven Finite state machine)**.** An event-driven finite state machine (FSM) is a 5-tuple:

$$M = \langle q_0, \Sigma, \delta, Q, Q_m \rangle \tag{1.1}$$

where:

- q_0 is the initial state;
- Σ is the set of events, σ is an event where $\sigma \in \Sigma$;
- $\delta : Q \times \{\sigma\} \rightarrow Q$ is the transition function, through which the system changes its current state;
- Q is the set of states, i.e; the state space;
- Q_m is the set of desired states to be reached.

Notice. The desired states Q_m are those which represent an achievement of a mission, such as the end of a task. This mechanism is not used in our work. As explained in the sequel, desired states are pointed out through formal specifications. Thus, in the sequel, when Q_m is not mentioned explicitely this means $Q_m = \emptyset$.

1.2.1.2 Common notions in event-driven modeling

- *Execution path*: A sequence of visited states $(q_0 q_1 q_2 ...)$. Noticing that a sequence is ordered set.
- *Reachable state*: a state (q) is called reachable if it is possibly visited starting from the initial state q_0 if there exist an execution path leading to it.
- *Next state*: given a current state (q) the next state $(q^{'})$ is the state visited when an event $\sigma \in \Sigma$ occurs. We can denote: $(q \times \{\sigma\} \rightarrow q^{'})$
- *Marked state*: a state q_m is a marked state, which represents a state that must be visited, starting from the initial state, i.e., there must be an execution path, which leads to this state.

The notion of reachable and marked states helps designers of discrete event systems to represent some required, inadmissible and possibly reached behavior of a system. A familiar example that we can find in our everyday life is microwave machine; the designer of a the control part of the microwave system can define a list of marked states that should be visited through the system life cycle like the "finished state".

Thus, the state marking adds a requirement expression mechanism to the event-driven FSM. In this work however, requirements are dissociated from the dynamic FSM model, and expressed separately. This choice is not fundamental, it is only based on the current practice in hardware design engineering. Thus, in the sequel, all the models we manipulate have $Q_m = \emptyset$.

For a given system, modeled by an event-driven FSM, The event-driven execution mechanism can be expressed by the following algorithm [19]:

Algorithm.1 Algorithmic description of the Event-driven paradigm

1: current state: $q \in Q$
2: next state: $q^{'} \in Q$
3: initialization: let $q = q_0$
4: **Begin loop**
5: wait for an event $\sigma \in \Sigma$
6: compute the reaction (next state computation): let $q^{'} = \delta(q, \sigma)$
7: finalize the transition: let $q = q^{'}$
8: **End loop**

The main restriction that applies when using event driven modeling to model hardware systems is the fact that it supposes that only one event can occur at a time, which is inadequate with the concurrent nature of a physical environment. Besides, hardware systems are inherently

concurrent. They are made of building blocks that run in parallel and possibly communicate with each other.

Building a global finite state machine model which represents the simultaneous behavior of all considered individual finite state machines is known as a parallel composition. More specifically, we focus on the *broadcast composition*. This notion is looser, because it does not add blocking phenomena to the composition. It is also conceptually close to the *synchronous product* [20] which is widely used in hardware design. The broadcast compositon requires the definition of the active event set for a given state.

Definition 1.3 (Active event set)**.** Let $M = \langle q_0, \Sigma, \delta, Q, Q_m \rangle$ be an event-driven FSM. The active event set $\Gamma : Q \rightarrow 2^{\Sigma}$ of a given state $q \in Q$ is defined as:

$$\Gamma(q) = \{\sigma \in \Sigma \text{ s.t. } \delta(q, \sigma) \text{ is defined } \}$$

The broadcast composition is defined as follows:

Definition 1.4 (Broadcast composition)**.** Let $M_1 = \langle q_{01}, \Sigma_1, \delta_1, Q_1, Q_{1m} \rangle$ and $M_2 = \langle q_{02}, \Sigma_2, \delta_2, Q_2, Q_{2m} \rangle$ be two finite state machines.

The broadcast composition of M_1 and M_2 [21] is denoted $M_1 \parallel M_2$ and is defined as follows:

$$M_1 \parallel M_2 = \langle (q_{01}, q_{02}),\ \Sigma_1 \cup \Sigma_2,\ \delta_{M_1 \parallel M_2},\ Q_{1m} \times Q_{2m} \rangle \tag{1.2}$$

The compound transition function is defined as follows:

$$\delta((q_1, q_2), \sigma) = \begin{cases} (\delta_1(q_1, \sigma), q_2) & if\ \sigma \in (\Gamma_{q_1} \setminus \Gamma_{q_2}) \\ (q_1, (\delta_2(q_2, \sigma)) & if\ \sigma \in (\Gamma_{q_2} \setminus \Gamma_{q_1}) \\ (\delta_1(q_1, \sigma), \delta_2(q_2, \sigma)) & if \sigma \in (\Gamma_{q_1} \cap \Gamma_{q_2}) \\ Undefined & Otherwise \end{cases} \tag{1.3}$$

Example 1.2. *Let M_1 and M_2 be two finite states machines. They are defined as follows.*
$M_1 = \langle q_{01}, \Sigma, \delta, Q, \emptyset \rangle$.
Where:
$q_{01} = A$, $\Sigma = (a, b, c)$, $Q = \{A, B, C\}$, *δ is given as follows:*

$$\delta(state, event) : \begin{cases} \delta(A, b) = B \\ \delta(B, c) = C \\ \delta(C, a) = A \end{cases} \tag{1.4}$$

$Q_m = \emptyset$ $M_2 = \langle q_{02}, \Sigma, \delta, Q, \emptyset \rangle$.
Where:
$q_{02} = A$, $\Sigma = (a, b, j)$, $Q = \{J, H, I\}$, δ *is given as follows:*

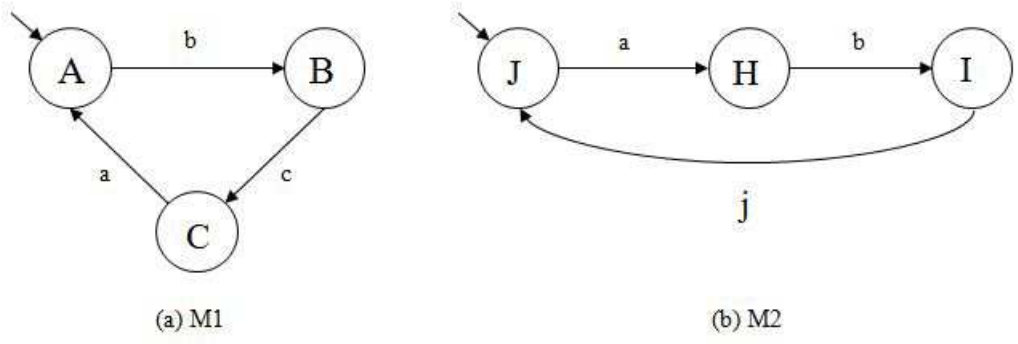
$$\delta(state, event) : \begin{cases} \delta(J, a) &= H \\ \delta(H, b) &= I \\ \delta(I, j) &= J \end{cases} \tag{1.5}$$

$Q_m = \emptyset$

Let a, b *be two shared events,* $\Sigma_1 \cap \Sigma_2 = \{a, b\}$. *A finite state model of the machines M1, M2 is represented in figure 1.1 (a), (b) respectively.*

The synchronous product of the two machines is illustrated in figure 1.2.

FIGURE 1.1: Two concurrent finite state machines



(a) M1 (b) M2

The synchronous product is a looser variant of the synchronized product, used in discrete-event modeling and denoted $\times$. It is defined as follows [21] :

Definition 1.5 (Synchronized product)**.** Given two finite state machines $M_1 = \langle q_{01}, \Sigma_1, \delta_1, Q_1, Q_{1m} \rangle$ and $M_2 = \langle q_{02}, \Sigma_2, \delta_2, Q_2, Q_{2m} \rangle$.

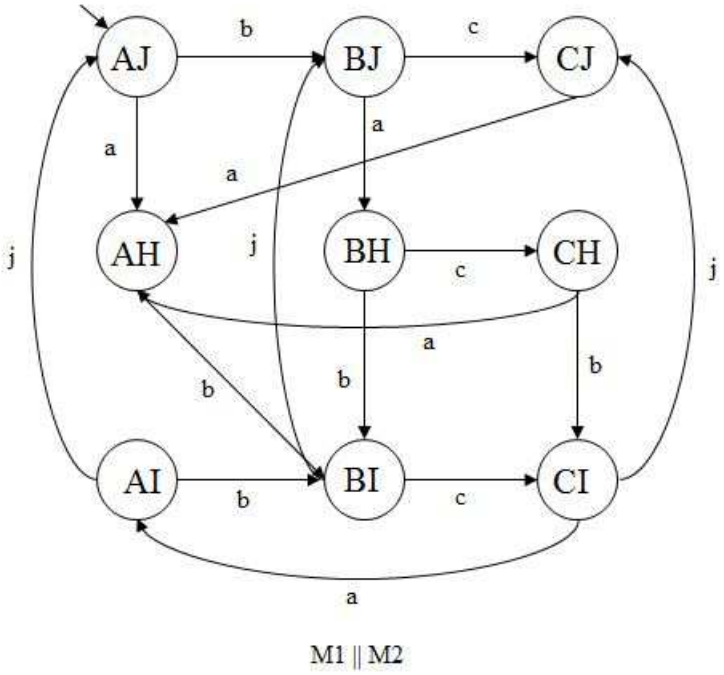
The synchronized product of the two machines is denoted $M_1 \times M_2$ and defined as follows:

$$M_1 \times M_2 = \langle (q_{01}, q_{02}),\ \Sigma_1 \cap \Sigma_2, \delta_{M_1 \times M_2},\ Q_1 \times Q_2, Q_{1m} \times Q_{2m} \rangle \tag{1.6}$$

where the transition function is :

$$\delta((q_1, q_2), \sigma) = \begin{cases} (\delta_1(q_1, \sigma), \delta_2(q_2, \sigma)) & if \sigma \in (\Sigma_1 \cap \Sigma_2) \\ Undefined & Otherwise \end{cases} \tag{1.7}$$

FIGURE 1.2: Synchronous product of two concurrent finite state machine M1, M2



The *synchronized product* is much more strict as a composition operation than the synchronous product as the global automaton can evolve if and only if there exist common events among the combined automata. In the case where $\Sigma_1 \cap \Sigma_2 = 0$ then the global combination automaton remains locked in its initial state. In our work we employ the *synchronous product* as our target is to combine components which do not necessary have events in common.

Example 1.3. *Let* M_1, M_2, M_3 *be three finite state machines.* $M_i = \langle q_0, \Sigma, \delta, Q, Q_m \rangle$. *Where :* $q_{01} = A$, $\Sigma_1 = \{a, b\}$, $Q_1 = \{A, B\}$ δ_1 *is given as follows:*

$$\delta(state, event) : \begin{cases} \delta(A, b) = B \\ \delta(B, a) = A \end{cases} \tag{1.8}$$

$q_{02} = C$, $\Sigma_2 = \{c, d\}$, $Q_2 = \{C, D\}$ δ_2 *is given as follows:*

$$\delta(state, event) : \begin{cases} \delta(C, d) = D \\ \delta(B, c) = C \end{cases} \tag{1.9}$$

$q_{03} = E$, $\Sigma_1 = \{a, e\}$, $Q_3 = \{E, F\}$ δ_3 *is given as follows:*

$$\delta(state, event) : \begin{cases} \delta(E, b) = F \\ \delta(F, e) = E \end{cases} \tag{1.10}$$

$Q_m = \emptyset$

A finite state model of each of the machines M_1, M_2, M_3 is illustrated in figure 1.3. The product operations of the machines M_1, M_2 and M_1, M_3 are shown in figure 1.4. Notice that the finite state automaton resulting of $M_1 \times M_2$ in the left figure is restricted to the initial state AC since the automata have no common events, whereas in the right figure the product result evolves from the state (AE) to (BF) due to the event (b).

FIGURE 1.3: finite state model of M1, M2, M3 machines

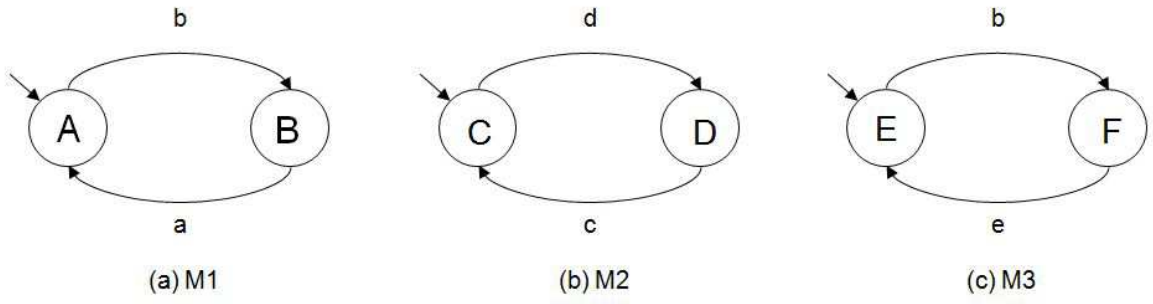
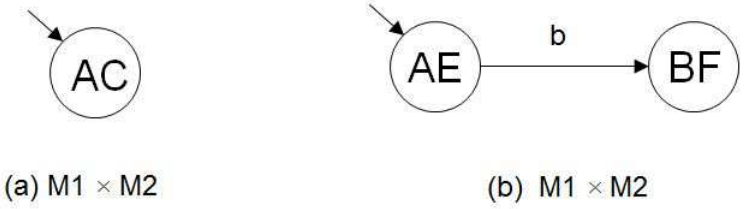


FIGURE 1.4: Product operation of finite state model for M1 x M3 machines

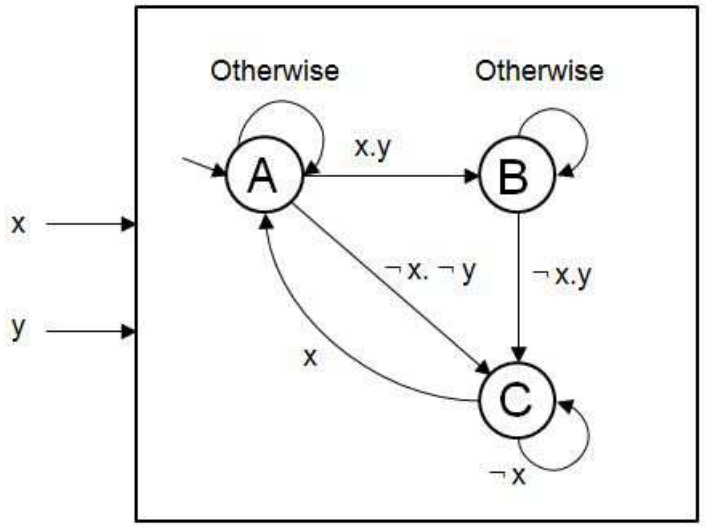


1.2.2 Sample-driven modeling

Sample-driven models handle values, instead of events. The main difference comes from the abstraction level between these two notions. Events are considered abstract modeling artifacts, and one possible implementation of an event synchronization mechanism is the continuous sampling of a value. Sampling requires a clock, and a sampling frequency. In the sample-driven modeling there exist only one event which is a hardware clock tick. At each clock tick, all system's inputs are sampled and their values are read and all transitions are triggered. The next state is calculated with respect to the current state and the values read from the environment. An implicit transition is modeled if no change of state is needed regarding the input sampling as shown in figure 1.5. The actual clock frequency remains abstract and remains to be subsequently determined at a physical implementation step. It is assumed, and this assumption remains to be

physically validated, that the clock frequency is sufficient for an accurate observation of the environment dynamics.

FIGURE 1.5: Sample-driven model of a FSM



Definition 1.6 (Sample-driven finite state machine)**.** A sample-driven model of a discrete event system can be presented as a FSM of a 4-tuple $\langle q_0, X, \delta, Q, Q_m \rangle$ where:

- q_0 is the initial state;
- X is set of Boolean input variables;
- Q is set of states, represented by state variables;
- δ is the transition function: $\delta : Q \times \mathbb{B}^{|x|}$.

The behavior of a discrete-event system modeled by a **sample-driven** finite state machine can be expressed by the algorithm [19]

Algorithm 2: algorithmic description of the Sample-driven paradigm

1: initialization: $i = 0$, $q^i = q_0$
2: **forever**, at each clock tick *i* **do**
3: read input variables x^i
4: calculate the next state: $q^{i+1} = \delta(q^i, x^i)$
5: update state $q = q'$
6: **end**

1.2.2.1 Translating event-driven into sample-driven models

We use in the following the common notation for the classical Boolean operators, such as : " $\wedge$ " for the logical "and", " $\vee$ " for the logical "or", $\neg a$ for the logical negation of a, "$\rightarrow$" for the logical implication and "$\Leftrightarrow$" for the Boolean equivalence. To simplify the figures we use " . " instead of " $\wedge$ "

In this work we focus on hardware systems, which are modeled using sample-driven finite state machines. However, we argue that the results presented in the sequel can also apply to event-driven models. These can be systematically translated into sample-driven representations [22]. The transformation method we consider, between an event-driven model into a sample-driven model, is quite straightforward, and simply recalled here.

Events can be translated to vectors of values, one vector per event. Thus, each event is mapped to a unique Boolean encoded value according to a function:

$$enc : \Sigma \rightarrow \mathbb{B}^{log_2(|\Sigma|)} \tag{1.11}$$

Applying the encoding function enc to any event-driven FSM $M^E = \langle q_0^E, \Sigma^E, \delta^E, Q^E, Q_m^E \rangle$ results in a sample-driven representation of the same machine $M^T = \langle q_0^T, X^T, \delta^T, Q^T, Q_m^T \rangle$, where:

- $Q^T = Q^E$;
- $q_0^T = Q_0^E$;
- $X^T = x_0^T x_1^T ... x_{n-1}^T$ where $n = log_2^{|\Sigma^E|}$;
- $\delta^T : Q^T \times \mathbb{B}^n \rightarrow Q^T$, the transition function, which is defined as follows

$$\delta^T(q^T, x^T) = \left\{ \begin{array}{ll} \delta^E(q^T, enc(\sigma) & \text{if } \sigma \in \Gamma_E(q) \\ q^T & \text{if } \sigma \in \Sigma_E \setminus \Gamma_E(q) \end{array} \right\} \tag{1.12}$$

- $Q_m^T = Q_m^E$

The act of sampling an event-driven model M involves a clock, which is required to be unique for the whole system, otherwise no synchronization is directly possible. It is assumed that the frequency of the clock is sufficiently high to capture the occurrence of any event from Σ. Under these assumptions of clock unicity and sufficient frequency, the clocking mechanism can be abstracted away from any sample-driven model. A physical clock signal is introduced only at a physical hardware implementation step.

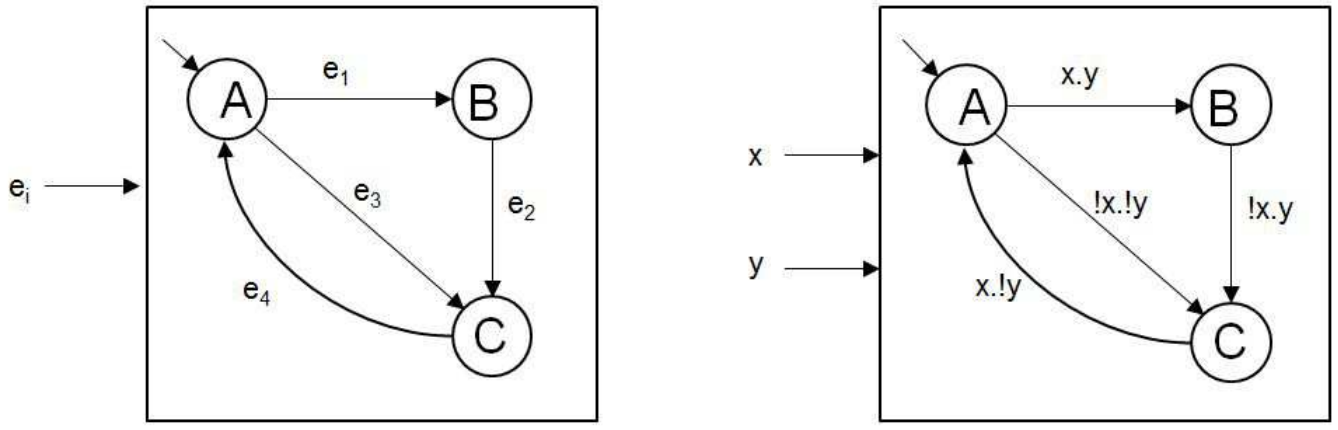
Example 1.4. *Let $M = \langle q_0, \Sigma, \delta, Q, Q_m \rangle$ be a three-states machine, where:*

- *The initial state is $q_0 = A$*
- *The list of expected events is $\Sigma = \{e_1, e_2, e_3, e_4\}$*
- *The set of states is $Q = \{A,\ B,\ C\}$*
- *The set of marked states is $Q_m = \emptyset$*
- *The transition function is $\delta : Q \times \Sigma \rightarrow Q$ where:*

$$\delta(q,(e_i)) = \begin{cases} (\delta(A,(e_1)) & = & B \\ (\delta(A,(e_3)) & = & C \\ (\delta(B,(e_2)) & = & C \\ (\delta(C,(e_4)) & = & A \end{cases} \tag{1.13}$$

A graphical representation of this machine is illustrated in figure 1.6

FIGURE 1.6: Sample-driven to event-driven modeling



As the machine receives $|\Sigma|$ = 4 events, then, $log_2(4) = 2$, two Boolean variables are needed and sufficient to encode the events and translate them into Boolean input variables. Let $X = \{x, y\}$. Let $enc(\sigma)$ be an encoding function computing values for the vector (x,y) and defined as:
$enc(e_1) = (1, 1)$;
$enc(e_2) = (0, 1)$;
$enc(e_3) = (0, 0)$;
$enc(e_4) = (1, 0)$

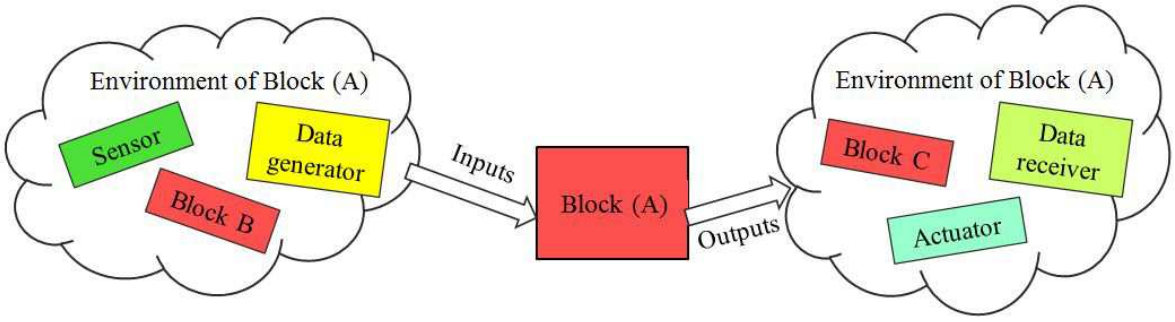
The translation of M from event-driven into a sample-driven model M^T is illustrated in figure 1.6 (a) and (b) respectively. All the remaining combinations of (x) and (y) not corresponding to $e_1 \dots e_4$ are self-loop transitions in M^T, left unrepresented for more clear readability of the figure. Through application of enc^{-1} they would fall into the abs event labeling a supplementary self-loop transition.

Notice that the inverse process of building an event-driven model from a sample-driven one through enc^{-1} does not necessarily yield the initial event-driven model. This happens because enc is not bijective; it is forced in that sense through the addition of the abs event, whose presence is not natural in an event-driven model.

1.2.2.2 Modeling interaction

Embedded systems are inherently concurrent systems. Concurrency is featured between building blocks, but also with the surrounding physical environment. This parallel execution requires communication abilities between the different elements, in order to establish interaction. For this purpose, the sample driven *FSMs* are extended with *outputs*.

FIGURE 1.7: Environment of a block



There exist two possibilities to model outputs: (1) An *implicit* representation of outputs i.e, the input alphabet of an event- or sample-driven FSM is defined as a *set of pairs* of input/output events. This technique has been applied in [21] for modeling concurrent control systems. (2) An *explicit* representation of outputs i,e outputs are separated from the inputs. In our work we adopt the second choice since it is more considered a more natural modeling practice by hardware design engineers. Thus, finite state machines communicating through outputs are defined as follows:

Definition 1.7. A FSM with outputs is a 6-tuple:

$$M = \langle q_0, X, Q, \delta, PROP, \lambda \rangle \tag{1.14}$$

where: q_0, X, Q, δ are exactly as defined above, and $PROP, \lambda$ are defined as follows:

- $PROP = \{p_1, ..., p_k\}$ is a set of k atomic Boolean propositions;
- $\lambda : Q \rightarrow \mathbb{B}^k$ is a labeling function, modeling the outputs of M;

Depending upon how outputs are handled during modeling, two approaches exist: associate output assignments to either states or transitions. Thus, *Moore*-like FSMs associate outputs to states, and *Mealy*-like FSMs associate outputs to transitions. More specifically, in a Moore machine, the output function $\lambda : Q \rightarrow \mathbb{B}^k$ is defined as:

$$For\ i = 1\ to\ k\ :\ \lambda^i(q)\ =\ 1 \text{ iff } p_i\ is\ true\ in\ state\ q. \tag{1.15}$$

In a Mealy machine, the output function calculates the value of an output variable for a given couple (state, input) (q, x), i.e., outputs are associated to transitions: $\lambda : Q \times \mathbb{B}^{|X|} \rightarrow \mathbb{B}^k$ is defined as

$$For\ i = 1\ to\ k\ :\ \lambda^i(q, x)\ =\ 1 \text{ iff } p_i \text{ is true in state q and the input equals } x \tag{1.16}$$

Let us define a reverse mapping λ^{-1} able to associate a given Boolean predicate defined on the elements of $PROP$ to the set of states in Q where this predicate holds. Let $\mathcal{P}$ represent the collection of all the possible Boolean predicates constructed with atomic propositions from $PROP$. We have $PROP \subset \mathcal{P}$. The reverse mapping $\lambda^{-1} : \mathcal{P} \rightarrow 2^Q$, is defined as follows:

- for $p_i \in PROP : \lambda^{-1}(p) = \{q \in Q | \lambda^i(q) \text{ is true } \}$
- $\lambda^{-1}(a \wedge b) = \lambda^{-1}(a) \cap \lambda^{-1}(b)$ with $a, b \in \mathcal{P}$;
- $\lambda^{-1}(a \vee b) = \lambda^{-1}(a) \cup \lambda^{-1}(b)$ with $a, b \in \mathcal{P}$;
- $\lambda^{-1}(\neg a) = Q \setminus \lambda^{-1}(a)$, with $a \in \mathcal{P}$;

Example 1.5. *Consider the finite-state machines shown in figure 1.8. They illustrate the distinctions between Mealy/Moore modeling styles. The character '*' denotes any Boolean value. Both models have the same set of states:* $\{I, W, A\}$*, and receive the Boolean inputs* $\{req, go, stop\}$*, and assign the output* $(done)$*. The machines are identical in behavior, they just differ in the moment of delivering the output.*

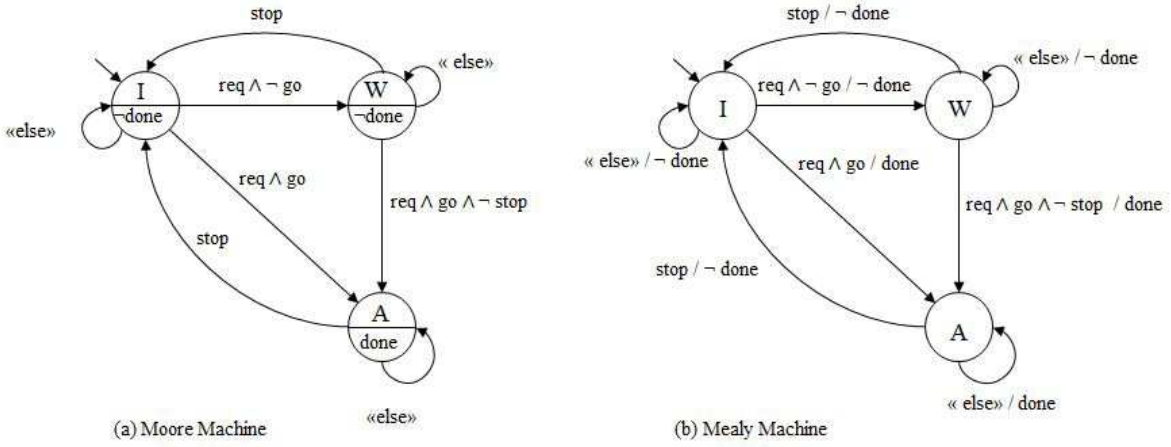
In the Moore machine, figure 1.8 (a) the output only depends on the system's state. We suppose the output function definition as follows:

$$\begin{aligned} \lambda(I) &= \{\neg done\}; \\ \lambda(W) &= \{\neg done\}; \\ \lambda(A) &= \{done\}. \end{aligned}$$

In the Mealy machine, figure 1.8 (b), outputs are associated to transitions. To make this Mealy machine identical to the Moore one its output function should be defined as follows:

$$\begin{aligned}
\lambda(I,(req,go,stop)) &= \{\neg done\},\ if\ (req,go,stop) = (*,0,*)\\
\lambda(I,(req,go,stop)) &= \{done\},\ if\ (req,go,stop) = (1,1,*)\\
\lambda(W,(req,go,stop)) &= \{done\},\ if\ (req,go,stop) = (1,1,0)\\
\lambda(W,(req,go,stop)) &= \{\neg done\},\ if\ (req,go,stop) = (*,*,1)\ or\ (*,0,*)\\
\lambda(A,(req,go,stop)) &= \{done\},\ if\ (req,go,stop) = (*,*,0)\\
\lambda(A,(req,go,stop)) &= \{\neg done\},\ if\ (req,go,stop) = (*,*,1)
\end{aligned}$$

FIGURE 1.8: Moore vs. Mealy FSM models



1.2.3 Synchronous product with interaction

When finite state machines work concurrently and interact, a synchronous product of the two machines $M_1||M_2$ is calculated under the assumption that the outputs of the product are uniquely assigned by either M_1 or M_2.

Interacting FSMs are composed according to the synchronous paradigm defined in [20]. This operation requires a preliminary input/output mapping, usually performed by the design engineer Let $M_i, i = 1, 2$ be two communicating finite state machines, where $M_i = \langle q_{0i}, X_i, Q_i, \delta_i, PROP_i, \lambda_i \rangle$. The set of inputs X_i of M_i is divided into 2 disjoint subsets : $X_i = L_i \cup I_i$. The machines M_1, M_2 communicate through $Prop_1^{out}, Prop_2^{out}$ where $Prop_1^{out} \subseteq PROP_1$ is a subset of Boolean propositions which connects M_1 to M_2 via L_2 and $Prop_2^{out} \subset PROP_2$, is a subset which connects M_2 to M_1 via L_1. This interconnection is illustrated in figure 1.9

The synchronous product $M_1||M_2$ is represented as:

$$M_1||M_2 = \langle (q_{01}, q_{02}), X_{12}, \delta_{12}, Q_{12}, PROP_{12}, \lambda_{12} \rangle \tag{1.17}$$

where:

- $Q_{12} = Q_1 \times Q_2$;
- $X_{12} = I_1 \cup I_2$;
- $\delta_{12} : Q_1 \times Q_2 \times \mathbb{B}^{|X_1|} \times \mathbb{B}^{|X_2|} \rightarrow Q_1 \times Q_2$ is defined as:

$$\begin{aligned} \delta_{12}(q_1, q_2, \mathbf{i_1}, \mathbf{i_2}) =& (\delta_1(q_1, \mathbf{i_1}, \lambda_2^1(q_2), \cdots, \lambda_2^{|L_1|}(q_2)), \\ & \delta_2(q_2, \mathbf{i_2}, \lambda_1^1(q_1), \cdots, \lambda_1^{|L_2|}(q_1))); \end{aligned}$$

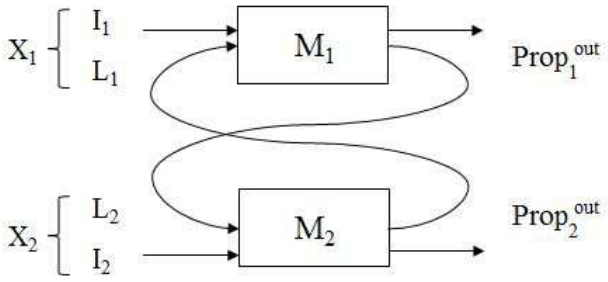
- $PROP_{12} = PROP_1 \cup PROP_2$;
- $\lambda_{12} : Q_{12} \rightarrow \mathbb{B}^{|PROP_1|+|PROP_2|}$ the output function is defined as:

$$\lambda_{12}(q_1, q_2) = (\lambda_1(q_1), \lambda_2(q_2)) \tag{1.18}$$

The expression of the reverse function $\lambda_{12}^{-1} : PROP_1 \cup PROP_2 \rightarrow 2^{Q_1 \times Q_2}$ is the following:

$$\lambda_{12}^{-1}(p) = \lambda_1^{-1}(p) \times Q_2 \cup \lambda_2^{-1}(p) \times Q_1. \tag{1.19}$$

FIGURE 1.9: Product of communicating FSMs



1.2.4 Efficient manipulation of symbolic models

Binary Decision Diagrams, that we abbreviate BDDs [23] have shown their efficiency in manipulating Boolean expressions. They have been successfully used for efficient FSM representation

and state exploration [15, 24, 25]. A BDD represents a Boolean expression as a direct acyclic graph, having two terminal nodes: true (1) and false (0). Their key advantage is the ability to provide a compact and canonic representation of a Boolean expression. Thus, constructing a BDD implicitly assess the satisfiability of the Boolean expression which is represented: a tautology is always represented by the terminal node (1), and a contradiction by the terminal node (0). Besides, all Boolean operations have a corresponding efficient BDD implementation.

For a given Boolean expression defined over a set of Boolean variables, building a BDD involves choosing an a priori total variable order. The key measure of BDD efficiency is the size of a BDD: the number of graph nodes required to build the diagram. The theoretical spatial complexity of a BDD is exponential in the number of Boolean variables contained.

The size of a BDD strongly depends on the initially chosen variable ordering. In practice, "good" variable orders can often be found, but unfortunately there is no tractable way to compute the best variable ordering. However, BDD packages implement interesting heuristic techniques for handling variable orderings dynamically yielding fairly good performance. Besides, for some known Boolean expressions, good variable orderings simply do not exist.

BDD-based formal techniques have reached their limits in terms of performance this is why most industrial and research efforts focus on the synergy between several techniques. Thus, BDDs are combined with SAT-based approaches (Satisfiability formal verification) [26] in order make the handling of large systems more tractable. Given a propositional formula φ, the Boolean Satisfiability problem posed on φ is to determine whether there exists a variable assignment under which φ evaluates to true in certain part of the total state space. If such assignment exists the φ is called satisfiable. Otherwise φ is said to be unsatisfiable.

In this work, an important part of our results rely on the symbolic Discrete Controller Synthesis technique, with a BDD-based implementation.

1.3 Behavior requirements specification

Modeling the behavior of hardware systems is usually achieved using hardware description languages like VHDL [27] and Verilog [28] or even system-level languages such as SystemC [29]. These are standard (IEEE) languages; they are dedicated for embedded hardware modeling and can be exploited by design tools for simulation, hardware synthesis, or formal verification.

However, any design process is a part of a more complex design flow, starting with requirement specifications. This process starts with functional requirements asserted informally (natural language). These requirements are first formalized, and then progressively refined, using UML or

SysML. Through such a process, a functional architecture (system-level) is refined into an organic architecture, and functional requirements are progressively mapped to behavioral requirements. These relate to the sequential behavior of a *component*, modeled as a communicating Boolean finite-state machine. Such requirements can be expressed formally either *logically*, using propositional and/or temporal logic, or *operationally*, by describing the desired behavior as a program.

1.3.1 Logic specifications

Logic specifications are Boolean assertions built over variable names, Boolean operations and *temporal operators*, expressing the relationship between the system behavior and abstract or explicit time. Such expressions extend the classical Boolean logic and are known as *temporal logic* formulæ, or temporal properties.

Temporal logic is an important specification tool, used by embedded design engineers as a means of formalizing an expected behavior of a system [30]. Its interest is confirmed by the fact that is has become a IEEE standard [31]. Besides, a number of commercial tools [32, 33] are able to process it. The most frequently used formulædenote the following properties:

- Safety: an unwanted behavior can never be observed during the system's life cycle. Such requirements express dangerous states or input/output sequences;
- Liveness: a desired behavior must eventually occur, at least once, during the system's life cycle. It can be a state that raises the accomplishment of certain mission or the end of a functional phase;
- Fairness property: this notion is close to the liveness, with a difference that the fairness requires that certain states of the system must be visited infinitely often during the system life cycle.

  Fairness properties are used to express repetitive behaviors either for the system or its environment. Indeed, reactive systems operate continuously hence they feature repetitive behaviors which are formalized by fairness properties. This mechanism is similar to the state marking used in event-driven modeling.

Among the variety of existing temporal logics, two variants are more frequently used, essentially because they are supported by most formal tools, either academic [15, 34] or industrial [32, 33]: the Linear-time Temporal Logic and the Computation Tree Logic.

1.3.1.1 Linear-time temporal logic (LTL)

Temporal logic has been defined by Pnueli in [35].

Temporal logic uses tense operators, like "always", "next time", "before" and "until", to express the evolution of a system through the time. The system's behavior is viewed as an infinite collection of linear traces. Each trace describes an infinite sequence of states. Each state inside a trace can have only one successor. Linear-time formulæare required to hold in the initial state of the system:

- Gp : p should hold forever;
- Fp : p should hold in some state in the future, at least once;
- Xp: p should hold in the immediate next state;
- p U q: p should hold until q holds, and q is required to hold eventually;
- p W q : p should hold until q holds, and q is not required to hold;
- p Before q: this is equivalent to $\neg q$ W $(p \wedge \neg q)$. This requirement states that q should never hold until p holds. If p occurs, q may occur afterwards.

1.3.1.2 Computation tree logic (CTL)

In the branching time logic, several executions are possible at a given point in time. Every state has a unique past and many future possibilities as shown in figure 1.10. CTL logic formulæ-combine the Boolean logic with two *path quantifiers* and *state quantifiers*.

The path quantifiers are summarized as follows:

- The universal path quantifier $A\varphi$: means the property φ should hold on all the branches of the computation tree starting from the current state.
- The existential path quantifier $E\varphi$: means the property φ should hold on at least one path of the computation tree starting from the current state.

The state quantifiers are summarized as follows:

- Next-time operator $X\varphi$: means that φ should hold in the exact next state;
- Future operator $F\varphi$: means that φ should hold at least once in the future ;

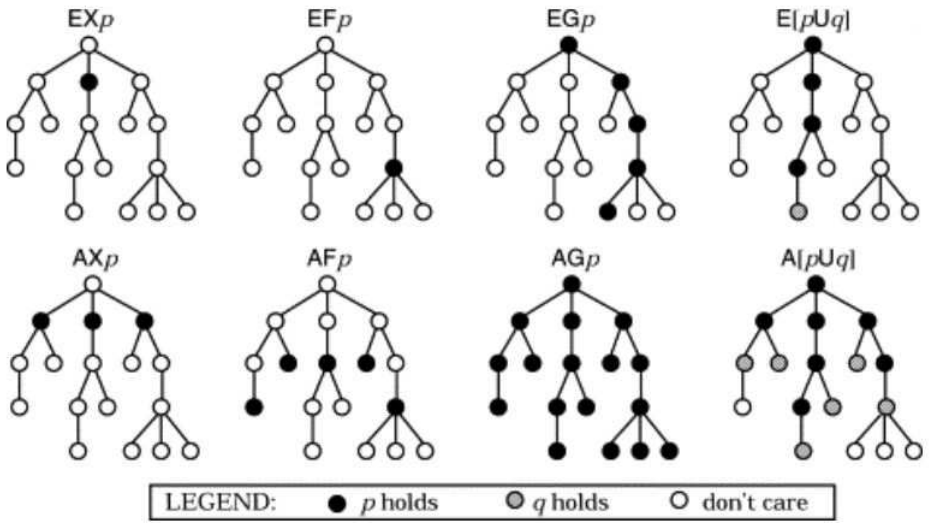
- Global operator $G\varphi$: means that φ should hold on all the states in the future;
- Until operator $\varphi U\psi$: means that φ should hold until ψ holds.

According to these elements, a CTL formula constructs as follows:

Definition 1.8. The syntax of CTL formulæis defined as follows:

- every Boolean atomic proposition is a CTL formula;
- if φ and ψ are CTL formulæ, then $\neg\varphi$, $\varphi \cdot \psi$, $AX\varphi$ $EX\varphi$, $A(\varphi U\psi)$, $E(\varphi U\psi)$ are CTL formulæ.

FIGURE 1.10: CTL tree logic, [1]



Branching time provides flexibility in expressing systems of complex behavior. It can express the functional behavior of a system through temporal properties:

1. Safety : $AG(\neg\varphi)$, given φ is a dangerous behavior of the system. As long as the property holds, the system is safe.
2. Liveness : $AF(\varphi)$, given φ is a required behavior of the system that must occur at least once in the system life cycle, the system is alive as long as the property holds.
3. Faireness : $AG\ AF\varphi$, this property means (φ) must occur eventually often during the system life cycle.
4. Acknowledgment : $AG(a \rightarrow AFb)$, given (a) a request that needs to be acknowledged by (b). this property usually used in control systems to verify that commands have been taken in account or not.

Logical specifications have a truth value for any state of a FSM model. We write $P, s \models \varphi$, meaning that formula φ is true in state s of P.

Logic specifications (LTL or CTL) are provided by most formal academic and commercial tools. However, LTL and CTL do not have the same expressive power. Most useful requirements can be expressed in both, but they also feature very subtle differences, which are quite confusing for design engineers. For example, the LTL formula FGp can be easily confounded with the CTL formula $AF\ AGp$, saying that they are logically equivalent on a the same system, which is not true. Anyway, the "flavor" choice, linear-time or branching-time, is mainly a matter of designer's preference. Besides, some commercial formal tools are restricted to very simple logic specifications, officially for efficiency and scalability reasons. In such situations, a possible workaround is writing operational spefications.

1.3.2 Operational specifications

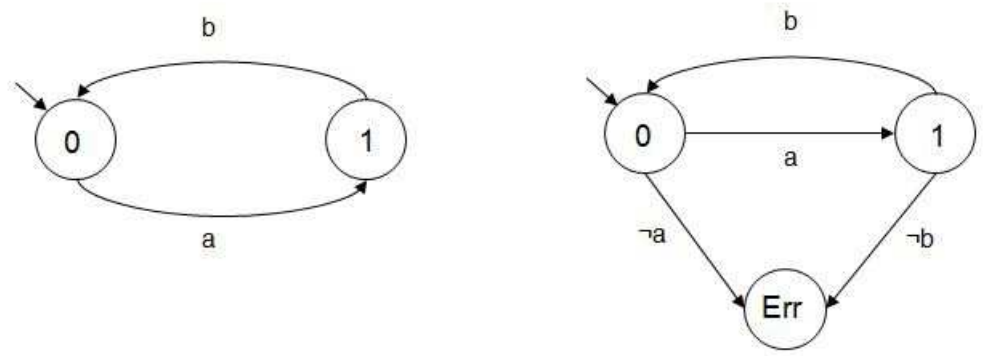
Operational requirements can be specified by a "program", modeled as one or more interacting finite state machines. Two modeling approaches co-exist in embedded design:

- a reactive program, modeled by a communicating FSM featuring the required input/output mechanisms, as well as the required dynamic behavior; such a program is regarded as a *"golden" or reference model*, and further implementations should relate to this model, formally or not. For instance, arithmetic operations can be specified as golden models, and their hardware implementation should be *proven equivalent* to this model;
- a reactive program equally modeled by a FSM, but whose purpose is merely observing values, and/or sequences of values, and asserting an error, by assigning a dedicated output, whenever an unwanted configuration is reached. Such a model is called an *observer* or a *monitor* [36]

Example 1.6. ***Monitor for alternative behavior*** *Let p be a requirement expressing alternation between two observed values (a, b). At the first cycle, a should be true, whatever the value of b, then immediately after, b should be true. This property is illustrated by the automaton shown in figure 1.11 (a). The monitor which observes this property is illustrated in figure 1.11 (b). As soon as the desired behavior does not happen any longer, the monitor switches to an error state.*

Monitors are usually written manually. They may be preferred to logic specifications if the requirement is easier to express operationally, such as desired sequences. It is equally possible, as shown later in this document, to transform formulæfrom a subset of the temporal logic into an equivalent monitors [37].

FIGURE 1.11: Alternative occurrence of events



(a) Alternative occurrence of a, b events (b) Property observer

Notice that a monitor only outputs one information: at each time moment t, it asserts whether the observed behavior has been true up to t or not. Hence, monitors can only express *safety* properties, and no liveness requirement can be translated into a monitor because of this lack of anticipation.

1.3.3 The Property Specification Language (PSL) standard

PSL [31] is the IEEE standard language allowing expression of behavioral requirements either as logic specifications or as operational specifications. It is a formal language developed by *Accellera*[1], which is an independent, non profit organization promoting modeling and verification standards for use by the electronics industry. PSL is heavily based on the proprietary language Sugar, developed by IBM and initially supported by their commercial formal verification tool RuleBase. Sugar is a syntactic layer over ordinary temporal logic, allowing shorthand expression of some common behaviors. These mechanisms were mere syntactic sugar, as they could always be translated into ordinary CTL. On the other hand, Sugar also supported operational specifications through a language heavily inspired from the CMU-SMV symbolic model checker's input language. The CMU-SMV language can express dynamic behaviors modeled as communicating synchronous Finite State Machines. Hence, PSL contains both linear-time and branching time temporal logic, and can express requirement specifications operationally.

Example 1.7. *A few properties written in PSL*

- *A Boolean expression:* $(start \vee go)$*, it means the system receives one of the two signals (start, go) or both at the instant* (t)*; All Boolean operators can be used in PSL expression like* $(\wedge, \vee, \neg, \rightarrow)$

[1] http://www.accelera.org

- *A PSL sequential expression:* $(req;\ ack;\ \neg req;\ \neg ack)$*; it means the signals req, ack appears in a certain sequence :* req *is asserted first, then* ack*, then* req *is de-asserted and then* ack *is de-asserted.*

- *A PSL property can be the combination of the two expressions together using the temporal operator like (always, next, until, before, ... etc):* $always\{start \vee go\} next(req;\ ack;\ \neg req;\ \neg ack)$ *which means always if (start) or (go) is asserted, it will be followed by the sequence of req, ack and their de-assertion.* $always\{request\}\ before\{acknowledge\}$ *which means the signal request appears before the signal acknowledge but not necessarily in the exact previous instant.*

The PSL standard is currently an input for commercial (formal) verification tools and has been successfully used within academic or industrial projects [38], [39], [40].

1.4 Verification of hardware embedded systems

Hardware designs are largely modeled using Hardware Description Languages (HDL). Correct behavior of the design needs to be verified. This amounts to establishing a relationship between the design itself and a requirement. This relationship expresses satisfaction and/or confidence in the design that has been produced. It can be established either semi-formally, or formally.

A *semi-formal verification* relies on an executable model of the design which is run for a given amount of hand-made scenarii and a given amount of time, and its outputs are inspected, more or less visually. The assessment is extremely partial, as simulation is only as exhaustive as the humans who wrote the scenarii.

More recently, since PSL has emerged, requirements can be *formalized*. The simulation can now be run together with and against executable models of PSL specifications. This can be achieved as PSL logic specifications can be translated into monitors:

- *safety requirements* can be directly translated into an equivalent monitor;

- *liveness requirements* monitors can only provide an approximate assessment. They are only able to assert that a desired configuration has been observed in the past, or not.

Formal verification requires a formal (mathematical) design model, and a formal specification. A translation process maps design language constructs to a formal model prior to the actual verification process.

Catching most design errors during the early phases of a design processes is vital. The next section presents a series of formal and semi-formal verification techniques which are frequently used in industrial projects.

1.4.1 Theorem proving

The theorem proving technique operates on the formal model of a design, which is handled as a set of axioms. The design needs to be translated into a set of mathematical definitions using first or higher order logic. The desired properties of the system are expressed as theorems. Theorem proving applies inference rules and induction to the system' formal model, attempting to prove the target theorem.

The advantages of the theorem proving technique are the following:

- it handles efficiently the size of the design's state space;
- it handles abstract data types, and induction, which makes it a powerful tool for either finite or infinite state systems; due to these qualities, it is a particularly effective tool for assessing the correction of arithmetic components;
- theorems are potentially more expressive than temporal logic.

Unfortunately, theorem provers are not user-friendly tools. As they can be applied in a general framework, with possibly models featuring an infinite number of states, decidability issues can occur. Besides, the proof process needs sometimes additional guidance from the designer, which is error-prone. It may also loop forever.

Some software tools have been developed to automate the process of theorem proving, in order to solve the problems in reasonable time and enhance the performance of the verification process (complexity, efficiency .. etc). The prototype verification system PVS [41], [42] is a platform supported by a specification language and verification tool based on theorem proving method. It has been successfully used to verify hardware designs [43], or a shutdown protocol of a nuclear system [44]. The Fischer's real-time mutual exclusion protocol and a railroad crossing controller [45] were also verified using PVS. Zenon [46] is another experimental automated verifying tool, invented and dedicated for Focal environment. It provides an execution OCaml code and a Coq certification code. Isabelle/HOL [47], HOL-TestGen [48] are classical theorem provers based on the higher order logic HOL. Although automated theorem proving has solved the problem of verification time, its main drawback is the need of user assistance to guide the proof. Besides, if a theorem proof fails, it is difficult to get diagnosis/debugging information.

1.4.2 Guided simulation

The guided simulation requires the formal specification of both the design requirements and simulation scenarii. The design is run *randomly* together with its formally specified requirements, according to the simulation scenarii.

Guided simulation is used to verify industrial systems. A bus arbiter controller was verified for mutual exclusion and conservativenes properties, using guided simulation method [49]. In [50], simulation essays were applied on the missile Titan IIIC digital flight control system in order to verify its correctness. According to the authors, the simulation scenarios were carefully chosen to cover the entire control system of the missile functionality. The author in [51], proposes a heuristic that avoids a blocking verification search by tracking multiple promising states and backing-off when getting stuck. An abstraction-guided simulation approach was proposed in [52] to combine the advantages of simulation and mathematical verification techniques, in order to find some hard to find states with a microprocessor checking. Author's contribution was building a Markov model of the studied system and apply simulation essays on it. They demonstrate by experimental results on microprocessors, that their approach is efficient in covering hard-to-reach states, for large and complex designs.

This technique can be used for medium or large sized designs, without any complexity issues. It provides acceptable coverage, and demonstrates efficiency in discovering design errors. It may however be argued that the verification is not exhaustive, and that no absolute correction guarantee can be offered.

1.4.3 Model checking

The model checking is another formal verification technique. It achieves an exhaustive exploration of the design's state space, in order to check the satisfaction of a requirement written in temporal logic. In case the design does not satisfy the requirement model checking tools provide a counterexample that illustrates how the requirement was violated.

The Model checking appeared to solve the problem of concurrent program verification, where concurrency errors are difficult to find since they do not reproduce in the program very often. At the beginning, proofs were conducted by hand using the Floyd-Hoare logic formalism. A system was proposed by Owicki and Gries [53] for reasoning about conditional critical regions.

The work of Pnueli [35], Owicki and Lamport [54] proposed the use of temporal logic for specifying concurrent programs. Although they still advocated hand constructed proofs, their work demonstrated convincingly that Temporal Logic was ideal for expressing concepts like mutual

exclusion, absence of deadlock, and absence of starvation. Clarke and Emerson [55] proposed an algorithm that automatically reasons about temporal properties of finite state system by exploring the state space.

K. McMillan in his PhD thesis develops the *symbolic model checking* technique, and the CMU-SMV tool [15] to verify the satisfaction of temporal properties in a modeled system. The tool is based on Binary Decision Diagram (BDD) structure instead of stat graph and uses a special input language to represent the models and the temporal logic for the specified properties. The symbolic method coupled to BDDs has shown outstanding performance figures, and most commercial formal verification tools exploit this principle.

Bounded model checking was introduced in 1999 [56] by A. Biere et al. The core of this strategy is to verify the satisfaction of a property of a system under fixed number of steps (k); one keeps increasing (k) as long as the property is satisfied and stops when the property is violated or when decide that k is enough to decide the satisfiability of the property. This method uses the state graph as data structure and can be applied to both safety and liveness properties, but it suffers from being always bounded, so one cannot validate a studied system except for limited number of states.

Edmund M. Clarke in [57] explains some advantages of model checking, we cite here some of them:

- fully automatic method: the user of a Model Checker does not need to construct a correctness proof. In principle, all that is necessary is for the user to enter a description of the circuit or program to be verified and the specification to be checked. The checking process is fully automatic;
- rigorous: when a specified property is not satisfied by the model studied, the model checker provides a counterexample illustrates where the property was violated, and this is one of the most important features of using model checking technique;
- powerful: employing temporal logic allows to consciously expressing complex properties, and it helps to reasoning concurrent systems while it is very hard to verify all possible cases manually.

Unfortunately, a serious problem of which all the model checkers suffer is combinational explosion. A system with n Boolean state variables has 2^n possible states. This blow-up happens during BDD construction/manipulation. Even though BDDs allow efficient manipulation of large sets of states, their theoretic complexity is still exponential in the number of Boolean variables handled. Thus it is sometimes impossible to explore the entire state space with limited resources of time and memory [58].

TABLE 1.1: comparison between different formal verification techniques

	Theorem proving	Guided simulation	Model checking
state space coverage	+	-	+
free software	+	-	+
counterexample	-	+	+
fully automatic verification	-	-	+
large state space	+	+	-
concurrency	+	+	+

Industrial and research efforts have concentrated on these performance issues. SAT-based techniques [59] have been developed, featuring considerable performance, and requiring much less memory. Nowadays commercial tools are able to combine several state traversal techniques, switching dynamically from one to the other. Hence, random guided exploration can narrow down the state space, getting "closer to the final solution"; then, an exhaustive BDD-based exploration can start, implemented by a parallel algorithm. Thus, model checking is still an important design tool which is able to uncover corner-case subtle design errors [60].

The following table illustrates a comparison between the theorem proving, guided simulation and model checking regarding some key features :

Regarding the 1.1 table we suggest in our contribution to use the model checking technique and take advantage of its features (fully automatic, full coverage of state space, counterexample providing, accept symbolic representation of a system), to find the design errors and rigorously get a trace when a property is violated by the original system specification.

1.5 Supervisor synthesis

1.5.1 Supervisory control

This work advocates the use of the supervisory control theory - also known as Ramadge and Wonham framework [61] - as a complementary embedded design tool. This theory has been developed on top of formal languages, more particularly the regular languages. Just like formal verification, this theory applies to *finite state/transition* models, directly derived from regular languages. Unlike verification, rather then checking whether a system satisfies a formally specified requirement, this control paradigm *enforces* requirements, when possible. To achieve this, a *supervisor* is automatically generated from the design at hand, also known as *the plant*, and the formally specified requirement. The supervisor acts as a new building block, running concurrently with the plant, and communicating with it: reading state information from the system and the environment, and feeding *control information* back to the system, in order to enforce

the desired requirements. Thus, a "closed-loop" control architecture is provided. Typically, a desired behavior is made either invariant (safety), or reachable. It is also possible to enforce unavoidability (liveness) in an approximate way, by solving this problem as an optimal path problem, such as proposed in [62]. In this work, safety is the only requirement that is considered. This limitation is mainly proposed in order to simplify the application of DCS to COTS-based designs. Extensions to reachability/liveness requirements are left as future developments.

The actual technique which performs the construction of a supervisor is called Discrete Controller Synthesis (DCS). In this work, safety is the only requirement that is considered. This limitation is mainly proposed in order to simplify the application of DCS to COTS-based designs.

The term "controller" either designates the supervisor, or a building block obtained from the supervisor. In this work, DCS refers to both the supervisor generation and the controller construction. This aspect is developed later in this chapter.

Two DCS variants exist according to the formal models upon which they have been built: event or sample-driven. Thus, the plant is either specified as a formal regular language, using exclusively states, events and transitions, or as a reactive program featuring inputs, outputs, state variables and some algorithmic structures implementing transition functions.

The event-driven DCS technique initially developed by Ramadge and Wonham [61] has been recently ported to symbolic BDD-based state traversal algorithms [63], [64]. The current work is based on sample-driven models, which is why the DCS technique used in the sequel is the one developed in [65].

1.5.2 Controllability in hardware systems

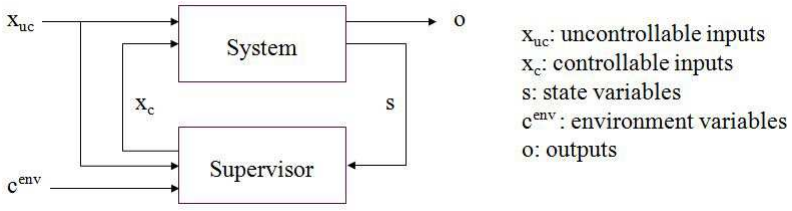
Prior to recalling the DCS basics, needed in the remaining of this work, it is important to clarify some terminology elements. The DCS control design technique is used in a hardware design context and hence, some keywords common to both automatic control design and hardware design become ambiguous. Even though they are conceptually close, they do not have the same significance.

In electronic hardware design, two types of building blocks are handled: (1) the *controller* and (2) the *datapath*, also known as the operative part. A datapath is a building block featuring a sequential dynamic behavior, typically arithmetic computations. The controller acts as a scheduler for the different arithmetic operations that are needed in order to implement a requirement. Hence, both the controller and the datapath are modeled using the same language, have the same underlying formal model, and most often are implemented in the same electronic chip.

The *high-level synthesis* [66] technique achieves automatic generation of both the controller and the datapath, according to a user-defined specification issued as an iterative algorithm. This context is close to the supervisory control theory. Two types of interactions exist: with the physical environment, through uncontrollable inputs (or events), and with the operative part, through controllable inputs, which are seen as actions taken by the controller. The conceptual distinction between these two kinds of inputs is obvious, and the partitioning too.

However, this case remains an exception. Many hardware designs do not feature a clear frontier between the control and operative parts. Besides, control functions are often complex operations, featuring concurrent behaviors. They are manually coded by design engineers, and due to their complexity this process is error-prone. In this work, such control functions are considered as candidates to control, in order to enforce a requirement. Unfortunately in this context, is is much more difficult to partition the input interface of a building block with respect to controlability. This distinction is unnatural for hardware design engineers: all available input variables are considered to be driven by the environment, and the design process does not anticipate the presence of a supervisor. However, in order to apply DCS, the input variable set needs to be partitioned into controllable and uncontrollable inputs, so that the control architecture presented in figure 1.12 can be used: $X = X_{uc} \cup X_c$. A method for achieving this partitioning is provided in Chapter 3.

FIGURE 1.12: Control architecture for hardware designs



1.5.3 Symbolic supervisor synthesis

The symbolic discrete controller synthesis method was first developed in [24] [65, 67]. The symbolic DCS algorithm makes invariant the set $\lambda^{-1}(spec)$ of states, satisfying the desired specification $spec$. To achieve this, it iteratevely prunes all the states from which an execution path can lead outside $\lambda^{-1}(spec)$ through uncontrollable input values. The result of this process is a set of states, called invariant under control $\mathcal{IUC}$. The calculation of the invariant under control iteratively calls a basic step: finding the list of controllable predecessors of a given set of states. This list can be obtained as follows.

Given a Boolean finite state machine $M = \langle s_0, X, S, \delta, PROP, \lambda \rangle$, and a set $E \subset \mathbb{B}^{|S|}$, the set of controllable predecessors of E is defined as:

$$CPRED(E,\delta) = \{\mathbf{s} \in \mathbb{B}^{|S|} \mid \forall \mathbf{x_u} \in \mathbb{B}^{|X_{uc}|}, \exists \mathbf{x_c} \in \mathbb{B}^{|X_c|}, \exists \mathbf{s'} \in \mathbb{B}^{|S|} : \mathbf{s'} = \delta(\mathbf{s}, \mathbf{x_{uc}}, \mathbf{x_c}, \mathbf{s}) \wedge \mathbf{s'} \in E\}$$

In other words, the state s is a controllable predecessor of a state $s' \in E$ iff for any uncontrollable value $\mathbf{x_{uc}}$, there exist a controllable value $\mathbf{x_c}$ such that the transition function δ leads to s'. Hence, it is always possible to go from state s to state s' by computing an adequate value for $\mathbf{x_c}$, according to s and $\mathbf{x_{uc}}$.

The DCS algorithm consists of a recursive application of $CPRED$ till a greatest fixed point is reached:

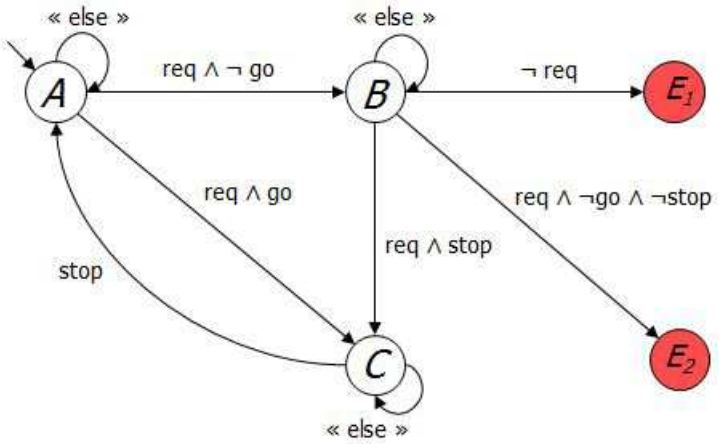
$$\begin{aligned} \mathcal{IUC}_0 &= \lambda^{-1}(spec) \\ \mathcal{IUC}_{k+1} &= \mathcal{IUC}_k \cap CPRED(\mathcal{IUC}_k, \delta) \end{aligned}$$

The greatest fixed point reached is the invariant under control $\mathcal{IUC}$ set. A control solution exists if this result is not the empty set. Besides, the initial state of the system must be included in the invariant under control $\mathcal{IUC}$. Under these conditions, the supervisor SUP is constructed from $\mathcal{IUC}$ as follows:

$$SUP = \{(\mathbf{s}, \mathbf{x_{uc}}, \mathbf{x_c}) \in \mathbb{B}^{|S|} \times \mathbb{B}^{|X_{uc}|} \times \mathbb{B}^{|X_c|} \mid \exists \mathbf{s'} \text{ s.t. } \mathbf{s'} = \delta(\mathbf{s}, \mathbf{x_{uc}}, \mathbf{x_c}) \wedge \mathbf{s'} \in \mathcal{IUC}\}$$

SUP is the set of all transitions of M leading to $\mathcal{IUC}$.

FIGURE 1.13: A 5-states design to be controlled using DCS



Example 1.8. *We consider the state-based design illustrated in the figure 1.13.*

Let $spec = G\neg(E_1 \vee E_2)$ *be the requirement which should be enforced on this model. Starting from the initial state* A*, the property can be broken in state* B*, uppon receiving the input* $req = 0$*.*

Supposing that go *is a controllable variable, and* $req, stop$ *are uncontrollable variables, there exist a winning strategy which assigns the value* 1 *to the variable* go *at the state* A*. Any other controllable variables are not enough to prevent the system from reaching the inadmissible state. By applying the computation algorithm of IUC we obtain :*

$$\begin{aligned}\mathcal{IUC}_0 &= \{A, B, C\}\\ \mathcal{IUC}_1 &= \{A, C\}\\ \mathcal{IUC}_2 &= \{A, C\}\\ \mathcal{IUC}_2 &= \mathcal{IUC}_1;\end{aligned}$$

A fixed point is reached and the final $\mathcal{IUC}$ *set is* $\{A, C\}$*.*

1.5.4 DCS for hardware designs

The supervisor provided by the symbolic DCS is a characteristic function:

$$\mathcal{SUP} : \mathbb{B}^{|S|} \times \mathbb{B}^{|X_{uc}|} \times \mathbb{B}^{|X_c|} \rightarrow \mathbb{B}$$

defined as:

$$\mathcal{SUP}(\mathbf{s}, \mathbf{x_{uc}}, \mathbf{x_c}) = 1 \text{ iff } (\mathbf{s}, \mathbf{x_{uc}}, \mathbf{x_c}) \in SUP$$

The actual control of M requires solving the equation

$$\mathcal{SUP}(\mathbf{s}, \mathbf{x_{uc}}, \mathbf{x_c}) = 1$$

continuously, for each reaction of M, and considering $\mathbf{X_c}$ as unknown variables. This is not directly implementable into hardware. We use the supervisor decomposition technique presented in [68] in order to obtain systematically the control architecture presented in Figure 1.12.

According to this decomposition technique, the characteristic equation representing the supervisor $\mathcal{SUP}$ is automatically decomposed into a vector $\hat{\mathcal{C}}$ of m Boolean functions, where m is the number of controllable variables.

$$\hat{\mathcal{C}} = \begin{pmatrix} f_1(\mathbf{s}, \mathbf{x_{uc}}, \mathbf{x_{c1}^{env}}, f_2, \ldots, f_m) \\ f_2(\mathbf{s}, \mathbf{x_{uc}}, \mathbf{x_{c2}^{env}}, f_3, \ldots, f_m) \\ \ldots\ldots \\ f_m(\mathbf{s}, \mathbf{x_{uc}}, \mathbf{x_{cm}^{env}}) \end{pmatrix}$$

The variables $\mathbf{x_c^{env}}$ hold the values that the environment "proposes" for the controllable variables $\mathbf{x_c}$. It is up to the controller $\hat{\mathcal{C}}$ to decide whether the environment proposal is accepted or not. In the sequel, the decomposed supervisor $\hat{\mathcal{C}}$ is referred to as the controller.

Example 1.9. *The supervisor decomposition for the previous example yields a controller containing one function which assigns values to the controllable input* go*:*

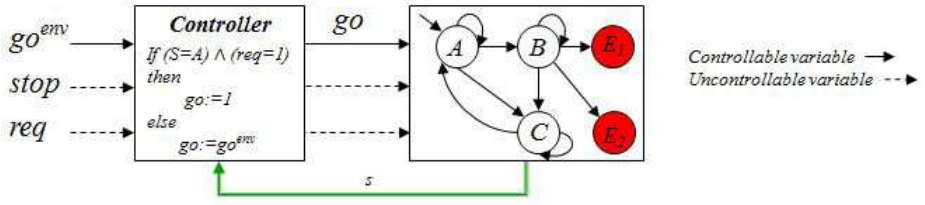
$$go = f(\mathbf{s}, req, go^{env})$$

where the expression of f *is :*

$$\begin{aligned} f(\mathbf{s}, req, go^{env}) = \\ & \textit{if } (\mathbf{s} = A) \wedge (req = 1) \textit{ then} \\ & \quad go := 1 \\ & \textit{else} \\ & \quad go := go^{env} \end{aligned}$$

Thus, if state A is active and the req *input is asserted then* go *is assigned to* 1*. Otherwise,* go *receives the value* go^{env} *assigned by the environment. The resulting controlled design is shown in Figure 1.14.*

FIGURE 1.14: A 5-states design assembled to the controller



Hence, the dynamics of the controlled design is presented by the following algorithm:

Algorithm 3: sample-driven execution of a controlled design

1: initialization: $i = 0$, $q^i = q_0$
2: **for** each clock tick *i* **do**
3: sample all uncontrollable variables $\mathbf{x_{uc}}$ **and** all desired controllable values $\mathbf{x_c^{env}}$
3: compute the controlled values $\mathbf{x_c} = \hat{\mathcal{C}}(\mathbf{s}, \mathbf{x_{uc}}, \mathbf{x_c^{env}})$
4: compute the next state: $s^{i+1} = \delta(\mathbf{s^i}, \mathbf{x_{uc}}, \mathbf{x_c})$
5: compute the outputs: $\mathbf{o^i} = \lambda(\mathbf{s^i}, \mathbf{x_{uc}}, \mathbf{x_c})$
6: **end for**

According to the dynamics described above, at each reaction, controllable inputs x_c are driven by the environment and controlled (modified) by the supervisor, according to the current state of the design and the satisfaction of the control specification.

Both M and the decomposed supervisor can be directly implemented on hardware targets. A concrete example is developed in Chapter 3 of this document.

1.6 COTS-based design

The term COTS is an abbreviation standing for Commercial Off The Shelf. A COTS is a generic, reusable, hardware or software component. Although, there is no component that can do everything and can work everywhere, COTS developers provide components with generic behavior, which can be used for different applications. A component is considered reusable COTS when it satisfies certain functional requirements when integrated in certain environments. The main advantage of using COTS in a design, is to economize time and production costs [69], and also to provide fault tolerance in circuits [70]. The companies use in-house COTS, fabricated by the company itself or out-house COTS, also called IP[2] components, that it buys from the existing market. Big industrial companies like military, avionic, transport companies, can also solicit smaller companies to fabricate special COTS suitable with their own interests and needs, especially if they do not own a development staff to do this mission.

1.6.1 COTS definitions

In the last decades, several definitions of COTS were given, starting from very general notion of re-usability going to a more precised characterization of reusable components. In 1997, [71], a COTS was defined as a black box, where no access to its behavior is available, nor fully access to its documentation is possible. In 1998 [72], COTS was considered as any pre-built

[2]Intellectual property

component, available for the public, which can be bought, leased or licensed, and integrated in a larger system in order to achieve certain mission. The federal acquisition regulation (FAR), in part 12, defines the commercial item as follows "An item is sold, leased, or licensed to the general public; offered by a vendor trying to profit from it; supported and evolved by the vendor who retains the intellectual property rights; available in multiple, identical copies; and used without modification of the internals." [73]. In 2002 [74], seven attributes were considered necessary to any component to be considered COTS, the attributes put frontiers to the vague definitions of COTS. (1) A COTS is generally a purchased component. (2) a license must be given to the customer. (3) the possible modification could be applied to the COTS are limited to parameterization. (4) the COTS should be stocked in a library. (5) medium size component. (6) the component must have an interface. (7) it must have vertical functionality, i.e; it must be used in many applications. A formal definition of COTS will be presented latter in this manuscript.

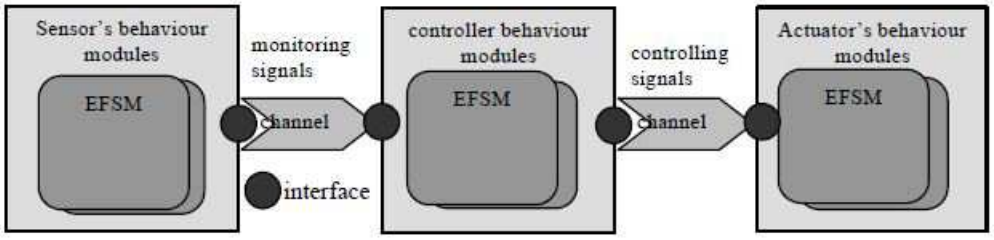
1.6.2 COTS integration in a design process, difficulties and solutions

The lack of formal specification of pre-built components behavior was an obstacle against using COTS in critical systems, thus, a formal definition of COTS interface was proposed in 1999 [75]. Formalizing contracts over the COTS interface isolates the system under construction of the COTS internal software and makes the interaction between the original system and the integrated COTS verified and controlled through the interface contracts.

In 2005 A formal definition of Component-based embedded system was proposed in [2]. A communication level specification of components, based on extended finite state machines (EFSM), was used. The studied embedded system is divided into three sorts of communicating blocks (1) sensors, (2) a controller and (3) actuators as shown in figure 1.15. Components are connected by communication points, i.e., inputs and outputs. sensor's outputs are input events for the controller, the controller's outputs are the inputs of the actuators. The system components are black boxes specified and tested using formal description technique, Estelle [76], [77]. The specification approach reuses an already existing methods to formally define embedded systems and profits of methods like automatic analysis, test data generation, validation and formal fault models, to enhance the components based development and follow the growing complexity of embedded system requirements especially safety and real-time properties.

Using COTS in a design, although it facilitates the cost of initializing a system design, it raises the risk of component maintenance, since the original developer may not be available for checking the components. If the vendor is not capable to keep up with the customer problems or product bugs, or even the shop is closed or stopped producing such COTS, negative consequences will affect the customer design and extra unexpected expenses will be needed for any system

FIGURE 1.15: Composition environment of embedded systems based on EFSMs [2]



upgrading. Thus, maintenance costs increases exponentially when integrating COTS inside the design, [78]. Thus, researchers worked to provide some heuristics in order to balance between the use of COTS in a design and the negative consequences of such use [78], [79].

1.6.3 Safety preserving formal COTS composition

Formal representation of components' composition has been evoked in the literature of software engineering of embedded systems. In [80], [81] a formal method for developing trustworthy real-time reactive systems is proposed. The system components are a set of communicating tasks modeled as communication finite state machines, and the system's trustworthy is evaluated by the system's preservation of some formal safety and security properties. The model checking is used to verify these properties over the studied system.

In [82] software components are modeled as communicating finite state machines and the safe communications are represented by LTL assertions. Informal lexical compatibility study is made over the components to conclude the system's safety represented by compatible components.

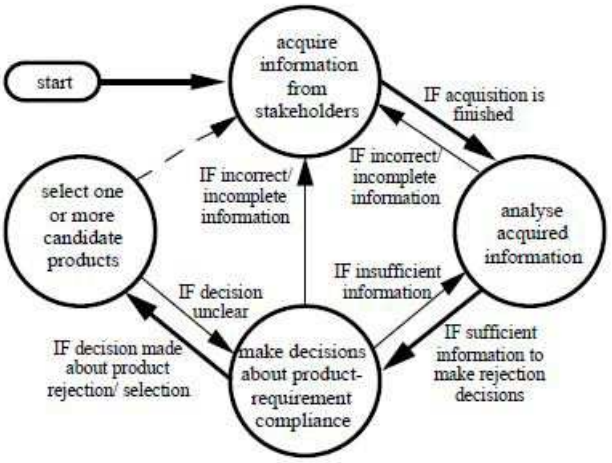
In [83] a monitoring data flow service is proposed to supervise the communication between some tasks modeled as FSMs. The system is composed by a synchronous product of the tasks FSMs, and the service can detect any unexpected communication over the system and signalize it as an error. Monitors are automatically generated using a particular tool named (*Enforcer*).

1.6.4 Safety in component-based development

In [84], safe COTS is defined as "a COTS which poses no threat or only a reduced threat in accordance with the nature of its use and which is acceptable in view of maintaining a high level of protection for the health and safety of persons".

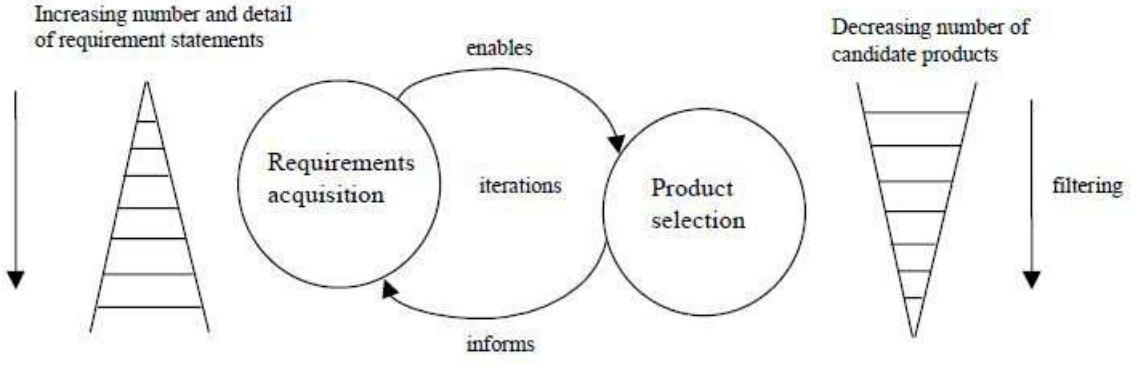
In 1999, PORE (Procurement-Oriented Requirements Engineering) [3], a guiding coherent technique for component-based requirements engineering was proposed. it provides the designing team with instructions to choose COTS components. It focuses on (1) the collection of the maximum information and requirements of the system under designing. (2) the selection of various candidate COTS exist in the market. Then (3) mapping the system requirements to the selected COTS to evaluate the components candidates. and finally (4) rejecting the components that satisfies the requirements the least, and keeping the components which are the best candidates to be used in the system design, as illustrated in figure 1.16.

FIGURE 1.16: High-level generic processes for PORE method [3]



The process of COTS evaluation is iterative one, i.e, requirements acquisition enables product selection and product selection informs requirements acquisition. As the number and detail of requirements increases, the number of candidate products decreases, as shown in figure 1.17

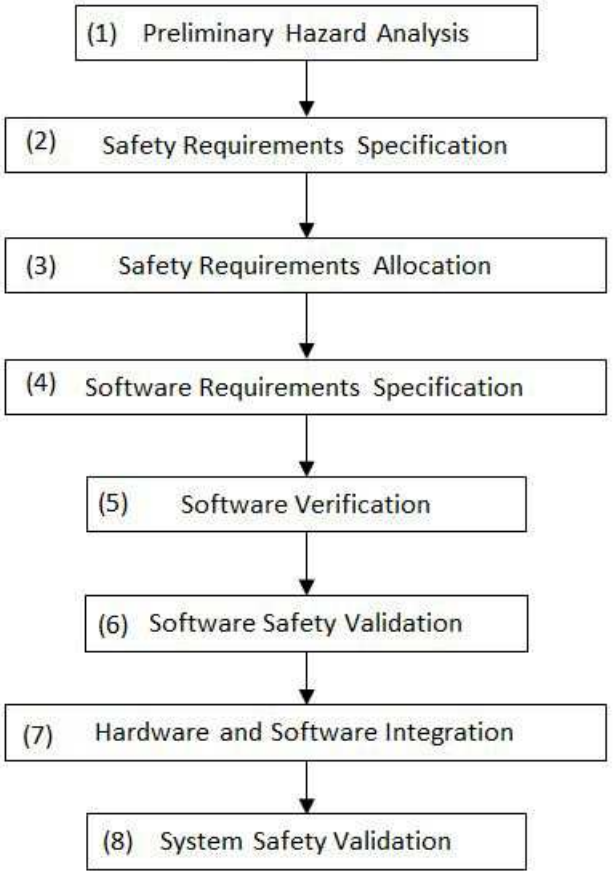
FIGURE 1.17: Overview of the PORE's iterative process [3]



In 2003, a component based development method [85] was proposed. Basing on analytic study of the efficiency of using COTS in system's development and the obstacles faced by such choice, the CARE method, shown in figure 1.18, comes up with evaluation, verification and validation steps, over the hardware and software parts, of the target system. The method contains eight

successive steps. It starts by a *Hazard analysis* and risk assessment on the conceptual system, in order to delineate hazards uncovered by the analysis. The second step is a *Safety requirements specification*, in this step safety verification is performed to ensure that system's specification reflects the safety requirements. The third step of the method is *safety requirements allocation*, in this step safety requirements are allocated to hardware and software system's parts. In step four, *Software requirements specification*, the software requirement specification is examined to ensure its inclusion of all safety requirements dictated by the former step. At step five, *Software verification*, a verification is performed at each phase of software development to ensure that the safety requirements are respected by the system specification. At the sixth step, *Software safety validation*, the software part of the system is validated if the verification step successfully passed and the safety requirements were fulfilled. In step seven, *Hardware and software integration*, both system parts are integrated and a verification is performed to ensure that the hardware part of the system which depends on the software part correctly operates. The eighth step, *Systems safety validation*, gives a final validation and assurance that the designed system is verified to fulfill the safety requirements specified for it and the system is ready to be used or supplied to the market.

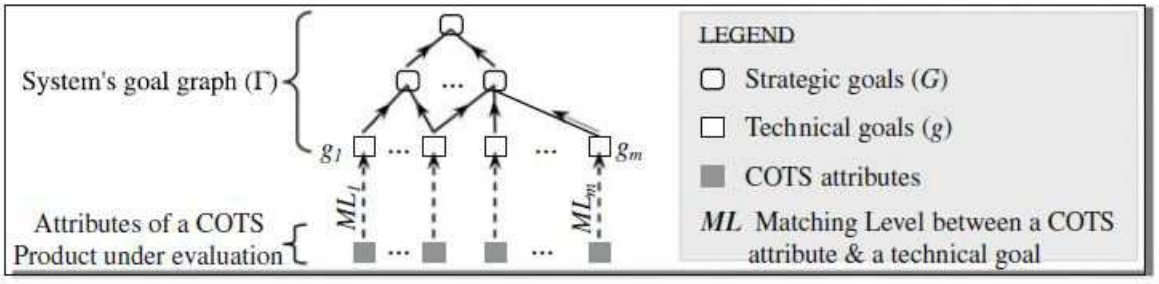
FIGURE 1.18: System safety V and V for COTS based systems



In 2004 a COTS aware requirements engineering technique, CARE [86], inspired by the PORE method was introduced. The proposal directs the designer to bridge between the native system and the requirements added by the foreign components. It raises some issues about the optimization of COTS using, like (1) determining the requirements which satisfies the final client. (2) matching the client requirements to the COTS capabilities. (3) selecting the best COTS candidate if different components exist in the market. (4) deciding whether building a system over COTS component is efficient or it is worthy to construct the system components from scratch. (5) studying the impact of COTS integration over the global system.

In 2007, a formal method, MiHOS [87] [4],[88], to evaluate the matching between a system requirements and COTS product was proposed. The goal of the method is to provide the best COTS candidates to a design. The method starts by analyzing strategic goals until reaching technical requirements. The strategic goals are the global behavior required by the final system, which cannot be performed by a single component ex. a secure communication between services. The technical requirement, is a COTS technical attribute which can achieve the strategic requirement, ex. supporting SSL communication protocol. The method depends on giving weights to the system requirements and evaluating which are the COTS that match the most these requirements. The method suggests five levels of matching, "(1) Zero match, when the COTS product fails to satisfy the requirements. (2) partial match, indicates partial satisfaction of requirements. (3) surplus, indicates that the COTS has extra functionality, not needed in the design. (4) overmatch, which indicates that the COTS functionality exhibits more capabilities than required. (5) equivalence which means, the COTS contribute to achieve the goal of the requirement, but it does not match the technical requirement itself." A hierarchical definition of a COTS evaluation with MiHOS method is illustrated in figure 1.19.

FIGURE 1.19: Hierarchical definition for COTS evaluation criteria [4]



A notion of trustworthy assembly of COTS is proposed in [89]. COTS were considered as black boxes and propose to solicit the UML method to represent the studied components; The work proposes to use the composite structure diagrams in order to express the architecture of components and their interfaces. They use the class diagrams, sequence diagrams and protocol state machines to describe the behavior of each component. The problem discussed is the interoperability of COTS through their interfaces. When assembling COTS, compatible interfaces are

needed, and since the COTS may be built in different companies, they probably do not share the same syntax languages, a mediator is necessary to organize the exchanges between assembled COTS. Trustworthy by verification, of hardware and software COTS based systems, was proposed in [90], where the model checking method is used to verifying if the satisfaction of some formal properties. The method validates a trusted system, if abstract models expressed in xUML or Verilog, of the real components satisfy properties written in a unified property specification language developed specially for the proposed method.

From a safety point of view, the above mentioned methods are quite robust, but, some points are missed. (1) The methods do not tackle the cases where the safety norms are not respected. (2) They do not suggest implementation tools. (3) They are manual tools and strongly depend on the designer competence in requirement engineering. (4) The last trust worthy method [90] is limited in application because it necessities a designer who masters well its language of property specification, otherwise, the model checking results cannot be trusted.

In our work, we fill those gaps by providing a complete method for safe design based on COTS. Our method is not an extension of the above ones, but, they are conceptually comparable to each other, as they both concern the safety of system based on COTS.

1.7 Conclusion

In this chapter we have presented an introduction about the hardware embedded systems, then the two modeling methods of the discrete behavior of these systems (event-driven and Sample-driven modeling). We have shown that the event-driven modeling is an abstract method and the Sample-driven modeling is more suitable to represent the hardware circuits, since it is closer to its material nature.

We have mentioned also some assisting design method which help the designer verify his design before implementing it like: (1) the theorem proving technique, since it is manual method, it depends strongly about the experience of the verifier and its mathematical competences. It is convenient for verifying systems of certain sizes. The ATP, as it does not provide a counterexample, it is useful only to confirm that the studied specification does not satisfy the studied property. No further benefits can be reached of this method. (2) the guided simulation is suitable method for verifying certain precise behavior in an industrial large design, but, it cannot provide a complete results about the full comportment of the studied system. (3) the model checking, is much useful for verifying medium to large systems, since it is fully automatic tool, its results are trusted, as long as it does not explode. When it provides a counterexample it may inspire the designer where he needs to correct the error. All those formal verification methods, whether

manual or automatic are limited to only finding design errors. No matter what technique the designer used, he is still obliged to correct the detected errors manually.

We have illustrated the discrete controller synthesis method, and its capability to synthesize certain behavior to discrete event systems. This method is still used only in the research domain, and we propose to integrate it in the application domain, take advantage of its services and figure out its shortcomings.

We have approached the evolution of the COTS notion through time, then the safety principles in using COTS components in system designs. The safety studied focuses basically on the matching of the components selected and the existing system, it depends on analytic studies of the system requirements and the components' behavior. If the matching is satisfied the COTS is taken for the design, otherwise it is rejected and another COTS is searched. In our work we advocate the possibility of automatically correcting mismatching COTS, through correct by construction code generation.

The following chapter presents a safe COTS-based design method for hardware embedded systems.

Chapter 2

The COTS-based design method

2.1 Introduction

The former chapter provides an overview of the state of the art in the domain of designing critical hardware systems; (1) modeling the behavior of hardware systems, (2) the formal verification of hardware systems and and its utility in finding design errors (3) the discrete synthesis tool and its capacity to automatically generating a correcting component named controller which once combined to the original system, it makes the resulting assembly satisfy safety properties, (4) the COTS-based design of embedded systems and the safety notion in COTS-based systems.

This chapter presents a novel COTS-based safe design method. Its novelty comes from the synergy between the formal verification and the Discrete Controller Synthesis techniques for building correct COTS-based designs. While formal verification is a mature technique in industry, DCS has never been previously in an industrial hardware design project. Besides the specific methodological aspects related to COTS design, the key issues in applying DCS to hardware designs are explained and solved.

The COTS components considered here are described by a behavioral model built by their original designer. The actual user has no knowledge about their implementation details, as sometimes the design code provided is not even readable. The user side perception of a COTS is a documented black box. The requirement specifications are formally expressed PSL language. These formal specifications are either given by the COTS provider, or built a posteriori by the COTS user, from the textual documentation.

The method proposed here encompasses all stages, from the initial specification to the target implementation prior to hardware synthesis. It is illustrated through a non-trivial yet pedagogic example: the generalized buffer system abbreviated as *GenBuf* [91].

The rest of the chapter is organized as follows : section 2.2, defines the notion of COTS, as used throughout this work. Basic reasoning steps in COTS-based design are presented in section 2.3, followed by the articulation with the Discrete Controller Synthesis technique.

DCS application to hardware designs needs a user-specified partition of the input signals set, in order to designate controllable inputs. As explained previously, this step is unnatural to hardware design engineers. A technique is developed in order to determine a set of candidates to controlability, automatically providing designers with a set of controllable candidates. These elements are aggregated within a global design method. Section 2.4 presents the example of a COTS-based hardware design. The application of the method developed here is illustrated on this example. We have presented this work in [92], [93]

2.2 Building COTS-based control-command systems

2.2.1 Stand-alone COTS

In this work, the notion of COTS is mainly related to reusability. This means that a COTS is a mature building block, implementing a non-trivial behavior, for which a certain amount of design and verification efforts have been spent. This is why a COTS is not only a component having a behavior, but is also documented: this documentation states both how the COTS should be used, and what it achieves. In practice, these elements are provided informally, as text. Even though this approach is familiar to design engineers, we highlight the need for formalizing them and associating them to the COTS behavior. Building new complex functions out of COTS calls for formal tools in order to achieve a safe design. COTS are usually structured into libraries, when developed and reused by the same team.

A stand-alone control-command COTS is the basic building block considered in this work.

It is characterized conventionally by four elements [94] :

1. COTS interface I
2. a set of preconditions A
3. a set of post-conditions G
4. COTS functional behavior M.

The set of post-conditions are satisfied by the COTS behavior if and only if

A stand-alone COTS is atomic: no structural decomposition into other elementary COTS is possible.

The COTS interface designates its set of inputs and output variables. The COTS preconditions A is a set of requirements expressed on the COTS' environment.

The need for environment assumptions. COTS are logical building blocks which are intended to be assembled in order to build the final control-command system driving an operative part, as shown in figure 2.1. Hence, the environment of a COTS is rarely physical, but most often logical, made of other COTS. Complex interactions may occur between COTS through communication. In order to handle this complexity within a design team where COTS components can be developed separately, a contract-based approach is generally used [95]. Consider the COTS C and C', developed separately by two designers. They each implement a specific function, and they interact with each other through inputs and outputs.

At the moment of designing C, the designer reasons as follows: the design at hand assign outputs according to the values of the inputs and possibly an internal state. It often happens that some (sequences of) inputs only make sense in some given configurations. There are three possible choices:

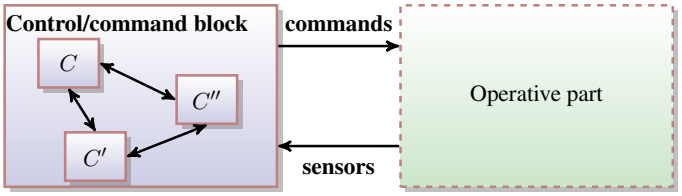
- an input value is unexpected. It should be processed and this should result in an error code;
- an input value or sequence comes at the wrong moment. Either answer with an error code, or memorize it till it can be processed.

  Either possibilities require adding additional, often very complex code, making sure that inputs are just as expected. Testing every possible situation is time-costly and error-prone.
- assume that the input values cannot be erroneous. Document this assumption.

The designer of C' follows the same reasoning. Hence, either both designers should add extra heavy code for testing the validity of the inputs, or they both assume that invalid inputs cannot occur. This is also known as "assume-guarantee" reasoning: when designing C, it is assumed that C' behaves correctly. The designer of C' should guarantee this correction. This approach is very cost effective even though this reasoning is not always valid: typically, if the correction of C depends on the correction of C' and vice-versa, the assume-guarantee reasoning cannot be applied. This issue is illustrated later in this document.

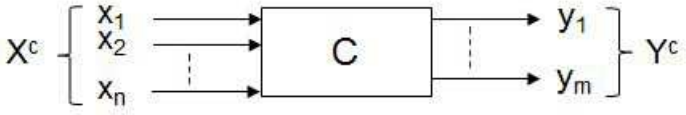
This work mainly addresses designs made of interconnected COTS, which is why the method we propose needs to reason about the environment behavior of a given COTS (or COTS assembly).

FIGURE 2.1: COTS-based control command system made of interacting blocks



The construction of the functional behavior formal model M is not mentioned in this document, as it is generally systematically and automatically extracted from the design code (VHDL or other). This is why, by an abuse of language, pieces of code are referred to as models.

FIGURE 2.2: Interface of a COTS represented by X^c, Y^c



Definition 2.1 (Stand-alone COTS)**.** We define a stand-alone COTS component C as a 4-tuple :

$$C = (I^c,\ M^c,\ A^c,\ G^c) \tag{2.1}$$

We denote X^c the list of COTS' inputs, $X^c = \{x_1^c,\ x_2^c,\ \ldots\ x_n^c\}$. Y^c the list of COTS outputs, $Y^c = \{y_1^c,\ y_2^c,\ \ldots\ y_m^c\}$. Thus, the component interface is :

$$I^c = X^c \cup Y^c \tag{2.2}$$

as shown in figure 2.2.

The COTS preconditions A^c describe the expected behavior of the environment of C. They are expressed logically and/or operationally.

$$A^c = \{\Phi_a^C, \mathbf{M_a^C}\} \tag{2.3}$$

where

- Φ_a^C is a set of logical specifications written in PSL, upon variables of I^C;
- $\mathbf{M_a^C}$ is a set of operational specifications. They can be either monitors, expressed over variables of I^C, or random FSM models, reading Y^C and specifying desired values for X^C.

The COTS post-conditions G^c is a similar set of formally specified assertions:

$$G^c = \{\Phi_g^C, \mathbf{M_g^C}\} \tag{2.4}$$

where Φ_g^C is a set of PSL formulæand $\mathbf{M_g^C}$ is a set of operational specifications. The post-condition set expresses *behaviors guaranteed to hold* on C, provided that the assumptions hold.

The COTS's behavior, given with the COTS documentation, is modeled by a finite state machine: $M^C = \langle s_0, X, S, \delta, PROP, \lambda_B \rangle$, where:

- $X = X^c$;
- $PROP = Y^c$;
- $s_0, S, \delta, \lambda_B$ are defined exactly as in Chapter 1.

The COTS behavioral model is usually automatically extracted from the design code as mentioned before.

The relationship between A^C, G^C and M^C is expressed as follows:

$$M^C, \langle a_1^C \dots a_k^C \rangle \models g^C \tag{2.5}$$

The FSM model M^C satisfies the requirement g^C provided that all the assumptions $a_1^C \dots a_k^C \in A^C$ are satisfied by the environment of M^C. This dependency $\models$ between assumptions and guarantees does not necessarily mean a logical implication. It happens that hardware designers express environment assumptions either as *sufficient* conditions or as *necessary* conditions, or sometimes both. Most often, they are perceived as necessary conditions. The difficulty of distinguishing between these two situations comes from the sequential complexity of the COTS' behavior. It would be theoretically very interesting to automatically "extract" the set of *necessary* environment assumptions a COTS needs in order to operate correctly; however, such an operation is not possible to the best of our knowledge.

The satisfaction relation $\models$ has the following signification:

- if g^C is a logical specification, then the satisfaction is established by model checking M^C against g^C by assuming $a_1^C \dots a_k^C$;
- if g^C is an operational specification, then there are two possibilities:

- g^C is a monitor. It has the form $(I, M_{g^C}, \emptyset, \phi^{Err})$, where Err is the single output of M_{g^C} and asserted true if the behavior observed by M_{g^C} violates the desired requirement. The predicate ϕ^{Err} is of the form "$always\ (\neg Err)$". In this case, $M^C||M_{g^C}$ is model checked against ϕ^{Err};
- g^C is a "golden" , or reference FSM model. M^C should be sequentially equivalent to g^C. In other words, M^C should have exactly the same behavior as g^C . This can also be established formally, by model checking.

It is important to note that the sets A^C and G^C contain two kinds of requirements: those expressed by the designers, which are considered as the most important, but also, in a large number, those left implicit; their exhaustive expression is both practically impossible and useless. However, some assumptions and/or guarantees which have been left implicit can simply characterize design errors, left uncovered. Indeed, uncovering a bug requires asking the "right question", and translate it logically or operationally. This totally depends on the verification engineer's skill, and thus, the chances of success are random. Hence, a COTS is not a "perfect" component; it can (and probably) have hidden bugs, and building designs out of existing COTS elements also amounts to mixing unwanted behaviors from each building block.

In the context of this work, COTS behaviors are usually specified using the VHDL hardware description language. For the specific needs of the Ferrocots project, the ControlBuild [96], [97] graphical design environment has been used, as it provides designers with a more user-friendly design framework. ControlBuild models are assumed, without loss of generality, to have a synchronous semantics, compliant to the models defined in Chapter 1 of this document. Moreover, all ControlBuild designs are translatable into synchronous VHDL, compliant with the IEEE standard "RTL synthesis subset level 1" [98]. This compliance provides an informal guarantee that a synchronous FSM semantics can be associated to any ControlBuild design.

Example 2.1. *Preconditions of a stand-alone COTS*

Let C be a component with the set $X^c = \{x_1^c, x_2^c\}$, $Y^c = \{y_1^c, y_2^c\}$. The COTS inputs work in mutual exclusion mechanism and the environment change at each instant. The inputs are represented by an interface preconditions set $A^c = \{a_1^c, a_2^c\}$ where: $(a_1^c : x_1^c \neq x_2^c)$, and $(a_2^c : x_1^c at\ the\ instant\ (t) = \neg x_1^c$ at the instant $(t-1))$.

In PSL these preconditions can be written as:

- a_1^c = $always\ ((x_1^c \wedge \neg x_2^c) \vee (\neg x_1^c \wedge x_2^c))$
- a_2^c = $always(\ (x_1^c \rightarrow\ X(\neg x_1^c)) \wedge (\neg x_1^c \rightarrow\ X(x_1^c)))$

a_1^c, a_2^c are illustrated by the truth table 2.1.

TABLE 2.1: Truth table illustrating the preconditions of a dynamic environment with a mutual exclusion behavior

x_1^c	next x_1^c	not next x_1^c	$x_1^c \rightarrow$ not next x_1^c	x_1^c	$\neg x_1^c$	next x_1^c	$\neg x_1^c \rightarrow$ next x_1^c	a_2^c	x_2^c	a_1^c
0	0	1	1	0	1	0	0	0	0	0
0	1	0	1	0	1	1	1	1	1	1
1	0	1	1	1	0	0	1	1	0	1
1	1	0	0	1	0	1	1	0	1	0

Example 2.2. *Post-conditions of a stand-alone COTS*

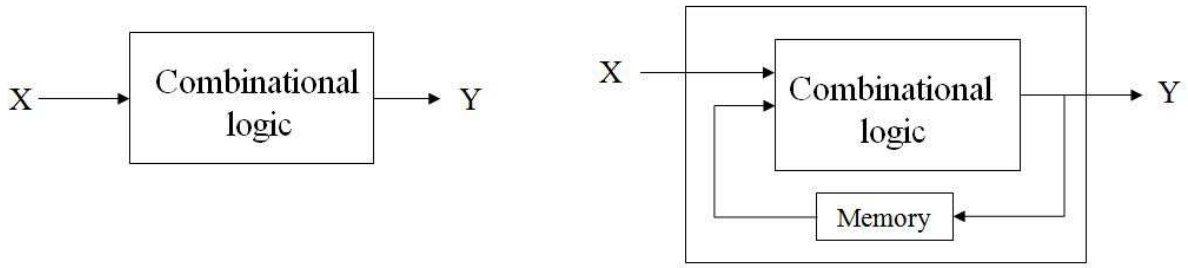
Let (C) *be a component with the set* $X^c = \{x_1^c, x_2^c\}$, $Y^c = \{y_1^c, y_2^c\}$ *with a precondition list* $A^c = \{a_1^c : x_1^c \neq x_2^c\}$, *a post condition that must be satisfied is* $G^c = \{g_1^c : y_1^c = y_2^c\}$.

These pre/post-conditions can be expressed in PSL as follows :

- $a_1^c = always\ ((x_1^c \wedge \neg x_2^c) \vee (\neg x_1^c \wedge x_2^c))$
- $g_1^c = always\ (y_1^c \wedge y_2^c)$

Some COTS have a combinational behavior, i.e, the outputs of the COTS depend only on the values of its inputs, As shown in figure 2.3 (a). Other components have a sequential behavior, i.e, the output of the COTS depends on the its internal state and the current values of inputs. This is shown in figure 2.3 (b).

FIGURE 2.3: COTS behavior



(a) Combinational COTS behavior (b) Sequential COTS behavior

2.2.2 COTS assembly

A COTS assembly is the act of combining components together, in order to produce a new behavior, which cannot be produced by any COTS, considered alone. The approach for assembling COTS proposed in this document relies on a rather classical conception of the component-based design: components have a behavior, a set of guarantees and and a set of assumptions, also known sometimes as contracts. The guarantees hold provided that their corresponding contracts

are fulfilled. The act of composing two components amounts to finding a matching between the guarantees of one component and the contracts of the other. An interaction is thus established, and some other guarantees may find themselves satisfied by simple component composition. These mechanisms are quite classical in software programming. This design approach is quite natural, and has proven its robustness. Most recent developments we are aware of, conducted within the AFSEC[1] research community, define contracts for reactive timed models [99] and the corresponding compositional reasoning with contracts. Other relevant contributions close to our work have been presented in 1.6.3; they are related to *software* COTS-based design, where COTS are communicating tasks, modeled by finite state machines and composed together by a synchronous product. The correctness of a COTS-based design is established by model checking.

In this work, we extend the classical component-based reasoning mechanisms, by identifying specific issues, characteristic for the component-based design of hardware embedded systems. We solve these issues both technically, by establishing a synergy between the formal verification and DCS techniques, and methodologically, by proposing a safe design method.

When assembling hardware COTSs, the assembly interface becomes a partial part of all the interfaces of assembled COTS, some input and output signals of a COTS become internal signals. We denote (X^{intern}, Y^{intern}) for internal inputs and internal outputs respectively. We denote I^{intern} to the union $X^{intern} \cup Y^{intern}$.

Prior to achieving a COTS assembly, two kinds of functional behaviors related to either standalone COTS or COTS assemblies need to be distinguished:

- the post-conditions, or guarantees, denoted by the letter G, which are originally satisfied by the COTS and cited in its documentation;
- the required properties, denoted by the letter P, which are user-specified, according to the specific use of an instantiated COTS, or COTS assembly. Their validity remains to be established.

Structural assessment of COTS interconnections. The act of interconnecting COTS relies on a one-to-one conceptual pattern. No matter how complex the target function to be achieved, designers usually consider COTS to be assembled in pairs.

Several compositional architectures are possible according to the interconnections that are considered between the assembled COTS, as shown in figure 2.4. In each of these situations, new

[1] *Approches Formelles des Systèmes Embarqués Communicants*: french research community on the topic of Formal methods for Embedded Systems Design

interactions are established between pairs of components, and questions arise about the safety of these interactions.

In case (1) COTSs are completely separated architecturally, even though once assembled global assembly properties may occur, for example: $y_1 \neq y_2$.

In cases (2,3) COTSs are semi-serially assembled, the difference between the two cases is that in (2) the output $y_2^{c_2}$ becomes an internal signal after the assembly, whereas in case (3) $y_2^{c_2}$ remains an output signal. In both cases, C_2 becomes part of the environment of C_1, and the compliance of C_2 with the assumption set of C_1 needs to be assessed.

In case (4) COTS C_1 and C_2 share the same environment behaviors, and it is important to assess whether the assumption sets of C_1 and C_2 are not contradictory.

In cases (5, 6, 8, 9) COTSs are cyclically assembled. In these assembly architectures each component represents a partial environment of the other. The assessment of these compositions is more delicate, as explained in example 2.3.

Case (7) combines the cases (2) and (4). On the one hand, C_1 and C_2 share the same environment. On the other hand, C_2 itself is part of the environment of C_1. The assessment of this configuration should ensure that the assumptions are not contradictory, and that the guarantees of C_2 do not contradict the assumptions of C_1.

Hence, it is generally required that in order to ensure a safe behavior of a COTS assembly, the architecture should be taken in consideration and post-conditions must not break preconditions. This is detailed in the sequel of this section.

Definition 2.2 (COTS assembly)**.** Let $\mathcal{LIB}$ be a COTS library modeled as a set of N individual COTS. Let $\mathcal{ASM} \subseteq \mathcal{LIB}$ be a set of COTS from COTS library, modeled as a set of $K \leq N$ COTS. We define a *compositional product* operation denoted $\|^C$, and we define the COTS assembly as follows:

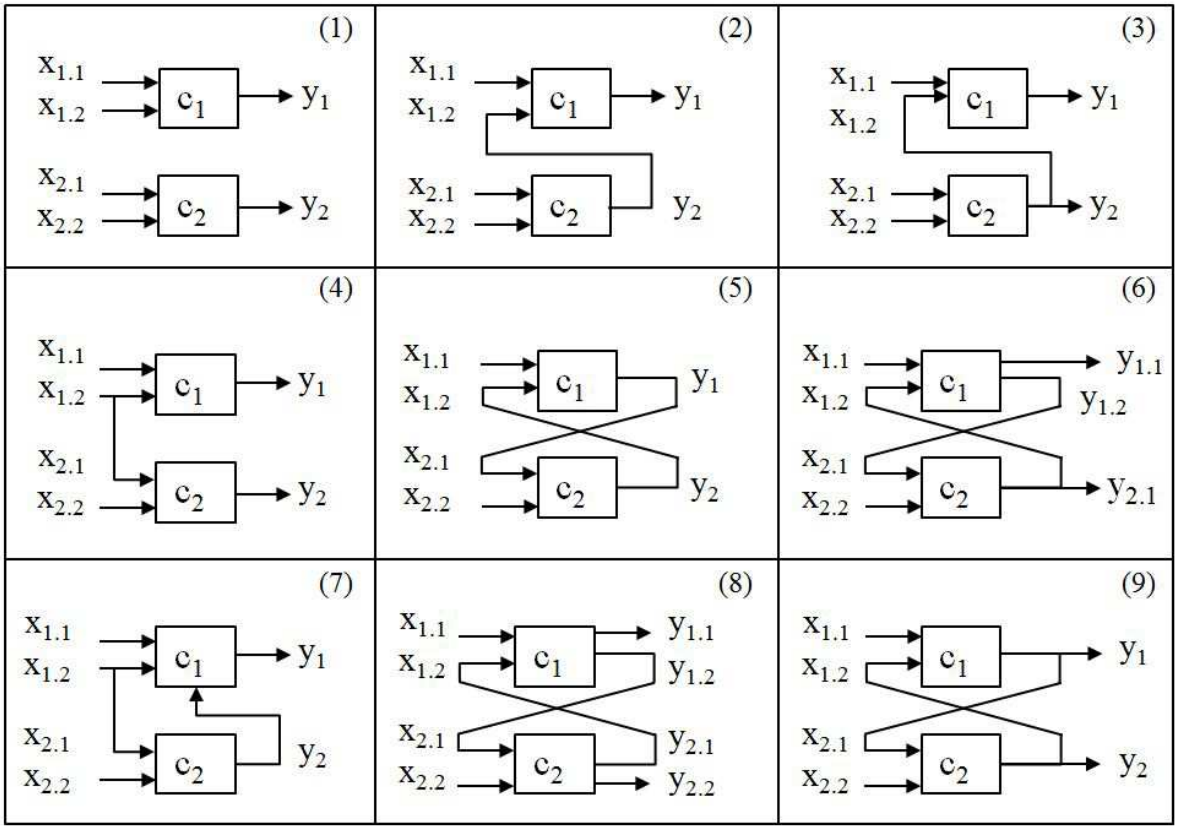
$$C_1 \|^C C_2 \|^C \ ... \|^C C_K = (I^{asm}, M^{asm}, A^{asm}, G^{asm}) \tag{2.6}$$

The assembly interface I^{asm} is defined as follows :

$$\begin{aligned} I^{asm} &= I^{c_1} \cup I^{c_2} \ ... \cup I^{c_K} \setminus I^{intern} \\ &= (X^{c_1} \cup X^{c_2} \ ... \cup X^{c_K} \cup Y^{c_1} \cup Y^{c_2} \ ... \cup Y^{c_K}) \setminus (X^{intern} \cup Y^{intern}) \end{aligned} \tag{2.7}$$

The assembly behavior model is the synchronous composition of the COTS s models, as the system evolves regarding any event, whether it is common or uncommon, which happens to any

FIGURE 2.4: Different COTS assembly architectures



COTS defined in 1.17.

$$M^{asm} = M^{c_1} \parallel M^{c_2} \ldots \parallel M^{c_K} \tag{2.8}$$

Let A^{asm} be the set of environment assumptions that must still hold after the assembly. We have $A^{asm} \subseteq A^{c_1} \cup A^{c_2} \ldots \cup A^{c_k}$.

Let G^{asm} be the set of post-conditions that hold after the assembly. We have $G^{asm} \subseteq G^{c_1} \cup G^{c_2} \ldots . \cup G^{c_k}$.

The exact definition of A^{asm} and G^{asm} depends on the interconnections achieved during the construction of M^{asm}. This is detailed below, in the section 2.2.3.

Assembling together interacting COTS is a delicate operation with few correction guarantees. Indeed, pairs of COTS are rarely designed with the specific purpose of working and *interacting* together. Thus, it is likely that they do not provide each other with the expected environment, needed to guarantee their respective post-conditions. In other words, some post-conditions or guarantees can be broken simply by COTS assembly. The following section gives the necessary reasoning mechanisms for building correct COTS assemblies.

2.2.3 Compositional reasoning

As mentioned earlier, building complex functions out of individual COTS cannot by achieved only structurally, by mere input/output interconnections. A behavioral assessment is required, in order to establish the conceptual correction of the COTS assembly. After the assembly operation, the set of post-conditions (or guarantees) must be preserved.

According to the design patterns identified in figure 2.4, the following situations are required to be detected. Some of them are unwanted and must be fixed.

Incompatibility between environment assumptions. This situation is highlighted in figure 2.4, pattern (4). According to this pattern, C_1 and C_2 share a common environment behavior through the input sets $x_{1.2}$ and $x_{2.1}$. Each COTS comes with its environment assumptions A_1 and A_2. In this situation, it must be ensured that:

$$\nexists(a_1, a_2) \in A_1 \times A_2 \text{ such that } (a_1 \wedge a_2) = false$$

This can only be achieved when the assumptions are explicit. Usually this compatibility between environment assumption can be established manually. If a pair of contradictory assumptions exists, than it is considered that the COTS assembly in question is not valid. Otherwise, we can define:

$$A^{asm} = A_1 \cup A_2 \text{ and}$$
$$G^{asm} = G_1 \cup G_2$$

Hence, the local guarantee sets are conserved on the assembly.

Contradiction between guarantees and environment assumptions This situation occurs when a component's outputs drive the inputs of another component. This is highlighted in figure 2.4, patterns (2,3,7). In these situations, it must be ensured that there is no guarantee in C_2 which contradicts an assumption in C_1:

$$\nexists(a_1, g_1, a_2, g_2) \in A_1 \times G_1 \times A_2 \times G_2 \text{ such that}$$
$$(M^{C_1}, a_1 \models g_1) \wedge (M^{C_2}, a_2 \models g_2) \wedge (g_2 \rightarrow \neg a_1)$$

If such a situation occurs, we say that a guarantee $g_1 \in G_1$ is broken by the COTS assembly. In practice, design engineers need to establish a matching between assumptions and guarantees. To do so, they compare visually the assume/guarantee sets of the COTS they are about to assembly in order to find matching assume/guarantee pairs. An assumption matches a guarantee if it is implied by that guarantee. On the other hand, designers also need to find the set of assumptions broken by the assembly. These operations are mostly conducted manually. This happens because the COTS requirements are not systematically formalized. Besides, to the best of our knowledge, there are no automatic matching tools to handle this operation. Thus, this check is achieved manually, on the explicit sets of assumptions and guarantees. If a guarantee is broken, then the COTS assembly is invalid.

Compatibility between guarantees and environment assumptions When the above check is successful, the COTS assembly preserves all the guarantees. This situation is the most desired in COTS-based design: after the COTS assembly operation, the guarantees do not need to be reassessed. Besides, all the environment assumptions required by C_1 and satisfied by C_2 do not need to be assumed anymore. In this situation we can define:

$$A^{asm} = (A_1 \cup A_2) \setminus \{a_1 \in A_1 | \exists g_2 \in G_2 : g_2 \rightarrow a_1\} \text{ and}$$
$$G^{asm} = G_1 \cup G_2$$

Cyclic reasoning. This situation is highlighted in figure 2.4, patterns (5,6,8,9). Such that:

$$M^{C_1}, a^{C_1} \models g^{C_1} \text{ and}$$
$$M^{C_2}, a^{C_2} \models g^{C_2} \text{ and}$$
$$g^{C_1} \rightarrow a^{C_2} \text{ and}$$
$$g^{C_2} \rightarrow a^{C_1}$$

In other words, C_1 provides guarantees g_1 that enable assumptions a_2 of C_2 and viceversa. This circular dependency cannot be exploited in a COTS assembly operation, because it can generally not be concluded that $M^{C_1} || M^{C_2} \models g^{C_1} \wedge g^{C_2}$, as explained in example 2.3 below. In this situation, we advocate the elimination of the assume/guarantees involved in the circular dependencies:

$$A^{asm} = (A_1 \cup A_2) \setminus \{a_1 \in A_1 | \exists g_2 \in G_2 : g_2 \rightarrow a_1\} \setminus \{a_2 \in A_2 | \exists g_1 \in G_1 : g_1 \rightarrow a_2\} \text{ and}$$
$$G^{asm} = \emptyset$$

By default, such an assembly does not preserve any guarantees, hence we consider that the set of guarantees local to each COTS needs to be re-established on the COTS assembly. Such circular reasoning situations are to be handled separately, through formal verification after the assembly. Circular reasoning can only be applied in some particular situations [100], [101]. In such situations, the actual proof requires an important amount of user assistance, and we consider that the technicality of such proofs is both error prone and beyond the skills of design engineers.

Hence, we consider that these design patterns require a re-evaluation of the set $G_1 \cup G_2$.

Example 2.3. *Circular reasoning*

Let C_1 and C_2 be defined by the following:

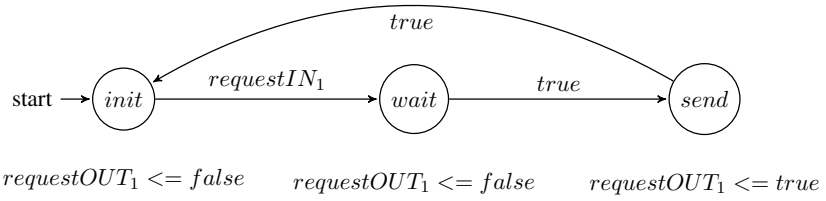
- $X_i = \{requestIN_i\}$;
- $Y_i = \{requestOUT_i\}$;
- M^{C_i} *behaviorally defined as shown in figure 2.5;*
- $A_i = \{always\ eventually\ (requestIN_i)\}$;
- $G_i = \{always\ eventually\ (requestOUT_i)\}$

for $i = 1..2$.

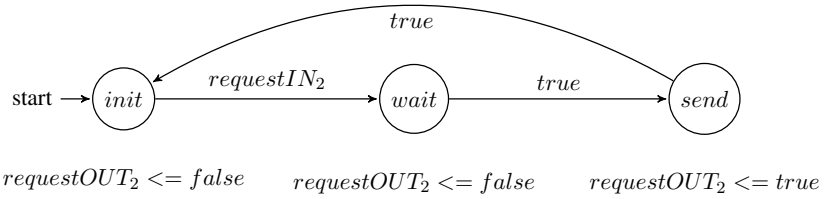
According to the behavioral models provided in figure 2.5, it can be visually determined that if $requestIN_i$ occurs infinitely often, as stated in A_i, then $requestOUT_i$ shall also occur infinitely often, as stated in G_i.

Consider the particular assembly of C_1 and C_2 where $requestIN_1$ is assigned the value of $requestOUT_2$ and vice versa, as shown in figure 2.6(c). By applying the circular reasoning, we would conclude that both assertions from G_1 and G_2 hold unconditionally after the assembly operation. However, even though they were locally true, after the assembly none remains true.

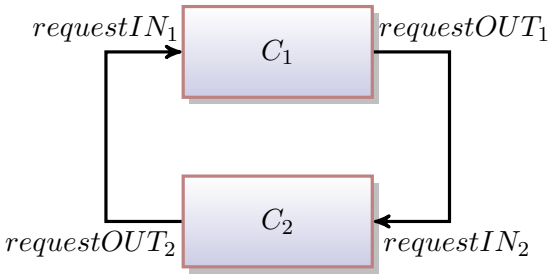
FIGURE 2.5: Circular reasoning for a COTS assembly



(a) C_1 behavioral model



(b) C_2 behavioral model



(c) Assembly of C_1 and C_2

Indeed, it can be seen that when both C_1 and C_2 are in the initial state, each shall wait forever for $requestIN_i$ to become true. Hence we have:

$$M^{C_1}, A_1 \models G_1 \textit{ and}$$
$$M^{C_2}, A_2 \models G_2 \textit{ but}$$
$$\neg G_1 \wedge \neg G_2$$

According to these compositional reasoning patterns, we can define the notion of *safe COTS assembly*.

Definition 2.3 (Safe COTS assembly)**.** An assembly of COTS is safe if and only if

- it conserves the satisfaction of all post-conditions:

$$\forall (a, g) \in A^{asm} \times G^{asm} \text{ either of the following must be true :}$$

$$1: \quad g \rightarrow a \qquad \text{i.e. } g \text{ enables } a, \text{ or}$$

$$2: \quad \neg(g \rightarrow \neg a) \qquad \text{i.e. } g \text{ and } a \text{ can hold together, or}$$

$$3: \quad a \qquad \text{i.e. } a \text{ holds regardless of } g$$

- it does not contain cyclic assumption/guarantee dependencies:

$$\nexists a_1, a_2 \in A^{asm}, \nexists g_1, g_2 \in G^{asm} : a_1 \rightarrow g_1 \rightarrow a_2 \rightarrow g_2 \rightarrow a_1$$

Definition 2.4 (Complex COTS)**.** Any safe COTS assembly is also a COTS, which we call a *complex COTS*. Its elements are defined exactly as above.

Example 2.4. *Precondition and post-condition for a COTS assembly*

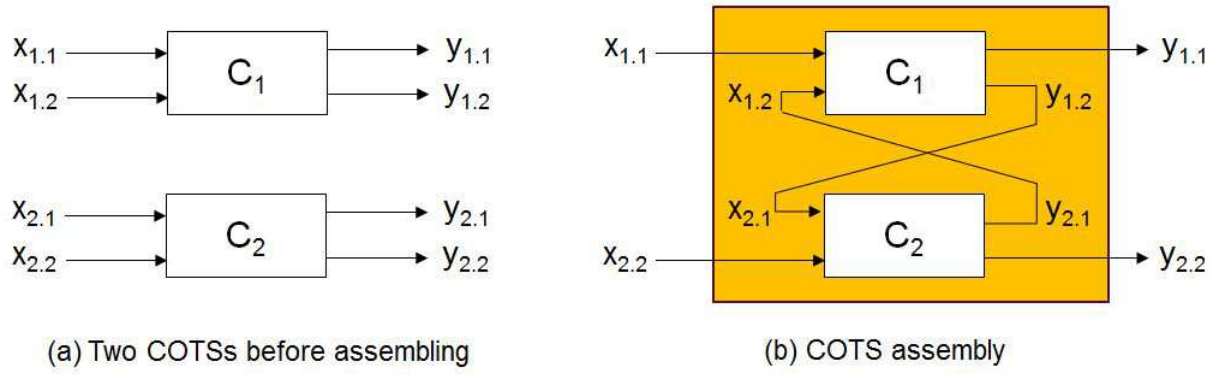
Let C_1, C_2 *be two components where* $I^{c_1} = \{x_{1.1}, x_{2.1}, y_{1.1}, y_{1.2}\}$ *and* $I^{c_2} = \{x_{2.1}, x_{2.2}, y_{2.1}, y_{2.2}\}$. *The components are assembled as illustrated in figure 2.6.*

Let $A^{c_1} = \emptyset$, $A^{c_2} = \emptyset$. *Let* $G^{c_1} = y_{1.1} = y_{1.2}$, $G^{c_2} = y_{2.1} \neq y_{2.2}$.

The assembly input set $X^{asm} = \{x_{1.1}, x_{2.2}\}$, *the output set* $Y^{asm} = \{y_{1.1}, y_{2.2}\}$. *The internal communication signals' set* $I^{intern} = \{x_{1.2}, x_{2.1}, y_{1.2}, y_{2.1}\}$.

A proposed global precondition assumption is $A^{asm} : x_{1.1} \neq x_{2.2}$. *The global post-condition of the assembly is* $G^{asm} = G^{c_1} \cup G^{c_2}$.

FIGURE 2.6: COTS interface before and after assembly



2.2.4 Adding context-specific requirements

The reuse of a set of COTS amounts to an instatiation of their generic behavior into a specific application context, featuring specific constraints and requirements. At the moment of reusing a (set of) COTS, the designer may ask the following questions:

- are there supplementary functional requirements P, specific to the application context, that should hold?
- what shall be the environment for this COTS (assembly) ?
- does this environment allow the satisfaction of P ?

These questions may actually be asked for an individual COTS, or for a COTS assembly. We designate such additional requirements as *local* or *global* properties. These requirements are associated to COTS at the moment they are used.

Definition 2.5 (Local property)**.** Let C be a COTS. We call a *local property* C, denoted P^C_{loc} an additional, undocumented requirement added by the designer, with respect to the usage intended for the COTS at the moment it is instantiated. Such a property is usually expressed logically, using PSL. A local property is not automatically guaranteed. Its satisfaction remains to be established.

$$P^C_{loc} = \{\Phi^{Ci}_p, \mathbf{M^{Ci}_p}\} \tag{2.9}$$

Definition 2.6 (Safe stand-alone COTS)**.** A safe stand-alone component is a COTS that satisfies both its local properties and its post-conditions before being assembled to other COTS, and after assembling. In other words, the satisfaction of the local properties is conserved through COTS composition.

The COTS C_i is safe with respect to $P^{c_i}_{loc}$ iff :

$$\begin{array}{rcl} Before\ assembly & : & M^{c_i}, A^{c_i} \models P^{c_i}_{loc} \cup G^{c_i} \\ After\ assembly & : & M^{asm}, A^{asm} \models P^{c_i}_{loc} \cup G^{c_i} \end{array} \tag{2.10}$$

Definition 2.7 (Global property)**.** The global property P^{asm} of a COTS assembly is a new user-defined requirement which is related to the global behavior of the assembly, and concerns the interaction between different components of the assembly.

Just like a local property, a global property is not guaranteed. Its satisfaction remains to be established.

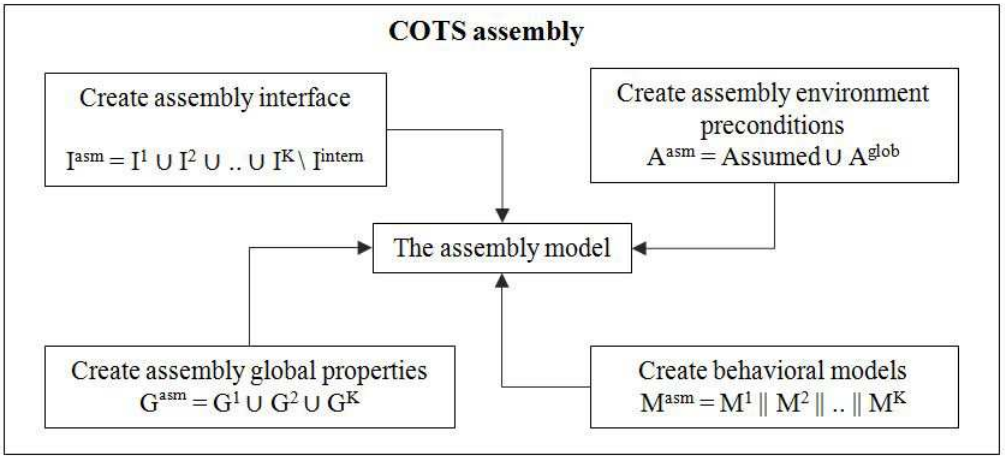
$$P^{asm} = \{\Phi_p^{asm}, \mathbf{M_p^{asm}}\} \tag{2.11}$$

As stated above, many assume/guarantee expressions related to a COTS can be implicit. The application of the compositional reasoning mechanisms requires explicit assumption/guarantee expressions, which is quite unrealistic. When these pre-requisites are not met, the truth of the guarantees and local/global properties must be re-established after a COTS assembly. This is performed automatically, typically using model checking. When this operation succeeds, all the guarantees that could not be established by compositionnal reasoning may be established by model checking. Hence, when successful, the result of this operation is a posteriori a *safe COTS assembly*. According to the results obtained, the following possibilities apply:

- all properties are verified. The COTS assembly is safe;
- some required liveness properties are false. This situation requires a redesign, since we are not aware of a technique able to enforce liveness properties over a pre-built component;
- some required safety local/global properties are false. These can be enforced through DCS, which produces a correcting controller which forces the COTS or COTS assembly to satisfy these requirements.

A global COTS assembly framework is shown in figure 2.7

FIGURE 2.7: COTS assembly framework



2.2.5 Design errors

We define a local error as the situation where a component does not satisfy a local property. We distinguish two types of local errors, (1); local stand-alone error, (2) local error cause by

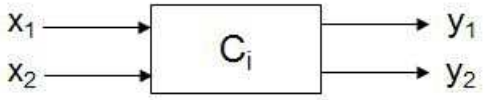
the assembly. The **local stand-alone error** occurs when we verify a stand-alone component against an additional property and we get negative results. Whereas the **local error caused by the assembly**, occurs when an assembly of components breaks a local property of a component whether this property is an original one (was satisfied before the assembly), or an additional one. Assembling COTS to each other, and exchanging values between each other, can entail errors, especially when some components send values which are not expected by the other components. This can negatively affect the functionality of the assembly components.

Example 2.5. *Local stand-alone error*

Let C_i be a stand-alone component as shown in figure where the set of preconditions $A^i = \{(a_1^i : x_1 = x_2)\}$ and a set of post-conditions $G^i = (g^i : y_1 = TRUE)$.

A local property $P_{loc}^{c_i} : always(y_1 = y_2)$, is to be verified over the component, for the input vector $(x_1, x_2) = (0, 0)$, the output vector result was $(1, 0)$, thus, it can be concluded that the property P_{loc} is not satisfied by the COTS behavior.

FIGURE 2.8: Local stand-alone error



Example 2.6. *: Local error caused by the assembly*

Let C_i, C_j be two components where $C_i = \{I^i, M^i, A^i, G^i\}$, $C_j = \{I^j, M^j, A^j, G^j\}$, they are assembled together as shown in figure 2.9, and $I^{intern} \subset I^{c_i}$, The component C_i has a precondition set $A^i = \{a_1^i : (x_{i.2} = TRUE)\}$. and a post-conditions set $G^i = \{g_1^i : (y_{i.1} = y_{i.2})\}$.

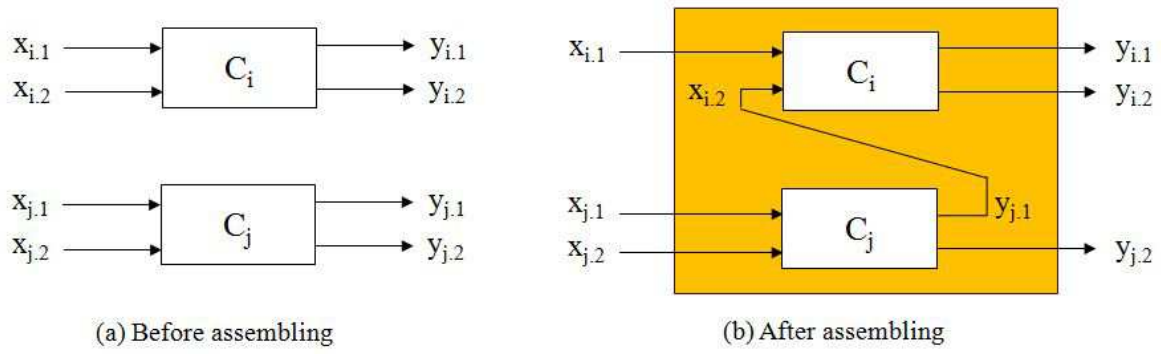
The component C_j has an empty preconditions' set $A^j = \{\emptyset\}$, and a post-condition set $G^j = \{g_1^j : (y_{j.1} = x_{j.1})\}$.

After assembling $C_i \parallel^C C_j$, for the vector of inputs $(x_{i.1}, x_{j.1}, x_{j.2})$ = $(1, 0, 0)$, the value of $y_{j.1} = 0$, it is transmitted to C_i, this violates the precondition of C_i, thus, the local post-condition g_1^i could be broken.

2.2.6 Global design error

When COTS are assembled together, some behaviors (global properties P^{asm}) related to the global behavior of the components, may be defined to be required. If the assembly behavior does not guarantee this global behavior, we call this situation a global error. A bad synchronization of

FIGURE 2.9: Local assembly error



components can be the cause of such error. The violation of any of the original post conditions G^{asm} of any of the assembled COTS is also a global error.

Let $C_i = \{I^i, M^i, A^i, G^i\}$, $C_j = \{I^j, M^j, A^j, G^j\}$ be two assembled components as shown in figure 2.9, given $I^{intern} \subset I^{c_i}$. Given a global post-condition G^{asm}. A global design error is denoted as : $M^{C_i} \parallel M^{C_j} \not\models G^{asm} \cup A^{asm}$.

We refer the violation of a global property to one among four possible reasons:

- The property expression P^{asm} is incorrect. i.e, it does not reflect the expected meaning of the verbal requirements. Although this situation can happen in practice, we assume that the requirement are correctly expressed;
- The COTS behavior is incorrect. But this contradicts the nature of COTS which are assumed to be correct reusable components;
- The assembly behavior cannot realize P^{asm};
- The environment assumptions of the assembly are incorrectly formalized.

Example 2.7. *global error*

Let C_i, C_j be the same two components illustrated in figure 2.9. let $A^{asm} = \{a_1^{asm} : (x_{i.1} = x_{j.1})\}$ be a precondition of the assembly. Let $P^{asm} : y_{i.1} = y_{j.2}$ be a global property over the assembly.

For a given vector of inputs $(x_{i.1}, x_{j.1}, x_{j.2}) = (1, 1, 0)$, satisfying the preconditions A^{asm}, the output vector observed is $(y_{i.1}, y_{i.2}, y_{j.2}) = (1, 1, 0)$. These values break the global property P^{asm}. This is named a global error. In this example it is supposed that the reason beneath this error is an incorrect formalization of the assumption set A^{asm}.

The design method we propose finds these kinds of errors and provides an automatic solution, if the error is caused by bad synchronization between components or incorrect definition of preconditions.

2.2.7 Enforcing local/global properties

2.2.7.1 Computing the controllable input set

The controllability problem for a hardware design has been highlighted in the previous chapter. Such designs are not explicitly built to interact with a DCS generated controller. Thus, enforcing a property by DCS requires to choose a set of controllable variables, among the input set of the design at hand.

We propose a systematic procedure allowing to construct, for a given COTS-based design, a collection of input variables which are candidates to be controlled. This procedure relies on several inputs: the designer's knowledge on the one hand, and the processing of the model checking results on the other hand. Typically, the designer is able to distinguish between the following situations:

- input variables driven by physical sensors with hard reactive processing constraints. The processing of such information should follow the rhythm of the environment. DCS should consider such inputs as uncontrollable;
- input variables driven by physical sensors with interactive processing constraints. When processing such inputs, it can be assumed that the physical environment can wait for a variable but reasonable amount of time. Typically, human users accept interactive processing constraints. Such inputs can be controlled by DCS;
- inputs carrying data. Such inputs should be considered as uncontrollable.

By exploiting the counterexample provided by the model checking tool, the designer can construct a list of candidate controllable signals. The counterexample is a simulation trace showing the sequence of values of the input/state/output variables leading to the violation of a property. The input variables displayed by a counter-example are clearly involved in this violation. It can be attempted to prevent this situation, by adequately controlling these inputs.

The question whether these variables are the only ones involved in the violation of the property is more delicate. For a given verification problem, a counterexample is rarely unique. Indeed, several paths may exist, leading to the same property violation but involving differrent input variables. Hence, in order to determine exhaustively the set of controllable candidates, we would

need to process several counterexamples for the same verification problem. Such a feature is not implemented by the model checking tools we are aware of. Hence, the unique counterexample obtained is processed in order to obtain a list of controllable candidates.

To construct the list of the controllable input variables X_c, the designer follows the procedure given below:

Construction the list of controllable signals X_c

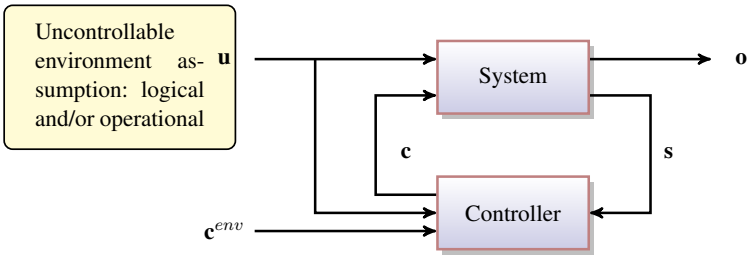
1. X_c initial = the complete list of signals provided by the counterexample.
2. remove from X_c the uncontrollable signals by following the rules:
3. **Begin removing:**
4. Every input variable intended to be driven by a sensor with hard reactive constraints must be eliminated from the X_c list;
5. Every input variable representing data must be eliminated from the X_c list;
6. Every input variable representing an alert or alarm information must be eliminated;
7. Every state variable must be eliminated;
8. Keep all the remaining signals and construct the list of controllable signals X_c.
9. **End removing**.

2.2.7.2 Environment-aware DCS

In the context of logic COTS-based design, environment assumptions are of great importance. Most likely, the environment of a given COTS consists of a collection of other COTS, and no direct connection exist to the "physical world", made of sensors and of actuators. In this situation, unlike a physical environment, the environment of a COTS must feature a precise behavior so that the COTS at hand can fulfill its function. Hence, most guarantees rely on assumptions.

For these reasons, enforcing a property P on a COTS or an assembly may also require the specification of an environment assumption for that COTS. This usually happens because the straightforward DCS application gives no control solution: the "plant", consisting of a COTS assembly, cannot be controlled in order to enforce P, as shown in figure 2.10.

FIGURE 2.10: Control architecture for hardware designs



This unfortunately makes the DCS step as tedious as formal verification. Indeed, for a given verification problem, the verification engineer spends a lot of time in finding the *proper environment assumptions*, needed to establish the proof of a requirement. Hence, the verification process is actually cyclic and often requires several runs for the same requirement.

Of course, it is worth attempting to enforce the target property P without modeling any environment assumptions, but if no DCS solution exists, then adding manually environment assumptions can be a cheaper solution than a COTS redesign.

The straightforward DCS technique we rely on is unable to take into account environment assumptions. A variant of the DCS algorithm is proposed in this section; it is able to take into account environment assumptions concerning the uncontrollable variables. The control solution, if it exists, assumes that the uncontrollable inputs can only feature the behaviors specified by the environment assumptions.

2.2.7.3 Environment modeling

Let C be a COTS and P be a local or global property to be enforced on C under the assumption $a \in A^C$: the objective is to establish $M^C, a \models P$. Both a and P are safety properties. For the specific needs of DCS, they must be translated into equivalent monitors [102]. This is straightforward, as recalled in Chapter 1.

Let $M_a = \langle X_a, S_a, s_{0a}, \delta_a, \lambda_a \rangle$ be the FSM model of the monitor recognizing a and $M_P = \langle X_P, S_P, s_{0P}, \delta_P, \lambda_P \rangle$ the model of the monitor recognizing P. The following properties are true:

$X_a \subseteq X^C_{uc}$ i.e. X_a is a subset of the uncontrollable variable set of C
$X_P \subseteq X^C \cup Y^C$
$\lambda_a \Leftrightarrow a$ i.e. λ_a is true iff a is true
$\lambda_P \Leftrightarrow P$ i.e. λ_P is true iff P is true

According to theses transformations of logical assumptions into operationally specified assumptions, the DCS problem to be solved becomes:

$$M_a||M^C||M_P \models always(\lambda_P), \text{ such that } always(\lambda_a) \tag{2.12}$$

This amounts to making invariant the set λ_P^{-1} on the model $M_a||M^C||M_P$, by considering only uncontrollable values which keep the set λ_a^{-1} invariant.

2.2.7.4 The environment aware DCS algorithm

The supervisor construction relies on a similar fixed point computation as in DCS. However, a supplementary constraint needs to be integrated in the DCS basic step $CPRED$: the controllable predecessors are computed under the assumption that the monitor M_a representing the environment assumption stays within the λ_a^{-1} set of states.

$$CPRED^{env}(E, \delta, \lambda_a^{-1}) = \{\mathbf{s} \in \mathbb{B}^{|S|} \mid \forall \mathbf{x_{uc}} \in \mathbb{B}^{|X_{uc}|}, , \exists \mathbf{x_c} \in \mathbb{B}^{|X_c|}, \exists \mathbf{s}' \in \mathbb{B}^{|S|} : \quad (2.13)$$

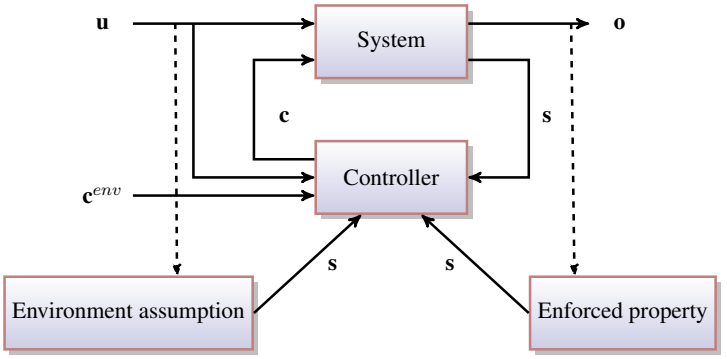
$$\mathbf{s}' = \delta(\mathbf{s}, \mathbf{x_{uc}}, \mathbf{x_c}) \wedge \lambda_a^{-1}(\mathbf{s}') \rightarrow \mathbf{s}' \in E\} \quad (2.14)$$

The environment aware discrete controller synthesis (EDCS) performs the same greatest fixed point computation as DCS, except that $CPRED^{env}$ is used instead $CPRED$. Notice that if no environment assumption is specified, $\lambda_a^{-1} = 1$.

The resulting controller is a Boolean function vector, as previously explained in 1.5.4.

The control architecture obtained is represented in Figure 2.11. Notice that EDCS operates on the composition between several models: the original design, the uncontrollable environment assumption, and the safety requirement to be enforced (since EDCS cannot enforce liveness requirements). These become part of the corrected design. Besides, the controller needs to observe their internal state.

FIGURE 2.11: Control architecture for hardware designs integrating environment assumptions



2.2.7.5 Applying EDCS to COTS-based designs

Let C be a COTS, or a COTS assembly and P be a requirement expressed as a safety property. The following situations are possible:

- P is a guarantee broken through COTS composition;
- P is a local or global property previously disproven by model checking.

The EDCS application on M^C, and P, possibly integrating an environment assumption A attempts to enforce the validity of P on M^C. The resulting correcting controller $\hat{\mathcal{C}}$ is composed by input/output interconnection with the original behavioral model M^C as well as the operational expression of the environment assumptions M_a. Since the controller $\hat{\mathcal{C}}$ is a Boolean function it can be represented by a finite state machine with a unique state. The result is guaranteed to satisfy the target property P, provided that the output λ_a of M_a is always true:

$$M_a||M^C||\hat{\mathcal{C}}, always(\lambda_a) \models P$$

Specific terminology for a EDCS-corrected COTS: Glue and Patch controllers. From the designer's point of view, we distinguish two situations:

- application of EDCS to enforce a local property on a stand-alone COTS. This often amounts to correcting a bug, discovered in that COTS. The resulting correcting controller is referred to as a *patch* for C.
- application of EDCS to enforce a global property on a COTS assembly. The resulting controller is referred to as a *glue* controller.

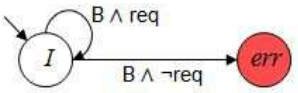
If all the broken guarantees can be enforced for a given COTS assembly, the result of this operation may be considered a posteriori as a *safe COTS assembly* and thus C may become a complex COTS. Additionally, if a local/global property is enforced on a safe COTS assembly, the result may also be considered as a safe COTS assembly. However, this decision cannot rely exclusively on the EDCS result: the act of enforcing a requirement may break the satisfaction of another requirement. In order to make this decision reliable, EDCS must be used in synergy with formal verification. This aspect is handled methodologically, and is detailed in section 2.3.

Example 2.8. *We apply the EDCS method to the system in example 1.8 shown in figure 2.13*

We remind that M is a model of 5 states machine. The state variable $s \in A, B, C, E_1, E_2$ The controllable input variable is go, the uncontrollable variables are $req, stop$. The safety property to be synthesized over the system is : $prop : never(E_1 \vee E_2)$. An environment assumption requires that once the system is at the sate B the variable req is activated. This assumption is represented as monitor FSM illustrated in figure 2.12 and formalized in PSL as follows:

$$prop : always(B \rightarrow req)$$

FIGURE 2.12: Environment monitor FSM B–> req



The environment assumption protects the system of reaching the error state E_1 and allows EDCS to generate a controller which manipulates the controllable variable go in order to prevent the system of reaching the error state E_2.

The controller generated can be expressed as follows: If $(s = B)$ then $go := 1$ Else $go := go^{env}$

The EDCS preserves an invariant under control set $\mathcal{IUC}$ = $\{A, B, C\}$ which is more permissive than the one generated by DCS.

The controller is assembled to the original system as illustrated in figure 2.14

FIGURE 2.13: A 5-states design to be corrected using DCS

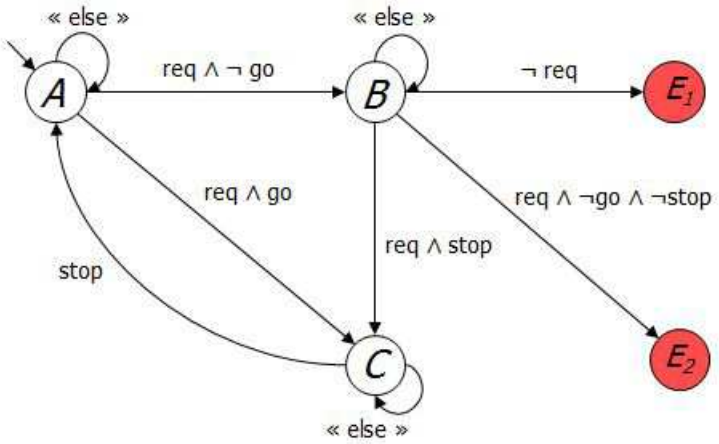
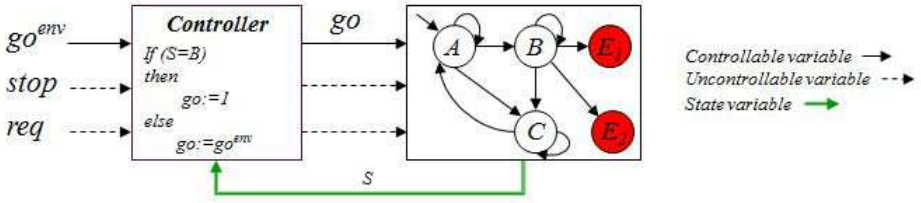


FIGURE 2.14: A 5-states design assembled to the controller



2.2.8 Implementing the control loop

In section 2.2.7.1, a guidance procedure is provided, in order to help hardware designers identify relevant candidates to control, prior to enforcing properties by EDCS. By applying this procedure, it has been found that even though it provides a way to cut down the number of controllable candidates, the designer's knowledge about the system remains preponderant in the choice of the controllable signals.

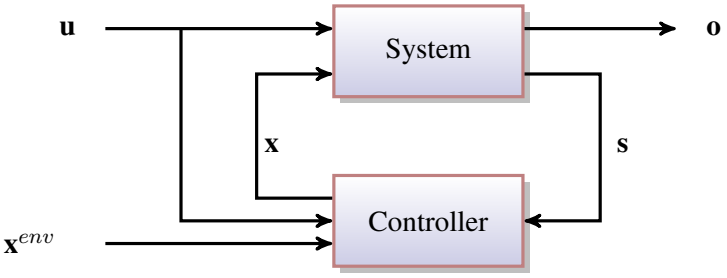
We have identified three particular control architectures, which we consider typical for the hardware design context. These architectures are structurally identical, but the behaviors they implement and the requirements they should satisfy are different, according to the knowledge available about their function. We recall that the hardware system behavior we considered are modeled following the sample-driven paradigm, where a unique clock exists for the whole system. The input values are sampled at each clock tick. In the examples presented below, a clock tick is associated to a rising edge of the clock signal.

The general control loop. This control architecture implements the classical closed-loop control. The input variable x, designated as a controllable input, is driven by the controller, according to the property enforced, and the value desired by the environment, as shown in figure 2.15 a. There is no a priori knowledge about the expected behavior of x^{env}. Hence, the possible behaviors of the controlled design are exemplified in figure 2.15 b: the controller can forward the input value, $x = x^{env}$ or mask it $x = \neg x^{env}$. It is up to the designer to judge whether the controller actions are appropriate.

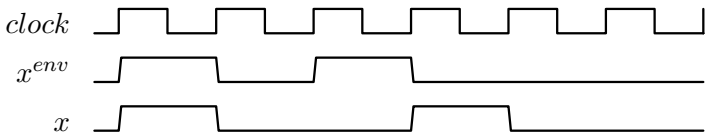
Controllable inputs with hard reactive constraints. Such inputs are driven by physical sensors. The decision procedure advocates to consider such inputs as uncontrollable. Let x^s be such an input. In practice, two possibilities exist:

- follow the decision procedure, and make an EDCS attempt by considering x^s as uncontrollable. If a control solution is found, continue with the verification process, as described below;

FIGURE 2.15: The generic control architecture



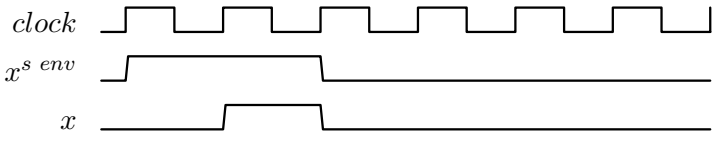
(a) Control architecture



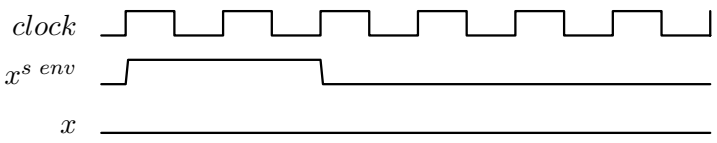
(b) Possible control behaviors

- if no control solution is found, then attempt an EDCS step by considering x^s as controllable. In this case, the meaning of the value of x^s must be taken into account: which value indicates that the sensor is active ? By convention, let $x^s = 1$ mean that the physical sensor is active. In this case, the only tolerated action for the controller is delaying the active values of x^s for an acceptable amount of time, short enough for sampling the sensor activation. This is shown in figure 2.16. This mechanism relies on a constraining assumption: the controlled system's speed should be high enough to be able to sample the changes of x^s. This requires a functional assessment, in order to establish that each time x^s rises, it shall eventually be sampled by the controlled system. In practice, the input x^s should only be *delayed for a bounded amount of time*. A performance estimation on the physical (FPGA) implementation of the controlled design is also required, in order to determine that bound, and assess if the actual system speed is sufficient.

FIGURE 2.16: Hard reactive constraints for a controllable input



(a) The physical sensor is eventually sampled

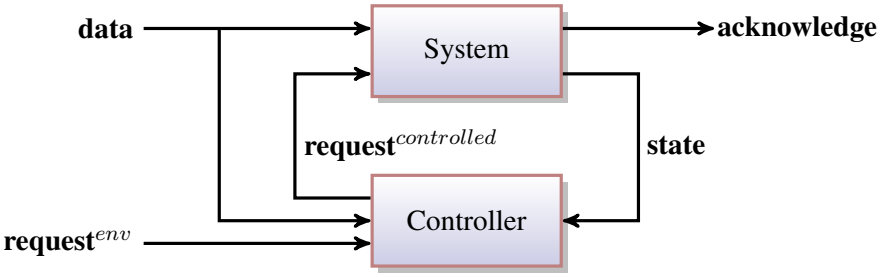


(b) The physical sensor' value is lost - the Controller is not valid

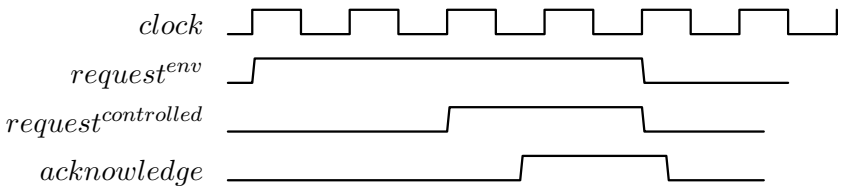
Due to its delicate implementation, this architecture is not recommended. However, if it cannot be helped otherwise (no control solution is found), an intermediate solution can consist of buffering the sensor information so that everything is memorized until it is processed. Of course, even buffering requires dimensionning, but this can be achieved statically, using the FIFO theory.

Controllable inputs with soft reactive constraints. In such situations, the decision procedure designates as controllable a *request* input, which is a part of a synchronization mechanism. The 4-phase handshake protocol is both a generic and representative mechanism in hardware design. It is used for data exchange and synchronization between components. It is implemented by a pair of Boolean signals: a request and an acknowledge. The handshake protocol starts when the request is activated. The acknowledge is then activated, followed by the request de-activation, and finally by the acknowledge deactivation. This sequence is called a transaction, and its duration can be arbitrarily long. The activation/deactivation events need to be associated to actual Boolean values, usually 1 for the active value and 0 for the inactive value. Typically, *transactions have an arbitrary delay*. It is only required that *they last a finite time*.

FIGURE 2.17: Controlling transactions



(a) Controlled transaction architecture



(b) Desired controller behavior

For this specific case, the desired control should implement one among the following behaviors: either prevent a transaction from starting if its beginning is likely to break the enforced requirement, or let it start otherwise. The transaction start is triggered by the environment, through the $request^{env}$ input variable. This value should be either forwarded or delayed by the controller, as shown in figures 2.17 a and b.

2.2.9 The "event invention" phenomenon

The application of DCS to sample driven models, and in particular to hardware models, runs sometimes into a delicate conceptual problem. Unlike input events, which trigger transitions only when they occur, input values are continuously sampled: at each (abstract) clock tick, a potentially new value can be sampled. Considered independently, a value has no significance whatsoever. Hence the notion of event occurrence needs to be reconstructed, by associating it to a (sequence of) value(s).

This semantic gap between events and values translates into an important difference between event-driven and sample-driven controllers: they forbid either events or values, but this action does not have the same significance. While the act of forbidding a controllable event is unambiguous, forbidding a controllable value can result in unwanted or even dangerous situations.

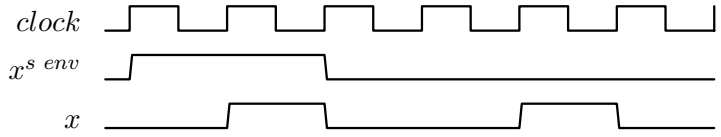
Let us re-examine the 4-phase handshake protocol. A controller which acts upon the $request$ input is not aware of the notions of activation or deactivation, but only manipulates the values 1 or 0 of this input. At some moments, the value 1 can be forbidden (and thus the value 0 is forced), or vice-versa. However, these two situations are not symmetric: in the first case, the transaction does not start, while in the second, the transaction is forced, or "invented"! This is illustrated in figure 2.18 a and b. Usually, transactions also carry data, and hence such a situation does not make sense. Obviously, this is unacceptable.

Possible workarounds can be the following:

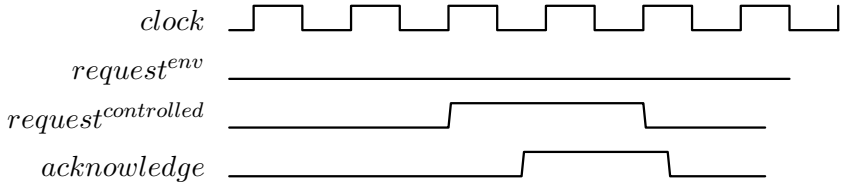
- consider transactions atomically, by making abstraction of the handshake. This is not satisfactory for hardware designers, who manipulate handshake signals explicitly. Moreover, it can happen that the phases of several transactions are required to be interleaved, for efficiency reasons, in which case this atomic point of view is inappropriate ;
- make sure a posteriori that the controller never "invents" transactions, and if it does, invalidate the control solution.

Hence it is vital to formally ensure the absence of "event-invention" phenomena. The expression of this requirement for a controllable varialbe x needs to mention systematically x^{env}, which is generated by the DCS, as explained in Chapter 1. However, the variable x^{env} does not exist at the moment DCS starts. It is not possible mention its name to express requirements over the resulting controller. This is why we cannot enforce event invention, but only detect it by model checking.

FIGURE 2.18: The event "invention" phenomenon



(a) Physical sensor "invention" by the controller



(b) Transaction "invention" by the controller

2.2.10 Detection of "event inventions"

This behavior is simply checked by the property:

$$NO\_EVENT\_INVENTION = always\neg(request^{controlled} \wedge \neg request^{env}) \quad (2.15)$$

In complement, it must also be established that a transaction cannot be delayed forever:

$$FINITE\_TRANSACTION = always(request^{env} \rightarrow eventually\ acknowledge) \quad (2.16)$$

In order to prove these requirements, it can be needed to assume that once the environment asserts the input request $request^{env}$, it is held until it is acknowledged:

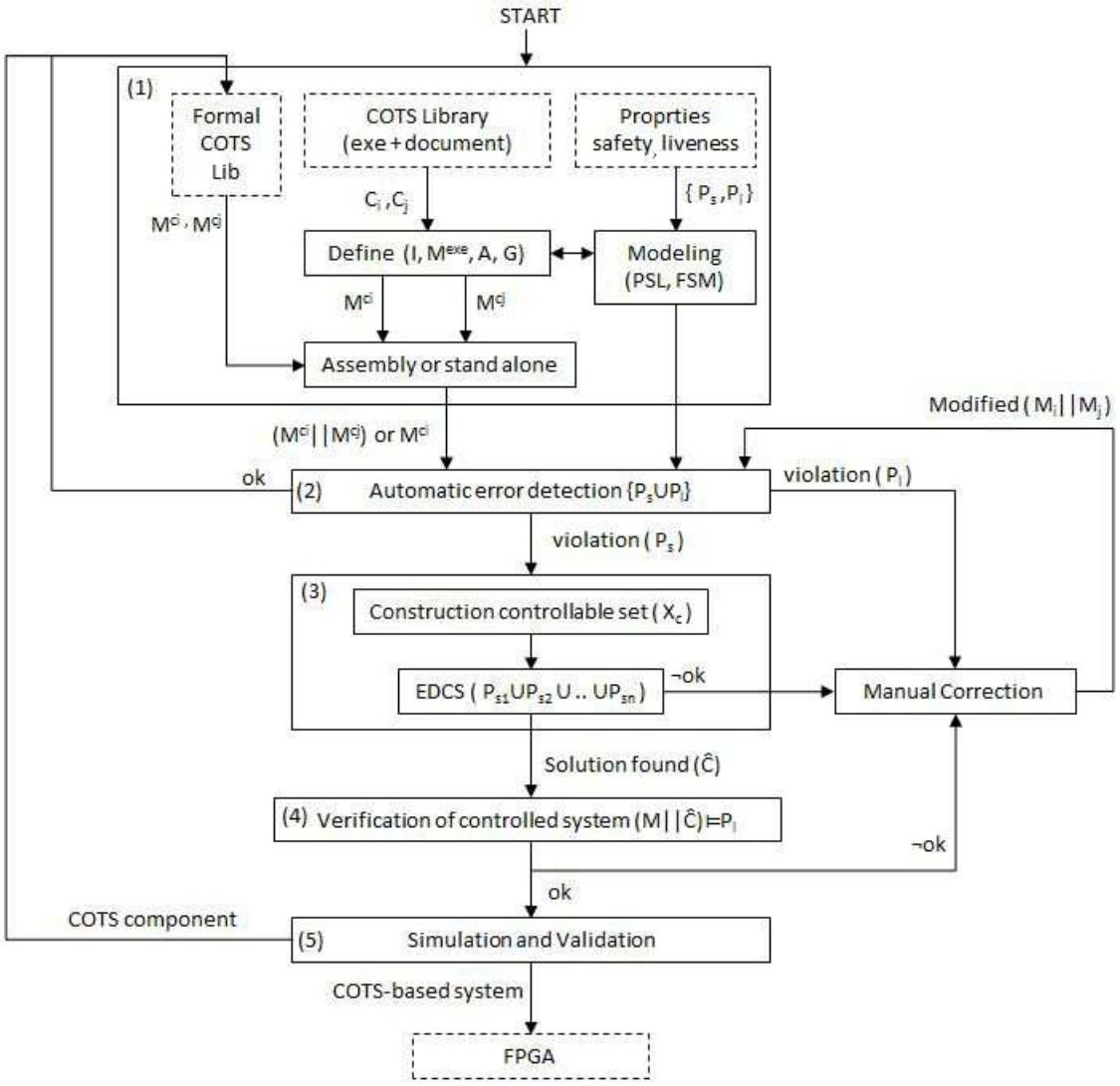
$$REQUEST\_STABLE : always(request^{env} \rightarrow ((nextstable(request^{env}))until acknowledge) \quad (2.17)$$

where the PSL operator $stable$ is a built-in PSL operator: $stable(x)$ evaluates to true in every cycle where x did not change its value with respect to the previous cycle.

2.3 The safe COTS-based design method

The method we propose is illustrated in figure 2.19. It combines several existing computer aided design approaches and advocates their use in synergy in order to build a correct by construction

FIGURE 2.19: Safe design method for hardware systems



COTS-based design. It provides the designer with the support of two formal tools: model checking and discrete controller synthesis. The method can be used for two purposes:

- synthesis of new properties (functional requirements) to a safe stand-alone component, yielding a new COTS likely to be stored inside a COTS library;
- construction of a safe COTS assembly.

We distinguish the components stored in the COTS library from those under development. As recalled previously, a COTS component comes with a set of guarantees, and thus should be locked for further modifications/redesign. Both kinds of components can be involved in a design process. It would not be realistic to assume that COTS libraries provide sufficient mechanisms for building new systems. Usually in any industrial project, design engineers need to mix COTS

reuse and code (re)writing. One advantage of this method is the ability to reduce the amount of manually written code.

The method proposed relies on five successive steps. Its starting point is a COTS library, together with a set of requirement expressions. The upstream design process is based on semi-formal model engineering, going from informal requirement specifications, through progressive refinements, towards a functional architecture, and finally to a compositional (organic) architecture. At this point, some existing components are refferred to as COTS, because of their re-usability. The components that have not been coded yet, possibly need to be purchased, and are also refferred to as COTS.

Step 1: Modeling. The first step of the design method is the formal modeling of the components. This step produces one *formal COTS* per component, by constructing the following association for each component:

- the input/output interface. This is straightforward, as it is identical to the interface of the component considered;
- the environment pre- and post-conditions. Two possibilities may occur:
  - the component is developed locally, and is a result of a top-down design process. The same design process has produced a set of informal pre- and post-conditions, which need only be formalized using as either logic or operational specifications.
  - the component has been purchased, and shipped most probably with an informal, textual documentation, which also needs to be translated into either logic or operational requirements.
- the COTS behavior: this is the design code describing its dynamics. Usually in our context, design languages such as VHDL, Verilog or SystemC are used. These languages are strictly limited to their hardware synthesizable subset [98], which is the only one supported for producing an actual hardware implementation. The same subset is required by model checking tools, which is a necessary restriction for preserving coherence between what is formally verified and what is finally implemented. This is why we identify the component behavior to its formal FSM model M, which can *always* be extracted from its design code.

At this point, formal COTS are the starting point for building new compound or stand alone functions. This action may result in the expression of additional requirements, as explained in Section 2.2.4, referred to as local or global properties.

Step 2: Automatic error detection. When two or more COTS are assembled, this may result in the breaking of some guarantees. This happens because of a possible contradiction between some guarantees/assumptions pairs which appear at the moment the COTS are assembled. Such problems can sometimes be detected visually: it can either be established that the assembly is safe, by applying compositional reasoning, or that there is a guarantee/assume contradiction induced by the assembly. However, compositional reasoning never provides exhaustive results, as it relies on the set of formally expressed requirements, which is never exhaustive itself. This is why compositional reasoning needs to be completed by an automatic model-checking step.

A COTS (or COTS assembly) which is required to satisfy a new local or global property P is also formally verified with respect to P. If the property is satisfied, the COTS can be safely used with respect to P.

The results of this step can be the following, according to the nature of the property which is verified:

- safety property; such a property is expressed using only the *always*, *next* and weak *until* operators. If a safety property is false, it can be attempted to correct it automatically using EDCS, as explained at Step 3;
- liveness property; such a property is expressed by combining the operators presented above with the *eventually* operator. If a liveness property is false, it cannot be automatically corrected, and thus this systematically requires a manual correction, when the COTS is locally produced, or external support, if the COTS has been purchased.

For all safety properties that do not hold, an automatic correction can be attempted, during the subsequent step.

Step 3: Automatic error correction. All the safety properties that do not hold are candidate for automatic error correction. This is achieved by EDCS, which operates on the same formal model and formal requirement expression as the model checking. The most delicate operation here, is the construction of the controllable input set, intended to be assigned by the controller which is generated. For this step, the designer can rely on the method provided in Section 2.2.7.1.

The EDCS application can enforce the satisfaction of the desired safety requirements. However, sometimes a control solution may not exist for the given COTS assembly and the user-defined controllable input set. EDCS fails to find a controller and the only correction possible is a *manual correction*, as shown in Figure 2.19. For such components the method needs to reiterate from Step 2.

Step 4: Formal verification. Even though a control solution is found, it should be noted that the automatic correction using EDCS can break the satisfaction of other previously established guarantees. Indeed, the behavior of the generated controller acts by disabling transitions on the "plant". When computing a new controller, EDCS does not take into consideration the existing guarantees, which is why they can be broken. If such a situation occurs, it actually invalidates the automatic correction step, and requires a manual correction and reiteration at Step 2.

Thus, the resulting controlled COTS assembly needs additional verification of its guarantee set which is expressed over the controllable input set, and thus, is likely to be broken by EDCS. It mainly focuses on verifying the liveness of the system. Besides, the non-restrictiveness and passiveness of the generated controller are systematically formally verified.

If this verification step is successful, the resulting controlled COTS is considered as a new formal COTS, providing new features, some of them established by EDCS, and conserving its guarantees. This new COTS embeds a EDCS generated controller and cannot be dissociated from it. Such a COTS can be stored in the library, and be reused as a new building block.

Step 5: Simulation. The fifth step is the final one in our method, it is a simulation of the controlled system. Validating critical systems requires a human-eyed validation and simulation of certain scenarios, interesting for the designer, to make sure that the final version of the system operates as the designer wants it to operate, before being installed in the physical environment or stocked in a formal library.

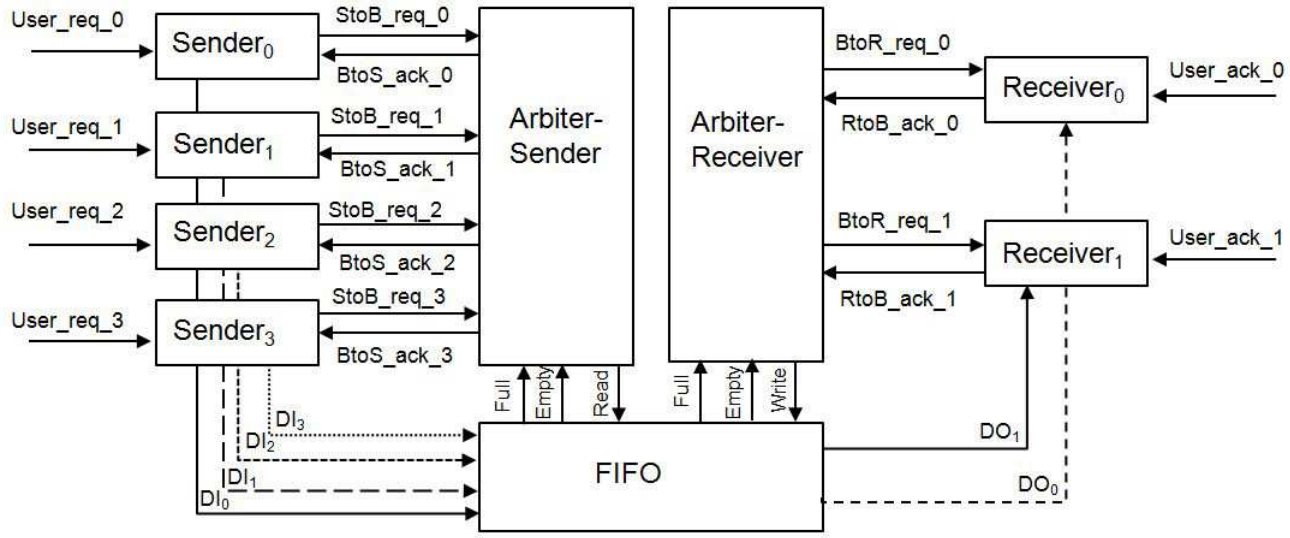
This method is generic for hardware designs. It is illustrated in detail in the remaining part of the chapter. We have chosen the GenBuf [103] system to explain our contribution because it is suitable to illustrate the different steps of our method. Indeed, this design is built of various reusable components which interact with each other in order to accomplish a global mission, which managing concurrent data transfers from a set of senders to a set of receivers, by using an arbitration mechanism and a FIFO unit.

2.4 Running example : the generalized buffer design

The Generalized Buffer "GenBuf" is a design block that queues words (32 bits) of data sent by four senders to two receivers. The queue is a depth 4 FIFO. The senders are equivalent, as are the receivers. The interface for each sender consists of a request input denoted StoB_req(i) for the i^{th} sender, an acknowledge output denoted BtoS_ack(i), and one point-to-point data bus denoted DI(i*32..(i+1)*32-1). The interface for each receiver consists of a request output

denoted BtoR_req(j), an acknowledge input denoted RtoB_ack(j), and one output data bus denoted DO(0..31), that is shared by both receivers. The figure 2.20 shows the block diagram of the system.

FIGURE 2.20: GenBuf block architecture



The GenBuf functional behavior. Each of the four senders S(i) demands to solicit the FIFO by sending a request signal $StoB\_req(i)$ to the sender's arbiter. The sender's arbiter acknowledges the senders by an acknowledgment signal $BtoS\_ack(i)$ allowing the sender to send its data DI(i) to the FIFO unit. The sender's arbiter commands the FIFO unit to read the data from the correspondent sender by the command signal $Read$. To upload the date onto the receivers, the receiver's arbiter sends a request signal $BtoR\_req\_(j)$. The receiver replies the arbiter by an acknowledgment $RtoB\_ack\_(j)$. The receiver's arbiter commands the FIFO unit to send upload the data DO(i) on the correspondent receiver by the command signal $Write$. The FIFO unit updates the sender's arbiter and the receiver's arbiter about its actual situation, whether it is full or empty by the signals $Full$, $Empty$ respectively. It is obvious that the FIFO must not read data from senders when it is full and it cannot upload data onto receivers if it is empty.

2.5 Step 1. Modeling

2.5.1 From text to formal requirement expressions

The COTS-based design method starts by building the actual COTS library. Initially, components are found as textually documented code. The modeling step first extracts the COTS

interface ports (inputs/outputs), preconditions and post-conditions from the documentation and builds the actual COTS model $M = (I, M, A, G)$. The modeling does not only concern the components selected from the library, but also, the additional functional requirements to be verified/enforced on a component or an assembly of components. We mention in this study two types of COTS libraries, (1) a generic library, where COTS are stored as design code and verbal documentation, (2) a formal library where COTS are stored as design code and formalized preconditions and post-conditions. During a design process, these libraries are subject to evolution. The modeling step relies an incremental refinement approach using SysML, also developed within the FerroCOTS project [2].

The benefits of the modeling step are twofold: first, all components are provided with a formal representation. This provides the designer with a unified view of all components. It also facilitates the COTS assembly process. The second advantage the fact that formalized COTS are more compatible with the formal tools (verification, EDCS) used throughout the design process.

As shown in figure 2.19 step (1), each component taken by the generic library is modeled as *(Interface, code, preconditions, post-conditions)*. The preconditions and post-conditions are modeled logically (PSL) or operationally (monitors). The same applies to the additional properties, to be synthesized over a stand-alone component or an assembly.

2.5.2 Example: modeling components of the GenBuf design

The documentation of the generalized buffer "GenBuf system" [37] [103], tells that the system consists of various reusable components, whose behavior is specified in hardware description language VHDL. We model each of these components regarding the generic model of COTS $C = \{I^c, M^c, A^c, G^c\}$. For each component we extract its interface, its preconditions, its post-conditions, and we keep its behavior described in VHDL as it is.

In the following we model each generic component then its instances The four generic COTS: Sender, receiver, arbiter, FIFO unit, are instanced as follows

- four instances of the COTS Sender, named Sender_(i), where $i = [0 - 3]$
- two instances of the COTS Receiver, named Receiver_(j), where $j = [0 - 1]$
- two instances of the COTS Arbiter, named Sender-Arbiter and Receiver-Arbiter
- one instant of the FIFO unit named FIFO

[2] Teamwork LOT2 FerroCOTS

TABLE 2.2: Mapping the generic names of the COTS interface to the interface names of the COTS instances

Interface	**generic**	**instance**
COTS	Sender	Sender(i)
input	$Request\_input$	$User\_req\_(i)$
input	$Acknowledge\_input$	$BtoS\_ack\_(i)$
output	$Request\_output$	$StoB\_req\_(i)$
output	$Data\_output$	DI_i
COTS	Receiver	Receiver(j)
input	$Request\_input$	$User\_ack\_(j)$
input	$Acknowledge\_input$	$BtoR\_req_(j)$
input	$Data\_input$	$DO_(j)$
output	$Request\_output$	$RtoB\_ack\_(j)$
COTS	Arbiter	Arbiter-Sender
input	$Request\_input\_(i)$	$StoB\_req\_(i)$
input	Full	Full
input	Empty	Empty
output	$Ack\_output\_(i)$	$BtoS\_ack\_(i)$
output	Go	Read
COTS	Arbiter	Arbiter-Receiver
input	$Request\_input\_(j)$	$RtoB\_ack\_(j)$
input	Full	Full
input	Empty	Empty
output	$Ack\_output\_(j)$	$BtoR\_req\_(j)$
output	Go	Write
COTS	FIFO	FIFO
input	$Read$	$Read$
input	$Write$	$Write$
input	$Data\_input$	DI_i
output	$Data\_output$	DO_j
output	$Full$	$Full$
output	$Empty$	$Empty$

The names of the interface ports in the generic formulæchange when using an instance of a generic COTS in a particular system. Table 2.2 illustrates the inputs and outputs' names of each COTS instance in GenBuf system.

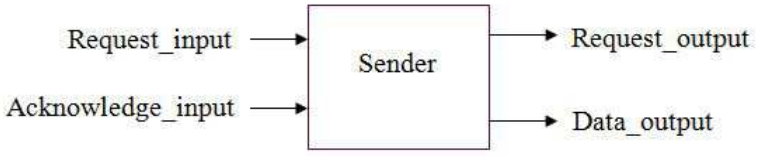
The model of GenBuf system is given as follows:

$$GenBuf = C^{sender_{0-3}} \parallel^c C^{receiver_{0-1}} \parallel^c C^{arbiter-sender} \parallel^c C^{arbiter-receiver} \parallel^c C^{FIFO}$$

1. **Four Senders**: The generic sender COTS exist in the library has certain interface ports, as shown in figure 2.21.

$$I^{sender} = (request\_input,\ Acknoledge\_input,\ request\_output,\ data\_output) \quad (2.18)$$

FIGURE 2.21: A sender COTS



The request_input is fed by the final user of the sender. The request_output transfers a sending request to the a new component (for example, an arbiter). The data_output named ports transfer the data to another component which stores data (for example a FIFO unit).

The sender component has only one precondition assumption in the precondition list $A^{sender} = \{a_1^{sender}\}$ to be respected by its environment, which is: an activation value (Boolean true) must occur infinitely often:

$$a_1^{sender} = always\ eventually(request\_input); \quad (2.19)$$

The sender component has a post-condition $G^{sender} = \{g_1^{sender}\}$ to be respected by its behavior, which is: the request demanded by the final user, should eventually be transferred to the arbiter connected to the sender:

$$g_1^{sender} = always(request\_input \rightarrow eventually\ (request\_output)); \quad (2.20)$$

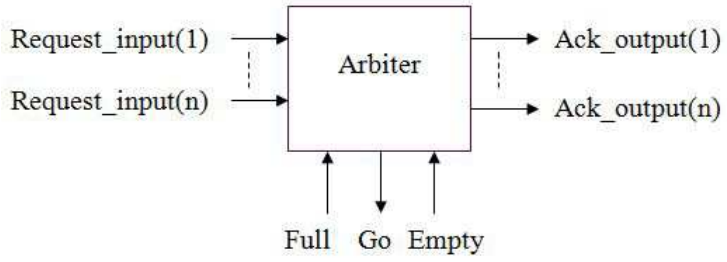
The instance $Sender\_i$ of the sender is modeled as follows :

- $I^{S_i} = \{User\_req\_(i), BtoS\_ack\_(i), StoB\_req, DI\_i\}$
- $A^{S_i} = \{always\ eventually(User\_req\_(i))\}$
- $G^{S_i} = \{always(User\_req\_(i) \rightarrow eventually(StoB\_req\_(i)))\}$

2. **Two arbiters:** The arbiter component is supposed to manage the connection between some components (for example senders or receiver) with another component (ex, a data store FIFO unit) as show in figure 2.22. The arbiter's interface is:

$$I^{arbiter} = (request\_input\_(i),\ Full,\ Empty,\ ack\_output\_(i),\ Go) \quad (2.21)$$

FIGURE 2.22: Arbiter COTS



An arbiter has an assumption over its input $A^{arbiter} = \{a_1^{arbiter}\}$, that supposes requests should occur infinitely often:

$$a_1^{arbiter} = always\ eventually(request\_input) \tag{2.22}$$

Also the arbiter has to guarantee a request received should be eventually acknowledged:

$$g_1^{arbiter} = always(request\_input\_(i) \rightarrow eventually\ (ack\_output\_(i))) \tag{2.23}$$

The generalized buffer contains two instances of the arbiter component: (1) Sender's arbiter, (2) Receiver's arbiter.

The sender's arbiter interface:

$$I^{Arbiter_S} = (StoB\_req\_(i),\ BtoS\_ack\_(i),\ Full,\ Empty,\ Read) \tag{2.24}$$

The sender is supposed to organize the access of the senders, to a FIFO unit. It sends an acknowledgment $BtoS\_ack\_(i)$ to the sender i which asked to send data to the FIFO. It sends a $Read$ command to the FIFO when it should starts reading data.

The **receiver's arbiter** interface is similar to the receiver:

$$I^{Arbiter_R} = (RtoB\_ack\_(j),\ BtoR\_req\_(j)Full,\ Empty,\ Write) \tag{2.25}$$

The receiver is supposed to organize the writing on the two receivers by the FIFO unit. It sends an acknowledgment to the receiver which asked to get data from the FIFO. It sends a $Write$ command to the FIFO when it should starts writing data on the receiver.

The process of reading and writing data is out the scope of our study as it is not a control process.

3. **A FIFO unit** The FIFO COTS receives data from the senders. It stores (8 x n) bits of Data, in our example $n = 4$. It writes the stored data to the receivers when demanded.

The generic FIFO COTS's interface is:

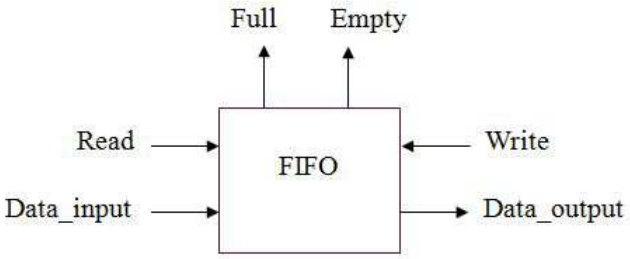
$$I^{FIFO} = (Data\_input,\ Data\_output,\ Read,\ Write,\ Full,\ Empty) \tag{2.26}$$

It is shown in figure 2.23

The interface of our FIFO instance is :

$$I^{FIFO} = (DI\_i,\ DO\_j,\ Read,\ Write,\ Full,\ Empty) \tag{2.27}$$

FIGURE 2.23: The FIFO unit COTS



The FIFO component has a precondition ($A^{FIFO} = \{a_1^{FIFO}\}$) supposed to be respected by its environment, which is a FIFO should be infinitely often solicited by some senders:

$$a_1^{FIFO} = always\ eventually(Read) \tag{2.28}$$

The FIFO component has some post-conditions ($G^{FIFO} = \{g_1^{FIFO}, g_2^{FIFO}, g_3^{FIFO}\}$) supposed to be respected by its behavior, if the precondition is respected.

- A FIFO should eventually send its contained data to some receivers

$$g_1^{FIFO} = always\ eventually(Write) \tag{2.29}$$

- A FIFO must not accept data if it is full

$$g_2^{FIFO} = always\neg(Read \wedge Full) \tag{2.30}$$

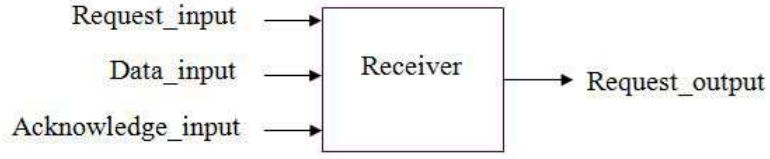
- A FIFO must not send packets of data if it is empty

$$g_3^{FIFO} = always\neg(Write \wedge Empty) \tag{2.31}$$

4. **Two Receivers :** A generic receiver COTS in the library has certain interface ports, as shown in figure 2.24:

$$I^{receiver} = (request\_input,\ Acknowledge\_input,\ data\_input,\ request\_output) \tag{2.32}$$

FIGURE 2.24: A receiver COTS



The request_input is fed by the final user of the receiver. The data_input ports transfer the data from a data store component, to the receiver. The request_output transfers a request to the a component (for example, an arbiter) in order to inform it that the receiver's user wants to get some data.

The receiver component has a precondition to be respected by its environment $A^{receiver} = \{a_1^{receiver}\}$, which is: a request activation value, (Boolean true), must eventually often appear:

$$a_1^{receiver} = always\ eventually(request\_input); \tag{2.33}$$

The receiver component has a post-condition $G^{receiver} = \{g_1^{receiver}\}$ to be respected by its behavior, which is: the request demanded by the final user, should eventually be transferred to the arbiter connected to the receiver:

$$g_1^{receiver} = always(request\_input \rightarrow Frequest\_output); \tag{2.34}$$

The receiver instance j in GenBuf is modeled as follows :

- $I^{R_j} = \{User\_ack_(j),\ DO\_j,\ BtoR\_req\_(j),\ RtoB\_ack\_(j)\}$
- $A^{R_i} = \{always\ eventually(User\_ack\_(j))\}$
- $G^{R_j} = \{always(User\_ack\_(j) \rightarrow eventually(RtoB\_ack\_(j)))\}$

We construct the assembly model of GenBuf regarding the assembly block diagram 2.20 as follows:

The signals: $StoB\_req\_(i), BtoS\_ack\_(i), BtoR\_req\_(j), RtoB\_ack\_(j), Read, Write, Full, Empty$ become internal signals in the assembly.

The input set of the assembly is : $X^{GenBuf} = \{User\_req\_(i), User\_ack\_(j)\}$.

The output set of the assembly is : $Y^{GenBuf} = \{\emptyset\}$.

Thus, the assembly of GenBuf interface is defied as follows :

$$I^{GenBuf} = X^{GenBuf} \cup Y^{GenBuf} = \{User\_req\_(i), User\_ack\_(j)\}. \tag{2.35}$$

The set of internal signals is defined as follows:

$$I^{intern}_{GenBuf} = \{StoB\_req\_(i), BtoS\_ack\_(i), BtoR\_req\_(j), RtoB\_ack\_(j), Read, Write, Empty, Full\} \tag{2.36}$$

The data signals $Data\_input, Data\_output$ are not considered in the interface as they are not control signals.

The set of assembly preconditions is :

$$A^{GenBuf} = A^{sender(i)} \cup A^{receiver(j)} \tag{2.37}$$

where : i, j represent the number of sender, receiver respectively.

The set of assembly post-conditions is :

$$G^{GenBuf} = \{G^{S_{[0-3]}} \cup G^{arbiter_S} \cup G^{arbiter_R} \cup G^{FIFO} \cup G^{R_{[0-1]}}\}. \tag{2.38}$$

The additional global requirements are expressed in the same manner, either logically or operationally. The satisfaction or the global requirements needs to be established: either formally verified, or enforced by EDCS, or both.

2.5.3 Exemple : writing global properties for the GenBuf design

We illustrate the idea of expressing temporal properties over the input and output signals over the GenBuf system.

- Each sender should be served and allowed to send its data. This property has a liveness nature. The signals known by the designer and related to this property are the the acknowledgments sent from the sender's arbiter to the senders to allow them send the data ($BtoS\_ack\_(i)$).

To be complete, each sender acknowledge should be related to a request made by the sender itself ($StoB\_req\_(i)$). Thus the property expression by integrating the senders requests has the following expression :

$$always\ (StoB\_req\_(i) \rightarrow eventually\ (BtoS\_ack\_(i))) \tag{2.39}$$

Read as follows, in every moment if there is a sender's request, it must eventually be acknowledged.

- All receivers must be solicited. This property also has a liveness nature. The interface signals over which the property can be expressed are : ($BtoR\_req\_(j)$).

  Like the situation on the sender's side, the property of receiver's liveness is related to a demand signal from the receiver itself ($RtoB\_ack\_(j)$).

  The property can be written with the use of (G, F), always and eventually LTL operators as follows and in PSL as follows :

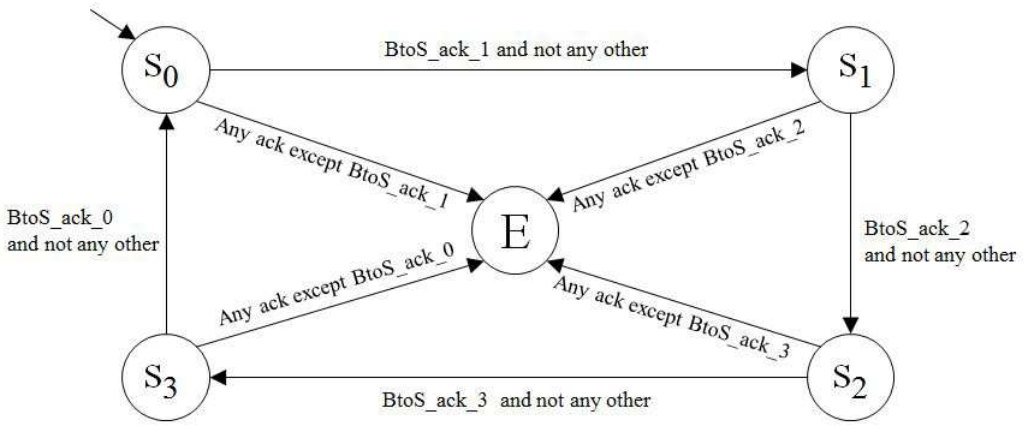
$$always\ (RtoB\_req\_(j) \rightarrow eventually\ (BtoR\_ack\_(j))) \tag{2.40}$$

- Senders alternation means the system must acknowledge the senders in the order ($S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3\ then\ S_0\ and\ so\ on$).

$$always\ (BtoS\_ack\_(i)\ before\ \ (BtoS\_ack\_(i+1)\ mod\ 4)), i \in [0..3] \tag{2.41}$$

To facilitate the verification process we represent the alternative behavior of the four senders as a FSM monitor as shown in figure 2.25. The monitor models the desired behavior of the senders and passes to an error state in this alternation is not respected.

FIGURE 2.25: Alternative senders' behavior

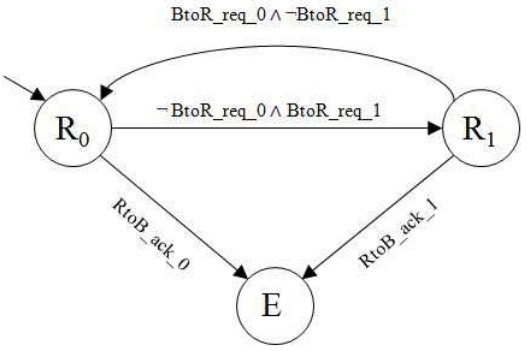


- Receiver's alternation means the arbiter must solicit the receivers in the order (R_0 then R_1) alternately. This property can also expressed in PSL as follows :

$$always\ (BtoR\_req\_(j) \rightarrow next\ (BtoR\_req\_(j+1)\ mod\ 2)), i \in [0..1] \tag{2.42}$$

The alternative behavior of the receivers as a FSM as shown in figure 2.26. As long as the alternative behavior is respected by the receivers the monitor is in accepted states, as soon as the property is violated, the monitor passes to an error state.

FIGURE 2.26: Alternative receivers' behavior



A correct functionality of the GenBuf system requires a coherent behavior of senders and receivers. In other words, the environment must respect some correct behaviors like :

- All senders should send requests to the sender's arbiter;
- All receivers should send requests to the receiver's arbiter
- No errors should appear during the system operation.

Each one of those preconditions is also expressed in PSL, with respect to the input signals related to it. They are expressed as follows :

- Every sender should send a request to send its data to the FIFO unit infinitely often:

$$always\ eventually\ (StoB\_req\_(i)) \tag{2.43}$$

- Every receiver should acknowledge their data infinitely often:

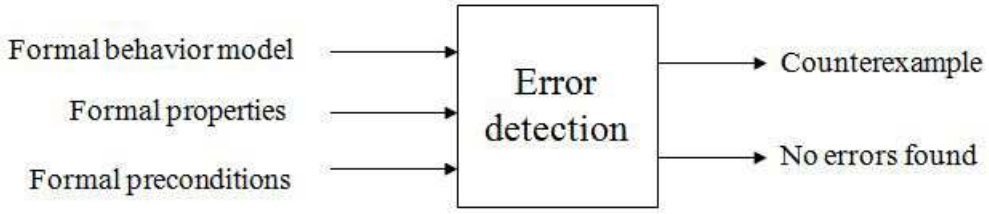
$$always\ eventually\ (RtoB\_ack\_(j)) \tag{2.44}$$

In the above steps the requirement formalization has been achieved in a straightforward manner, by extracting relevant assertions from the available documentation. For a rigorous requirement formalization approach, the reader should refer to [104].

2.6 Step 2. Automatic error detection

In the former step the designer models any unwanted behavior as a safety property and any required behavior as a liveness property. In this step he detects automatically if the design satisfies these requirements or not. The step is fully automatic, it takes as input the symbolic model extracted from the COTS/assembly code behavior and the properties to be verified. If the property is not satisfied, a counterexample is provided under the form of a simulation sequence. This mechanism is recalled in figure 2.27. The verification tool highlighted here is model checking. The choice is motivated by its ease of use, and the fact that it fully supports the compositional reasoning steps defined in this method.

FIGURE 2.27: Error finding for DES systems

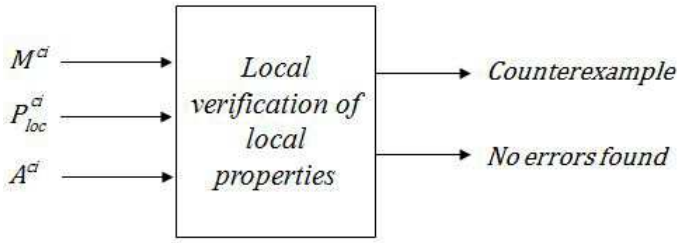


2.6.1 Local verification of local properties

This verification detects the stand-alone errors, when the designer's goal is only to synthesize new properties to a component. The designer presents all the properties he wants to verify and verify it under the component preconditions (A^{c_i}) as shown in figure 2.28. There is no need to verify the component post-conditions (G^{c_i}) as they are already respected and no change in the component preconditions happened which can break them.

When a new local property is introduced to the component it must be locally verified under the component local preconditions. If the property is broken locally by the component behavior, this encourages the designer to correct the error locally, before introducing the component in the global design.

FIGURE 2.28: Local verification of local stand-alone error



2.6.2 Global verification of local properties

This verification should be used to detect the local errors caused by the assembly. Consider a COTS assembly consisting of of K communicating components $C_i \mid i = [1, ..., K]$ with the corresponding behavioral model $M^{asm} = \{M^{C_1} \parallel M^{C_2} \parallel M^{C_K}\}$. The designer provides the model checking tool with the behavioral model of the assembly M^{asm} and the set of the new properties to be verified P^{asm}

Note that the assembly operation needs to be safe. Hence, the designer should establish the conservation of the local post-conditions of each stand-alone component (G^{c_i}) through the assembly. This is achieved either by applying the compositional reasoning previously defined, or, if such a reasoning cannot be applied, by formally verifying each of them. Typically, the compositional reasoning cannot be applied if it is circular, or if the sets of assumed/guaranteed requirements are insufficient. This decision procedure is illustrated in figure 2.29.

If no error is detected the designer can validate the component assembly as a safe component. If the new global properties are respected and one or more original post-conditions of any component is broken, this indicates the designer that the components have conflicting behavior, in this case the designer can decide whether the broken post-conditions are acceptable or not: validate the design with new global properties and less local post-conditions, or not. In case he validated the assembly with its new situation, a new COTS needs to be created, featuring less guarantees.

2.6.3 Global verification of global properties

Let $(C_a,\ C_b)$, be two assembled COTS, let P^{asm} be a global property concerning the global behavior of the two components, we construct the assembly behavioral model $M^{asm} = M^{C_a} \parallel M^{C_b}$, and we verify if: $M^{C_a} \parallel M^{C_b} \models P^{asm}$ under the assembly preconditions A^{asm} as shown in figure 2.30.

FIGURE 2.29: Global verification of local error caused by assembly

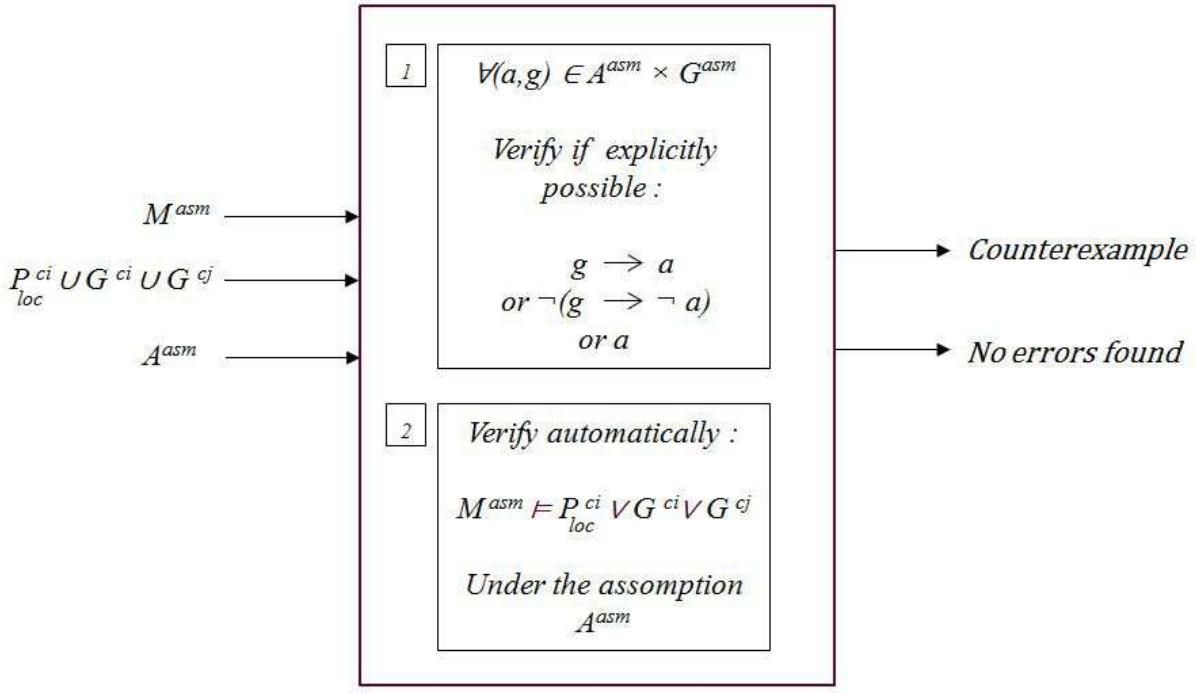
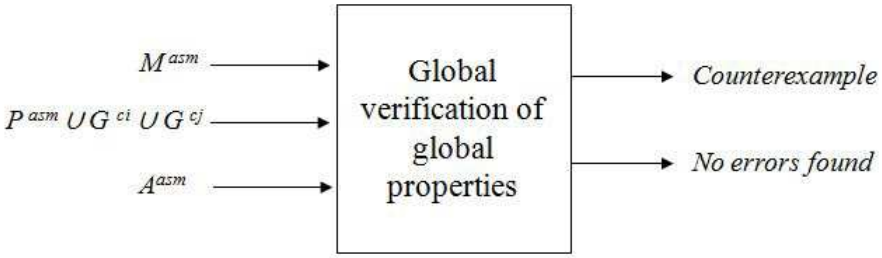


FIGURE 2.30: Global verification of global error



A global property P^{asm} is probably not satisfied by the assembly unless the components are built to work together, which is difficult to be ensured when buying COTS from different companies. Thus, receiving a verification counterexample is an expected result.

Example 2.9. *To illustrate the step of error detection we have verified two liveness global properties over the GenBuf system :*

- *acknowledgment of all senders.*

$$For\, i = 0\, to\, 3\, Verify\, the\, assertion : always\, (StoB\_req\_(i) \rightarrow eventually\, (BtoS\_ack\_(i))); \tag{2.45}$$

- *solicitation of both receivers*

$$For\ j = 0\ to\ 1\ Verify\ the\ assertion : always\ (RtoB\_ack\_(j) \rightarrow eventually\ (BtoR\_req\_(j)));$$ (2.46)

The formal verification results show that both properties are respected by the system design, under a precondition assumption that imposes that there will always be a request from all senders to send data

$$a_1^{GenBuf} : always\ eventually(User\_req\_(0) \vee User\_req\_(1) \vee USer\_req\_(2) \vee User\_req\_(3))$$ (2.47)

If a liveness property is broken, a manual correction is required. If a safety property is broken locally by a component, our method calls for an automatic correction as shown in figure 2.19.

Example 2.10. *To verify the property of alternative senders' behavior, and alternative receiver's behavior, we assemble the FSM models of the monitors to the behavior model of the GenBuf and verify the following safety assertion:*

$$always \neg (sender\_error \vee receiver\_error);$$ (2.48)

The formal verification results show that both safety properties are broken by the design. The model checking tool provides two counterexamples which illustrate where exactly the assembly behavior breaks each property.

2.7 Step 3. Automatic error correction

This step aims to automatically correct the design errors detected by the model checking tool. The manual correction of design errors is time consuming when the system is large, especially when the system components are built by other design teams. This step replaces the manual correction by an automatic generation of a piece of code to correct each discovered error. Thus the correction time is reduced.

2.7.1 Automatic synthesis of a correcting controller

This step requires the definition of a list of controllable variables. According to the procedure defined earlier in this chapter, the controllable set is determined with respect to the counterexample showing that the property to be enforced is not yet satisfied.

TABLE 2.3: Set of counterexample variables candidates to be controllable

Signal name	Signification
StoB_req_(i)	sender to sender's arbiter request
RtoB_ack_(j)	receiver to receiver's arbiter request
User_req_(i)	the input of a sender component
User_ack_(j)	the input of a receiver component
DI[0-31]	input data sent from the senders to the FIFO unit
DO[0-31]	output data sent from the FIFO unit to the receivers

Example 2.11. *In GenBuf system, the model checking tool has provided a counterexample for the alternative behavior of senders and receivers. The list of signals appeared in the counterexample is illustrated in table 2.3. Among these signals, with regard to the rules of constructing the list of controllable signals we remove the DI[0-31], DO[0-31] signals as they represent data information and* $StoB\_req\_(i), RtoB\_ack\_(j)$ *as they are internal variables. The resulting list of controllable signals for GenBuf is*

$$X_c^{GenBuf} = \{User\_req\_(i), User\_ack\_(j)\} \tag{2.49}$$

After calculating the list of controllable signals X_c, the designer generates automatically the controller (if it exists). EDCS needs the following pre-requisites to operate: the symbolic model of the COTS assembly, the controllable variable list, the requirement to be enforced and possibly the environment assumptions that should be taken into account. The act of providing pre-conditions to EDCS is similar to a model checking verification process:

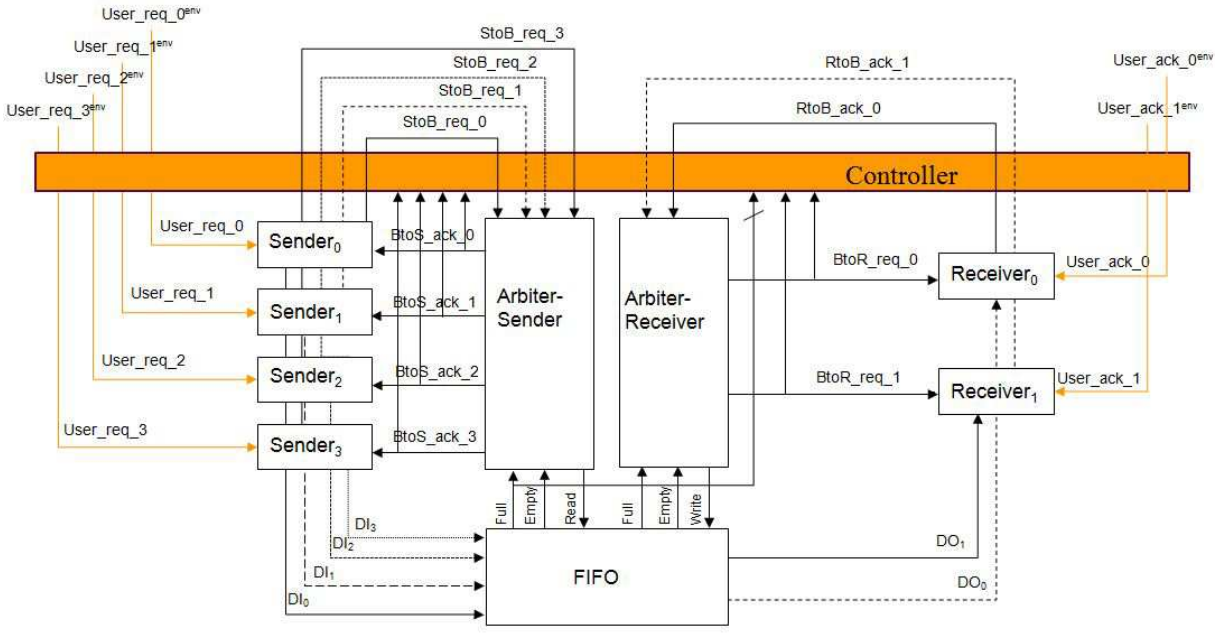
1. the first EDCS attempt is performed without environment assumptions. If a satisfactory control solution is found, then stop;
2. if no control solution was found, or if the solution is too constraining, provide manually an environment assumption. Go to step 1.

Example 2.12. ***Generation of a correcting controller for the global property "senders and receivers alternation"***

We assemble the monitors of the safety properties (alternative behavior of senders, alternative behavior of receivers) to the model of the system GenBuf. In this particular example, the design does not have safety preconditions to be supplied to the DCS tool with the safety properties. The DCS tool generates a controller for safety property that combines both properties $(always\neg(error\_sender \vee error\_receiver))$.

The assembly (original system-controller) is illustrated in figure 2.31, the orange parts represent the controller which reads all the system variables (controllable, uncontrollable and internal variables), and affect only the controllable variables which are represent in orange arrows. The different styles of arrows (dash, dot, line) are only used to distinguish the source and destination of each signals.

FIGURE 2.31: The controlled GenBuf system, the arrow with a diagonal bar ↛ combines figuratively the signals: Full, Empty, Read and Write only to keep the figure visible



The automatic error correction concerns only the correction of safety nature errors, whereas in the cases of liveness nature errors are detected a manual modification is required and the designer must recall the error detection step. Also if the EDCS methods fail in finding a correcting solution the manual modification becomes unavoidable step and the designer must remount to step (2) as shown in figure 2.19.

2.8 Step 4. Verification of the corrected/controlled system

EDCS generated controllers, if they exist, only guarantee the satisfaction of the enforced properties. However, it can happen that enforced properties are satisfied in a very "restrictive" manner. Typically and caricaturally, a design which gives no answer to its stimuli can be considered as a safe design, but this behavior is not satisfactory! Hence, the designer needs to ensure that after automatic correction by EDCS, the global operation of the design remains satisfactory. A particular care is needed in order to establish the conservation of the liveness guarantees, as

they characterize by nature the creativity of the design. There are two kinds of post-synthesis requirements that need to be checked:

- technical requirements, related to the implementation of the control loop. Typically, these amount to expressing formally the absence of "event invention" phenomena;
- functional requirements: after enforcing a property by a correcting controller, all guarantees need to be assessed once again.

Assessing all previously checked guarantees is an expensive process. Fortunately, not all of them actually need to be re-assessed. Consider M^{abs} a model obtained from a COTS assembly, and P be a global property to be enforced on M^{abs}, by composing it with the controller $\hat{\mathcal{C}}$. The following situations may occur, either allowing the conservation of a guarantee g, or requiring a new assessment.

The guarantee is a safety property with no assumptions. In this situation, we have:

- before synthesis: $M^{asm} \models g$. The assembly model satisfies the guarantee g, with no environment assumption;
- after synthesis: $M^{asm}||\hat{\mathcal{C}} \models g \wedge P$. Enforcing the satisfaction of P has no influence on the satisfaction of g. This is true because initially, g holds whatever the behavior of the environment of M^{abs}. The generated correcting controller is just a particular environment of M^{abs}.

Hence, the satisfaction of g needs not be assessed anymore.

The guarantee is a safety property relying on assumptions. In this situation, we have:

- before synthesis: $M^{asm}, a \models g$. The assembly model satisfies the guarantee g, provided that the assumption a holds for the environment of M^{abs};
- after synthesis: $M^{asm}||\hat{\mathcal{C}} \models P$. It cannot be concluded that g is conserved after synthesis. Indeed, g relies on a particular environment behavior, which can be in contradiction with the behavior of the generated controller enforcing P.

Hence, the satisfaction of g needs to be re-assessed. This is achieved in two steps. First, as the controller provides an environment behavior for M^{asm} it can first be checked that the controlled

assembly $M^{asm}||\hat{\mathcal{C}} \models g$. If this proof is not successful, it can be checked whether $M^{asm}||\hat{\mathcal{C}}, a \models g$.

The same reasoning applies to liveness properties, according to the situations where they hold free from environment assumptions or not.

Example 2.13. *Ensuring the absence of "event invention".*

In the GenBuf system, we verify that the controller system does not invent a sender request unless the final user commits a request to use the FIFO. Similarly, the controller system does not invent a request to write from the FIFO on a receiver unless the final user of this latter commits a request to do so. Those properties are specified in PSL as follows :

$$No\ sender's\ request\ invention\ : assert\ always\ \neg(User\_req\_(i) \wedge User\_req\_(i)^{env}); \tag{2.50}$$

$$No\ receiver's\ request\ invention\ : assert\ always\ \neg(User\_ack\_(i) \wedge \neg User\_ack\_(j)^{env}); \tag{2.51}$$

Example 2.14. *Permissiveness of the controlled GenBuf*

After synthesizing the alternation properties of senders and receivers over the GenBuf system, we verify that all senders are capable to send data to the FIFO unit, by verifying the property :

$$always\ eventually(BtoS\_ack\_(i))$$

We also verify that both receivers are solicited in the controlled system, by verifying the property :

$$always\ eventually(BtoR\_req\_(j))$$

In case the designer is not satisfied by the verification results the manual correction is recommended and after that the designer remounts to step (2), in order to verify whether the modifications he made entailed design error or not as shown in figure 2.19.

2.9 Step 5. Simulation

This step is the last one in our method before taking the decision to validate the result component. The goal of this step is to ensure that the controlled system is still operating as needed.

Since in the error correction step some behaviors were removed by the DCS, the designer needs to acquire supplementary insight on the controller decisions; in particular, as a complement to formal liveness verification, simulation can witness the fact that the controller is not too restrictive. In this step the designer can monitor the evolution of the resulting controlled system before the final hardware implementation. Beyond the subjective need of visualizing the system behavior before implementing it, simulation is used when the formal verification reaches its complexity limits. The same temporal properties can be checked by guided simulation, with much better scalability. Of course, the result is not exhaustively established, and the validity of a "proof" depends on the bound chosen for the simulation time, and thus for the simulation efforts.

2.10 Conclusion

In this chapter, we have presented a safe design method based on COTS design and reuse. The integration of the Discrete Controller Synthesis technique based on assembly discussion in this design flow represents the major contribution of our work.

The method presented takes advantage of the existing computer aided design methods and techniques and uses them in synergy, filling major gaps left by each method considered alone. Our method is semi-automatic as it depends on software tools, and in the same time the designer intervention is indispensable in certain phases, like the writing the functional requirements to be respected and those to be synthesized, the visual validation of the simulation results. The method consists of six steps, starting from the textual documentation of the target design till the final system ready to be installed on an FPGA chip.

The method can be used for two main goals: (1) addling new properties to a stand-alone component and save it in a formal COTS library. (2) assembling components together and adding new properties to be respected by the assembly, then either implement the assembly design or save it in a formal COTS library.

The modeling step (1) concerns the formal modelization of hardware reusable components, COTS, which are usually treated by only their textual documentation and their behavioral model which is a black box for the COTS user. in the same step requirements of COTS are classified in two categories preconditions and post-conditions (a notion that is not provoked in Ramadge/-Wonham framework), then modeled in formal assertions/models (PSL or FSM). The designer also formalizes the new safety properties that must be satisfied by the studied design (stand-alone COTS or an assembly). This step must be realized carefully since it represents the first

formal step in the design and it must be fully correct translation of the system documentation and the contract between the designer and the client.

The error detection step (2) concerns the verification of the properties formalized above using model checking tool. When the method is used to add properties for a stand-alone COTS, the designer verifies the new properties under the preconditions of the component, if the properties are respected he validates the design and save it in a formal COTS library after adding the new properties to its documentation. Else, if he receives a counterexample he passes to the error correction step. If the method is used to assemble some components together and add new properties to the assembly, then the designer need to automatically verify the local post-conditions and the new global properties under the preconditions of the assembly. In case all post-conditions and properties are respected, he can either save the assembly in the formal library or implement it. Else, if he receives a counterexample he must pass to the step of error correction.

The error correction step (3) achieves removing the errors detected in step two. If the error is a violation of a liveness property, then a manual correction is required, else if the error is a violation of a safety property, then an automatic solution may be found by using the an environment aware DCS method. The designer automatically corrects the errors (the violation of local or global properties), in case the cause of this error is a mistake in defining the component or assembly of components preconditions or a bad synchronization between the assembled COTSs. The DCS tool needs to manipulate some inputs of the design, named controllable inputs, in order to correct the error. When the controllability of inputs of the system is not predefined, the designer needs guidance in order to determine what to control. Our method guides the designer in his choice by suggesting the use of the counterexample as it contains, only and all, the signals involved in causing the error. The designer starts by the full set of signals of the counterexample and exclude some signals from being controllable regarding an algorithm for building the controllable list. We argue that although this algorithm may cut down remarkably the number of controllable candidates, but, the designer knowledge of the system remains indispensable. Thus, we propose an implementation of the control loop; three control architectures typical for the hardware systems which can make the decision of signals controllability more reliable.

After building the controllable inputs' set, the designer uses the DCS tool and generates a correcting controller which is a Boolean function, we define a *patch* controller to correct the local errors and a *glue* controller for the global errors, with preserving the safety preconditions and post-conditions, the patch and the glue are conceptually the same, they difference only in the error's type which they correct.

The controlled system verification step (4) concerns the livness verification of the controlled system, because of the shortcoming of the DCS tool, since it cannot synthesize liveness assertions. Moreover, the designer needs to verify the passiveness of the controller, i.e, the controller does not invent decisions the user did not take before.

The final step (5) is the simulation of the controlled system against certain interesting scenarios defined by the designer. It provides the designer with a final look over the design in order to reassure that the system behaves in a safe manner. No matter how trusted are the used tools for the model checking and discrete controller synthesis, the human validation remains an indispensable step for critical industrial systems like transport, military and medical devices.

Our contribution combines the applicable efforts existing in the literature of designing hardware systems and the research studies in this domain. We aim to profit of the advantages of using COTS components in the design, like time and money consuming. Contrarily to the existing COTS based methods, we do not reject the selected COTS in case it achieve certain parts of the required job, but, we generate an additional correcting component, which enforces a desired behavior on the original design, without modifying the internal behavior of the reused components.

Chapter 3

Application on an industrial system

3.1 Introduction

Applying the academic proposals on real systems is a challenge that researchers face due to several reasons. Firstly, the size of real systems is sometimes beyond the reach of academic tools. Secondly, it is often necessary to spend considerable engineering efforts, in order to make academic tools compliant with the industrial needs. These efforts concern various aspects such as user interface enhancement, input language support, optimization, and most certainly *certification*.

The goal of this chapter is to apply our proposal on a real industrial design, in order to demonstrate its feasibility, and the ability of the DCS technique to handle industrial control/command problems. The design project considered here is provided by Bombardier Transport Company, and has been used during the FerroCOTS project [105]. It is detailed enough to illustrate all the design steps of our method. The developments presented in this chapter have been conducted at the Ampere laboratory, in collaboration with the Ferrocots partners. We have presented this industrial case in [106].

3.2 FerroCOTS: Presentation and Goal

The initial documentation of the project describes the control/command system as an interconnection of logical gates and relays, presented either textually or graphically, as logic data-flow diagrams.

The modeling tools had a UML flavor, using a variant of the UML activity diagrams to represent the control/command dynamics of the system under design. According to our understanding,

the design process starts with a textual requirement expression and refines it into a set of "activity" diagrams. The translation between the refined requirements formalized as activity diagrams and the final logic diagram is left unspecified. The actual implementation of the design amounts to the production of a hard wired board. Simulation traces have never been available. A possible explanation can be the fact that the actual logic diagram were expressed textually, as an *intermediate specification*, not intended to be fed to a simulation tool. Thus, the verification process was mainly performed manually (or "brainually"), by the design engineers.

The perception of this process can be somehow caricatural, but its results were satisfactory as long as the control/command functions were not extremely complex, and thus, it could be afforded to achieve some design steps manually. Moreover, this process was validated, and certified. Recall that these control/command functions were (and some are still) implemented inside real and *safe* trains. All the key steps were relying on human expertise, which was enough for guaranteeing safety.

In this context, FerroCOTS is an innovation project which aims to implement the train control-command systems by a design method based on reusable/reused COTS. This objective also requires an architectural evolution: replace the hard wired relays and combinational logic boards, by logic components implemented by an FPGA (Field Programmable Gate Array) chip. This architectural and methodological upgrade brings several advantages, such as reconfigurability, requirement traceability, as well as the ability to integrate formal tools and design methods, already mature for hardware designs. Last but not least, FPGA boards turn out to be lighter, and easier to manipulate by maintenance employees. The resulting design method is entirely supported by SysML: from the requirement expression, through a systematic application of progressive refinement rules, to a functional decomposition, and finally an organic decomposition into COTS, featuring an interface and formally specified. Our role in this project is to develop the COTS-based safe design method.

We base our proposals on the documentation supplied by Bombardier, and included in Appendix 1 of this document.

3.3 The Passengers Access System

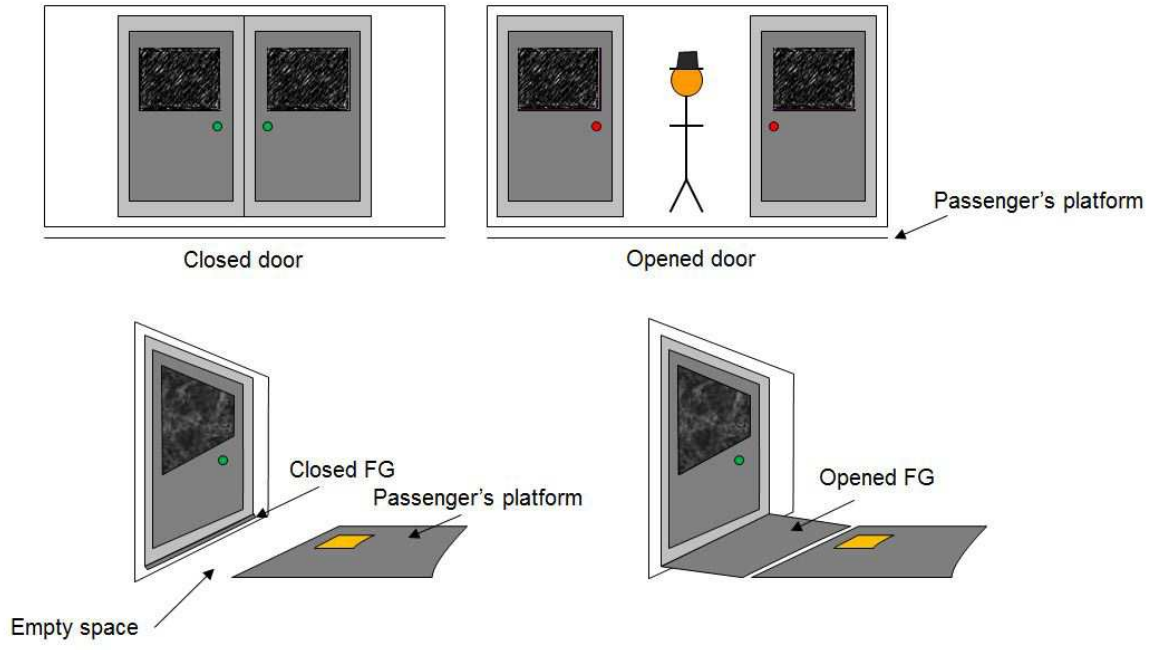
3.3.1 Design objectives

The system consists of two main parts. (1) The operative part (O.P) shown in figure 3.1, represents the physical door (D) and filling gap (FG). The filling gap is used to fill the empty space between the train coach and the platform. It is used to facilitate the access of people

in wheelchairs or mothers with baby strollers. (2) The control part (CC.P) which is the logic system responsible of commanding the (O.P) with respect to the environment signals. The operative part behaves according to the commands arriving from the CC.P and sends notifications, which are read by the control part.

Our work concerns only the control part, while the operative part is considered only by the logical abstraction of its physical state: sensors are read as Boolean input variables, and commands are issued as Boolean output variables. The objective of this case study is to build the

FIGURE 3.1: Physical environment of the train in the station



control/command function for the passenger access system, by reusing three previously defined COTS:

- the door controller $Manage\_open\_close$ COTS, illustrated in figure 3.2, handles the physical signals for opening and closing the door. Upon request ($Demand\_open$), it issues the command to start the door engine at a "fast" speed. When the door stop point becomes near, a sensor is activated and upon that event, the control sets the door engine to a "low" speed. The door stop point is signaled by a stop sensor. The reverse process is achieved similarly;
- the filling gap controller $Manage\_FG$ COTS, illustrated in figure 3.5 deploys and retracts the filling gap, upon request. The deployment and retraction are performed by controlling adequately the filling gap engine, and by reading the filling gap stop point sensors.

TABLE 3.1: Manage_open_close signals' signification

Signal	Signification
Demand_open	Opening door request
Demand_close	Closing door request
DemandUrgentUnlock	Urgent close request sent only in emergency cases
CmdFunctionPanels	command the panels to function
CmdSensPanels	the value of this signal determines the direction of the panels movement, (open panels if 0, close panels if 1)
CmdSpeedPanels	the value of this signal determines the speed of the panels movement, (slow movement if 0, fast movement if 1)
SnsApproachClose	a sensor to indicate the imminent closing of the panels
SnsApproachOpen	a sensor to indicate the imminent opening of the panels
SnsClose	a sensor to indicate the full closing of the panels
SnsOpen	a sensor to indicate the full opening of the panels
SnsClosedLocked	a sensor to indicate that doors are locked
CR_open	the value of this signal corresponds to the Sns_open which needs to be transferred to SEQ_DOOR COTS
CR_close	the value of this signal corresponds to the Sns_close which needs to be transferred to SEQ_DOOR COTS

- the $Open\_authorization$ COTS determines whether the doors may be open, according to the train state and the security requirements.

By combining these functions, it is required to build a passenger access control system featuring the following requirements:

- for security reasons, the door should not be open if the filling gap is not deployed, otherwise accidents can happen;
- when closing doors, there should be a delay between the closing request and the closing action. This delay is fixed by the specification document at 2 seconds.

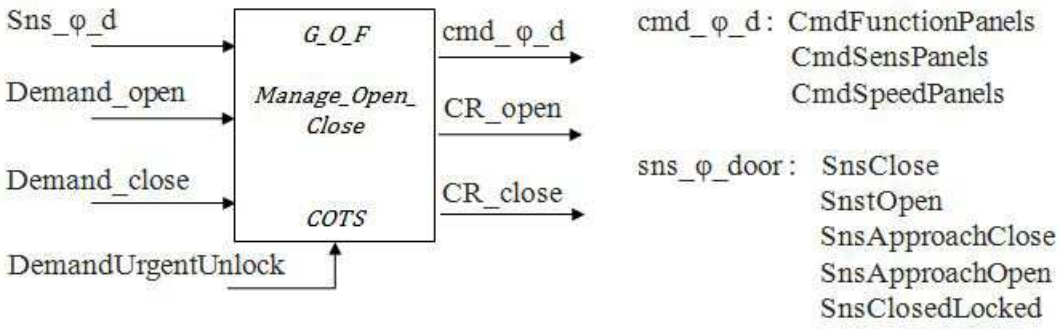
We attempt to establish these desired requirements by using both model checking and EDCS. This produces a new $Passenger\ access$ COTS, featuring new enforced guarantees.

3.3.2 Structural description of the available COTS

The signification of the component's signals is given exhaustively in table 3.1.

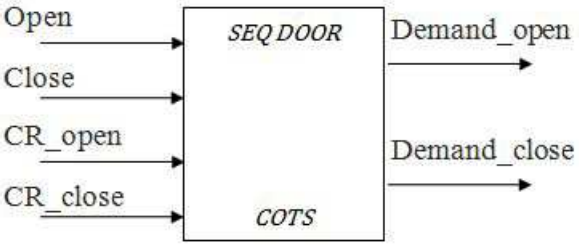
The specific timing constraints that apply at door closing are actually handled by another COTS available: SEQ_DOOR, shown in figure 3.3. This component forwards open requests, but delays close requests. The actual delay mechanism is target specific. For FPGA, explicit timing

FIGURE 3.2: Manage_open_close COTS



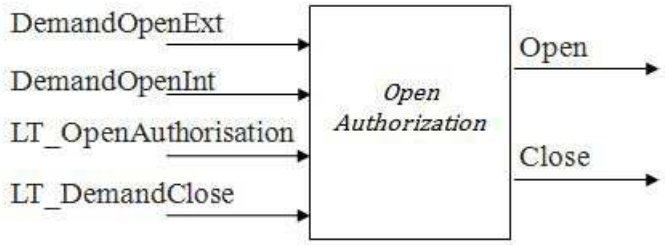
constraints are handled by counters, which need to be configured with respect to the final clock frequency at implementation time.

FIGURE 3.3: Operational constraint SEQ_DOOR



The Open-authorization component shown in figure 3.4 provides the COTS Manage_open_close with the open and close requests. It's inputs, output signals' signification is illustrated in table 3.2.

FIGURE 3.4: Open authorization component



The filling-gap control component Manage_FG shown in figure 3.5 controls the deploying (opening) and withdrawing (closing) of the physical filling gap. It receives a deploy / withdraw request from the train driver $(Deploy, Withdraw)$ respectively and reads the values of sensors $(SnsFGout, SnsFGin)$, which indicate the full opening and the full closing of the physical filling-gap, in order to generate the commands of the physical filling-gap. The signals' significations of the component are explained in table 3.3

TABLE 3.2: Open Authorization signals' signification

Signal	Signification
DemandOpenExt	a demand to open the door by the passenger from the external
DemandOpenInt	a demand to open the door by the train conductor
LT_DemandOpenAuthorisation	an authorization signal to allow the opening demands to be treated
LT_DemandClose	a demand to close the doors by the conductor
Open	the final opening request signal (authorized opening weather it is internal or external demand)
Close	the close request signal which will be treated by $Manage\_Open\_Close$ in order to command the doors to close

FIGURE 3.5: Manage_FG COTS

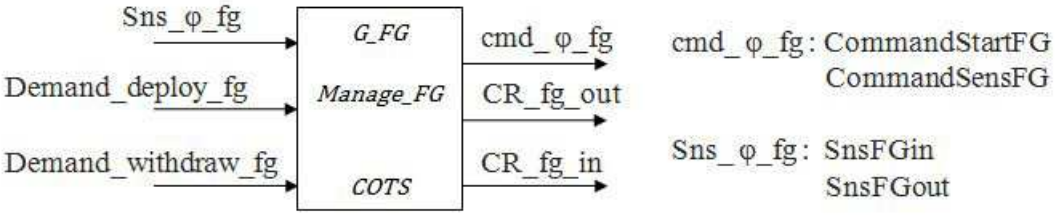


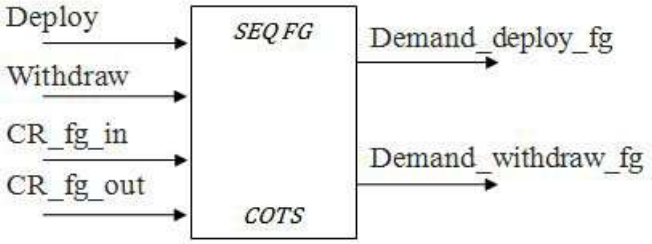
TABLE 3.3: Manage_FG signals' signification

Signal	Signification
Demand_deploy_fg	deploying filling-gap request
Demand_withdraw_fg	withdrawing filling-gap request
CommandStartFG	the physical command to function the filling-gap
CommandSnsFG	the value of this signal determines the sens of the filling-gap movement the values (0,1) correspond to withdrawing, deploying movement respectively
SnsFGin	indicates the full withdrawing of the filling-gap
SnsFGout	indicates the full deploying of the filling-gap
CR_fg_out	the value of this signal corresponds to the value of SnsFGout
CR_fg_in	the value of this signal corresponds to the value of SnsFGin

We suppose that the internal functionality of each component is correct and no local errors occur. The global behavior of the component's assembly has to obey a global safety temporal property which guaranties that the door must not start opening before the full opening of the filling-gap.

The same timing constraint could be provided for the filling gap. It is implemented the component SEQ_FG shown in figure 3.6. The specification document does not require any waiting for the filling gap, but for genericity reasons, this COTS is used with a waiting time configured to 0.

FIGURE 3.6: Operational constraint SEQ_FG



3.3.3 Behavioral description of the COTS assembly

When the train is in $parking\ mode$ at the station the driver in his cabin commands an automatic opening and closing of the doors and the filling-gaps. The driver sends requests (Demand_open, Demand_close, Demand_deploy_fg, Demand_withdraw_fg) and let the control system send the commands to the physical parts in the right moment in order to keep the passengers access into the train a safe operation.

The component Manage_open_close controls the panels movement direction and speed, according to the information sent by the physical sensors Sns_φ_d. It sends commands to the physical environment ($CmdFunctionPanels, CmdSensPanels, CmdSpeedPanels$) regarding the driver requests. $CmdFonctionPanels$ commands the panels of the door to function, $CmdSensPanels$ commands the direction of the door panels (opening direction / closing direction), $CmdSpeedPanels$ commands the speed of the panels movement (quick / slow movement).

Manage_FG component controls the movement of the filling-gap and its direction, basing on information received from physical sensors Sns_φ_fg. It returns signals at the end of the operation CR_fg_in, CR_fg_out which indicate whether the filling-gap is withdrawn of deployed.

Authorization opening component produces a control opening and closing of applications sent by the driver.

In order to preserve safe behavior of the system, the door must not start opening before a full opening of the filling-gap. Since the components are not principally designed to work together the fact of combing them to each other will not take in consideration this requirement, thus, a global safety property must be specified and enforced over the assembly of the components in order to let it operate safely.

In the rest of this chapter we design -step by step- the safe passengers' access system through our design method.

3.3.4 Modeling and formal specification

The first step in the design method is the modeling of the control-command part of each component. First we model each individual component and then the assembly of COTS.

3.3.4.1 The stand-alone door component

With respect to the model of a stand-alone component, the door component can be modeled as follows :

$$C^d = \{I^d, M^d, A^d, G^d\} \tag{3.1}$$

Where :

- I^d is the set of interface inputs and outputs.
  We model the interface using Boolean variables as follows :
  $I^d = \{SnsClose, SnsOpen, SnsApproachClose, SnsApproachOpen, SnsClosedLocked, Demand\_open, Demand\_close, CmdFunctionPanels, CmdSensPanels, CmdSpeedPanels\}$;
- A^d is the set of environment preconditions: (1) the door is eventually requested to open, (2) the door is eventually requested to close.
  A^d = $\{a_1^d,\ a_2^d\}$
  The formal representation of these preconditions is given by two PSL assertions :
  - a_1^d : $always\ eventually(CmdFunctionPanels \wedge CmdSensPanels \wedge CmdSpeedPanels)$
  - a_2^d : $always\ eventually(CmdFunctionPanels \wedge CmdSensPanels \wedge \neg CmdSpeedPanels)$
- G^d is the set of environment post-conditions : (1) if the door is requested to open then it eventually opens, (2) if the door is requested to close then it eventually closes. G^d = $\{g_1^d,\ g_2^d\}$
  The formal representation of these preconditions is two PSL assertions :
  - g_1^d : $always(Demand\_open \rightarrow eventually(CmdFunctionPanels \wedge CmdSensPanels \wedge CmdSpeedPanels))$
  - g_2^d : $always(Demand\_close \rightarrow eventually(CmdFunctionPanels \wedge CmdSensPanels \wedge \neg CmdSpeedPanels))$
- M^d is the behavioral model of the door component. It is modeled by two finite state machines illustrated in figure 3.7. The top FSM models the behavior of the Manage_open_close component. The middle FSM models the behavior of SEQ_DOOR component. The 2-second delay is modeled by a parametric 2-states finite state machine illustrated in figure 3.8.

FIGURE 3.7: Behavioral model of the door COTS

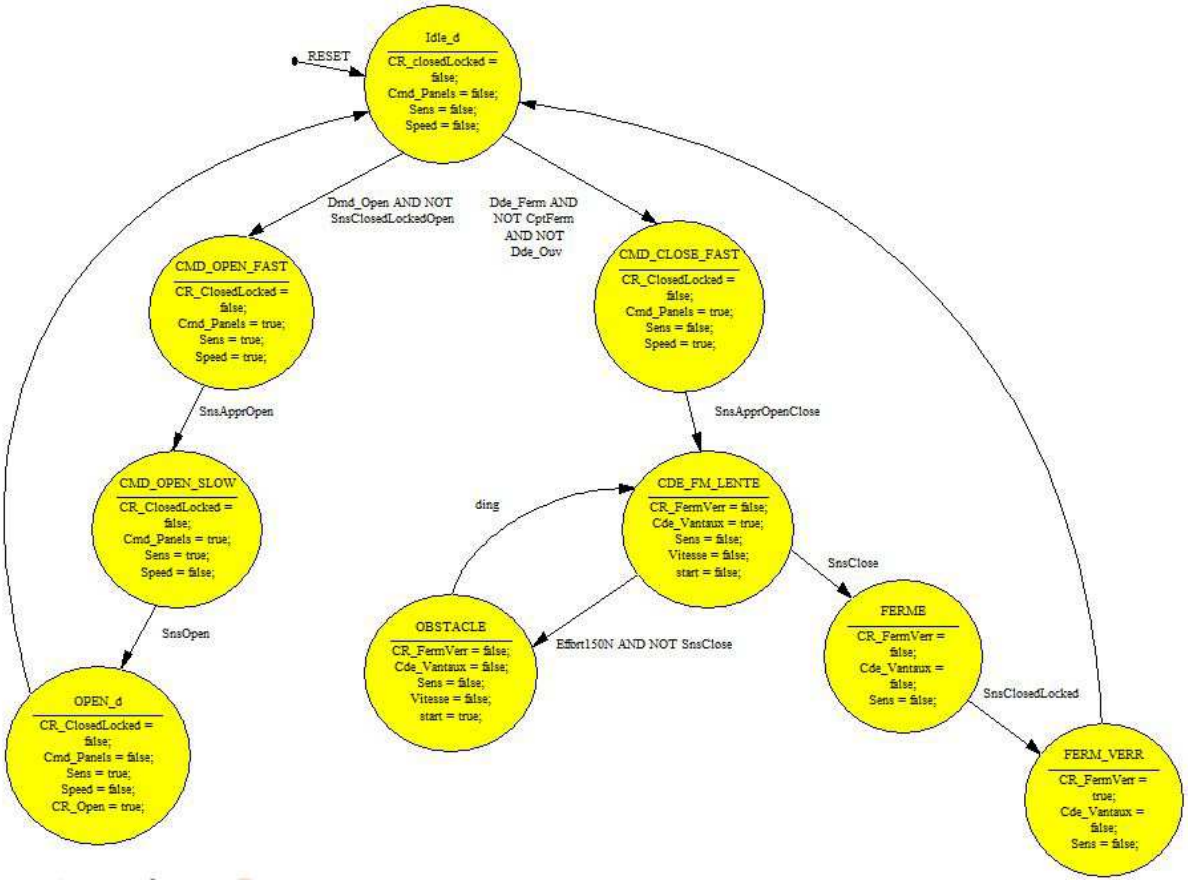
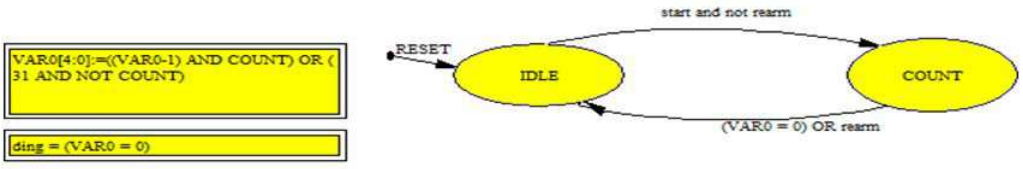


FIGURE 3.8: FSM model for 1s delay



3.3.4.2 Stand-alone filling-gap component

Similarly to the door component, we model the stand-alone filling-gap COTS:

$$C^{fg} = \{I^{fg}, M^{fg}, A^{fg}, G^{fg}\} \tag{3.2}$$

Where :

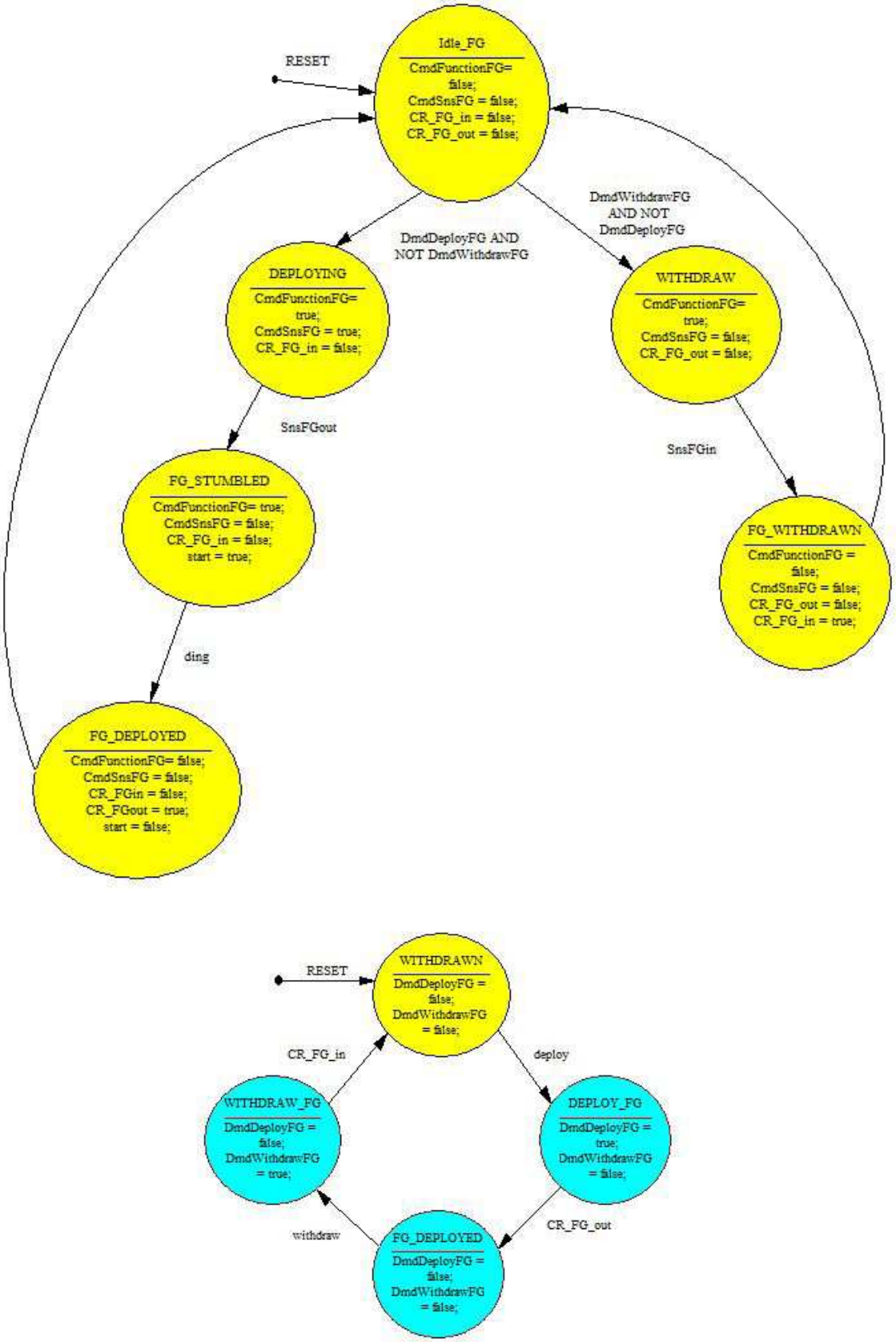
- I^{fg} is the set of interface inputs and outputs.
  We model the interface using Boolean variables as follows :
  $I^{fg} = \{(SnsFGin, SnsFGout, Demand\_deploy\_fg, Demand\_withdraw\_fg, CommandStartFG, CommandSensFG, CR\_fg\_out, CR\_fg\_in)\}$;
- A^{fg} is the set of environment preconditions which contains two precondition: (1) the filling-gap is eventually requested to open, (2) the filling-gap is eventually requested to close.
  A^{fg} = $\{a_1^{fg},\ a_2^{fg}\}$
  The formal representation of the preconditions is two assertions in PSL :
  - a_1^{fg} = $always\ eventually(Demand\_deploy\_fg)$
  - a_2^{fg} = $always\ eventually(Demand\_withdraw\_fg)$
- G^{fg} is the set of environment post-conditions which contains two post-condition : (1) the filling-gap eventually opens, (2) the filling-gap eventually closes. G^{fg} = $\{g_1^{fg},\ g_2^{fg}\}$
  The formal representation of the preconditions is two assertions in PSL :
  - g_1^{fg} = $always(Demand\_deploy\_fg \rightarrow eventually\ CommandStartFG \wedge CommandSensFG)$
  - g_2^{fg} = $always(Demand\_withdraw\_fg \rightarrow eventually\ CommandStartFG \wedge \neg CommandSensFG)$
- M^{fg} is the behavioral model of the stand-alone component, it is represented as a finite state machine illustrated in figure 3.9.

  Together, these automata build the behavioral model of the filling-gap control-command part.

The open authorization component is modeled as follows: M^{O_AUT} = $\{I^{O\_AUT}, M^{exe}, A^{O\_AUT}, G^{O\_AUT}\}$ where:

- I^{O_AUT} = $\{DemendOpenExt, DemandOpenInt, LT\_OpenAuthorization, LT\_DemandClose, Open, Close\}$;

FIGURE 3.9: Behavioral model of the filling-gap COTS



- A^{O_AUT} = $\{always\ eventually(DemendOpenExt \vee DemandOpenInt),$
  $always\ eventually(LT\_close)\}$;
- G^{O_AUT} = $\{always(DemendOpenExt \vee DemandOpenInt \rightarrow eventually(Open)),$
  $always(LT\_close \rightarrow eventually\ Close)\}$.

The behavior of this component is purely combinational, its behavioral model M^{exe} is expressed by its output functions:

- M^{exe} is only given by its output functions assigning $Open$ and $Close$ where:
  $Open$ = $(DemandOpenExt \vee DemandOpenInt) \wedge (LT\_OpenAuthorisation)$
  $\wedge(\neg LT\_DemandClose)$;
  $Close$ = $LT\_DemandClose$.

3.3.4.3 The Door / Filling-gap assembly

The goal of the assembly is to create a global system which contains the door and the filling-gap and ensure a safe behavior of the assembly with preserving the local characteristics of each component.

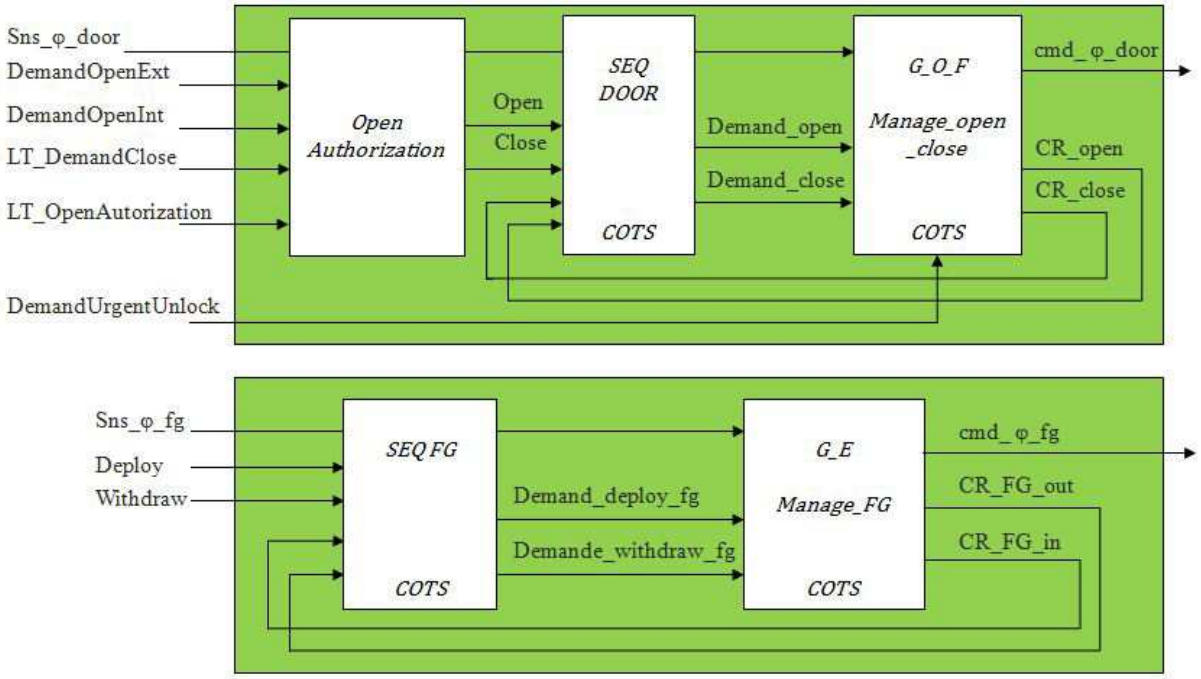
The assembly of components is illustrated in figure 3.10 and the assembly model is formalized as follows:

$$C^{asm} = \{I^{asm}, M^{asm}, A^{asm}, G^{asm}\} \tag{3.3}$$

Where :

- I^{asm} is the set of interface inputs and outputs of both the door, the filling-gap and the open authorization from which the internal signals are excluded.
  I^{asm} = $I^d \cup I^{fg} \cup I^{O\_AUT} \setminus I^{intern}$.
- A^{asm} is the set of environment preconditions which contains the union of the door and the filling-gap preconditions.
  $$A^{asm} = A^d \cup A^{fg}$$
- G^{asm} is the set of the assembly environment post-conditions which is the union of post condition of both components.
  $$G^{asm} = G^d \cup G^{fg}$$
- M^{asm} is the behavioral model of the assembly.

FIGURE 3.10: Door_Filling-gap assembly



3.3.4.4 Functional requirements of the door - filling-gap assembly

To ensure safe behavior of the door and the filling-gap together in the train platform a global safety property must be respected by the assembly model. The property is (P^{asm}) : a closed door must not start opening before a full extraction of the filling gap and an extracted filling-gap must not start closing before a full closing of the door. These requirements are modeled in PSL as follows:

$$FG\_deploy\_before\_door\_open : always(CR\_fg\_out\ before\ CR\_open)$$
$$door\_close\_before\_FG\_retract : always(CR\_close\ before\ CR\_fg\_in)$$

The global property P^{asm} is the Boolean conjunction of the predicates given above.

3.3.5 Error detection

After modeling the assembly of components, the designer should verify that each component still satisfies its local guarantees, and also verify the satisfaction of the global properties which must be respected by the assembly. We apply model checking by using the Cadence SMV tool [15].

Hence, we verify the local guarantees (G^d) of the door, the local guarantees (G^{fg}) of the filling-gap, and the global property (P^{asm}) of the assembly under the preconditions of the assembly (A^{asm}). The model checking attempt gives a negative result and a counterexample which indicates the violation of the the global safety property (P^{asm}). The counterexample shows that the closed door can open even if the filling-gap is withdrawn, which is quite obvious for this example, because the door and filling gap controllers were not designed to operate together.

As the safety global property is not satisfied by the components assembly, the designer attempts an automatic correction step. The violation of the safety property is considered a global design error.

3.3.6 Error correction

3.3.6.1 Controllable variables

To prepare for the automatic correction step the designer should construct (X_c) the set of controllable variables and (X_{uc}) the set of uncontrollable variables. For the available set of interface variables figuring in the counterexample we construct the initial set of controllable variables:

$X_c^{init} = \{DemandOpenInt, DemandOpenExt, LT\_OpenAuthorization, SnsClose, SnstOpen, SnsApproachClose, SnsApproachOpen, SnsClosedLocked, Deploy, Withdraw, SnsFGin, SnsFGout, CommandStartFG, CommandSensFG, CmdFonctionPanels, CmdSensPanels, CmdSpeedPanels, CR\_fg\_out, CR\_fg\_in, CR\_open, CR\_close$

We apply the elimination rules:

- remove all the sensor variables:
  $\{SnsClose, SnstOpen, SnsApproachClose, SnsApproachOpen, SnsClosedLocked, SnsFGin, SnsFGout\}$;
- remove all state variables:
  $\{CR\_fg\_out, CR\_fg\_in, CR\_open, CR\_close\}$;
- remove all output variables which command the operative part of the system :
  $\{CommandStartFG, CommandSensFG, CmdFonctionPanels, CmdSensPanels, CmdSpeedPanels\}$

As a result, we obtain the set of controllable variables:

$$X_c = \{DemandOpenExt, DemandeOpenInt, LT\_DemandClose, Deploy, Withdraw\}$$

TABLE 3.4: Controllable variables and their environment corresponding

Controllable variable	Environment variable
$DemandOpenExt$	$O\_DemandOpenExt$
$DemandOpenInt$	$O\_DemandOpenInt$
$LT\_DemandClose$	$O\_LT_DemandClose$
$Deploy$	O_Deploy
$Withdraw$	$O\_Withdraw$

The remaining interface variables are considered uncontrollable: $X_{uc} = X_c^{init} - X_c$.

3.3.6.2 Correcting controller generation

We provide the DCS tool with three elements: (1) the behavioral assembly model. (2) the sets of controllable and uncontrollable variables X_c, X_{uc} respectively. (3) the global property $\{P^{asm}\}$. We obtain a controller in the form of a vector of Boolean functions, each assigning one controllable variable.

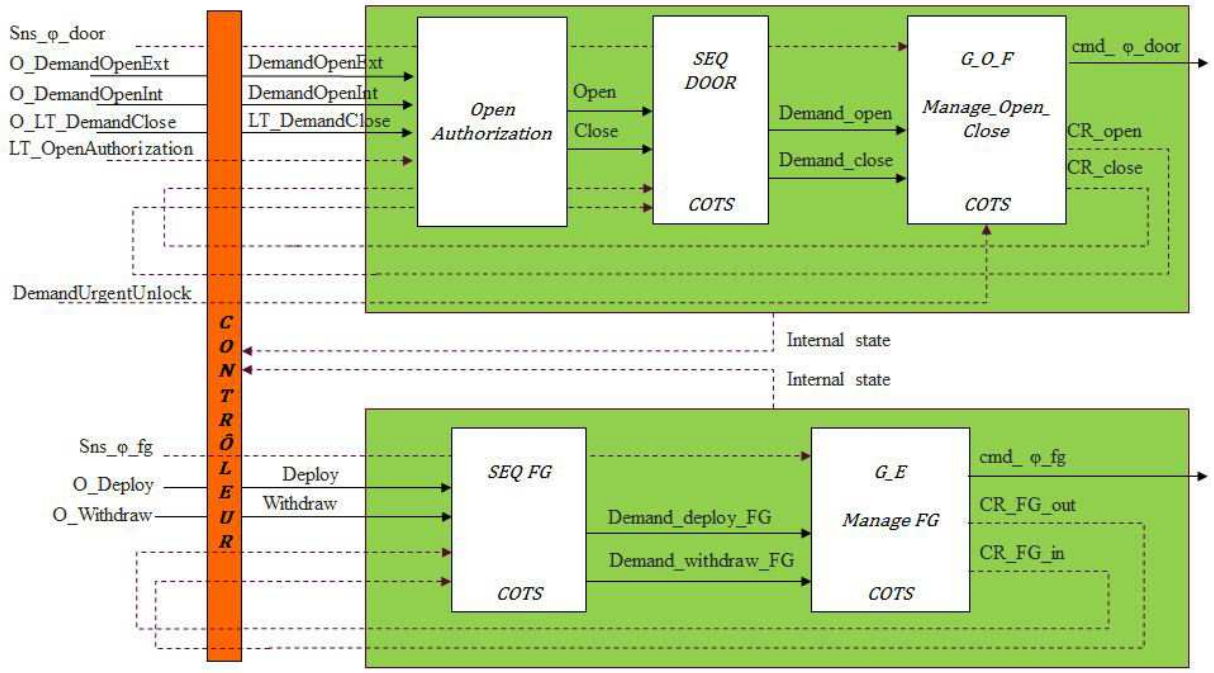
For our system, the correcting controller we obtain controls the values of $DemandOpenExt$, $DemandeOpenInt$, $LT\_DemandClose$, $Deploy$, $Withdraw$. Each of these variables is connected to an environment variable prefixed by $O_variable_name$ the table 3.4 illustrates each controllable variable and its corresponding environment controllable. According to the global state of the controlled system, and the values of uncontrollable variables the controller decides whether the environment values are to be forwarded or masked by the controller, as explained in Chapter 2. The code of the controller provides the expression for the 5 Boolean control functions, unfortunately unreadable due to the number of variables involved.

The assembly obtained by the controller synthesis allows several possible implementations of the compound function door / Filling-gap. The presence of the controller allows the arrival of the opening and deployment requests in any order, with the guarantee of a correct sequence, according to P^{asm}.

Figure 3.11 illustrates the assembly of the correcting controller to the original control-command system.

Overview of the generated controller. An extract of the generated controller is provided below. The generation produces synthesizable VHDL. The controller is constructed as an entity, **top_controller** featuring input and output ports. Its behavior is expressed by an associated architecture named **rtl**. The controller is a function without side effects, implemented by a VHDL *combinatinal process*, which continuously assigns the outputs of **top_controller** each time an

FIGURE 3.11: Passengers' access controlled system



input changes. There are two types of assignments: variable assignments ":=" are local to the process (variables cannot be referenced from outside a process); signal assignments "<=" assign the outputs of the entity.

```
entity top_controller is
  port (CR_door_open : in bit; CR_door_close : in bit;
    CR_FG_in : in bit; CR_FG_out : in bit; error_FG_door_open : in bit;
    error_door_FG_close : in bit; o_dmdopenInt : in bit;
    o_dmdopenExt : in bit; o_LT_demandeClose : in bit;
    o_deploy : in bit; o_withdraw : in bit;
    -- controllable variables are ouputs of the controller:
    demandOpenInt : out bit; dmdopenExt : out bit;
    LT_demandeClose : out bit; deploy : out bit; withdraw : out bit);
end entity top_controller;
architecture rtl of top_controller is
begin
    update : process (CR_door_open,
                  CR_close, CR_approche_open, CR_approche_close,
                  CR_door_close, CR_FG_in, CR_FG_out,
                  LT_OpenAuthorization,
                  ...,
                  FG_2_sHell1_FG_sreg, p_dmdOpenAuthorization,
                  o_demandOpenInt, o_LT_demandeClose, o_deploy,
                  o_withdraw)
    variable sub_51 : bit;
    variable sub_50 : bit;
    ...
variable sub_660 : bit;  variable DemandOpenInt : bit;
  begin
    sub_51 := '0'; sub_660 := '1';
    ...
    case FG_2_sHell1_FG_sreg1l0R is
      when '1' => sub_2092 := '0';
      when '0' => sub_2092 := sub_1844;
    end case;
    demandOpenInt <= sub_2483;
    deploy <= sub_2486;
    withdraw <= sub_2487;
  end process update;
end architecture rtl;
```

3.3.7 Verification of controlled passenger access system

The controller synthesized in the former step ensures that the controlled system satisfies the safety properties provided to the DCS tool. These properties are enforced by the controller, by assigning the controllable inputs of the COTS assembly. However, controlling an input can generate a conflict with respect to other requirements expressed over the same input. Typically, as DCS is unable to process environment preconditions expressing liveness, such preconditions can be broken by the controller, which is totally unaware of liveness requirements. Hence, by breaking a liveness assumption (or pre-condition), all liveness guarantees making this assumption are automatically broken. Thus, we verify the list of the assembly guarantees (G^{asm}) under the assembly preconditions (A^{asm}).

We also verify the absence of "event-invention". For example, we verify that the controlled system does not command the doors to open or close without the driver's request. Similarly, we verify that the filling-gap is never commanded to deploy or withdraw without a driver's request.

Let us verify the assembly guarantees G^{asm}, and some liveness and passiveness properties. The liveness properties required for the controlled system are the follows :

- If the filling-gap is requested to withdraw it will finally be withdrawn
  $live_1 : always(Deploy \rightarrow eventually\ \ CR\_FG\_out)$
- If the door is demanded to open by the train conductor or by a client it should eventually obey the request and open
  $live_2 : always((DemandeOpenInt \vee DemandeOpenExt) \rightarrow eventually\ \ CR\_open)$
- If the door is requested to close, the panels should eventually become fully closed
  $live_3 : always(LT\_DemandClose \rightarrow eventually\ CR\_close)$
- If the filling-gap is requested to close, it should eventually close
  $live_4 : always(withdraw \rightarrow eventually\ \ CR\_FG\_in)$

The passiveness controller properties are formalized in PSL as follows:

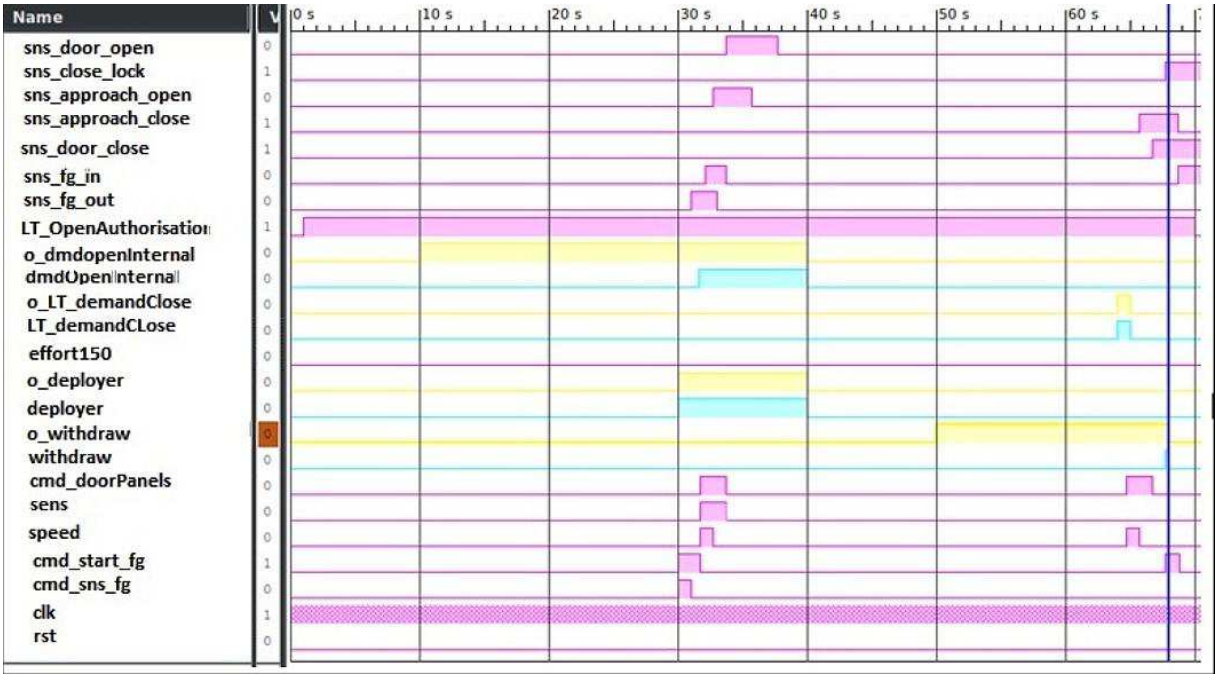
- $passive_1 : always\neg(DemandOpenExt \wedge \neg O\_DemandOpenExt)$;
- $passive_2 : always\neg(DemandOpenInt \wedge \neg O\_DemandOpenInt)$;
- $passive_3 : always\neg(LT\_DemandClose \wedge \neg O\_LT\_DemandClose)$;
- $passive_4 : always\neg(Deploy \wedge \neg O\_Deploy)$;
- $passive_5 : always\neg(Withdraw \wedge \neg O\_Withdraw)$;

The verification of the controlled system provide positive results: all the above properties are satisfied. Thus we conclude that the controlled system of the door and filling-gap assembly satisfies the safety, passiveness properties and liveness guarantees defined by the designer and it could eventually be validated.

3.3.8 Simulation

The controller is assembled manually to the original design model. At this step, the simulation acts on synthesizable VHDL descriptions automatically translated from the designed Control-Build models. The different translation steps and the tool-chain used are explained below and illustrated in figure 3.13. The simulation trace obtained shows the safe behavior of the Passenger access system. In figure 3.12, we can notice that at the second 10 the designer requests to open the door while the filling-gap is not yet deployed. The controller prevents this request until the second 31 where the filling-gap sensor provides the information that it is fully deployed and here the request to open the door is allowed to pass to the COTS. The reader can observe the actions taken by the correcting controller. Regardless of the order in which the driver requests the doors and filling gap operation, they always operate in the right order.

FIGURE 3.12: Simulation of the controlled Door_Filling-gap system



3.4 Comparison: assembly controlled synthesis vs. the initial assembly

On the original system, the order of control actions can be any interleaving of the filling-gap deployment / doors opening / doors closing / filling-gap withdrawal. The automatically generated controller ensures the implementation of the same operations in the desired safe order specified by P^{asm}.

In the simulation presented above, the doors and filling gap are explicitely handled by the driver, and they can be operated one after the other, regardless of the order. Another possible (and trivial) choice of implementation would enable the opening / deployment simultaneously on one hand, and closing / withdrawal on the other hand and the controller ensures the correct sequence of these actions. Thus, the COTS assembly obtained by DCS allows several possible implementations of the doors / filling-gap composite function. The presence of the controller allows the train conductor to request the opening and deployment in any order, with the guarantee of a correct sequence.

3.5 Implementation

The figure 3.13 illustrates the tool-chain used throughout the safe design process. According to the Ferrocots project' requirements, the Controlbuild tool is used for designing either individual COTS, or COTS assemblies. This environment supports assumption and guarantee embedding, but cannot handle temporal logic yet. Controlbuild is able to handle several kinds of models and translate them into the same PLC/Open pivot representation. In this project, we advocate the only use of state/transition models, graphically expressed as Grafcet models, and dataflow models featuring logical gates and state variables.

Design models are automatically translated into synthesizable VHDL, via the hdlgen tool, which is a part of Controlbuild. At the end of this PhD work, all this tool-chain has become unavailable to us, due to licensing issues. It has been replaced other free of access tools for modeling and simulation: Xilinx StateCAD can model graphically synchronous state based designs, simulate them, and translate them into synthesizable VHDL.

Our toolchain starts from synthesizable VHDL designs. We are able to read and translate them into several tool-specific formats: the z/3z, SMV and nuSMV formats all have in common an input/state variable/transition function/output representation.

The first step is achieved by the Design Compiler (DC) tool. DC is a commercial tool provided by Synopsys. This compilation step, also known as RTL synthesis, performs the transformation of random synthesizable VHDL, possibly containing high level programming mechanisms, into

a set of state variable and their corresponding transition functions. Everything is converted into the Boolean representation.

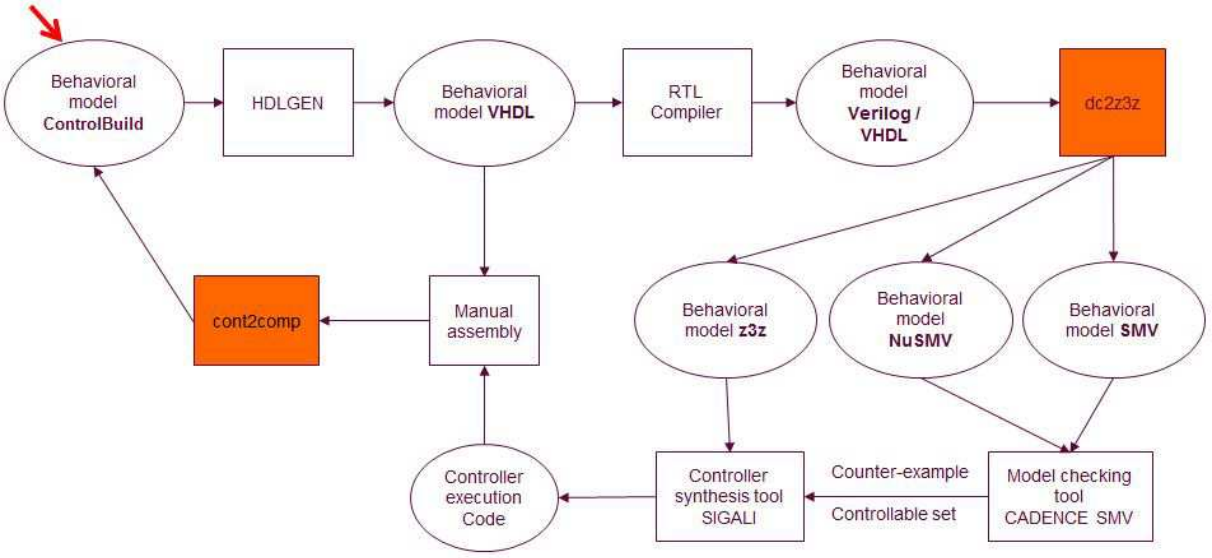
This result is further compiled and translated into the tool specific formats, in order to be able to feed the same design to either DCS or formal verification. This is achieved by the dc2z3z tool, developed during this Phd work.

The DCS and the formal verification are achieved using academic free tools: Sigali for DCS, Cadence SMV (or alternatively, NUSMV) for formal verification.

The synthesized controller is assembled manually to the original COTS, once translated into synthesizable VHDL. The controller can also be re-translated into ControlBuild through cont2comp tool (locally developed) for global simulation. This step has not been tested. All simulations were performed on VHDL models.

In the current implementation state, the functional requirements for either formal verification or DCS are manually handled by the designer, at the time of verification or synthesis. Recall that this association can be done in a more elegant way by using the Controlbuild tool. However, this is left as a future development possibility of this work.

FIGURE 3.13: Chain of design tools



3.6 Conclusion

We have shown in this chapter the applicability of our proposed method on a real industrial railway system. The system consists of two hardware COTS, a door and a filling-gap, commanded by their control-command parts. Through the proposed method we have modeled the

control-command part of each component and created the assembly of those COTSs. We have verified the assembly through the model checking method, (Cadence SMV tool), to detect the design errors. After that, we have managed to use the discrete controller synthesis method, (the Sigali 2.4 tool), to automatically generate a glue in order correct the discovered errors. Then, we verified the resulting controlled system to check the preserving of the local guarantees of the components and the passiveness of the glue. Finally, we have simulated the controlled system to ensure an admissible behavior of the final system before saving it in a formal COTS library or installing it on an FPGA chip.

It is interesting to note that the application of formal tools to the models presented above did not face performance issues. This is certainly due to the fact that the models we have handled remain medium or even small sized. However, it is important to highlight that even for such models, the automatic generation of correcting code is much faster and less error-prone than writing such code manually. Whatever the future developments on the DCS performance, it shall remain bounded by its exponential complexity, either in space or in time. So it is reasonable to assume that, just like model checking tools, DCS shall be able to handle small to medium sized designs but is shall be difficult to go beyond.

Still we argue that at the human designer's level, handling a small set of COTS, the DCS tool is more relevant than at a system level: even for "small" designs, manually finding and correcting design errors can be tricky and the time required is unpredictable, whereas DCS is guaranteed to succeed in a reasonable amount of time.

General conclusion

The work presented in this thesis concerns the safe design of hardware embedded systems, with applications to the design of train automation systems. Such systems are considered safety critical, and the aspect developed in this work is the safety with respect to design errors. This usually calls for specific design methods and tools, featuring that formal tools have a great power in establishing correctness or finding design errors.

The safe design hardware systems requires a thorough verification step. In practice, verification can count for up to 70% of the total time of the design process. This is due the cruciality to finding most design errors, on the one hand, and the time-costly process to do so on the other hand.

- Building a system basing on pre-built components, COTS, is an important development in the design process, since it economize loads of efforts for designing, documenting and coding components. Even though, integrating a strange component in a design or building a system over separately built components risks of confronting incompatibility and bad synchronization between the communicating components.
- Modeling a hardware system with a very abstract model enlarges the gap between the model and the real system. Thus, a model close to the reality is demanded to raise the trustworthy of the study's results.
- The size of hardware systems, (medium, large to extra large systems), makes a manual detection of the design errors using the *theorem proving* such a tough mission and augments the probability of time and efforts wasting. Applying *the guided simulation* method is such a good technique to automatically verifies the design for certain scenarios, but it is still considered not sufficient to cover all the potential error cases. The model checking is yet the most suitable technique to detect all the system predefined errors, thanks to the exhaustive search it makes on the system's state space which can detect any error state and provide an analytic counterexample about it. This method is very well in finding errors, however, the question raised is: "Who is going to correct the error detected and how?".

- The manual correction of errors is erroneous critical mission by itself. However, diving in a huge code and trying to correct it is critical enough to preventing any designer of making any modification in a design to avoid causing other errors. Any modification in the system design can have serious affects on other parts in the system and tracking such affection is not easy, especially in industrial hardware systems.

- The formal design of critical embedded hardware COTS-based systems evokes the indispensability of creating formal models of the system components, where formal models are, to our knowledge, limited to software COTS-based systems.

- The discrete controller synthesis technique proposed by Ramadge and Wohman is quite useful for synthesizing new requirements over discrete event event systems, but, this technique is preserved to researches and limited by the academic assumptions and contexts.

In attempt to tackle the above points we come up with a safe design method for hardware embedded systems based on COTS. It takes in consideration the above problems and aims to provide an integral solution, which takes on its charge to do the most hard work of the designer, without forgetting the importance of the human role to validating the system before installing it in the real environment. The method with its five steps takes solicit to the maximum the advantages provided by some existing methods and combine them together. The method is addressed to the domain of commercial off the shelf components. Since these components exist in the market in form of generic COTS libraries, where components are stocked in form of executable code and textual documentation, our method aims to build a formal library for COTS, where each component is formally defined by its interface, preconditions and post-conditions.

The designer can use the method either to add new properties to a COTS and save it in the formal library in order to enrich the library and use this component later, or to build a subsystem/system over COTS.

The method proposed stands on five successive steps:

The first step is the formal modeling. It concerns modeling the COTS in question; modeling their interfaces, environment preconditions, and post-conditions, whereas the behavioral model is already given to the designer with the textual documentation. If the designer's goal is to build a subsystem/system over COTS, then the components are formally assembled together A formal hardware COTS assembly is proposed in this step, it is similar but not identical to the software COTS formal composition. The assembly architecture, influencing the assembly interface, preconditions and post-conditions, remarkably characterizes the hardware formal assembly model, the The additional local and/or global properties are also modeled in this step.

The second step of the method concerns the detection of errors. Using the model checking, the designer verifies whether the original design of the COTS-assembly still guaranties the post-conditions of the assembled COTS, or the composition have yielded some errors. In this step also the designer verifies the satisfaction of the additional properties over a COTS or an assembly of COTS. The model checking features of providing a counterexample that illustrates where a property was violated, we profit of this counterexample and consider it a heuristic information source for the designer to choose the controllable candidates' set necessary for the automatic correction of the error.

The third step in the method, is the automatic correction of the detected errors. The environment aware discrete controller synthesis technique *EDCS* attempts to correct automatically the design errors with regard to the COTS/COTS-assembly's environment. Since the controllability of control-command COTS' inputs cannot be predefined as in the case of Ramadge and Wohman framework we propose certain heuristic rules and three architectures, typical for hardware context, helping the designer decide the signals controllability, in order to generate the most reliable controller. Since the design of COTS-based systems distinguishes between the local properties and the global properties, we define two kinds of controllers; the patch and the glue which corrects the local/ global errors respectively. By using the EDCS the designer corrects the safety errors only, if possible, but not the liveness ones, since the discrete controller synthesis methods available, till the date of this report, cannot synthesize liveness properties. Thus, if a liveness error is detected then a manual modification is inevitable, which can be considered a weakness point in the method.

The fourth step is the verification of the controlled system. It focuses on verifying the liveness of the system and the achievement of certain behaviors, since the EDCS cuts some behaviors to prevent the arrival to the error states. The passiveness of the generated controller is also verified; this verification is needed in Ramadge and Wohman framework as the controller in that case can only prevent/delay events, whereas in EDCS the controller which is a Boolean function can provide any Boolean value to the controlled signals, thus, it is possible that it evokes some dangerous behaviors. However, we verify the controlled system against certain properties to ensure that the controller does not invent events which are not given by the environment.

The fifth step is final one in our method, it is a simulation of the controlled system. We suppose that validating critical systems requires a human-eyed validation and simulation of certain scenarios, interesting for the designer, to make sure that the final version of the system before being installed in the physical environment or stocked in a formal library.

The application of our method on the passengers access system can partially validate its applicability in the real industrial domain. We still need to try some larger real systems.

The domain of COTS-based design system is large enough to bring to our minds many other questions about safety of a hardware design like: the components' redundancy and diversity in a design to provide a fault tolerant systems. There exist a technique [37] which can build a system from its functional behavior (safety and liveness requirements), but, this technique is not based on the DCS method. A perspective to our work is providing an enhancement to the EDCS method in order to achieve the synthesis of liveness properties.

Appendix A

Cahier des charges fonctionnel Système d'accès voyageurs

Références: EN 14752, TSI Loc and Pas, TSI PRM

1 Le système accès voyageur

Les portes et emmarchements mobiles sont à commande automatique avec blocage en fermeture et verrouillage mécanique pendant la marche du train. Le service des portes sera asservi à la sélection par l'agent de conduite du côté d'ouverture. L'autorisation d'ouverture sera conditionnée par l'information de vitesse du véhicule inférieure ou égale à $3km/h$ et la sélection du ou des côtés d'ouverture. Le fonctionnement d'ouverture des portes est du type libre service. Toutes les portes pourront être ouvertes individuellement de l'intérieur comme de l'extérieur du train sur demande d'un voyageur par appui sur le bouton poussoir d'ouverture, après autorisation préalable d'ouverture, donnée par l'agent de conduite. En cas d'absence d'énergie, un système d'ouverture manuel doit être prévu pour permettre le déverrouillage mécanique des vantaux rendant possible leur ouverture manuelle. La commande de fermeture de toutes les portes en même temps est provoquée par l'agent de conduite. Lorsqu'une porte n'est pas contrôlée fermée, tout déplacement de la rame (dérive ou traction) doit être impossible en mode normal. En service voyageurs, pour autoriser le départ du train et lors de la circulation du train, les portes sont bloquées et verrouillées en position fermée. Cette situation est appelée "maintien fermeture". L'ouverture manuelle des portes se fait avec assistance, après commande de préparation d'ouverture par le conducteur, par actionnement du système de déverrouillage. Une fois les portes ouvertes, la re-fermeture manuelle est possible.

L'ouverture devra s'effectuer selon le processus suivant :

- Ouverture par manoeuvre volontaire par le système de commandes des portes à disposition des voyageurs (lorsque les autres conditions d'autorisation sont réunies).
- Phase amortie sur les derniers centimètres de course, de façon à ce que les vantaux abordent les butées sans choc excessif.

L'ouverture des portes est conditionnée par le déploiement intégral préalable de l'emmarchement mobile. Ce déploiement sera considéré comme achevé lorsque l'emmarchement mobile, après avoir été en contact avec le quai ou en butée, se sera immobilisé après un recul à définir par le constructeur.

La fermeture des portes devra s'effectuer en deux temps :

- Phase rapide et régulière jusqu'à ce que l'écartement des vantaux soit d'environ 200 mm
- Phase lente avec amortissement, jusqu'à la fermeture complète

Après fermeture, les portes devront être normalement maintenues bloquées et verrouillées. La fermeture du comble-lacune intervient à l'issue de ce verrouillage.

Les portes voyageurs, doivent comporter des dispositifs de sécurité permettant :

- d'informer les voyageurs de la fermeture imminente des portes par un dispositif sonore et visuel, débutant au moins 1s avant le début de fermeture, et durant au minimum 2s.
- d'informer le voyageur de l'autorisation d'ouverture, par un signal lumineux clignotant.
- d'informer le voyageur de la prise en compte de sa demande d'ouverture, par un dispositif sonore et visuel, d'une durée minimale de 5s. La tonalité doit être différente de celle annonçant la fermeture des portes.
- de verrouiller mécaniquement chaque porte après sa fermeture,
- de n'autoriser le démarrage du véhicule que lorsque toutes les portes sont fermées et verrouillées ou condamnées,
- la détection de la présence de voyageurs entre les montants de cette porte,
- la détection d'obstacle,

L'obtention d'une ouverture de secours doit nécessiter l'une des deux actions suivantes, à vitesse $< 10km/h$:

- par le personnel muni d'une clé carré femelle sans rupture de plomb;
- par un voyageur en provoquant la rupture d'un plomb.

2 Choix techniques et interfaces fonctionnelles

Les composants, actionneurs et interfaces à considérer sont les suivants :

2.1 Porte

Système à deux vantaux pilotés par un moteur unique assurant les fonctions de fermeture et de verrouillage. Le moteur est équipé d'une partie opérative permettant de gérer le sens de rotation et la vitesse, et sera pilotée par les entrées binaires suivantes :

- Fonctionnement
- Sens (0 : Fermeture, 1 : Ouverture)
- Vitesse (0 : basse vitesse, 1 : vitesse normale)

Les vantaux et la transmission mécanique disposent des capteurs suivants :

- Effort $>$ 150N
- Fin de course ouverture
- Fin de course fermeture
- Approche fin de course ouverture
- Approche fin de course fermeture
- Position " fermée verrouillée "

A destination des voyageurs, les composants suivants sont mis en place :

- Effort $>$ 150N
- Fin de course ouverture
- Bouton poussoir lumineux d'ouverture
- Lampe d'annonce d'imminence fermeture
- Buzzer 1 Fermeture
- Buzzer 2 Ouverture

2.2 Emmarchement mobile

L'emmarchement est du type " déploiement total puis recul ", sans verrouillage spécifique. Son moteur est doté d'une partie opérative pilotée par les entrées suivantes :

- Marche
- Sens (0 : Fermeture, 1 : Ouverture)

Cette partie opérative met à 1 une sortie " effort maximal " lorsque l'intensité appelée dépasse un certain seuil, donnant ainsi l'information d'une fin de course ou d'un obstacle.

2.3 Cabine/train

Le pilotage de l'ensemble est effectué par le conducteur sur la base des boutons poussoirs au pupitre :

- autorisation d'ouverture porte gauche
- fermeture portes

Les signaux suivants sont à disposition :

- Ligne de train seuil de vitesse $3km/h$ (à 1 pour une vitesse $< 3km/h$)

A destination du train, les signaux suivants doivent être pilotés par le système portes :

- Ligne de train " porte non fermée verrouillée ", à 0 lorsque le verrouillage n'est pas réalisé

Appendix B

Notation table

In this table we propose a synthesis of the used notations, formal models, functions, operators and abbreviations, in order to provide the reader with an quick index for the significations of the used notations.

TABLE B.1: Notation table 1/3

Abbreviation	Signification
DCS	Discrete Controller Synthesis
$EDCS$	Environment aware Discrete Controller Synthesis
DES	Discrete Event System
FSM	Finite State Machine
COTS	Commercial Off The Shelf
FPGA	Field Programmable Gate Array
ASIC	Application Specific Integrated Circuit
GenBuf	Generalized Buffer
FG	Filling-gap component
D	Door component
LTL	Linear Temporal Logic
CTL	Computation Tree Logic
PSL	Property Specification Language
Notation	**Signification**
σ	Event
ϵ	Empty event
Σ	An *alphabet*, as set of events of a DES
$\|\Sigma\|$	Cardinality of sequence of events
Σ^*	All the possible finite a sequences of events from Σ
$L = \{\epsilon, \sigma_1, \therefore \sigma_n\}$	Language over an alphabet
$M = \{q_0, \Sigma, \delta, Q, Q_m\}$ q_0 Q Q_m $\delta: Q \times \sigma \to Q$	Event driven FSM Initial state Set of states Set of marked states Transition function
q	Current state
$q' \mid q \times \sigma \to q'$	Next state
$M_1 \parallel M_2$	Synchronous product of two FSM
$\Sigma_1 \cap \Sigma_2$	Intersection of two alphabets
$M_1 \times M_2$	Synchronized product of two FSM
$Q_1 \times Q_2$	Cartesian product of two sets of states Q_1, Q_2
$Q_{1m} \times Q_{2m}$	Cartesian product of two sets of marked states Q_{1m}, Q_{2m}
$M = \{q_0, X, Q, \delta\}$ X $\delta: Q \times \mathbb{B}^{\|x\|}$	Sample driven FSM Set of Boolean input variable Boolean transition function
$enc: \Sigma \to \mathbb{B}^{log_2(\|\Sigma\|)}$	Encoding function translates set of events to set of Boolean variables
abs	The absence of events
$enc^{-1}: \mathbb{B}^{log_2(\|\Sigma\|)} \to \Sigma \cup \{abs\}$	Inverse of encoding function, maps the set of Boolean variables to the corresponding events
$M^E = (Q^E, \Sigma^E, q_0^E, \delta^E, Q_m^E)$	Event-driven state FSM
$M^T = (Q^T, X^T, q_0^T, \delta^T, Q_m^T)$	sample-driven state machine

TABLE B.2: Notation table 2/3

Notation	Signification
$M = (q_0, X, Q, \delta, PROP, \lambda)$ $PROP = \{p_1, ..., p_k\}$ $\lambda : Q \rightarrow \mathbb{B}^k$	Finite state machine with outputs Set of k atomic Boolean propositions is a labeling function, modeling the outputs of M
$F_S : Q \rightarrow \mathbb{B}^n$	Function translates states to n Boolean state variables
$F_X : Q \rightarrow \mathbb{B}^m$	Function translates inputs to m Boolean input variables
$s_{B0} = F_S(q_0)$	Boolean initial state
$\delta_B : \mathbb{B}^n \times \mathbb{B}^m \rightarrow \mathbb{B}^n$	Boolean transition function
$\lambda_B : \mathbb{B}^n \rightarrow \mathbb{B}^k$	Boolean labeling function
$\mathcal{C}_s(\langle s_1, s_2, \cdots, s_n \rangle) = \prod_{j=1}^{n} (s_j \Leftrightarrow F_S^j(q))$	Characteristic function of Boolean state q
$\mathcal{C}_E(\langle s_1, s_2, \cdots, s_n \rangle) = \bigvee_{s \in E} \prod_{j=1}^{n} (s_j \Leftrightarrow F_S^j(s))$	Characteristic function of set of states E
$\mathcal{T}(s, x, s') = \prod_{j=1}^{n} s'_j \Leftrightarrow \delta_B^j(s, i)$	Symbolic transition function
$\mathcal{M} = (\mathcal{C}_{s0}, \mathcal{X}, \mathcal{S}, \mathcal{T}, \mathcal{PROP}, \lambda_B)$ $\mathcal{C}_{s0}$ $\mathcal{X} = X_B$ $\mathcal{S} = S_B$ $\mathcal{T} : \mathbb{B}^n \times \mathbb{B}^m \times \mathbb{B}^n \rightarrow \mathbb{B}$ $\mathcal{PROP}$ $\lambda_B : \mathbb{B}^n \rightarrow \mathbb{B}^k$	Symbolic model of a FSM Characteristic function of the initial state Input variables' set State variables' set Transition relation Atomic propositions' set Output (labeling) function
$\mathcal{C}_{p_j}(s) = \bigvee_{s \in \mathbb{B}^n} \mathcal{C}_s(s) \wedge \lambda_B^j(s)$	Characteristic function of set of states where variable $pj \in PROP$ is true
$\mathcal{CPRED}(\mathcal{C}_E, \mathcal{T})(\mathbf{s})$	Set of controllable predecessors of E
$\mathcal{IUC}$	Set of invariant under control
$\mathcal{SUP}$	Supervisor Characteristic function representing the set of all transitions leading to $\mathcal{IUC}$
$\hat{\mathcal{C}}$	Boolean function of a controller
Models and Abbreviations of Safe design method	
$C = (I^c,\ M^c,\ A^c,\ G^c)$ I^c M^c A^c G^c	Model of stand-alone COTS COTS interface Behavioral model of COTS Set of COTS C preconditions Set of COTS C post-condition
a^c	One precondition of COTS C
g^c	One precondition of COTS C
Φ_g^C	Set of logical post-conditions specifications of COTS C
Φ_a^C	Set of logical preconditions specifications of COTS C

TABLE B.3: Notation table 3/3

Notation	Signification
$\mathbf{M_a^C}$	Set of operational preconditions specifications of COTS C
$\mathbf{M_g^C}$	Set of operational post-conditions specifications
X^{intern}	Set of COTS inputs which became internal in the assembly
Y^{intern}	Set of COTS outputs which became internal in the assembly
I^{intern}	Internal interface
$\|\|^C$	Compositional product
I^{asm}	Assembly's interface
M^{asm}	Behavioral model of an assembly
A^{asm}	Set of assembly preconditions
G^{asm}	Set of assembly post-conditions
$P_{loc}^{c_i}$	Local property of a COTS model M^{C_i}
P^{asm}	Global property of an assembly M^{asm}
G^{asm}	Set of assembly post-conditions
$\mathcal{M}_a = \{X_a, \mathcal{S}_a, \mathcal{C}_{s_{0a}} \mathcal{T}_a, \lambda_a\}$	symbolic FSM model of the monitor M_a recognizing a
$\mathcal{M}_P = \{X_P, \mathcal{S}_P, \mathcal{C}_{s_{0P}}, \mathcal{T}_P, \lambda_P\}$	Symbolic model of the monitor M_P recognizing P
X_c	Set on controllable inputs
X_{uc}	Set on uncontrollable inputs
$\mathcal{CPRED}^{env}(\mathcal{C}_E, \mathcal{T}, \mathcal{C}_{\lambda_a^{-1}})(\mathbf{s})$	Set of controllable predecessors of E in $EDCS$
$M_a \|\| M^C \|\| \hat{\mathcal{C}}$	Model of a controlled COTS C corrected by $EDCS$
$request^{env}$	Environment request
$request^{controlled}$	Controlled request
$\mathcal{M}_a \|\| \mathcal{M}^C \|\| \mathcal{M}_P$	Behavioral model of COTS C with the monitors of the preconditions and post-conditions
Operator	**Signification**
$X\varphi$	Next state temporal operator
$F\varphi$	Eventually in the future temporal operator
$G\varphi$	Globally in the future temporal operator
$\varphi U \psi$	Until operator φ is true until ψ is true
$A\psi$	Universal quantifier, i.e, ψ holds on all branches
$E\psi$	existential quantifier, i.e, ψ holds on some branches
$\vee$	Logical disjunction
$\wedge$	Logical conjunction
$\rightarrow$	Logical implication
$\neg$	Logical negation
Operation over models	**Signification**
$\times$	Synchronized product
$\|\|$	Synchronous product
$\|\|^C$	Compositional product

Bibliography

[1] Ciardo G., Jones R. L. III, Miner A. S., et al. Logic and stochastic modeling with smart. Perform. Eval., vol. 63, #6, 2006, pp. 578–608.

[2] Guerrouat Abdelaziz Richter Harald. A component-based specification approach for embedded systems using fdts. **In** : Proceedings of the 2005 conference on Specification and verification of component-based systems, SAVCBS '05. New York, NY, USA: ACM, 2005. ISBN 1-59593-371-9.

[3] Ncube Cornelius Maiden Neil A. M. PORE: Procurement-oriented requirements engineering method for the component-based systems engineering development paradigm. 1999.

[4] Mohamed Abdallah, Ruhe Guenther, Eberlein Armin. Mihos: an approach to support handling the mismatches between system requirements and cots products. Requir. Eng., vol. 12, #3, 2007, pp. 127–143.

[5] Simon David E. An Embedded Software Primer. 1st ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN 020161569X.

[6] Crnkovic Ivica. Component-based approach for embedded systems. **In** : Ninth International Workshop on Component-Oriented Programming. 2004.

[7] Edwards S., Lavagno L., Lee E.A., et al. Design of embedded systems: formal models, validation, and synthesis. Proceedings of the IEEE, vol. 85, #3, 1997, pp. 366–390.

[8] Morton T.D. Embedded Microcontrollers. Pearson Education, 2001. ISBN 9788129702265.

[9] Wilmshurst Tim. Designing Embedded Systems With PIC Microcontrollers: Principles and Applications. Newnes, 2006. ISBN 9780750667555.

[10] Henzinger Thomas A. Sifakis Joseph. The embedded systems design challenge. **In** : Proceedings of the 14th International Symposium on Formal Methods (FM), Lecture Notes in Computer Science. Springer, 2006, pp. 1–15.

[11] Wolf Wayne. Computers as Components, Second Edition: Principles of Embedded Computing System Design. Morgan Kauffman Publishers Inc., 2008.

[12] Gamatie Abdoulaye. Designing Embedded Systems with the SIGNAL Programming Language. Springer, 2010. ISBN 9781441909428.

[13] Feiler P.H. Model-based validation of safety-critical embedded systems. **In** : Aerospace Conference, 2010 IEEE. 2010, pp. 1–10.

[14] Strug J., Deniziak S., Sapiecha K. Validation of reactive embedded systems against temporal requirements. **In** : Engineering of Computer-Based Systems, 2004. Proceedings. 11th IEEE International Conference and Workshop on the. 2004, pp. 152–159.

[15] McMillan Kenneth Lauchlin. Symbolic model checking: an approach to the state explosion problem. Ph.D. thesis, Pittsburgh, PA, USA, 1992. UMI Order No. GAX92-24209.

[16] Ren Mingming. An incremental approach for hardware Discrete Controller Synthesis. Ph.D. thesis, INSA de Lyon, France, 2011.

[17] Chomsky Noam. On certain formal properties of grammars. Information and Control, vol. 2, #2, 1959, pp. 137 – 167.

[18] Cassandras Christos G. Lafortune Stephane. Introduction to Discrete Event Systems. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006. ISBN 0387333320.

[19] Halbwachs N. Synchronous programming of reactive systems, a tutorial and commented bibliography. **In** : Tenth International Conference on Computer-Aided Verification, CAV'98. Vancouver (B.C.): LNCS 1427, Springer Verlag, 1998.

[20] Milner Robin. Calculi for synchrony and asynchrony. Theor. Comput. Sci., vol. 25, 1983, pp. 267–310.

[21] Cassandras Christos G. Lafortune Stephane. Introduction to Discrete Event Systems. Softcover reprint of hardcover 2nd ed. 2008 ed. Springer, 2010. ISBN 1441941193.

[22] Gorgen R., Oetjens J., Nebel W. Transformation of event-driven hdl blocks for native integration into time-driven system models. **In** : Specification and Design Languages (FDL), 2012 Forum on. 2012, pp. 152–159.

[23] Bryant R. Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers, 1986.

[24] Balemi S, Kozak P, Smedinga R. Discrete event systems: modeling and control : proceedings of a joint workshop held in prague, august 1992. Birkhauser, 1993. ISBN 0817628452 9780817628451 3764328452 9783764328450.

[25] Miremadi Sajed, Lennartson Bengt, Akesson Knuta. A bdd-based approach for modeling plant and supervisor by extended finite automata. IEEE Trans. Contr. Sys. Techn., vol. 20, #6, 2012, pp. 1421–1435.

[26] Jain Himanshu Bryant Randal E. Verification using Satisfiability Checking, Predicate Abstraction, and Craig Interpolation. Ph.D. thesis, Carnegy Mellon University, 2008.

[27] Ashenden P.J. of Adelaide. Dept. of Computer Science University. The VHDL Cookbook. Department of Computer Science, University of Adelaide, 1991.

[28] Thomas Donald Moorby Philip. The Verilog Hardware Description Language. 5th ed. Springer Publishing Company, Incorporated, 2008. ISBN 0387849300, 9780387849300.

[29] Black David C. Donovan Jack. SystemC: From the Ground Up. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005. ISBN 0387292403.

[30] Boonpranchoo V., Kongratana V., Tipsuwanporn V., et al. Design of temporal logic embedded controller for small oven process. **In** : Control, Automation and Systems (ICCAS), 2011 11th International Conference on. 2011, pp. 1354–1357.

[31] IEEE, ed. IEEE Standard for Property Specification Language (PSL). IEEE Std 1850-2005 (2005). 2005.

[32] Beer Ilan, Ben-David Shoham, Eisner Cindy, et al. Rulebase: Model checking at ibm. **In** : Grumberg Orna, ed., CAV, vol. 1254 of *Lecture Notes in Computer Science*. Springer, 1997. ISBN 3-540-63166-6, pp. 480–483.

[33] Fisler Kathi Kurshan Robert P. Verifying vhdl designs with cospan. **In** : Formal Hardware Verification. 1997, pp. 206–247.

[34] Group The VIS. Vis: A system for verification and synthesis. **In** : Proceedings of the 8th International Conference on Computer Aided Verification, Lecture Notes in Computer Science. Springer, 1996, pp. 428–432.

[35] Pnueli Amir. The temporal semantics of concurrent programs. **In** : Proceedings of the International Sympoisum on Semantics of Concurrent Computation. London, UK, UK: Springer-Verlag, 1979. ISBN 3-540-09511-X, pp. 1–20.

[36] Shimizu Kanna, Dill David L., Hu Alan J. Monitor-based formal specification of pci. **In** : In Formal Methods in Computer-Aided Design. Springer-Verlag, 2000, pp. 335–352.

[37] Bloem Roderick, Galler Stefan, Jobstmann Barbara, et al. Interactive presentation: Automatic hardware synthesis from specifications: a case study. **In** : Proceedings of the conference on Design, automation and test in Europe, DATE '07. Nice, France: EDA Consortium, 2007. ISBN 978-3-9810801-2-4, pp. 1188âĂŞ–1193. ACM ID: 1266622.

[38] Launiainen Tuomas, Heljanko Keijo, Junttila Tommi. Efficient model checking of psl safety properties. **In** : Proceedings of the 2010 10th International Conference on Application of Concurrency to System Design, ACSD '10. Washington, DC, USA: IEEE Computer Society, 2010. ISBN 978-0-7695-4066-5, pp. 95–104.

[39] Pnueli Amir Zaks Aleksandr. Psl model checking and run-time verification via testers. **In** : FM. 2006, pp. 573–586.

[40] Chang Kai-Hui, ting Tu Wei, jong Yeh Yi, et al. A simulation-based temporal assertion checker for psl. **In** : In Proc. 46th IEEE IntâĂŹl Midwest Stmp. on Circuits and Systems. IEEE Computer Society, 2003, pp. 1528–1531.

[41] Owre S., Rushby J. M., , et al. PVS: A prototype verification system. **In** : Kapur Deepak, ed., 11th International Conference on Automated Deduction (CADE), vol. 607 of *Lecture Notes in Artificial Intelligence*. Saratoga, NY: Springer-Verlag, 1992, pp. 748–752.

[42] Owre S., Rushby J. M., Shankar N., et al. A tutorial on using PVS for hardware verification. **In** : Kumar Ramayya Kropf Thomas, eds., Theorem Provers in Circuit Design (TPCD '94), vol. 901 of *Lecture Notes in Computer Science*. Bad Herrenalb, Germany: Springer-Verlag, 1994, pp. 258–279.

[43] Owre Sam, Rushby John, Shankar N., et al. Pvs: An experience report. **In** : Hutter Dieter, Stephan Werner, Traverso Paolo, et al., eds., Applied Formal Methods—FM-Trends 98, vol. 1641 of *Lecture Notes in Computer Science*. Boppard, Germany: Springer-Verlag, 1998, pp. 338–345.

[44] Kim Taeho, Stringer-Calvert David, Cha Sungdeok. Formal verification of functional properties of an scr-style software requirements specification using PVS. vol. 2280, 2002, pp. 205–220.

[45] Shankar N. Verification of real-time systems using PVS. **In** : Courcoubetis Costas, ed., Computer Aided Verification, CAV '93, vol. 697 of *Lecture Notes in Computer Science*. Elounda, Greece: Springer-Verlag, 1993, pp. 280–291.

[46] Bonichon Richard, Delahaye David, Doligez Damien. Zenon : An extensible automated theorem prover producing checkable proofs. **In** : LPAR. 2007, pp. 151–165.

[47] Nipkow Tobias, Paulson Lawrence C., Wenzel Markus. Isabelle/HOL — A Proof Assistant for Higher-Order Logic, vol. 2283 of *LNCS*. Springer, 2002.

[48] Brucker Achim D. Wolff Burkhart. On theorem prover-based testing. Formal Aspects of Computing, 2012.

[49] Ruf J., Hoffmann D., Kropf T., et al. Simulation-guided property checking based on a multi-valued ar-automata. **In** : Proceedings of the conference on Design, automation and test in Europe, DATE '01. Piscataway, NJ, USA: IEEE Press, 2001. ISBN 0-7695-0993-2, pp. 742–748.

[50] Hu Yunwei. A guided simulation methodology for dynamic probabilistic risk assessment of complex systems. 2005.

[51] De Paula Flavio M. Hu Alan J. An effective guidance strategy for abstraction-guided simulation. **In** : Proceedings of the 44th annual Design Automation Conference, DAC '07. New York, NY, USA: ACM, 2007. ISBN 978-1-59593-627-1, pp. 63–68.

[52] Zhang Tao, Lv Tao, Li Xiaowei. An abstraction-guided simulation approach using markov models for microprocessor verification. **In** : Proceedings of the Conference on Design, Automation and Test in Europe, DATE '10. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2010. ISBN 978-3-9810801-6-2, pp. 484–489.

[53] Owicki Susan S. Gries David. Verifying properties of parallel programs: An axiomatic approach. Commun. ACM, vol. 19, #5, 1976, pp. 279–285.

[54] Owicki Susan Lamport Leslie. Proving liveness properties of concurrent programs. ACM Trans. Program. Lang. Syst., vol. 4, #3, 1982, pp. 455–495.

[55] Clarke E. M., Emerson E. A., Sistla A. P. Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst., vol. 8, #2, 1986, pp. 244–263.

[56] Biere A., Cimatti A., Clarke E. M., et al. Symbolic model checking using sat procedures instead of bdds. **In** : Proceedings of the 36th annual ACM/IEEE Design Automation Conference, DAC '99. New York, NY, USA: ACM, 1999. ISBN 1-58113-109-7, pp. 317–320.

[57] Clarke E. M. 25 years of model checking. chap. The Birth of Model Checking. Berlin, Heidelberg: Springer-Verlag, 2008. ISBN 978-3-540-69849-4, pp. 1–26.

[58] Clarke E. M. Grumberg O. Avoiding the state explosion problem in temporal logic model checking. **In** : Proceedings of the sixth annual ACM Symposium on Principles of distributed computing, PODC '87. New York, NY, USA: ACM, 1987. ISBN 0-89791-239-X, pp. 294–303.

[59] Eén N. SAT Based Model Checking. Ph.D. thesis, Chalmers University of Technology and Göteborg University, 2005.

[60] Yang Junfeng, Twohey Paul, Engler Dawson R., et al. Using model checking to find serious file system errors. ACM Trans. Comput. Syst., vol. 24, #4, 2006, pp. 393–423.

[61] Ramadge P.J.G. Wonham W.M. The control of discrete event systems. Proceedings of the IEEE, vol. 77, #1, 1989, pp. 81–98.

[62] Dumitrescu Emil, Girault Alain, Marchand Hervé, et al. Multicriteria optimal reconfiguration of fault-tolerant real-time tasks. **In** : Workshop on Discrete Event Systems, WODES'10. Berlin, Allemagne: IFAC, 2010, pp. 366–373.

[63] Wonham W.M. Supervisory control of discrete-event systems. ECE 1636F/1637S 2009-10, 2010.

[64] Akesson K., Fabian M., Flordal H., et al. Supremica - an integrated environment for verification, synthesis and simulation of discrete event systems. **In** : Discrete Event Systems, 2006 8th International Workshop on. 2006, pp. 384–385.

[65] Marchand Herve, Bournai Patricia, LeBorgne Michel, et al. Synthesis of discrete-event controllers based on the signal environment. **In** : In Discrete Event Dynamic System: Theory and Application. 2000, pp. 325–346.

[66] Coussy P., Gajski D.D., Meredith M., et al. An introduction to high-level synthesis. Design Test of Computers, IEEE, vol. 26, #4, 2009, pp. 8–17.

[67] Asarin Eugene, Maler Oded, Pnueli Amir. Symbolic controller synthesis for discrete and timed systems. **In** : Hybrid Systems II, LNCS 999. Springer, 1995, pp. 1–20.

[68] Dumitrescu Emil, Ren Mingming, Piétrac Laurent, et al. A supervisor implementation approach in discrete controller synthesis. **In** : ETFA. 2008, pp. 1433–1440.

[69] Shrum E.V. Use of RT CORBA in the U.S. army. **In** : Fourth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2001. ISORC - 2001. Proceedings. 2001, pp. 268 –269.

[70] Chau S.N., Alkalai L., Tai A.T., et al. Design of a fault-tolerant COTS-based bus architecture. IEEE Transactions on Reliability, vol. 48, #4, 1999, pp. 351–359.

[71] Vigder Mark R. Dean John. An architectural approach to building systems from cots software components. **In** : Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research, CASCON '97. IBM Press, 1997.

[72] Oberndorf Patricia. Cots and open systems. **In** : SEI Monographs on the Use of Commercial Software in Goverment Systems. 1998.

[73] Reform United States Office of the Deputy Under Secretary of Defense for Acquisition. Commercial item acquisition: considerations and lessons learned. Office of the Deputy Under Secretary of Defense for Acquisition Reform, 2000.

[74] Morisio Maurizio Torchiano Marco. Definition and classification of cots: A proposal. **In** : Proceedings of the First International Conference on COTS-Based Software Systems, ICCBSS '02. London, UK, UK: Springer-Verlag, 2002. ISBN 3-540-43100-4, pp. 165–175.

[75] Addy Edward A. Sitaraman Murali. Formal specification of cots-based software: a case study. **In** : Proceedings of the 1999 symposium on Software reusability, SSR '99. New York, NY, USA: ACM, 1999. ISBN 1-58113-101-1, pp. 83–91.

[76] Fecko M.A., Uyar M.ÃIJ., Amer P.D., et al. A success story of formal description techniques: Estelle specification and test generation for mil-std 188-220. Computer Communications, vol. 23, #12, 2000, pp. 1196–1213.

[77] Budkowski S. Dembinski P. An introduction to estelle: A specification language for distributed systems. Computer Networks and ISDN Systems, vol. 14, #1, 1987, pp. 3–23.

[78] Abts Chris. Cots-based systems (cbs) functional density – a heuristic for better cbs design. **In** : Proceedings of the First International Conference on COTS-Based Software Systems, ICCBSS '02. London, UK, UK: Springer-Verlag, 2002. ISBN 3-540-43100-4, pp. 1–9.

[79] Reifer Donald J., Basili Victor R., Boehm Barry W., et al. COTS-Based systems âĂŞ twelve lessons learned about maintenance. **In** : Goos Gerhard, Hartmanis Juris, Leeuwen Jan, et al., eds., COTS-Based Software Systems, vol. 2959. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. ISBN 978-3-540-21903-3, 978-3-540-24645-9, pp. 137–145.

[80] Mohammad Mubarak Alagar Vangalur. Specification and verification of trustworthy component-based real-time reactive systems. **In** : Proceedings of the 2007 conference on Specification and verification of component-based systems: 6th Joint Meeting of the European Conference on Software Engineering and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, SAVCBS '07. New York, NY, USA: ACM, 2007. ISBN 978-1-59593-721-6, pp. 89–93.

[81] Mohammad Mubarak Alagar Vangalur. A formal approach for the specification and verification of trustworthy component-based systems. J. Syst. Softw., vol. 84, #1, 2011, pp. 77–104.

[82] Etienne J. Bouzefrane S. Utilisation de l'approche par composants pour la conception d'applications temps réel. **In** : (RJCITR 05) Premières Rencontres des Jeunes Chercheurs en Informatique Temps Réel, Nancy. 2005.

[83] Cotard S., Faucou S., Bechennec J.-L., et al. A data flow monitoring service based on runtime verification for autosar. **In** : High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on. 2012, pp. 1508–1515.

[84] Arias-Chausson Carlos. The necessary legal approach to cots safety and cots liability in european single market. **In** : Proceedings of the 4th international conference on COTS-Based Software Systems, ICCBSS'05. Berlin, Heidelberg: Springer-Verlag, 2005. ISBN 3-540-24548-0, 978-3-540-24548-3, pp. 36–42.

[85] Clough A., (U.S.) John A. Volpe National Transportation Systems Center, Laboratory Charles Stark Draper. Commercial-off-the-shelf (COTS) Hardware and Software for Train Control Applications: System Safety Considerations. U.S. Department of Transportation, Federal Railroad Administration, 2003.

[86] Chung Lawrence Cooper Kendra. Defining goals in a cots-aware requirements engineering approach: Regular paper. Syst. Eng., vol. 7, #1, 2004, pp. 61–83.

[87] Mohamed Abdallah, Ruhe Guenther, Eberlein Armin. Decision support for handling mismatches between COTS products and system requirements. **In** : Proceedings of the Sixth International IEEE Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems. IEEE Computer Society, 2007. ISBN 0-7695-2785-X, pp. 63–âĂŞ72. ACM ID: 1259760.

[88] Mohamed Abdallah, Ruhe Guenther, Eberlein Armin. Mismatch handling for COTS selection: a case study. Journal of Software Maintenance and Evolution: Research and Practice, vol. 23, #3, 2011, pp. 145–178.

[89] Lanoix Arnaud Souquieres Jeanine. A Trustworthy Assembly of COTS Components, 2006.

[90] Xie Fei, Yang Guowu, Song Xiaoyu. Component-based hardware/software co-verification for building trustworthy embedded systems. Journal of Systems and Software, vol. 80, #5, 2007, pp. 643–654.

[91] IBM research | IBM haifa research lab | RuleBase parallel edition. http://www.research.ibm.com/haifa/projects/verification/RB_Homepage/publications.html.

[92] Hajjar Salam, Dumitrescu Emil, Niel Eric, et al. A component-based safe design method for train control systems. **In** : Embedded Real Time Software and Systems ERTS. Toulous, France: 3AF - SEE, 2012.

[93] Hajjar Salam, Dumitrescu Emil, Niel Eric, et al. Safe design method of embedded control systems based on cots. **In** : Actes de le 2ème Conférence en IngénieriE du Logiciel. Nancy, France, 2013, pp. 35–45.

[94] Beydeda S. Gruhn V. Testing Commercial-off-the-Shelf Components and Systems. Springer, 2005. ISBN 9783540218715.

[95] Richard Mitchell Jim McKim. Design by Contract: by example. Addison-Wesley, 2002.

[96] GeenSoft. Controlbuild, innovative environment for designing and validating critical control software applications, 2013.

[97] GeenSoft. Controlbuild, design, simulate & deploy automation & embedded control systems with higher efficiency, 2010.

[98] Ieee standard for vhdl register transfer level (rtl) synthesis. IEEE Std 1076.6-2004 (Revision of IEEE Std 1076.6-1999), 2004, pp. 1–112.

[99] Dragomir Iulia, Ober Iulian, Percebois Christian. Safety contracts for timed reactive components. **In** : Actes des Cinquiemes journées nationales du Groupement De Recherche CNRS du Génie de la Programmation et du Logiciel. 2013, pp. 37–49.

[100] Mcmillan K. L. Circular compositional reasoning about liveness. **In** : Advances in Hardware Design and Verification: IFIP WG10.5 International Conference on Correct Hardware Design and Verification Methods (CHARME 99), volume 1703 of Lecture Notes in Computer Science. Springer-Verlag, 1999, pp. 342–345.

[101] Henzinger Thomas A., Qadeer Shaz, Rajamani Sriram K. You assume, we guarantee: Methodology and case studies. Springer-Verlag, 1998, pp. 440–451.

[102] Borrione D., Morin-Allory K., Oddos Y. Design Technology for Heterogeneous Embedded Systems, chap. Property-Based Dynamic Verification and Test. Springer, 2012.

[103] Bloem Roderick, Galler Stefan, Jobstmann Barbara, et al. Specify, compile, run: Hardware from PSL. Electron. Notes Theor. Comput. Sci., vol. 190, #4, 2007, pp. 3–16.

[104] Pétin Jean-François, Evrot Dominique, Morel Gérard, et al. Combining SysML and formal methods for safety requirements verification. **In** : 22nd International Conference on Software & Systems Engineering and their Applications. Paris, France, 2010, p. CDROM.

[105] Jadot Jean-Yves. Ferrocots, from cable to chip.

[106] Hajjar Salam, Dumitrescu Emil, Niel Eric, et al. Safe design method of embedded control systems : Case study. **In** : 5èmes Journées Doctorales / Journées Nationales MACS Ecole en Modélisation, Analyse et Conduite des Systèmes dynamiques. Strasbourg, France, 2013.