Luigi Di Guglielmo

# Realizability of embedded controllers: from hybrid models to correct implementations

Ph.D. Thesis

April 17, 2012

Università degli Studi di Verona
Dipartimento di Informatica

Advisors:
prof. Franco Fummi
prof. Tiziano Villa

Series N°: **TD-04-12**

Università di Verona
Dipartimento di Informatica
Strada le Grazie 15, 37134 Verona
Italy

*To my family*

# Contents

# 1 Introduction

An *embedded controller* is a reactive device (e.g., a suitable combination of hardware and software components) that is embedded in a dynamical environment and has to react to environment changes in real time. Embedded controllers are widely adopted in many contexts of modern life, from automotive to avionics, from consumer electronics to medical equipment. Noticeably, the correctness of such controllers is crucial. When designing and verifying an embedded controller, often the need arises to model the controller and also its surrounding environment. The nature of the obtained system is *hybrid* because of the inclusion of both discrete-event (i.e., controller) and continuous-time (i.e., environment) processes whose dynamics cannot be characterized faithfully using either a discrete or continuous model only. Systems of this kind are named cyber-physical (CPS) or hybrid systems.

Different types of models may be used to describe hybrid systems and they focus on different objectives: detailed models are excellent for simulation but not suitable for verification, high-level models are excellent for verification but not convenient for refinement, and so forth. Among all these models, *hybrid automata* (HA) [8, 77] have been proposed as a powerful formalism for the design, simulation and verification of hybrid systems. In particular, a hybrid automaton represents discrete-event processes by means of finite state machines (FSM), whereas continuous-time processes are represented by using real-numbered variables whose dynamics is specified by (ordinary) differential equation (ODE) or their generalizations (e.g., differential inclusions).

Unfortunately, when the high-level model of the hybrid system is a hybrid automaton, several difficulties should be solved in order to automate the refinement phase in the design flow, because of the classical semantics of hybrid automata. In fact, hybrid automata can be considered perfect and instantaneous devices. They adopt a notion of time and evaluation of continuous variables based on dense sets of values (usually $\mathbb{R}_{\geq 0}$ and $\mathbb{R}$, resp.). Thus, they can sample the state (i.e., value assignments on variables) of the hybrid system at any instant in such a dense set $\mathbb{R}_{\geq 0}$. Further, they are capable of instantaneously evaluating guard constraints or reacting to incoming events by performing changes in the operating mode of the hybrid system without any delay. While these aspects are convenient at the modeling level, any model of an embedded controller that relies for its correctness on such precision and instantaneity cannot be implemented by any hardware/software device, no matter how fast it is. In other words, the controller is un-realizable, i.e., un-implementable.

## 1.1 Aims of the thesis

This thesis proposes a complete methodology and a framework that allows to derive from hybrid automata proved correct in the hybrid domain, correct realizable models of embedded controllers and the related discrete implementations. In a realizable model, the controller samples the state of the environment at periodic discrete time instants which, typically, are fixed by the clock frequency of the processor implementing the controller. The state of the environment consists of the current values of the relevant variables as observed by the sensors. These values are digitized with finite precision and reported to the controller that may decide to switch the operating mode of the environment. In such a case, the controller generates suitable output signals that, once transmitted to the actuators, will effect the desired change in the operating mode. It is worth noting that the sensors will report the current values of the variables and the actuators will effect changes in the rates of evolution of the variables with bounded delays.

In particular, given a hybrid automaton-based model $M$ of a hybrid system, the proposed methodology focuses on:

- Identifying the existence of a realizable model $M_{\mathcal{R}}$ which includes a controller that is *implementable* and is still able to safely handle the surrounding environment. In other words, this step aims at automating the synthesis of a *relaxed* or *lazy* control strategy for $M$ which takes into account the digital and imprecise aspects of the hardware device on which the actual control strategy is being executed. In particular, the synthesis establishes the *performance bounds* to be satisfied by any conservative concrete hardware/software device that will implement the controller. So, if a realizable model $M_{\mathcal{R}}$ is identified, then its control strategy can be translated into an embedded software that has to be executed on a hardware device satisfying the synthesized performance bounds.
- Simplifying the translation of the control strategy of $M_{\mathcal{R}}$ into the embedded software that is implementing such a realizable control strategy. Instead of manually defining the code implementation, the methodology proposes to adopt a *model-driven design* approach [126], i.e., to define the control strategy behaviors by means of discrete abstract graphic formalisms (e.g., FSMs, hierarchical FSMs). The gain offered by the adoption of a such an approach is the capability of synthesizing the code implementing the embedded software in a systematic way, i.e., it avoids the need of manual writing, analyzing and modifying the code, because the corresponding code is correct by construction, that is, it implements correctly the specified model behaviors.
- Supporting the embedded software model-driven design with an integrated functional verification environment. In fact, even if the model-driven design simplifies the generation of the software code implementation, it does not prevent the designer to wrongly define the software behaviors using the graphic formalisms. In particular, the methodology supports the functional verification

of embedded software by means of *dynamic assertion-based verification* [64]. Such a kind of verification uses formal temporal assertions for checking the functional and temporal correctness of the embedded software model and, thus, of its implementation.

- Supporting the qualification of the set of assertions used to verify the embedded software. In particular, the proposed methodology supports the *vacuity checking* [16, 98] that aims at identifying the presence of vacuous assertions, i.e., assertions that are trivially satisfied. Such assertions can lead designers to a false sense of safety because the embedded software implementation could be erroneous even if all the defined assertions are satisfied.

The methodology summarized above is thoroughly described in the following chapters. Each chapter highlight a specific problem and describes how the proposed methodology aims to solve it. Moreover, each chapter clearly reports which are the novelties of the proposed work with respect to the state of the art approaches.

## 1.2 Thesis overview

This thesis is divided in three main parts: the first part focuses on the problem of synthesizing an implementable control strategy from hybrid automata to enable the subsequent refinement phases typical of embedded controllers design flows; the second part focuses on the problem of modeling and verifying embedded software for guaranteeing the correctness of the software that model the implementable control strategy previously identified; finally, the third part focuses on evaluating the effectiveness of the verification phase which guarantee the real correctness of the implemented embedded software.

In Chapter 2, we discuss the problem of synthesizing implementable control strategies for hybrid models described by means of hybrid automata. In particular, in Section 2.3 we describe an existing solution for synthesizing implementable control strategies for sub-classes of HA and propose a framework that extends its applicability on general classes of HA. This is based on a joint work with prof. Tiziano Villa, Dr. Davide Bresolin and Dr. Luca Geretti [31]. In Section 2.4, instead, we defined a new approach for synthesizing implementable control strategies for *lazy linear hybrid automata* which is a relevant class of HA whose semantics differs from the traditional one and it is suited for modeling discrete controller embedded into physical environments. This approach results from a recent joint work with prof. Tiziano Villa and prof. Sanjit Seshia which has not been published yet.

In Chapter 3 we present a suitable combination of model-driven design and dynamic assertion-based verification as an effective solution for embedded software development. Besides, we describe the characteristics of a framework composed by two environments that has been developed for supporting this integrated approach. The model-driven design environment, named radCASE, provides the designer with a comprehensive approach to cover the complete modeling and synthesis process of embedded software. The dynamic assertion-based verification environment, named radCHECK, automates the simulation-based verification making dynamic assertion-based verification really practical. This work is based on collaborations with Dr. Giuseppe Di Guglielmo, prof. Masahiro Fujita, prof. Franco Fummi and prof. Graziano Pravadelli [49]. Preliminary results have been published also in [48, 52].

In Chapter 4 we tackle the problem of assertion qualification focusing in particular on the identification of vacuously satisfied assertions which may hide errors in the verification phase. In particular, we propose the first methodology that, given a set of assertions satisfied by a design implementation (either hardware or software), enables to perform vacuity detection in the context of dynamic assertion-based verification. In fact, all the existing approaches in literature are suited for static verification. This work is based on a collaboration with prof. Franco Fummi and prof. Graziano Pravadelli [53, 54].

Finally, a summary of the contributions made by the thesis is reported in Chapter 5. This chapter concludes the thesis by identifying future research directions.

# 2

# Synthesis of implementable control strategies for HA

## 2.1 Introduction

There is a non-negligible semantic gap between hybrid models described by means of hybrid automata and discrete implementations. First, the notion of variable used in hybrid automata is continuous, defining variables which take their values from a dense set (e.g., $\mathbb{R}$), while implementations can only use digital and finitely precise variables. Second, hybrid automata react instantaneously to events and guard constraints while implementations can only react within a given, usually small but non-zero, reaction delay. Third, hybrid automata may describe control strategies that are unrealistic, such as Zeno-strategies or strategies that require the system to act faster and faster. For these reasons, a control strategy that has been proven correct may not be implementable (at all) or it may not be possible to systematically turn it into an implementation that is still correct [36]. Thus, the development of a technique for synthesizing an implementable control strategy (if it exist) for a hybrid automaton-based model is really valuable.

This chapter tackles this problem by proposing:

- a complete framework, based on the methodology proposed in [46], that makes *practical* the applicability of the Almost-ASAP semantics for synthesizing implementable control strategies for relevant classes of hybrid automata for which the reachability problem is not decidable.
- a new methodology, again supported by tools, for the synthesis of implementable control strategies for the interesting class of *lazy linear hybrid automata*. For such a class of hybrid automata the reachability problem is decidable.

In particular, the chapter is organized as follows. Section 2.2 summarizes the background, the current results related to verification and synthesis of implementable control strategies for hybrid automata and languages for hybrid automata specification. Section 2.3 provides the main concepts related to the Almost-ASAP methodology and the model manipulations implemented into the proposed framework that enables to synthesize implementable control strategies for generic classes of hybrid automata. Experimental results show the usability of the framework. Section 2.4 introduces, instead, the fundamental definitions and semantics of lazy linear hybrid automata that motivates the assumptions at the base of the new methodology proposed for synthesis of implementable control strategies described in Section 2.4.2. Also in this case, experimental results underline the applicability of the methodology. Finally, Section 2.5 is devoted to concluding remarks.

## 2.2 Background

A hybrid automaton is a mathematical model for precisely describing systems in which computational processes tightly interact with the physical world. The following formal definition has been proposed in [77].

**Definition 2.1 (Hybrid Automaton).** *A hybrid automaton $H$ is a tuple $\langle X, Q, Q_0, init, inv, flow, E, jump, update, \Sigma\rangle$. The components of a hybrid automaton are as follows:*

- Variables. *A finite set $X = \{x_1, \dots, x_n\}$ of real-numbered variables. The number $n$ is called the dimension of $H$. $\dot{X}$ stands for the set $\{\dot{x}_1, \dots, \dot{x}_n\}$ of dotted variables (which represent the time derivatives of continuous variables during continuous change), and $X'$ stands for the set $\{x'_1, \dots, x'_n\}$ of primed variables (which represent values of continuous variables at the conclusion of discrete change).*
- Control modes. *A finite set $Q$ of control modes. $Q_0 \subseteq Q$ denotes the set of initial modes.*
- Initial conditions. *A labeling function init that assigns to each control mode $q \in Q_0$ an initial predicate. Each initial condition $init(q)$ is a predicate whose free variables are from $X$.*
- Invariant conditions. *A labeling function inv that assigns to each control mode $q \in Q$ an invariant predicate. Each invariant condition $inv(q)$ is a predicate whose free variables are from $X$.*
- Flow conditions. *A labeling function $flow$ that assigns to each control mode $q \in Q$ a flow predicate. Each flow condition $flow(q)$ is a predicate whose free variables are from $X \cup \dot{X}$.*
- Control switches. *A set $E$ of edges $(q, q')$ from a source mode $q \in Q$ to a target mode $q' \in Q$.*
- Jump conditions. *An edge labeling function $jump$ that assigns to each control switch $e \in E$ a predicate. Each jump condition $jump(e)$ is a predicate whose free variables are from $X$.*
- Update conditions. *An edge labeling function update that assigns to each control switch $e \in E$ a predicate. Each update condition $update(e)$ is a predicate whose free variables are from $X \cup X'$.*
- Events. *A finite set $\Sigma$ of events, and an edge labeling function $event : E \to \Sigma$ that assigns to each control switch an event.*

In literature different classes of hybrid automata have been proposed which specify particular restrictions on the initial, invariant, flow, jump and update conditions.

- ***Timed automata***. A timed automaton (TA) is a hybrid automaton in which continuous variables are "clocks", i.e., variables whose time derivatives are constrained by the flow conditions to 1 (e.g., $\dot{x} = 1$) in each control mode. In a TA, the initial, invariant and jump conditions consist of clock constraints of the form
$$\phi = x \leq c \mid c \leq x \mid \neg\phi \mid \phi_1 \wedge \phi_2$$
whose free variables are from $X$ and $c \in \mathbb{Q}$. The update conditions, instead, are restricted clock constraints whose free variables are from $X'$ and variables can be only reset to 0 (e.g., $x' = 0$).
- ***Linear hybrid automata***. A linear hybrid automaton (LHA) is an hybrid automaton in which all the predicates assigned by $init$, $inv$, $flow$, $jump$ and $update$ functions are rectangular formulas. A rectangular inequality is of the form $x_i \sim c$ with $c \in \mathbb{Z}$ and $\sim \in \{<, \leq, >, \geq\}$. A rectangular formula is a conjunction of rectangular inequalities. In a LHA initial, invariant and jump conditions are given by rectangular formulas over continuous variables (e.g., $x_i = [l, u]$, where $l$, $u \in \mathbb{Z}$ are the lower and upper-bound for $x_i \in X$, respectively), update conditions are given by rectangular formulas over primed variables (i.e., $X'$) and flow conditions are given by rectangular formulas over continuous variables time derivatives (i.e., $\dot{X}$).
- ***Affine hybrid automata***. An affine hybrid automaton (AHA) is an hybrid automaton in which all the predicates assigned by $init$, $inv$, $flow$, $jump$ and $update$ functions are finite linear formulas. A linear expression is of the form $\sum_i a_i x_i + b$ and a convex linear formula is a finite conjunction of constraints $\sum_i a_i x_i + b \sim 0$, with $a_i$, $b \in \mathbb{Z}$, $x_i \in X$ and $\sim \in \{<, \leq, =\}$. Then, a linear formula, also referred to as non-convex linear formula, is a finite disjunction of convex linear formulas. In an AHA initial, invariant and jump conditions are given by linear formulas over continuous variables (i.e., $X$), update conditions are given by linear formulas over continuous variables and the primed variables (i.e., $X \cup X'$) and, finally, flow conditions are given by linear formulas over the continuous variables and their time derivatives (i.e., $X \cup \dot{X}$).

To formalize the concept of semantics of hybrid automata, several notions needs to be defined.

**Definition 2.2 (Valuation for continuous variables).** *Let $X = \{x_1, \ldots, x_n\}$ be a set of continuous variables. A valuation $V$ for the variables in $X$ is a member of $\mathbb{R}^n$ such that $V$ prescribes a real value $V(i)$ to each variable $x_i$.*

**Definition 2.3 (State of a HA).** *Let $H$ be a hybrid automaton. A* state *of $H$ is a pair $(q, V)$, where $q \in Q$ is a control mode and $V \in \mathbb{R}^n$ is a valuation for the continuous variables in $X$. A state $(q, V)$ is initial if and only if $q \in Q_0$ and $V \vDash init(q)$.*

Intuitively, the semantics of a HA corresponds to a sequence of transitions from one state to another, alternating continuous and discrete evolutions. In continuous evolution, the control mode does not change while the time passes and evolution of the variables follows the dynamic law associated with the current mode. A discrete evolution step consists of the activation of a discrete transition that can change both the current mode and value of the variables, in accordance with the update function associated with the transition. The interleaving of continuous and discrete evolutions is decided by the invariant of the mode, which must be true for the continuous evolution to keep on going, and guard predicates, which must be true for a discrete transition to be activated. Guards and invariants are not necessarily complements of each other: when both the invariant and one or more guards are true, both the continuous evolution and activation of discrete transitions are allowed, and the behavior of the automaton becomes nondeterministic.

Formally, the semantics of a HA is defined as follows.

**Definition 2.4 (Transition relation of a HA).** *Let $H$ be a hybrid automaton and let $S_H$ the set of states of $H$. The semantics of $H$, noted as $[\![H]\!]$, is given by the* transition relation $\Longrightarrow \subseteq S_H \times (\Sigma \cup \mathbb{R}_{\geq 0}) \times S_H$ *defined as follows:*

- Continuous transition. *For each nonnegative real number $t \in \mathbb{R}_{\geq 0}$, define $(q, V) \overset{t}{\Longrightarrow} (q', V')$ iff $q = q'$ and there is a differentiable function $f : [0, t] \to \mathbb{R}^n$ with the first derivative $\dot{f} : [0, t] \to \mathbb{R}^n$ such that $f(0) = V$, $f(t) = V'$ and $\forall t' \in [0, t]$, $f(t') \vDash inv(q)$ and $\dot{f}(t') \vDash flow(q)$.*
- Discrete transition. *For each event $\sigma \in \Sigma$, define $(q, V) \overset{\sigma}{\Longrightarrow} (q', V')$ iff there is a control switch $e \in E$ such that $e = (q, q')$, $V \vDash jump(q)$, $(V, V') \vDash update(q)$ and $\sigma = event(e)$.*

From the semantics define above, it is possible to derive the following notions of trajectory of a HA, reachability relation between states and product of transition relations of HA.

**Definition 2.5 (Trajectory of a HA).** *Let $H$ be a hybrid automaton and let $(q, V)$ be a state of $H$. A* trajectory *of $H$ from $(q, V)$ is a sequence of states $(q_i, V_i)$, with $i > 0$, such that $(q_0, V_0) = (q, V)$ and $(q_{i-1}, V_{i-1}) \overset{\alpha}{\Longrightarrow} (q_i, V_i)$ with either $\alpha \in \mathbb{R}_{\geq 0}$ or $\alpha \in \Sigma$.*

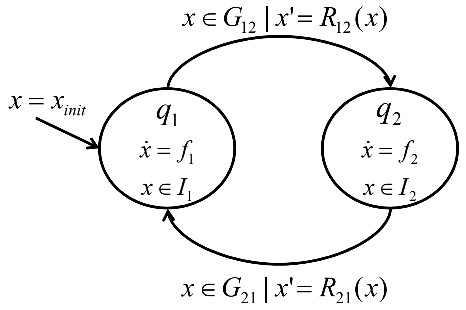


Fig. 2.1: Example of hybrid automaton model.

**Example** (*A generic hybrid automaton*). Figure 2.1 sketches an example of a hybrid automaton $H$ composed of two control modes $q_1$ and $q_2$. The set $X$ of continuous variables is $X = \{x\}$. Let $i$ and $j$ be such that $i, j \in \{1, 2\}$ and

$i \neq j$. Each invariant condition $inv(q_i)$ is defined as a subset $I_i$ of $\mathbb{R}^n$ (in this case $n = |X| = 1$). The automaton can stay in a control mode $q_i$ if the valuation of the variable $x$ satisfies the invariant condition, i.e., $x \in I_i$. The jump condition of a control switch $e_{ij} = (q_i, q_j)$, instead, is specified by a guard set $G_{ij}$ and an update function $R_{ij}$ that defines how the continuous variable $x$ may change when $H$ switches from $q_i$ to $q_j$. For keeping things simple in the example, let assume that both $R_{12}$ and $R_{21}$ are the identity function. Finally, each flow condition $flow(q_i)$ is specified by a function $f_i$, thus, $f_i$ constraints the continuous evolution of the variable $x$ until the HA stays in the control mode $q_i$.

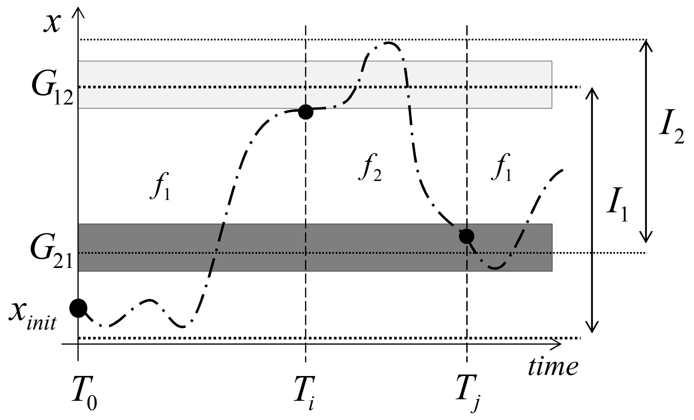


Fig. 2.2: Example of a possible trajectory of the HA in Figure 2.1.

Besides, Figure 2.2 shows part of a possible trajectory of $H$ from the initial state $(q_1, x_{init})$, which at first follows the dynamics $f_1$. As soon as the trajectory reaches the guard set $G_{12}$ (light-gray box), e.g., at time instant $T_i$, the control switch $e_{12}$ is enabled. Due to the fact that at $T_i$ the invariant condition $I_1$ is still satisfied, the HA can either switch to the mode $q_2$ or remain in $q_1$. The former is the case in this example. Because of $R_{12}$ is the identity function, the trajectory of $H$ starts following the dynamics $f_2$ from the same state reached at $T_i$. Then, it keeps following such a dynamics until the invariant $I_2$ is violated or the jump condition $G_{21}$ (dark-gray box) is satisfied. This is the case at the time instant $T_j$, when the automaton takes the switch $e_{21}$, and moves back to $q_1$ where the trajectory evolves again under the dynamics $f_1$.

**Definition 2.6 (Reachability relation between states of a HA).** *Let $H$ be a hybrid automaton. A state $(q, V)$ reaches a state $(q', V')$ if there exists a finite trajectory of states $(q_i, V_i)$, with $0 \leq i \leq n$, such that $(q_0, V_0) = (q, V)$ and $(q_n, V_n) = (q', V')$. $\mathcal{R}_H(q, V)$ is used to denote the set of states reachable from the state $(q, V)$. $\mathcal{R}_H(S)$ is used to denote the set $\bigcup_{(q,V)\in S} \mathcal{R}_H(q, V)$, i.e., the set of all the states reachable from each state $(q, V)$ in the set $S$.*

**Definition 2.7 (Product of transition relations of HA).** *Given two hybrid automata $H_1$ and $H_2$, let $S_{H_1}$, $S_{H_2}$ denote the set of states, and $\Sigma_1$, $\Sigma_2$ denote the* *set of events of $H_1$ and $H_2$, respectively. The product of two transition relations $[\![H_1]\!]$, $[\![H_2]\!]$, noted as $[\![H_1]\!] \parallel [\![H_2]\!]$, is a third transition relation $\Longrightarrow \subseteq (S_{H_1} \times S_{H_2}) \times (\Sigma_1 \cup \Sigma_2 \cup \mathbb{R}_{\geq 0}) \times (S_{H_1} \times S_{H_2})$ defined as follows:*

- *for any $\sigma \in \Sigma_1 \cup \Sigma_2 \cup \mathbb{R}_{\geq 0}$, $((s_1, s_2), \sigma, (s'_1, s'_2)) \in \Longrightarrow$ iff one of the following conditions hold:*
  - *$\sigma \in (\Sigma_1 \setminus \Sigma_2)$ and $(s_1, \sigma, s'_1) \in [\![H_1]\!]$ and $s_2 = s'_2$;*
  - *$\sigma \in (\Sigma_2 \setminus \Sigma_1)$ and $(s_2, \sigma, s'_2) \in [\![H_2]\!]$ and $s_1 = s'_1$;*
  - *$\sigma \in (\Sigma_1 \cap \Sigma_2) \cup \mathbb{R}_{\geq 0}$ and $(s_1, \sigma, s'_1) \in [\![H_1]\!]$ and $(s_2, \sigma, s'_2) \in [\![H_2]\!]$.*

### 2.2.1 Verification of HA

**Definition 2.8 (Safety property).** *A safety property $\varphi_{sp}$ for a hybrid automaton $H$ consists of a subset of states of $H$, i.e., $\varphi_{sp} \subseteq (Q \times \mathbb{R}^n)$. A state $s$ of $H$ is said to be safe if and only if $s \in \varphi_{sp}$. The hybrid automaton $H$ is safe with respect to $\varphi_{sp}$ if and only if all the states reachable from its set $S_0$ of initial states are safe, i.e., $\mathcal{R}_H(S_0) \subseteq \varphi_{sp}$. $[\![H]\!] \vDash \varphi_{sp}$ is alternative notation to state that $H$ is safe w.r.t. $\varphi_{sp}$.*

Given such a definition, it is clear that the problem of verifying safety properties on hybrid automata reduces to the reachability problem [77].

Unfortunately, in a large variety of settings, the reachability problem on HA is undecidable (e.g., for the general classes of LHA and AHA). Hence, many tools for reachability analysis of general hybrid automata are based on approximation techniques. The challenge for these tools [14,67,78] is to find the "best" approximations of continuous states. A precise overview of the boundary between decidable and undecidable aspects of hybrid automata is drawn in [80, 122].

The works in [76, 106, 121] suggest that except under very restrictive settings (such as the case of TA), it is not possible to attain decidability if the continuous variables are not reset during mode switches and flow rates change as a result of the mode change. From a point of view of digital controllers that interact with plants through sensors and actuators, the resetting requirement severely restricts the modeling expressiveness.

Due to the inherent discrete nature of a controller of a hybrid system, several works have moved to focus on the discrete time behavior of hybrid systems [4–6, 9, 24, 25, 79]. Also in this context a number of undecidability results have been reported in the literature. These undecidability results are mainly related to piecewise-affine systems with infinite precision [24, 25].

On the contrary, positive results have been reported in [4–6, 9, 79]. In particular, the authors in [4–6] define a class of LHA, named lazy linear hybrid automata, for which the control state reachability problem is decidable, also allowing the continuous variables to retain their values during mode changes. This is attained by focusing on discrete time behavior and requiring finite precision for variables. Because of such assumptions, this formal model allows a rich class of guards and cope with lazy sensors and actuators that have bounded delays associated with them. However, its discrete behavior depends on the sampling frequency of the controller as well as the precision of variables, and hence, the discretized representations are very large and any enumerative analysis would not be feasible for systems of appreciable size. This problem has been overcome in [89]. The authors propose a symbolic representation for lazy linear hybrid automata which can be used for reachability analysis based on bounded model checking. Further, the authors practically show the scalability of their method on interesting real case studies.

### 2.2.2 Synthesis of implementable control strategies for HA

Whereas the control state reachability problem has been thoroughly investigated, only few works in literature focus on the synthesis of an implementable control strategy [7, 46, 95, 125] for the hybrid models.

In particular, the work in [95], proposes a digitalized semantics for timed automata based on the non-instant observability of events. This semantics models the fact that a physical environment cannot be observed continuously, but only at discrete time instants by an implementable control strategy. The authors consider timed automata as timed specifications of the hybrid system and, thus, they study in what sense the corresponding timed traces can be recognized by a digital controller. To do this, they introduce the concept of *time-triggered automata*. A time-triggered automaton is essentially a time table for a digital controller describing what the controller should do at a given time point. The advantage of such time-triggered automata is that they can be easily transformed into executable programs (i.e., the concrete control strategies). The main result shown into the work is that the authors can effectively decide whether a time-triggered automaton correctly recognizes the timed traces, i.e., the time-triggered automaton implements the timed specification of the hybrid system. Unfortunately, the authors underlined that the systematic synthesis of time-triggered automata from timed automata in such a way they recognize the corresponding timed traces is still an open issue.

On the contrary, in [46] the authors propose an alternative semantics for timed automata that allows to synthesize from a timed automaton $C$ of a digital controller an implementable control strategy in a systematic way. The new semantics, named Almost-ASAP, relies on the continuous time behavior and infinite precision of HA. However, it takes into account the digital and imprecise aspects of the hardware in which the timed automaton $C$ is being executed by relaxing the "synchrony hypothesis": the semantics does not impose the controller to react instantaneously, but it is supposed to react within $\Delta$ time units when a synchronization or a control action has to take place. The designer acts as if the synchrony hypothesis was true, i.e., he/she models the environment $Env$ and the controller $C$ without referring to the reaction delay. The reaction delay is taken into account during the verification phase: the authors use reachability analysis to look for the largest value $\Delta$ for which the relaxed controller is still receptive w.r.t. the environment in which it will be embedded and it is still correct w.r.t. the properties the original instantaneous model $C$ has to enforce. Such a $\Delta$-relaxed controller represents an implementable control strategy if $\Delta > 0$. However, it is worth noting that the Almost-ASAP approach requires the models of the control strategy $C$ and environment $Env$ to be separate automata interacting each other only by means of synchronization events. Moreover, due to the adoption of a continuous time and infinite precision semantics, the problem of synthesizing such a value $\Delta$ may not be decidable.

The work in [125] proposes a similar approach for synthesizing implementable control strategies from timed automata. Given a timed automaton $C$ modeling the control strategy, instead of looking for a conservative abstraction of $C$ that does not contain behaviors which falsify the original specifications, the authors look for a conservative refinement that includes all the non-blocking behaviors of $C$. To determine the existence of such a conservative refinement (i.e., shrinkability problem) the authors construct from $C$ a new timed automaton $C'$ that shrinks the guards of the original automaton so that all behaviors of $C'$ under relaxation (performed as proposed in [46]) are included in those of $C$. Thus, the so obtained timed automaton $C'$ preserves the properties proven for $C$. This means that all timing requirements satisfied by $C$, such as critical deadlines, are strictly respected by $C'$. However, such a shrinking may remove too many behaviors and even introduce deadlocks in $C'$. For this reason, the authors have theoretically investigated the constraints which guarantee the preservation of the desired behaviors in $C'$, and have shown that deciding the shrinkability problem of a timed automaton $C$ is EXPTIME.

Notice that modifying the semantics may not be the only way to enforce the implementability. Indeed, in [7] the authors ask the question whether similar results can be obtained without introducing a new semantics, but acting on modeling instead, thanks to the introduction of new assumptions on the program type or execution platform by means of changes, in a modular way, on the corresponding models. The authors propose an implementation methodology for timed automata which allows to transform a timed automaton into a program and to check whether the execution of this program on a given platform satisfies a desired property. Unlike the works in [46,125], an open problem of this approach is how to guarantee that, when a platform $P$ is replaced by a "better" platform $P'$, a program proved correct for $P$ is also correct for $P'$; the authors reported examples where this does not hold for a reasonable assumption of a "better" platform, namely, when $P$ and $P'$ are identical, but $P'$ provides a periodic digital clock running twice as fast as the one of $P$; the reason is that a program using the faster clock has a higher "sampling rate" and thus may generate more behaviors than a program using the slower clock, so this situation may result in a violation of properties.

### 2.2.3 Languages for HA specification

The large number of modeling formalisms for hybrid models specification makes hard the integration of different techniques and tools in a comprehensive design framework for hybrid systems. In fact, a model for simulation (e.g., Simulink [137], Ptolemy II [141], Modelica [110], gPROMS [120]) is very likely to be transformed into another formalism for verification purposes (e.g., Uppaal [17], PHAVer [67], Ariadne [14]) and, again, a model suited for either simulation or verification is likely to be manipulated for other analysis intends (e.g., control optimization with MUSCOD II [88], and so on). This lack of integration capability is the major challenge towards a broad industrial acceptance of hybrid systems tools.

A promising approach for achieving inter-operability between hybrid systems tools is to develop automatic translations of their formalisms via a general *interchange format* with sufficiently rich syntax and semantics.

The most interesting interchange formalism is the *Compositional Interchange Format* (CIF) [129]. Although it has been developed in the recent years, the CIF formalism is already integrated into different simulation [110,120,136,137] and verification [14,17,67] frameworks via manipulation tools. This interchange formalism has several interesting characteristics which can be summarized as follows:

- it has a formal and compositional semantics which allows property preserving model transformations;
- its concepts are based on mathematics, and independent of implementation aspects such as equation sorting, and numerical equation solving algorithms;
- it supports arbitrary differential algebraic equations, algebraic loops, steady state initialization, switched systems such as piecewise affine systems;
- it supports a wide range of concepts originating from hybrid automata, including different kinds of urgency, such as urgency predicates, deadline predicates, triggering guard semantics, and urgent actions [17];
- it supports parallel composition with synchronization by means of shared variables and shared actions;
- it supports hierarchy and modularity to allow the definition of parallel modules and modules that can contain other modules (hierarchy), and to allow the definition of variables and actions as being local to a module, or shared between modules.

Moreover several ad hoc tools have been developed to support directly the design and analysis of CIF models:

- a graphical editor that speeds up the definition of the hybrid automaton-based model by dragging and dropping modes and edges. Once the model is completed, the editor automatically synthesizes the corresponding CIF code;
- a compiler that takes as input the CIF code and translates it to an abstract format suited for simulation;
- a stepper that takes as input a CIF abstract format and calculates its dynamic behavior resulting in a hybrid transition system that consists of action and time transitions.
- a simulator that provides a front-end to the stepper. Several options exist to customize the output of the simulator, such as the visualization of the trajectories of the model variables during and/or after the simulation, or the visualization of the performed discrete actions. For simulation purposes, the action and delay transitions are calculated using symbolic and/or numerical solvers. Besides, the simulator can be run in different modes:
  - User-guided mode: In this mode, the simulator shows all possible transitions, and the user may choose which transition to execute.
  - Automatic mode: In this mode, the non-deterministic choices between transitions are resolved automatically by the simulator, or the simulator can simulate all possibilities (exhaustive simulation/state-space generation). In both modes, the simulator can be parameterized with the solvers to be used, including solver-specific options, and the requested simulation output.

Summarizing, the CIF formalism represent a good input language for guaranteeing the inter-operability of new hybrid system frameworks with the already existing ones.

## 2.3 Synthesis of implementable control strategies for generic HA

### 2.3.1 Problem definition

The widely adopted design paradigm for hybrid systems can be summarized as follows:

1. construct a timed/hybrid model $\mathcal{E}$ of the environment;
2. state clearly which are the safety properties $\varphi_{sp}$ the control strategy (i.e., the controller) should respect;
3. design a timed/hybrid model $C$ of the controller implementing the control strategy;
4. verify the correctness of the whole model, i.e., $[\![C]\!] \parallel [\![\mathcal{E}]\!] \vDash \varphi_{sp}$, that is check that the composition of the controller with the environment is safe with respect to the safety properties.

Such a verification step is performed using the traditional semantics for hybrid automata. Unfortunately, this relies on unimplementable assumptions, such as the *synchrony hypothesis*, i.e., the capability of performing any computation in zero time units and forcing a change in the dynamics of the model without delays. Thus, the verification returns correctness results for the control strategy which cannot be used in practice.

To overcome such a problem, the work in [46] proposes an alternative semantics for interpreting the control strategy described into a hybrid model. The new semantics takes into account the digital and imprecise aspects of the hardware device on which the actual control strategy is being executed. In particular, it concerns:

- *the relaxation of the variable precision*: continuous variables can be modeled only with a finite precision and consequently they are rounded according to the HW platform characteristics;
- *the relaxation of the instantaneousness of reaction to timeouts and events*: any reaction to a timeout and an incoming or outgoing event introduces delays that depend on the HW platform characteristics.

Such a semantics is named *Almost-As-Soon-As-Possible* (AASAP) semantics.

The authors have formally proved that by verifying the correctness of the control strategy using such a semantics, it is possible to determine if the control strategy of the modeled hybrid system is implementable.

However, none of the existing model checking tools for hybrid systems is able to natively verify hybrid models against safety properties using the AASAP semantics. Thus, the only way to work around this difficulty consists of transforming the original hybrid model into another one that, interpreted using the classical semantics for hybrid automata (Section 2.2), represents a conservative abstraction of the original model interpreted using the AASAP semantics (Section 2.3.3).

Intuitively, given a control strategy $C$, the conservative abstraction of the control strategy, noted as $C^\delta$, is characterized by a *relaxing* parameter $\delta$ such that:

- *$\delta$ relaxes the continuous variable precision.* Any guard constraint modeling a decision for the control strategy is relaxed by some small amount which depends on $\delta$, simulating values digitization;
- *$\delta$ relaxes the reactions to outgoing events.* Any decision that can be taken by the control strategy becomes urgent only after a small delay bounded by $\delta$. This simulates the actuation delays;
- *$\delta$ relaxes the reactions to incoming events.* A distinction is made between the occurrence of a synchronization event in the sender (occurrence) and the acknowledgement of the event by the receiver (perception). The time difference between the occurrence and the perception of the event is bounded by $\delta$. This simulates the sensing delays.

Notice that the abstracted control strategy exhibits a superset of the original behavior, the latter corresponding to a relaxing value $\varDelta = 0$ for the parameter $\delta$. Consequently, in order for the control strategy $C^\delta$ to be implementable at all, a necessary condition is that there exists a value $\varDelta > 0$ for which $C^{\delta=\varDelta}$ allows the hybrid model to be safe. Then, given $\varDelta$, the expression $\varDelta > 4\varDelta_P + 3\varDelta_L$ proved in [46] relates it to the actual constraints the implementable control strategy has to adhere such as clock period $\varDelta_P$ and the worst-case-time required for treating incoming and outgoing synchronization events $\varDelta_L$.

Thus, the problem of synthesizing an implementable control strategy reduces to the problem of synthesizing a suitable value for the parameter $\delta$, i.e., checking the existence of a value $\varDelta$ for which $[\![C^{\delta=\varDelta}]\!] \parallel [\![\mathcal{E}]\!] \vDash \varphi_{sp}$.

It is worth noting that the verification of $[\![C^{\delta=\varDelta}]\!] \parallel [\![\mathcal{E}]\!] \vDash \varphi_{sp}$ is based on the analysis of the reachable set, i.e., the set of all states that can be reached under the dynamical evolution of the systems starting from a given set of initial states. However, the state of a hybrid automaton consists of the pairing of a discrete location with a vector of continuous variables, therefore it has the cardinality of continuum. Due to this fact, the reachable set is, in general, not decidable [80]. To face this problem, many approximation techniques and tools to estimate the reachable set have been proposed in the literature. In particular, the framework in [46] implementing the AASAP analysis is based on the HyTech tool [78]. A key feature of HyTech is its ability to perform parametric analysis, i.e. to determine the values of design parameters (e.g., $\varDelta$) for which a linear hybrid automaton satisfies a safety property (e.g., $\varphi_{sp}$). Although HyTech natively includes an engine for synthesizing the desired value $\varDelta$, its analysis is limited to LHA, and, thus, it limits the applicability of the proposed methodology.

Thus, a possible way to exceed such a limitation consists of implementing a framework that makes practical the applicability of the AASAP approach on classes of hybrid automata more complex than LHA. The framework should include:

- tools for the manipulation of hybrid descriptions to generate, given the original control strategy, the *conservative abstraction* of such a control strategy according to the AASAP semantics;
- a parameters synthesis procedure that can be solved using different state-of-the-art hybrid domain model checkers that support generic classes of hybrid automata (e.g., AHA).

It is worth noting that the approach [46] requires the hybrid models of the control strategy and environment to be separate automata and the model of the control strategy to be an *elastic controller*. Such a model is a timed automaton featuring the following restrictions:

1. Only urgent transitions are allowed;
2. The guards must be closed clock expressions;
3. Communication with the environment is allowed only through events.
4. Continuous variables are restricted to clocks (i.e. continuous variables measuring the elapsing of time).

It must be remarked that the restrictions above are perfectly reasonable from a controller implementation viewpoint and do not represent a major limitation in terms of applicability of the method. Moreover, no restrictions are applied on the model of the environment, thus, it can be either a TA or a LHA or an AHA.

### 2.3.2 Contributions

The main contributions of Section 2.3 consists of a framework that makes practical the applicability of the AASAP approach proposed in [46]. It is worth noting that currently no theoretical extensions have been introduced.

In particular, the novelties of the framework can be summarized as follows:

- it adopts the CIF standard language that eases the modeling of hybrid systems and ensures the inter-operability of the developed framework with already existing hybrid system tools;
- it includes a tool, called *s-extract*, that automatically transforms a CIF description into another CIF description suited for the synthesis of the implementable control strategy;
- it includes two tools, called *cif2phaver* and *cif2ariadne*, which aim to translate CIF descriptions into the formalisms required to use the PHAVer [67] and Ariadne [14] model checkers able to handle linear and non-linear affine hybrid automata, respectively.
- it implements a synthesis procedure that, starting from a set of feasible values for the relaxing parameter $\delta$ and exploiting either PHAVer or Ariadne, identifies the maximum value which enable the relaxed control strategy to satisfy its safety properties. The two model checkers are used according to the complexity of the hybrid model dynamics to analyze.

### 2.3.3 The AASAP semantics [46]

First of all, any approach for the synthesis of an implementable control strategy based on the AASAP semantics requires the hybrid models of the control strategy and environment to be separate automata:

- the model of the control strategy has to be an *Elastic controller*;
- the model of the environment may be either a TA or a LHA or an AHA.

Intuitively, an Elastic controller is a timed automaton suited for modeling an embedded controller which has to react *As Soon As Possible* to stimuli coming from the surrounding environment. Such an embedded controller may contain timers whose expiration may require the generation of stimuli for the surrounding environment. Notice that actual timers are variables with finite precision whose value increments with the elapsing of time. The only legal updates on timers are resets (usually to 0). Expiration constraints can be effectively modeled using closed rectangular predicates over the timers.

**Definition 2.9 (Closed rectangular formula).** *A closed rectangular formula over a set of (continuous) variables $X$ is a finite formula $\varphi$ of the form*

$$\varphi = true|false|x \leq c \mid c \leq x \mid \phi_1 \wedge \phi_2$$

*where $x \in X$ and $c \in \mathbb{Q}$.*

In what follows, the usual notation $x \in [a, b]$ denotes the predicate $a \leq x \leq b$ and $Rect_c(X)$ denotes the set of all the possible closed rectangular formulas over the set $X$.

Now, an Elastic controller can be formally defined as follows:

**Definition 2.10 (Elastic Controller).** *An Elastic controller $C$ is a tuple $\langle X, Q, Q_0, init, inv, flow, E, jump, update, \Sigma\rangle$. The components of an Elastic controller are as follows:*

- Clock variables. *A finite set $X = \{x_1, \ldots, x_n\}$ of clocks. $\dot{X}$ stands for the set $\{\dot{x}_1, \ldots, \dot{x}_n\}$ of dotted variables and $X'$ stands for the set $\{x'_1, \ldots, x'_n\}$ of primed variables.*
- Control modes. *A finite set $Q$ of control modes. $Q_0 \subseteq Q$ denotes the set of initial modes.*
- Initial conditions. *A labeling function init that assigns to each control mode $q \in Q_0$ an initial predicate. Each initial predicate $init(q)$ is a closed rectangular formula over the variables in $X$.*
- Invariant conditions. *A labeling function inv that assigns to each control mode $q \in Q$ an invariant predicate. Each invariant predicate $inv(q)$ is a closed rectangular formula over the variables in $X$.*
- Flow conditions. *A labeling function $flow$ that assigns to each control mode $q \in Q$ a flow predicate. Each flow predicate $flow(q)$ is a predicate whose free variables are from $\dot{X}$ and constraints such variables to 1, e.g., $\dot{x} = 1$.*
- Control switches. *A set $E$ of edges $(q, q')$ from a source mode $q \in Q$ to a target mode $q' \in Q$.*
- Jump conditions. *An edge labeling function jump that assigns to each control switch $e \in E$ a predicate. Each jump condition $jump(e)$ is a closed rectangular formula whose free variables are from $X$.*
- Update conditions. *An edge labeling function update that assigns to each control switch $e \in E$ a predicate. Each update condition $update(e)$ is a closed rectangular formula whose free variables are from $X'$ in which variables can be only reset to 0 (e.g., $x' = 0$).*

- Events. *A finite set $\Sigma$ of events, and an edge labeling function event* $: E \to \Sigma$ *that assigns to each control switch an event. The set is partitioned into the set $\Sigma^{in}$ of input events, $\Sigma^{out}$ of output events and $\Sigma^{\tau}$ of internal events.*

To keep the notation compact, in what follows, $(q, q', g, \sigma, R) \in E$ is used to denote that there exists a control switch $e = (q, q')$ with $g = jump(e)$, $\sigma = event(e)$ and $R = update(e)$ in $C$.

Let $V \in \mathbb{R}^{n}_{\geq 0}$ be a valuation of the clocks in $X$, where $n = |X|$. $V - t$ denotes a valuation $V'$ such that for each $x_i \in X$, $V'(i) = V(i) - t$. Intuitively, $V + t$ is defined in a similar way.

**Definition 2.11 (True Since).** *Let $X$ be a set of clocks, $g$ be a closed rectangular formula over $X$ and $V$ a valuation for clocks in $X$. The function "True Since", noted as $TS : \mathbb{R}^{\geq 0} \times Rect_c(X) \to (\mathbb{R}^{\geq 0} \cup -\infty)$, is defined as follows:*

$$TS(V, g) = \begin{cases} t \text{ if } V \vDash g \wedge V - t \vDash g \wedge \forall t' > t : V - t' \nvDash g \\ -\infty \text{ otherwise} \end{cases}$$

**Definition 2.12 (Guard Enlargement).** *Let $g(x)$ be the closed rectangular formula $x \in [a, b]$. The closed rectangular formula ${}_{\Delta}g(x)_{\Delta}$ with $\Delta \in \mathbb{Q}^{\geq 0}$ denotes the formula $x \in [a-\Delta, b+\Delta]$ if $a-\Delta \geq 0$ and $x \in [0, b+\Delta]$ otherwise. Let $g$ be a convex closed rectangular predicate, i.e., the finite conjunction of closed rectangular formulas, then ${}_{\Delta}g_{\Delta}$ denotes the set of closed rectangular formulas $\{{}_{\Delta}g(x)_{\Delta} \mid g(x) \in g\}$.*

The input events (e.g., $\sigma \in \Sigma^{in}$) of the controller represent the possible synchronization requests the controller may receive from the environment. In what follows, every synchronization request is distinguished in an *occurrence*, i.e., the issuing of the request by the environment, and a *perception*, i.e., the viewing of such a synchronization request by the controller. In particular:

**Definition 2.13 (Perception of Events).** *Let $\Sigma^{in}$ be the set of input events. $\widetilde{\Sigma^{in}}$ denotes the set of perceptions of input events. To denote a link between an occurrence of an input event and its perception, the notation $\sigma$ is used for the occurrence and $\tilde{\sigma}$ for the perception.*

**Definition 2.14 (State of an Elastic controller).** *Let $C$ be an Elastic controller. A state of $C$ is the tuple $(q, V, I, d)$ where:*

- *$q \in Q$ is a control mode;*
- *$V \in \mathbb{R}^{\geq 0}$ is a valuation for the clocks in $X$;*
- *$I \in (\mathbb{R}^{\geq 0} \cup \{\bot\})^m$, with $m = |\Sigma^{in}|$, is a vector that records, for each input event $\sigma$, the time elapsed since its oldest* untreated *occurrence. The absence of an* untreated *occurrence of an input event is denoted with $\bot$. The notation $I(\sigma)$ prescribes the value of the component of $I$ corresponding to $\sigma$, whereas $I[\sigma := i]$ denotes a new vector $I'$ such that $I'(\alpha) = i$ for $\alpha = \sigma$ and $I'(\alpha) = I(\alpha)$ for $\alpha \neq \sigma$;*
- *$d \in \mathbb{R}^{\geq 0}$ records the time elapsed since the last mode change.*

*Finally, $\iota = (q_0, V_0, I, 0)$ denotes an initial state for $C$ where $V_0$ is such that $V_0 \vDash init(q_0)$, and $I$ is such that for any $\sigma \in \Sigma^{in}$, $I(\sigma) = \bot$;*

Now is possible to define the AASAP semantics for the Elastic controllers.

**Definition 2.15 (AASAP semantics).** *Let $C$ be an Elastic controller, $S_C$ be the set of states of $C$ and $\Delta \in \mathbb{Q}_{\geq 0}$. The AASAP semantics of $C$, noted $[\![A]\!]_{\Delta}^{AASAP}$ is given by the transition relation $\Longrightarrow \subseteq S_C \times (\Sigma \cup \widetilde{\Sigma^{in}} \cup \{\tau\} \cup \mathbb{R}_{\geq 0}) \times S_C$ where:*

1. *for the discrete transitions, it is necessary to distinguish five cases:*
   a) *let $\sigma \in \Sigma^{out}$.*
      $((q, V, I, d), \sigma, (q', V', I, 0)) \in \Longrightarrow$ *iff there exists* $(q, q', g, \sigma, R) \in E$ *such that* $V \models_{\Delta} [g]_{\Delta}$ *and* $(V, V') \models R$*;*
   b) *let $\sigma \in \Sigma^{in}$.*
      $((q, V, I, d), \sigma, (q, V, I', d)) \in \Longrightarrow$ *iff*
      - *either* $I(\sigma) = \bot$ *and* $I' = I[\sigma := 0]$*;*
      - *or* $I(\sigma) \neq \bot$ *and* $I' = I$*.*
   c) *let $\tilde{\sigma} \in \widetilde{\Sigma^{in}}$.*
      $((q, V, I, d), \sigma, (q', V', I', 0)) \in \Longrightarrow$ *iff there exists* $(q, q', g, \sigma, R) \in E$ *such that* $V \models_{\Delta} [g]_{\Delta}$, $I(\sigma) \neq \bot$, $(V, V') \models R$ *and* $I' = I[\sigma := \bot]$*;*
   d) *let $\sigma \in \Sigma^{\tau}$.*
      $((q, V, I, d), \sigma, (q', V', I, 0)) \in \Longrightarrow$ *iff there exists* $(q, q', g, \sigma, R) \in E$ *such that* $V \models_{\Delta} [g]_{\Delta}$*, and* $(V, V') \models R$*;*
   e) *let $\sigma = \tau$.*
      *For any* $(q, V, I, d) \in S_C$, $((q, V, I, d), \tau, (q, V, I, d)) \in \Longrightarrow$.
2. *for the continuous transitions:*
   a) *for any $t \in \mathbb{R}^{\geq 0}$,*
      $((q, V, I, d), t, (q, V + t, I + t, d + t)) \in \Longrightarrow$ *iff the two following conditions are satisfied:*
      - *for any edge* $(q, q', g, \sigma, R) \in E$ *with* $\sigma \in \Sigma^{out} \cup \Sigma^{\tau}$*:*
$$\forall t' : 0 \leq t' \leq t :$$
$$(d + t' \leq \Delta \vee \mathbf{TS}(V + t', g) \leq \Delta)$$
      - *for any edge* $(q, q', g, \sigma, R) \in E$ *with* $\sigma \in \Sigma^{in}$*:*
$$\forall t' : 0 \leq t' \leq t :$$
$$(d + t' \leq \Delta \vee \mathbf{TS}(V + t', g) \leq \Delta$$
$$\vee$$
$$I(\sigma) + t' \leq \Delta)$$

*Remark 2.16 (The AASAP semantics intuitively).* According to the AASAP semantics, control strategies are intended to try to take a control switch as soon as they can, thus, almost as soon as possible. The idea of *almost* is modeled by a value $\Delta$, which is used to upper-bound the delays for sensing and emitting synchronization events from and to the environment, and, moreover, to upper-bound the time imprecisions of the clocks. In particular:

- The *Case (1.a)* defines when it is allowed for the Elastic controller to emit an output event. The only difference with the classical semantics is that the guard is enlarged by $\Delta$, i.e., the guard is evaluated by supposing imprecisions in clocks evaluation.

- The *Case (1.b)* defines how inputs are treated by the Elastic controller, in other words, what happens when a synchronization event is emitted by the environment. The Elastic controller maintains, through the vector $I$, a list of *occurrences* of input events, i.e., events that have occurred and are not treated yet. An input event $\sigma$ can be received at any time, but if there are more occurrences of the same input $\sigma$ before the controller had the chance to treat the first one, the semantics simply ignores them and only the age of the oldest untreated $\sigma$ is stored in the vector $I$. Notice that in this *Case* the automaton does not change the control mode, i.e., $q$ and also $V$ and $d$ are unchanged at that point.
- The *Case (1.c)* defines when inputs are sensed by the Elastic controller. An input $\sigma$ is sensed when a transition with an enlarged guard and labeled with $\tilde{\sigma}$ is fired. Once $\sigma$ has been treated, the value of $I(\sigma)$ is reset to $\perp$.
- The *Case (1.d)* is similar to *Case (1.a)* states how an internal synchronization event $\sigma$ can be emitted.
- The *Case (1.e)* states that a synchronization $\tau$ can always be performed.
- Finally, *Case (2.a)* specifies how time can elapse. Intuitively, time can pass as long as no transition from the current mode is urgent. Given a control mode $q$, a transition labeled with an output or an internal event is urgent when the control strategy has stayed in $q$ for more than $\Delta$ time units, i.e., $d + t' > \Delta$, and the guard of the corresponding control switch has been true for more than $\Delta$ time units, i.e., $TS(v + t', g) > \Delta$. Given a control mode $q$, a transition labeled with an input event $\sigma$ is urgent when the control strategy has been in $q$ for more than $\Delta$ time units, i.e., $d + t' > \Delta$, the guard of the related control switch has been true for more that $\Delta$ time units, i.e., $TS(v + t', g) > \Delta$, and the last untreated occurrence of $\sigma$ has been emitted by the environment at least $\Delta$ time units ago, i.e., $I(\sigma) + t' > \Delta$.

Three problems can be formulated about the AASAP semantics of an Elastic controller.

**Definition 2.17 (Parametric safety verification problem [46]).** *Let $C$ be an Elastic controller, $\mathcal{E}$ be a hybrid automaton and $\varphi_{sp}$ a safety property. The parametric safety verification problem asks:*

- *[**Fixed**] whether $[\![C]\!]^{AASAP}_{\Delta} \parallel [\![\mathcal{E}]\!] \models \varphi_{sp}$ for a given fixed value of $\Delta$;*
- *[**Existence**] whether there exists $\Delta > 0$ such that $[\![C]\!]^{AASAP}_{\Delta} \parallel [\![\mathcal{E}]\!] \models \varphi_{sp}$;*
- *[**Maximization**] to maximize $\Delta$ such that $[\![C]\!]^{AASAP}_{\Delta} \parallel [\![\mathcal{E}]\!] \models \varphi_{sp}$.*

In particular, the problem [Fixed] is useful when the characteristics of the hardware on which the control strategy that will be implemented are known, the problem [Existence] is useful to determine if the control strategy is implementable at all and the problem [Maximization] is useful to determine what is the slowest hardware on which the controller can be implemented.

Unfortunately, none of the current existing model checkers are able to solve the parametric safety control problem summarized above. To overcome this, the following theorem has been proposed:

**Theorem 2.18 (Practical verification using traditional timed automaton semantics).** *For any Elastic controller $C$, for any $\Delta \in \mathbb{Q}^{>0}$, it is possible to* *effectively construct a timed automaton* $C^\delta = \mathcal{F}(C,\delta)$ *such that* $[\![C]\!]_\Delta^{AASAP} \sqsubseteq [\![C^{\delta=\Delta}]\!]$ *and* $[\![C^{\delta=\Delta}]\!] \sqsubseteq [\![C]\!]_\Delta^{AASAP}$.

The full proofs are available in [46].

Intuitively, Theorem 2.18 states that $C^\delta$ represents a conservative abstraction of the Elastic controller $C$ interpreted using the AASAP semantics. Thus, it follows:

**Corollary 2.19.** *For any Elastic controller* $C$, *for any* $\Delta \in \mathbb{Q}^{>0}$, *for any hybrid automaton* $\mathcal{E}$ *and safety property* $\varphi_{sp}$, $[\![C]\!]_\Delta^{AASAP} \parallel [\![\mathcal{E}]\!] \models \varphi_{sp}$ *iff* $[\![C^{\delta=\Delta}]\!] \parallel [\![\mathcal{E}]\!] \models \varphi_{sp}$.

In practice, Corollary 2.19 can be used to reduce the parametric safety verification problem based on the AASAP semantics to the traditional reachability problem on hybrid automata.

### 2.3.4 Conservative abstraction of the control strategy [46]

In what follows all the manipulation rules implemented into the *s-extract* tool are described. Such rules allow to derive from an Elastic controller $C$, modeling the control strategy, its conservative abstraction $\mathcal{F}(C,\delta)$ in terms of a timed automaton which enables to solve the parametric safety problem (thus, synthesizing an implementable control strategy for the hybrid model) by using already existing hybrid domain model checker. An example is reported to explain how the construction of $\mathcal{F}(C,\delta)$ works.

**Theorem 2.20 (Generation of $\mathcal{F}(C,\delta)$).** *Let* $C$ *be the Elastic automaton*

$$\langle X, Q, Q_0, init, inv, flow, E, jump, update, \Sigma\rangle,$$

*and let* $\mathcal{F}(C,\delta)$ *be the timed automaton*

$$\langle X_1, Q_1, Q_1^0, init_1, inv_1, flow_1, E_1, jump_1, update_1, \Sigma_1\rangle,$$

*such that:*

1. $X_1 = X \cup \{y_\sigma \mid \sigma \in \Sigma^{in}\} \cup \{d\}$.
2. $Q_1 = \{(q,b) \mid q \in Q \wedge b \in \{\top,\bot\}^m, m = |\Sigma^{in}|\}$.
3. $Q_1^0 = \{(q,b_\bot) \mid q \in Q_0 \wedge b_\bot = \{\bot\}^m, m = |\Sigma^{in}|\}$.
4. $init_1$ *is such that for every* $(q,b) \in Q_1^0$ , $init_1((q,b)) = init(q)$;
5. $flow_1$ *is such that for every* $(q,b) \in Q_1$ , $flow_1((q,b)) = \bigwedge_{x\in X_1} \dot{x} = 1$;
6. $\Sigma_1 = \Sigma^{in} \cup \Sigma^{out} \cup \Sigma^{\tau} \cup \widetilde{\Sigma^{in}}$;
7. $E_1$, $jump_1$ *and* $update_1$ *are defined in such a way* $((q,b),(q',b'),_\delta [g]_\delta, \sigma, R') \in E_1$ *iff one of the following conditions holds:*
   a) $\sigma \in \Sigma^{in}$ *and*
      - $q' = q$
      - $b(\sigma) = \bot$
      - $b' = b[\sigma := \top]$
      - $g = \text{true}$
      - $R' = \bigwedge_{x\in X_1\setminus\{y_\sigma\}} (x' = x) \wedge (y_\sigma = 0)$

*b)* $\sigma \in \Sigma^{in}$ *and*
- $q' = q$
- $b(\sigma) = \top$
- $b' = b$
- $g = \text{true}$
- $R' = \bigwedge_{x \in X_1}(x' = x)$

*c)* $\sigma \in \Sigma^{out}$ *and*
- *there exists* $(q, q', g, \sigma, R) \in E$
- $b' = b$
- $R' = R \wedge (d = 0)$

*d)* $\sigma \in \Sigma^{\tau}$ *and*
- *there exists* $(q, q', g, \sigma, R) \in E$
- $b' = b$
- $R' = R \wedge (d = 0)$

*e)* $\sigma = \tilde{\alpha} \in \widetilde{\Sigma^{in}}$ *and*
- *there exists* $(q, q', g, \alpha, R) \in E$
- $b(\alpha) = \top$
- $b' = b[\alpha := \bot]$
- $R' = R \wedge (d = 0)$

*f)* $\sigma = \tau$ *and*
- $q' = q$
- $b' = b$
- $g = \text{true}$
- $R' = \bigwedge_{x \in X_1}(x' = x)$

8. *Let* $EVT((q,b)) = \{((q, q', g, \sigma, R) \in E \mid \sigma \in \Sigma^{in} \wedge b(\sigma) = \top\}$. *Let* $ACT((q,b)) = \{(q, q', g, \sigma, R) \in E \mid \sigma \in \Sigma^{out} \cup \Sigma^{\tau}\}$.
*Then* $Inv_1((q,b)) = \varphi_1(q,b) \wedge \varphi_2(q,b)$ *where:*

$$\varphi_1(q,b) = \bigwedge_{(q,q',g,\sigma,R) \in EVT} (d \leq \delta \vee \neg({}^{\delta}g) \vee y_\sigma \leq \delta)$$

$$\varphi_2(q,b) = \bigwedge_{(q,q',g,\sigma,R) \in ACT} (d \leq \delta \vee \neg({}^{\delta}g))$$

*and* ${}^{\delta}g(x)$ *is the constraint* $x \in (a + \delta, b]$ *if* $g(x)$ *is of the form* $x \in [a, b]$.

*Then,* $\mathcal{F}(C, \delta)$ *satisfies the properties of Theorem 2.18.*

*Remark 2.21 (Generation of* $\mathcal{F}(C, \delta)$ *intuitively).* The above theorem describes how to transform an Elastic controller into a timed automaton that, interpreted using the traditional semantics, preserves all the behaviors of the initial Elastic controller interpreted using the AASAP semantics. The so obtained timed automaton is enriched with new control modes, support variables and a parameter $\delta$ to correctly model the concept of *almost as soon as possible* synchronization. This transformation proposed in [46] is easy to implement but not very efficient: indeed, the number of modes is larger by an exponential factor in the number of input events of the initial Elastic controller. In particular:

- *Rule (1)* defines the set $X_1$ of continuous variables of $\mathcal{F}(C, \delta)$. $X_1$ contains all the clocks (i.e., $x \in X$) of $C$ and new variables: $\{y_\sigma\}$ and $d$. Each $y_\sigma$ is a clock used to store the age of the oldest untreated input event $\sigma$, i.e., the time elapsed since the oldest untreated $\sigma$ has occurred. Instead, $d$ is a clock that stores the time spent in the current control mode.
- *Rule (2)* defines the set $Q_1$ of control modes of $\mathcal{F}(C, \delta)$. To each mode $q \in Q$, $\mathcal{F}(C, \delta)$ associates a set of modes whose cardinality depends on the number of input events of $C$. In particular, each mode $(q, b) \in Q_1$ models the fact that the original Elastic controller $C$ can remain in the mode $q$ even if several input events $\sigma$ have occurred and are untreated (i.e., $b(\sigma) = \top$). The non-untreated events are specified as $b(\sigma) = \bot$.
- *Rule (3)* defines the set $Q_1^0$ of initial control modes of $\mathcal{F}(C, \delta)$. A mode $(q, b) \in Q_1$ is initial if $q$ is an initial mode for $C$ and none of the input events is untreated (i.e., for all $\sigma \in \Sigma^{in},\ b(\sigma) = \bot$).
- *Rule (4)* defines the $init_1$ function of $\mathcal{F}(C, \delta)$. For all the initial modes $(q, b_\bot)$ of $\mathcal{F}(C, \delta)$ the initial conditions are such that $init_1((q, b_\bot)) = init(q)$, i.e., they coincide with the initial conditions of the corresponding initial modes of $C$.
- *Rule (5)* defines the $flow_1$ function of $\mathcal{F}(C, \delta)$. This rule is trivial because all the continuous variables in $X_1$ are clocks.
- *Rule (6)* defines the set $\Sigma_1$ of synchronization events of $\mathcal{F}(C, \delta)$. $\Sigma_1$ contains all the events (i.e., $\sigma \in \Sigma^{in} \cup \Sigma^{out} \cup \Sigma^{\tau}$) of $C$ and a new set of events (i.e., $\widetilde{\Sigma^{in}} = \{\tilde{\sigma} | \sigma \in \Sigma^{in}\}$) that models the perception of input events of $C$.
- *Rule (7)* defines the set of edges (and the related *jump* and *update* functions) of $\mathcal{F}(C, \delta)$. In particular:
  - *Rule (7.a)* defines, for any input event $\sigma \in \Sigma^{in}$ of $C$, the existence of an edge between two distinct modes $(q, b)$ and $(q', b')$, where $q = q'$, $b(\sigma) = \bot$ and $b'(\sigma) = \top$, with jump constraint *true*, synchronization event $\sigma$ and update constraint $R'$. Intuitively, these edges model the fact that when an input event $\sigma$ occurs in a mode in which it is non-untreated (i.e., $b(\sigma) = \bot$), $\mathcal{F}(C, \delta)$ moves to an appropriate mode in which $\sigma$ is marked as untreated (i.e., $b'(\sigma) = \top$). Moreover, the update constraint resets the variable $y_\sigma$ to 0, starting to record the time elapsed since $\sigma$ has occurred.
  - *Rule (7.b)* defines, for any input event $\sigma \in \Sigma^{in}$ of $C$, the existence of a self-loop on the mode $(q, b)$, where $b(\sigma) = \top$, with jump constraint *true*, synchronization event $\sigma$ and update constraint $R'$. Intuitively, these edges model the fact that when an input event $\sigma$ occurs in a mode where $\sigma$ is already untreated, $\mathcal{F}(C, \delta)$ remains in such a mode without changing any clock (i.e., $R'$ is the identity constraints).
  - *Rule (7.c)* defines, for any output event $\sigma \in \Sigma^{out}$ of $C$, the existence of an edge between two modes $(q, b)$ and $(q', b')$, where $b' = b$, with jump constraint obtained by enlarging the guard $g$, synchronization event $\sigma$ and update constraint $R'$ if $(q, q', g, \sigma, R)$ is a control switch in $C$. Intuitively, these edges model the fact that whether the initial Elastic controller $C$ can issue an output event $\sigma$ moving from a control mode $q$ to another one $q'$ when a guard $g$ is satisfied, $\mathcal{F}(C, \delta)$ can make the same move, but, in this case, the guard is evaluated by supposing imprecisions in clocks evaluation. Moreover, the recorded untreated input events are preserved (i.e., $b' = b$)

and the update constraints reset the variable $d$ to 0, starting to record the time elapsed since the target mode $(q', b')$ has been entered.

- *Rule (7.d)* is similar to *Rule (7.c)* and defines, for any internal event $\sigma \in \Sigma^\tau$ of $C$, an edge between two modes $(q, b)$ and $(q', b')$, where $b' = b$, with jump constraint obtained by enlarging the guard $g$, synchronization event $\sigma$ and update constraint $R'$ if $(q, q', g, \sigma, R)$ is a control switch in $C$. Intuitively, these edges model the fact that whether the initial Elastic controller $C$ can issue an internal event $\sigma$ moving from a control mode $q$ to another one $q'$ when a guard $g$ is satisfied, $\mathcal{F}(C, \delta)$ can make the same move, and also in this case the guard is evaluated by supposing imprecisions in clocks evaluation. Moreover, the recorded untreated input events are preserved (i.e., $b' = b$) and the update constraints reset the variable $d$ to 0, starting to record the time elapsed since the target mode $(q', b')$ has been entered.
- *Rule (7.e)* defines, for any event $\tilde{\sigma} \in \widetilde{\Sigma^{in}}$ of $C$, an edge between two modes $(q, b)$ and $(q', b')$, where $b(\sigma) = \top$ and $b'(\sigma) = \bot$, with jump constraint obtained by enlarging the guard $g$, synchronization event $\tilde{\sigma}$ and update constraint $R'$ if $(q, q', g, \sigma, R)$ is a control switch in $C$. Intuitively, this edge models the fact that whether the initial Elastic controller $C$ can handle an input event $\sigma$ moving from a control mode $q$ to another one $q'$ when a guard $g$ is satisfied, $\mathcal{F}(C, \delta)$ can make the same move only when it reaches a state in which $\sigma$ is untreated and then it issues a perception of such an event (i.e., $\tilde{\sigma}$). Notice that the guard of the corresponding edge is evaluated by supposing imprecisions in clocks evaluation and the update constraints reset the variable $d$ to 0, starting to record the time elapsed since the target mode $(q', b')$ has been entered.
- *Rules (7.f)* simply states that $\mathcal{F}(C, \delta)$ can always perform a synchronization $\tau$.

• Finally, *Rule (8)* defines the *inv* function, i.e., how time can elapse. Intuitively, in each control mode $(q, b)$, the invariant condition is given by the conjunction of two constraints $\varphi_1$ and $\varphi_2$.
In particular, $\varphi_1$ states that time can elapse in a mode $(q, b)$ in which an input event $\sigma$ is untreated if every outgoing control switch labeled with an event $\tilde{\sigma}$ is not urgent, that is, $\mathcal{F}(C, \delta)$ has been in $(q, b)$ for less than $\delta$ time units (i.e., $d \leq \delta$), or the guard of the control switch has been true for less than $\delta$ time units (i.e., $\neg({}^\delta g)$) or the oldest untreated occurrence of $\sigma$ has been emitted by the environment less than $\delta$ time units ago (i.e., $y_\sigma \leq \Delta$).
$\varphi_2$, instead, states that time can elapse in a mode $(q, b)$ if every outgoing control switch labeled with an output or internal event $\sigma$ is not urgent, that is, $\mathcal{F}(C, \delta)$ has been in $(q, b)$ for less than $\delta$ time units (i.e., $d \leq \delta$), or the guard of the control switch has been true for less than $\delta$ time units (i.e., $\neg({}^\delta g)$)

To graphically show the result of such transformations, Figure 2.4 depicts the timed automaton $\mathcal{F}(C, \delta)$ obtained by applying the rules summarized above on the elastic controller $C$ of Figure 2.3. For readability, self loops on modes are not depicted.

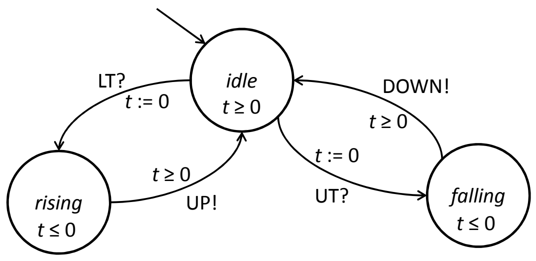


Fig. 2.3: Example of Elastic controller $C$.

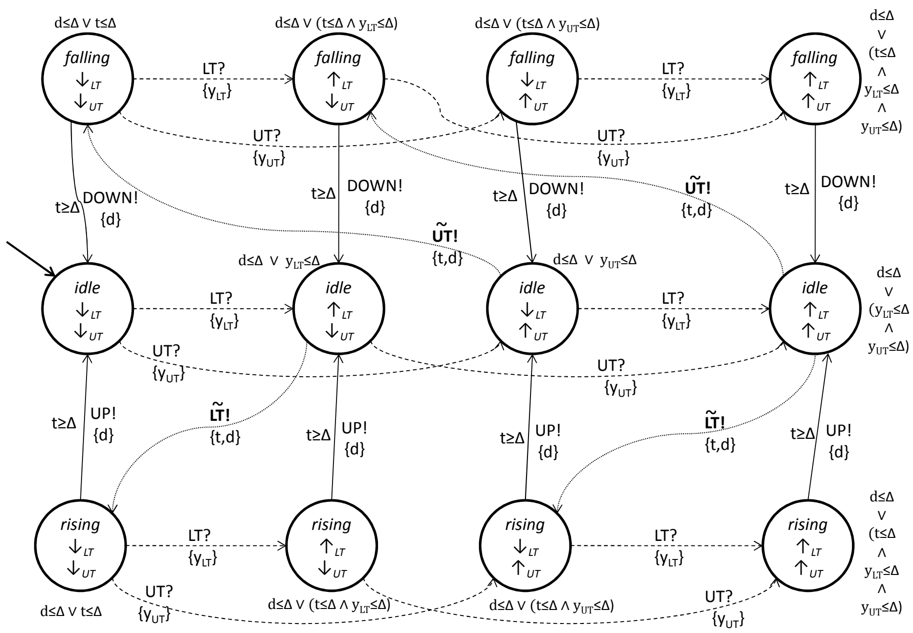


Fig. 2.4: Parameterized timed automaton derived from the Elastic controller $C$.

### 2.3.5 Synthesis procedure

The synthesis engine aims at identifying the maximum value $\Delta$ of the parameter $\delta$ for which the model $M^{\delta=\Delta}$, whose semantics is given by $[\![M^{\delta=\Delta}]\!] = [\![C^{\delta=\Delta}]\!] \parallel [\![\mathcal{E}]\!]$, satisfies the safety property $\varphi_{sp}$, i.e., $[\![M^{\delta=\Delta}]\!] \models \varphi_{sp}$.

The engine identifies the intended value $\Delta$ by using a bisection method on a finite interval of feasible values for the parameter $\delta$. The bisection method in mathematics is a method for finding a solution which repeatedly bisects an interval and then selects a subinterval in which a solution must lie for further processing. For this reason, the synthesis procedure asks the user to specify the initial interval

$[a, b]$ of values which represents the initial search space and the maximum number $N$ of interval bisections. A sound initial search space likely guarantees better performances and accuracy in identifying the value $\Delta$. Algorithm 1 reports the pseudo-code implementing the value synthesis procedure.

**Algorithm 1:** The value synthesis procedure for AASAP-based abstracted control strategies.

```
   procedure find_value($M^\delta$, $\varphi_{sp}$, $a$, $b$, $N$)
   input: the model $M^\delta$, the safety property $\varphi_{sp}$, the interval [$a$,$b$] of feasible values
          for $\delta$.
   output: maximal value $\Delta$ for which the model $M^{\delta=\Delta}$ satisfies $\varphi_{sp}$, $\Delta = 0$
           otherwise
1  $it = 0$;
2  $\Delta = 0$;
3  $mid = (a + b)/2$;
4  while ($it < N$) do
5      if $\mathcal{MC}(M^\delta, \varphi_{sp}, mid) == true$ then
6          $\Delta = mid$;
7          $a = mid$;
8      else
9          $b = mid$;
10     $mid = (a + b)/2$;
11     $it = it + 1$;
12 return $\Delta$;
```

At each step, the procedure performs a model checking process $\mathcal{MC}(M^\delta, \varphi_{sp}, mid)$ to determine if $[\![M^{\delta=mid}]\!] \vDash \varphi_{sp}$, in other words, the midpoint $mid$ of the current subinterval of $[a, b]$ is a good value for $\delta$ that enables $M^\delta$ to satisfy the safety property $\varphi_{sp}$. Now, according to the model checking result, the procedure registers the current midpoint as a candidate value for $\delta$ and selects the subinterval to be used in the next step. In particular, if $mid$ lets $M^{\delta=mid}$ to satisfy the safety property, $mid$ becomes a *candidate* maximal value $\Delta$ (i.e. $\Delta = mid$) and the new interval of search will be $[mid, b]$. For this reason the new $mid$ value is $(mid + b)/2$, i.e., the procedure will check the satisfiability of the property on new values for $\delta$ greater than the current midpoint. Otherwise, the procedure has to look for smaller values contained in the subinterval $[a, mid]$, i.e., the new $mid$ value is $(a + mid)/2$. In this way the interval that contains the desired value of the parameter $\delta$ is reduced in width by 50% at each step. The process is continued until the maximum number $N$ of iterations is reached.

### 2.3.6 Experimental results

This section reports the results obtained by applying the proposed framework on two case studies. All experiments have been performed on a workstation with Intel Xeon 2.53 GHz processors and 16GB RAM. The hybrid automaton-based models of the case studies have been described by means of CIF language. The *s-extract* tool has been used to automatically generate, from such models, the CIF parametric-models implementing the abstracted control strategies. The *cif2ariadne* and *cif2phaver* translators have been developed to automatically transform the different descriptions into the Ariadne [14] and PHAVer [67] formalisms. Then, the two hybrid domain model checkers have been used for the parameter synthesis (i.e., synthesis of the implementable control strategies) purposes. In particular, for each case study their performances have been compared.

## Watertank Control System

The watertank control system [22] is depicted in Figure 2.5, where four different automata are shown: a tank, a valve, an evaluator and a timed controller. Briefly, the system is centered on a water tank, which is characterized by an uncontrolled outbound water flow, while the inbound water flow is controlled by the aperture of a valve. The controller acts on the aperture of the valve $a$ in order to keep the water level $x$ in a safe interval.

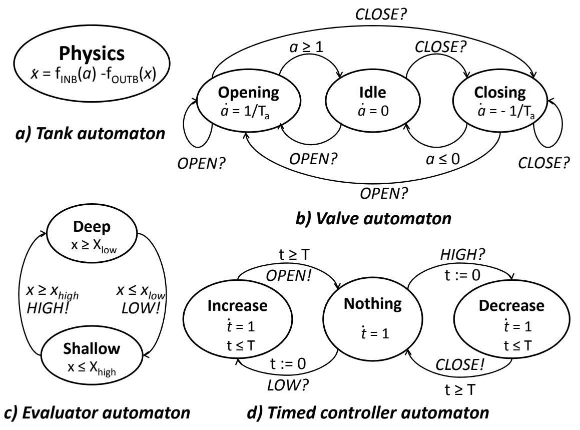


Fig. 2.5: Hybrid model of the watertank system.

The *tank* automaton is a hybrid automaton characterized by a single control mode and no transitions. The dynamic of $x$ can be kept purposely generic, since it is possible to introduce any positive (non-)linear function for $f_{OUTB}(x)$ and $f_{INB}(a)$, where $f_{OUTB}$ represents the outbound flow term and $f_{INB}$ the inbound flow term. In this case we defined the inbound and the outbound flow respectively as $f_{INB}(a) = 0.3 * a$ and $f_{OUTB}(x) = 0.02 * x$. The *valve* automaton increases (in the *Opening* mode) or decreases (in the *Closing* mode) the aperture of the valve $a$ between the two extremes 0.0 (closed) and 1.0 (opened). If the valve is closed, no inbound water flow is present, whereas if the valve is opened then the inbound flow is maximum. The automaton reacts to input events *OPEN* and *CLOSE* for starting the operation of opening or closing the valve, respectively. Notice that an input event is identified by the question mark suffix, while an output event is identified by the exclamation mark suffix. The opening and closing times that result from the dynamics are fixed and equal to $T_a$. The *evaluator* automaton evaluates the value of $x$: if it is higher than an upper threshold $x_{high}$ such that $x_{high} < x_{max}$, it issues an *HIGH* event, while if $x$ is lower than a lower threshold $x_{low}$ such that $x_{low} > x_{min}$, it issues a *LOW* event. The timed automaton *controller* models the control strategy. It receives the *HIGH* and *LOW* events and, after a fixed time $T$, issues the corresponding *CLOSE* and *OPEN* events for the valve.

The safety property $\varphi_{sp}$ used for defining the desired instance of the parametric safety verification problem (i.e., $[\![C^{\delta=\Delta}]\!] \parallel [\![\mathcal{E}]\!] \vDash \varphi_{sp}$) on the watertank control system has been $\varphi_{sp} = x < x_{max}$ & $x > x_{min}$. The property requires that the water level $x$ is always kept between the safe bounds $x_{max} = 8.25$ and $x_{min} = 5.25$ (while $x_{high} = 8.0$ and $x_{low} = 5.5$).

Given such an instance of the problem, the synthesis engine returned an upper bound for $\Delta$ equal to *0.289898872375*, for which the relation $\Delta > 4\Delta_P + 3\Delta_L$ must hold. For example, this means that the control strategy which needs to be executed on an hardware device characterized by a clock frequency equal to *14 Hz* (i.e. $\Delta_P = 1/14$ sec.), needs the latency for the correct handling of the incoming and outgoing synchronization events to be upper bounded by *1 ms* (i.e. $\Delta_L = 1/1000$ sec.). In fact, given the same frequency of *14 Hz*, if the synchronization latency reaches the *2 ms*, the control strategy is unable to promptly provide output signals to the environment, thus not preventing it from entering an error state. If on the other hand, given the same *1 ms* latency, the device executing the control strategy has a clock frequency of *13 Hz*, then it is not possible to correctly sample the incoming events: the controller might detect the perception of an event too late, and consequently it could miss firing a transition; this situation would in turn cause the environment to enter an error state while waiting on a synchronization.

Table 2.1: Results of the parametric safety verification problem for the watertank control system.

| $\mathcal{MC}$ | **Search space** | **# Bisect.** | **Time (sec)** |
|---|---|---|---|
| *PHAVer* | [0, 0.5] | 15 | 89635.72 |
| *Ariadne* | [0, 0.5] | 15 | 121255.06 |

Table 2.1 shows a comparison between the amounts of time required for synthesizing the value $\Delta$ by using the two model checkers. In particular, column $\mathcal{MC}$ reports the name of the model checkers; columns *search space* and *# Bisect.* report the initial search space and the maximum number of bisections used for synthesizing the value $\Delta$; finally, column *Time* reports the total time (in seconds) required for the synthesis.

According to the results, PHAVer has turned out to be faster w.r.t. Ariadne in the synthesis of the value $\Delta$. This is only due to the fact that PHAVer is a hybrid domain model checker optimized for analyzing linear affine dynamics, as the ones described by the watertank model, whereas Ariadne is suited for more complex dynamics. So, Ariadne seems to pay the gain of handling non-linear ODE in terms of performances when used for verifying models with linear ODE.

## Power Supply Selector Control System

The hybrid model depicted in Figure 2.6 represents the Power Supply Selector (PSS) control system included in the MAGALI platform [18]. The basic behavior of the PSS is to control the supply voltage $V_c$ of a generic unit of the platform. More precisely, due to DVFS (Dynamic Voltage and Frequency Scaling) operations, the supply voltage can switch dynamically between two values, *High* and *Low*. During such transitions, the supply voltage $V_c$ (which supplies the considerable load given by the equivalent resistance of the core circuit) must follow a linearly rising/falling reference voltage $V_r$ as closely as possible. Essentially, a controller provides periodic $UP$ or $DOWN$ events that ultimately make the supply voltage rise or fall by a fixed step of voltage: by ensuring that the controller issues events at a properly high frequency, the core voltage can follow the reference voltage, guaranteeing a bounded voltage difference $V_d$ between $V_r$ and $V_c$.

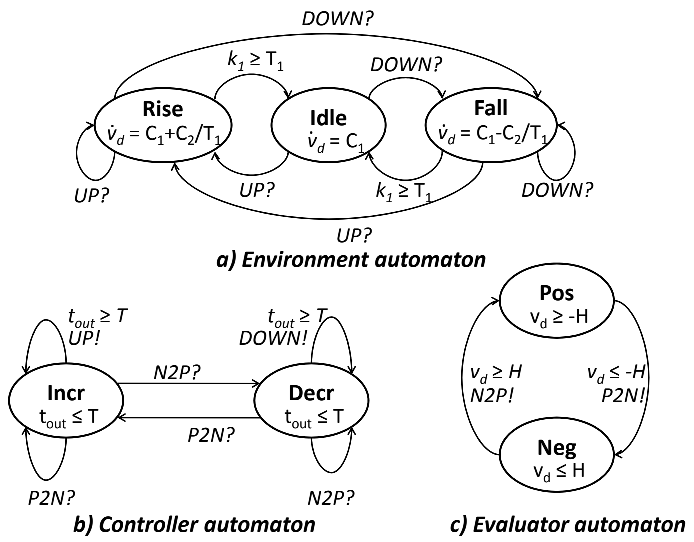


Fig. 2.6: Hybrid model of the Power Supply Selector system.

The hybrid system consists of three automata: environment, controller and evaluator. In the *environment*, the behavior of the voltage difference $V_d$ between the supply and reference voltages is modeled. The hybrid automaton can operate in an *Idle* mode, in which the derivative of $V_d$ is constant. In the *Rise* mode, an additional constant positive component makes $V_d$ temporarily increase in time, while in the *Fall* mode there is an additional negative component which temporarily increments the rate of decrease. The switch between modes is provided by the $UP$ and $DOWN$ commands from the controller. This automaton behavior abstracts the platform behavior in [18] when modulating the series-resistance between the core resistive load and the supply voltage, in order to control the difference of potential provided to the core. The *evaluator* simply translates the variations of sign of the $V_d$ variable into *N2P* (negative-to-positive) and *P2N* (positive-to-negative) events that are received by the controller, thus jumping between the *Pos* (positive) and *Neg* (negative) modes. The evaluator is able to produce a result as soon as $V_d$ is greater than a small threshold $H$ (which is sensibly lower than the operating core voltage). The *controller* features two modes: *Incr* (increase) and *Decr* (decrease), where a periodic *UP* or *DOWN* event is produced, respectively. Essentially, it compensates a *N2P* (*P2N*) event by periodically issuing a *DOWN* (*UP*) event to the environment, until a *P2N* (*N2P*) event is received and the behavior is reversed.

The parameters used for describing the automata timings (i.e. $T$, $T_1$, the slope of the $V_d$ variable in the locations of the environment) are: $T = T_1 = 0.01s$, $H = 0.001$ (where $V_{high} = 1.2$ and $V_{low} = 0.8$), the dynamics for the time derivative of $V_d$ are:

- *Idle*: $\dot{V_d} = C_1$;
- *Rise*: $\dot{V_d} = C_1 + C_2/T_1$;
- *Fall*: $\dot{V_d} = C_1 - C_2/T_1$;

where $C_1 = -0.4$ (which corresponds to a relatively slow falling reference voltage, since the core voltage is essentially stable on a specific value) and $C_2 = 0.05$ (which is the absolute value of the supply voltage step taken after an *UP* or *DOWN* event has been handled by the environment).

To check the parametric safety verification problem on the PSS system, the property chosen has been $\varphi_{sp} = (V_d >= -L \ \& \ V_d <= L)$, meaning that the voltage difference $V_d = V_r - V_c$ must maintain an absolute value not greater than $L = 0.1$.

For the PSS system, the synthesis engine retrieved an upper bound for $\Delta$ equal to *0.00112487792969*. Again, for the same considerations reported into the previous case study, the control strategy that needs to be executed on a device whose clock frequency is *10 Mhz* (i.e. clock precision of *10 $\mu s$*), requires the synchronization latency to be bounded by *60 $\mu s$*.

Table 2.2: Results of the parametric safety verification problem for the PSS system.

| $\mathcal{MC}$ | **Search space** | **# Bisect.** | **Time (sec)** |
|---|---|---|---|
| *PHAVer* | [0, 0.002] | 15 | 4273.55 |
| *Ariadne* | [0, 0.002] | 15 | 7436.61 |

Table 2.2 shows a comparison between the total time required for synthesizing the value $\Delta$ by using the two model checkers. In particular, column $\mathcal{MC}$ reports the name of the model checkers, columns *search space* and *# Bisect.* report the initial search space and the maximum number of bisections, respectively, which

are used for synthesizing the value $\Delta$. Finally, column *Time* reports the time (in seconds) required for the synthesis.

Also on this case study, PHAVer outperformed Ariadne. Again the main motivation of such a difference in the synthesis times relies on the different engines which computes the set of reachable states for the hybrid model. While PHAVer is suited for rectangular and linear affine dynamics, as the ones used in the PSS model, Ariadne is suited for non-linear affine dynamics. So, again Ariadne pays its capability of handling very complex dynamics in terms of performances on models with simpler dynamics.

## 2.4 Synthesis of implementable control strategies for LLHA

### 2.4.1 Lazy Linear Hybrid Automata [5]

In what follows the class of lazy linear hybrid automata is being described. This particular class of hybrid automata has the interesting property that its discrete time behavior can be computed and represented as finite state automaton as shown in [5].

**Definition 2.22 (Lazy Linear Hybrid Automaton).** *A finite precision lazy linear hybrid automaton (LLHA) is a tuple $\langle X, Q, Q_0, init, inv, flow, E, jump, Act, D, \epsilon, B, P\rangle$. The components of LLHA are as follows:*

- Variables. *A finite set $X = \{x_1, \ldots, x_n\}$ of real-numbered variables. $\dot{X}$ stands for the set $\{\dot{x}_1, \ldots, \dot{x}_n\}$ of dotted variables and $X'$ stands for the set $\{x'_1, \ldots, x'_n\}$ of primed variables.*
- Control modes. *A finite set $Q$ of control modes. $Q_0 \subseteq Q$ denotes the set of initial modes.*
- Initial conditions. *A labeling function init that assigns to each control mode $q \in Q_0$ an initial predicate. Each initial predicate $init(q)$ is a convex (non-)linear formula over the variables in $X$.*
- Invariant conditions. *A labeling function inv that assigns to each control mode $q \in Q$ an invariant predicate. Each invariant predicate $inv(q)$ is a convex (non-)linear formula over the variables in $X$.*
- Flow conditions. *A labeling function $flow$ that assigns to each control mode $q \in Q$ a flow predicate. Let $\dot{X}_q$ be the finite set $\{\rho_q^1, \rho_q^2, \ldots, \rho_q^k\}$ of possible rate vectors $\rho_q \in \mathbb{Q}^n$ which specify the rate $\rho_q(i)$ at which each variable $x_i$ evolves when the automaton is in the control mode $q$. Each flow predicate $flow(q)$ is of the form $(\dot{x}_1 = \rho_q^j(1)) \wedge \ldots \wedge (\dot{x}_n = \rho_q^j(n))$ with $\rho_q^j \in \dot{X}_q$.*
- Control switches. *A set $E$ of edges $(q, q')$ from a source mode $q \in Q$ to a target mode $q' \in Q$.*
- Jump conditions. *A labeling function jump that assigns to each control switch $e \in E$ a predicate. Each jump predicate $jump(e)$ from the control mode $q$ to $q'$, is given by a convex (non-)linear formula over the variables in $X$.*
- Update conditions. *A labeling function update that assigns to each control switch $e \in E$ a predicate. Each update predicate $update(e)$ from the control mode $q$ to $q'$, is given by the identity predicate over the variables in $X \cup X'$ (e.g., $x'_i = x_i$).*

- Actions. *A finite set Act of actions and an edge labeling function action* $: E \to$ *Act that assigns to each control switch an action.*
- Delay parameters. $D = g, \delta_g, h, \delta_h$ *is the set of delay parameters such that* $0 \leq g \leq g + \delta_g < h \leq h + \delta_h \leq P$, *where h denotes the sensing delay, g denotes the actuation delay and P is the sampling interval of the controller.*
- Precision. $\epsilon_i$ *is the precision of measurement of variable* $x_i$.
- Range. $B_i = [B_{i_{min}}, B_{i_{max}}]$ *is the range of the variable* $x_i$.
- Period. *P represents the time period associated with the discrete controller, i.e., control mode switches take place at times* $T_0$, $T_1$, $T_2$, ... *where* $T_{k+1} = T_k + P$.

To keep the notation compact, in what follows, $q \xrightarrow{a,\varphi} q'$ is used to denote that there exists a control switch $e = (q, q')$ with $q \neq q'$, $a = action(e)$ and $\varphi = jump(e) \wedge update(e)$ in $A$.

It is worth noting that, unlike the conventional definition of linear hybrid automata [79], invariants and guards in LLHA can be non-linear. The flows in linear hybrid automata are represented using rectangular formulas over only the rates of change of variables. Under the assumption of finite precision, such flows can be considered as a finite set of constant values of rate of change of different continuous variables. Thus, LLHA can be used for representing hybrid systems with rectangular flows [5].

Let $A$ be a lazy linear hybrid automaton as defined above. The semantics of $A$ is being defined in terms of an associated transition relation over its states.

**Definition 2.23 (State of a LLHA).** *A* state *of a lazy linear hybrid automaton A is a triple* $(q, V, \hat{q})$ *where* $q$, $\hat{q}$ *are control modes and V is a valuation. q is the control mode holding at the current time instant and* $\hat{q}$ *is the control mode that held at the previous time instant. V captures the actual values of the variables at the current instant. The state* $(q, V, \hat{q})$ *is feasible if and only if* $V(i) \in [B_{min_i}, B_{max_i}]$ *for every i.*

The initial state is, by convention, the triple $(q_{init}, V_{init}, q_{init})$ where $q_{init} \in Q_0$ and $V_{init}$ is such that $V_{init} \vDash init(q_{init})$. It is assumed without loss of generality that the initial state is feasible. Let $S_A$ denote the set of states of $A$. To better understand the dynamics of $A$ it is useful to distinguish each move of the LLHA into a time-passage move followed by an instantaneous transition.

At the time instant $T_0$, the automaton will be in its initial state. Suppose that $A$ is in the state $(q_k, V_k, \hat{q}_k)$ at $T_k$. Then, let the time pass. At time instant $T_{k+1}$, the automaton will make an instantaneous move by performing an action $a$ or the silent action $\tau$ and move to a state $(q_{k+1}, V_{k+1}, \hat{q}_{k+1})$. The silent action $\tau$ is used to record that no mode change has taken place during this move. On the contrary, each action $a \in Act$ labeling a control switch in $A$ is used to record that a mode change has taken place during the move. The two distinct sub-steps of a move (unit-time-passage followed by an instantaneous transition) highlighted above can be merged into one time-abstract transition labeled by a member of $Act$ or by $\tau$. With this scheme in mind, it is possible to define the transition relation $\Longrightarrow$ of $A$ as follows.

**Definition 2.24 (Transition relation of a LLHA).** $\Longrightarrow \subseteq S_A \times (Act \cup \{\tau\}) \times S_A$ *is such that:*

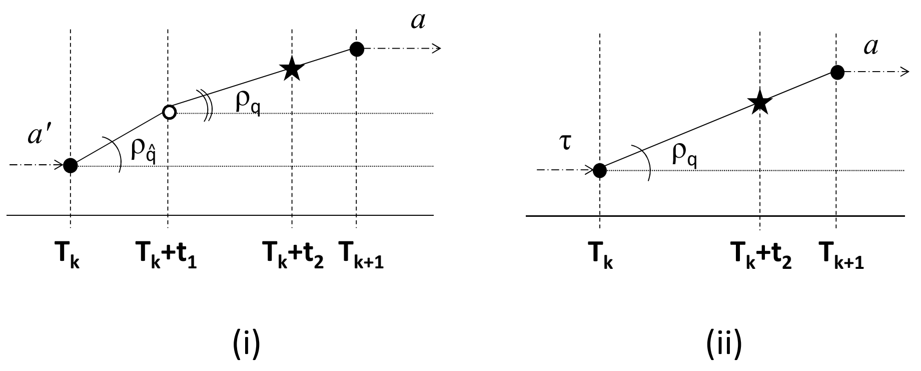


Fig. 2.7: Lazy transitions with sensing delays.

- *Let $(q, V, \hat{q})$, $(q', V', \hat{q}') \in S_A$ be states and $a \in Act$. Then $(q, V, \hat{q}) \overset{a}{\Longrightarrow} (q', V', \hat{q}')$ if and only if $\hat{q}' = q$ and there exist a control switch of the form $q \xrightarrow{a,\varphi} q'$ in $A$ and $t_1 \in [g, g + \delta_g]^n$, $t_2 \in [h, h + \delta_h]^n$ such that the following conditions are satisfied:*
  1. *Let $v_i = V(i) + \rho_{\hat{q}}(i) \cdot t_1(i) + \rho_q(i) \cdot (t_2(i) - t_1(i))$ for each $i$. Then $(\langle v_1 \rangle, \ldots \langle v_n \rangle)$ satisfies $\varphi$ and each $\langle v_i \rangle$ represents the digitized value of the variable $x_i$ that have been rounded using the value of $\epsilon_i$.*
  2. *$V'(i) = V(i) + \rho_{\hat{q}}(i) \cdot t_1(i) + \rho_q(i) \cdot (P - t_1(i))$ for each $i$.*
- *Let $(q, V, \hat{q})$, $(q', V', \hat{q}') \in S_A$ be states. Then $(q, V, \hat{q}) \overset{\tau}{\Longrightarrow} (q', V', \hat{q}')$ if and only if $q' = \hat{q}' = q$ and there exists $t_1 \in [g, g + \delta_g]^n$ such that:*
  1. *$V'(i) = V(i) + \rho_{\hat{q}}(i) \cdot t_1(i) + \rho_q(i) \cdot (P - t_1(i))$ for each $i$.*

Basically there are four possible transition types depending on whether $q = \hat{q}$ and $\alpha \in Act$.

Suppose, as shown in Figure 2.7, that $(q, V, \hat{q}) \overset{a}{\Longrightarrow} (q', V', \hat{q}')$ with $a \in Act$ and assume that $q \xrightarrow{a,\varphi} q'$ in $A$ and $t_1 \in [g, g + \delta_g]^n$ and $t_2 \in [h, h + \delta_h]^n$ are as specified above. Notice that $q' \neq q$ by definition of $\longrightarrow$ in $A$, while the requirement $\hat{q}' = q$ follows from the convention that $\hat{q}'$ is the control mode that held in the previous instant and it is known this was $q$. First consider the case $q \neq \hat{q}$ and suppose that the state $(q, V, \hat{q})$ holds at $T_k$ (Figure 2.7.(i)). Then $q \neq \hat{q}$ means that a mode change from $\hat{q}$ to $q$ has just taken place (instantaneously) at $T_k$ based on the observations that were made available at $T_k$. However, at $T_k$, the automaton will continue to evolve at the rate dictated by $\rho_{\hat{q}}$. Indeed, each $x_i$ will, starting from $T_k$, evolve at rate $\rho_{\hat{q}}(i)$ until some $T_k + t_1$ with $t_1 \in [g, g + \delta_g]$. It will then start to evolve at rate $\rho_q(i)$ until $T_{k+1}$. Consequently, at $T_{k+1}$, the value of $x_i$ will be $V'(i) = V(i) + \rho_{\hat{q}}(i) \cdot t_1 + \rho_q(i) \cdot (P - t_1)$. On the other hand, $q' \neq q$ implies that another instantaneous mode change has taken place at $T_{k+1}$ based on the measurements made in the interval $[T_k + h, T_k + h + \delta_h]$. Suppose $x_i$ was measured at $T_k + t_2$ with $t_2 \in [T_k + h, Tk + h + \delta_h]$. Then in order for the transition $q \xrightarrow{a,\varphi} q'$ of $A$ to be enabled at $T_{k+1}$, it must be the case that the observed values of $x_i$ at $T_k + t_2$ satisfy the guard $\varphi$. Then these values are $\langle v_i \rangle$ with $v_i = V(i) + \rho_{\hat{q}}(i) \cdot t_1 + \rho_q(i) \cdot (t_2 - t_1)$. This explains the demands placed on the transition $(q, V, \hat{q}) \overset{a}{\Longrightarrow} (q', V', \hat{q}')$.

It is worth noting that if $q = \hat{q}$, i.e., no mode change has taken place at $T_k$ (Figure 2.7.(ii)), then $V'(i) = V(i) + \rho_q(i) \cdot t_1 + \rho_q(i) \cdot (P - t_1) = V(i) + \rho_q(i) \cdot P$ as it should be. Furthermore, $v_i = V(i) + \rho_q(i) \cdot t_1 + \rho_q(i) \cdot (t_2 - t_1) = V(i) + \rho_q(i) \cdot t_2$.

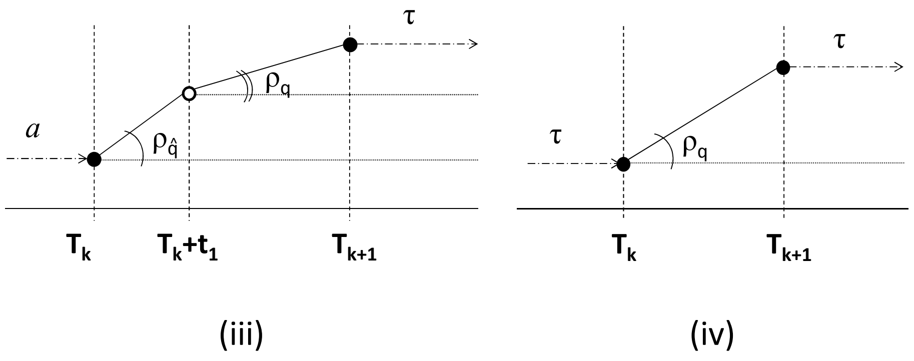


Fig. 2.8: Lazy transitions with actuation delays.

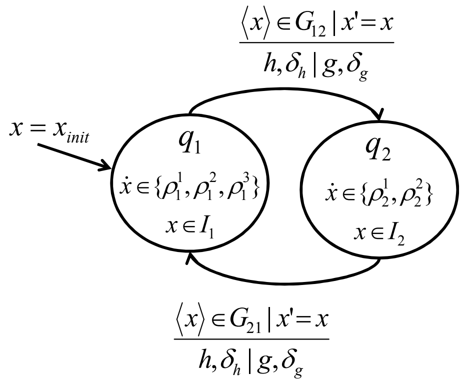
As shown in Figure 2.8, similar (and simpler) considerations motivate the demands placed on transitions of the form $(q, V, \hat{q}) \overset{\tau}{\Longrightarrow} (q', V', \hat{q}')$ where no mode changes take place at $T_{k+1}$. Here again, in case $q \neq \hat{q}$, i.e., a mode change has taken place at $T_k$ (Figure 2.8.(iii)), there exists $t_1 \in [g, g + \delta_g]$ such that $V'(i) = V(i) + \rho_{\hat{q}}(i) \cdot t_1 + \rho_q(i) \cdot (P - t_1)$.

On the contrary, in case $q = \hat{q}$ (Figure 2.8.(iv)), $V'(i)$ is determined uniquely, namely, $V'(i) = V(i) + \rho_q(i) \cdot P$.

From the semantics defined above, it is possible to derive the notions of trajectory of a LLHA and the reachability relation between states.

**Definition 2.25 (Trajectory of a LLHA).** *Let $A$ be a lazy linear hybrid automaton and let $(q, V, \hat{q})$ a state of $A$. A trajectory of $A$ from $(q, V, \hat{q})$ is a sequence of states $(q_i, V_i, \hat{q}_i)$ with $i \geq 0$, such that $(q_0, V_0, \hat{q}_0) = (q, V, \hat{q})$ and $(q_{i-1}, V_{i-1}, \hat{q}_{i-1}) \overset{\alpha}{\Longrightarrow} (q_i, V_i, \hat{q}_i)$ for some $\alpha \in Act \cup \{\tau\}$.*

**Example** (*A generic lazy linear hybrid automaton*). Figure 2.9 sketches a lazy linear hybrid automaton $A$ with two control modes $q_1$ and $q_2$. Let $i$ and $j$ be such that $i, j \in \{1, 2\}$ and $i \neq j$. Like in the classic hybrid automaton model, an invariant condition $inv(q_i)$ is defined as a subset $I_i$ of $\mathbb{R}^n$ and the LLHA can stay in the control mode $q_i$ if the valuation of the variable $x$ satisfies the invariant condition. The jump condition of a control switch $e_{ij} = (q_i, q_j)$ is specified by a guard set $G_{ij}$ and a reset function that, by definition of LLHA, is always the identity function. Unlike in the classic HA model, the control switch $e_{ij}$ is enabled only if the digitized values $\langle x \rangle$ detected by the sensor belong to $G_{ij}$ and, moreover, sensing and actuation delays (i.e., $h, \delta_h$ and $g, \delta_g$, respectively) are associated to



$\frac{\langle x\rangle \in G_{12} \mid x'=x}{h,\delta_h \mid g,\delta_g}$
$x = x_{init}$
$q_1$
$\dot{x} \in \{\rho_1^1, \rho_1^2, \rho_1^3\}$
$x \in I_1$
$q_2$
$\dot{x} \in \{\rho_2^1, \rho_2^2\}$
$x \in I_2$
$\frac{\langle x\rangle \in G_{21} \mid x'=x}{h,\delta_h \mid g,\delta_g}$

Fig. 2.9: Example of a lazy linear hybrid automaton model.

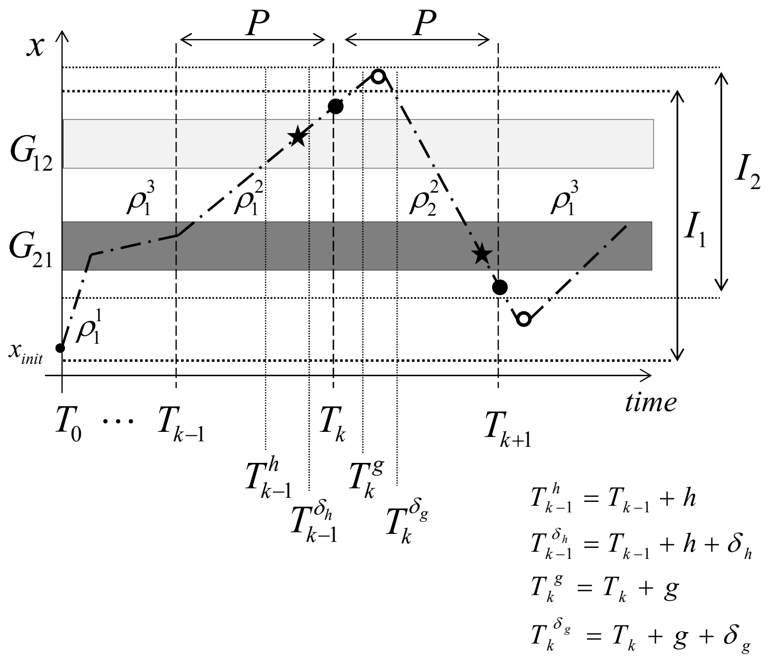


Fig. 2.10: Example of a possible lazy linear hybrid automaton trajectory.

the control switch. Finally, each flow condition $flow(q_i)$ constraints the evolution of the continuous variable $x$ to one of the possible rates $\rho_i^n$ allowed in the mode (e.g., $\{\rho_1^1, \rho_1^2, \rho_1^3\}$ in $q_1$).

Figure 2.10 sketches, instead, part of a trajectory of such a LLHA starting from the initial state $(q_1, x_{init})$. In the example, the trajectory keeps following the dynamics $flow(q_1)$ until the time instant $T_k$. In fact, the control switch $e_{12}$ is not enabled as soon as the trajectory reaches the guard set $G_{12}$ because of the semantics of LLHA: a jump condition can be evaluated only at periodic time points and by considering the digitized values detected by the sensor at some instant (marked with $\star$) in the interval $[T^h_{k-1}, T^{\delta_h}_{k-1}]$. As shown in the figure, at $T_k$, the invariant condition of the mode $q_1$ is still satisfied, thus, the LLHA can either switch to $q_2$ or continue with the dynamics of $q_1$. Let assume that the automaton performs a control switch (marked with $\bullet$) and moves in $q_2$. When LLHA switches from $q_1$ to $q_2$, it resets the continuous variable $x$ according to the predicate specified by the jump condition, i.e., the identity function. Thus, in this case, the trajectory starts from the same state reached at $T_k$. Notice that the trajectory keeps following the dynamics of $q_1$ due to the presence of an actuation delay (i.e., $g, \delta_g$) on the control switch. In fact, only at some time (marked with $\circ$) in the interval $[T^g_k, T^{\delta_g}_k]$ the trajectory changes according to the rates specified by the flow condition of $q_2$ (i.e., $\dot{x} \in \{\rho^1_2, \rho^2_2\}$). Then the trajectory follows that flow rate until the invariant $I_2$ is violated or the jump condition $G_{21}$ is satisfied allowing the automaton to jump back in the mode $q_1$.

**Definition 2.26 (Reachability relation between states of a LLHA).** *Let $A$ be a lazy linear hybrid automaton. A state $(q, V, \hat{q})$ reaches a state $(q', V', \hat{q}')$ if there exists a finite trajectory of states $(q_i, V_i, \hat{q}_i)$, with $0 \leq i \leq n$, such that $(q_0, V_0, \hat{q}_0) = (q, V, \hat{q})$ and $(q_n, V_n, \hat{q}_n) = (q', V', \hat{q}')$. $\mathcal{RC}(q, V, \hat{q})$ is used to denote the set of states reachable from $(q, V, \hat{q})$. $\mathcal{RC}$ is used to denote the set of all the possible states reachable from the initial ones.*

### 2.4.2 Problem definition

The synthesis of implementable control strategy for a LLHA consists of determining if there exist legal values for the clock precision (i.e., $P$), and upper-bounds for sensing and actuation delays (i.e., $T_{SD}$ and $T_{AD}$, respectively) for which the control strategy modeled in the LLHA is able to satisfy the safety properties the hybrid system has to ensure.

Let $A$ be a lazy linear hybrid automaton such that $\mathcal{S}_A$ is the set of the possible states, $Init$ is a function that constraints the control modes and the variables value in the set of initial states, and $\mathcal{TR}$ is the transition relation that models the lazy behavior of $A$. Let $\varphi_{safe}$ be a function that specifies the constraints for identifying the states satisfying the safety properties for the hybrid system. The synthesis problem summarized above can be formalized by a Quantified Boolean Formula (QBF), i.e., a formula in which propositional variables can be either quantified existentially or universally, as follows:

$$\forall S_0, \ldots, S_n, \exists P, T_{SD}, T_{AD} : Init(S_0) \wedge \bigwedge_{i=0}^{n} \mathcal{TR}(S_i, S_{i+1}, P, T_{SD}, T_{AD}) \rightarrow \bigwedge_{i=0}^{n} \varphi_{safe}(S_i) \tag{2.1}$$

where each $S_i$ is in $\mathcal{S}_A$. Intuitively, the formula states that every state $S_i$ reachable from one of the initial states satisfies the safety property $\varphi_{safe}$.

An efficient way to solve this problem consists of deriving from Formula 2.1 a reachability problem on $A$. Such a reachability problem should focus on identifying the existence of *bad states*, i.e., states $S_i$ violating the safety properties and are reachable from the initial states of $A$:

$$\forall S_0, \dots, S_n, \exists P, T_{SD}, T_{AD} : Init(S_0) \wedge \bigwedge_{i=0}^{n} \mathcal{TR}(S_i, S_{i+1}, P, T_{SD}, T_{AD}) \wedge \bigvee_{i=0}^{n} \neg\varphi_{safe}(S_i) \tag{2.2}$$

where each $S_i$ is in $\mathcal{S}_A$.

The unsatisfiability of Formula 2.2 proves the validity of Formula 2.1. Notice that the satisfiability of Formula 2.2 may be proved or disproved by applying a Satisfiability Modulo Theory (SMT) [15] decision procedure on its propositional part. Unfortunately, the number of copies of the transition relation $\mathcal{TR}$ in the formula coincides with the number of steps being checked. As a consequence, for a complete check, the SMT procedure must be invoked on formulas containing an exponential number of copies of the transition relation $\mathcal{TR}$ leading to the typical memory explosion.

To overcome such a problem, it is possible to limit the analysis to a bounded reachability problem: the transition relation may be unrolled only a finite number $k$ of times, where $k$ is the *reachability diameter* [96] of the LLHA. Thus, the formula checks if a bad state $S_{i \leq k}$ is reachable from the initial state $S_0$. This is an instance of the classical Bounded Model Checking (BMC) [23] problem, and there exist well-know techniques and tools to solve it.

In what follows, a symbolic BMC encoding for modeling Formula 2.2 is proposed. Then, a synthesis procedure for identifying the maximum values of clock precision ($P$), sensing and actuation delays ($T_{SD}$,$T_{AD}$ respectively) for satisfying the safety specification $\varphi_{safe}$ is proposed.

### 2.4.3 Contributions

In what follows it is shown that, by using LLHA, the problem of synthesizing a discrete-time and finite-precision control strategy for a hybrid system is decidable. In particular, the main contributions of Section 2.4 are as follows:

- it defines a symbolic encoding of the set $\mathcal{RC}$ of reachable states of a LLHA that reduces the synthesis of implementable control strategies for LLHA to the state reachability problem on LLHA. In fact the symbolic encoding of $\mathcal{RC}$, unlike the one proposed in [89], explicitly models the finite precision of clock and sensors, as well as the sensing and actuation delays by means of parameters. Then, by verifying the safety properties as reachability queries, it is possible to identify values for such parameters which make the control strategy implementable, i.e., the control strategy is able to handle the continuous plant by following finite-precision and discrete-time behaviors.
- it proposes a synthesis procedure that, starting from a set of feasible values for the different parameters, identifies for each of them the maximum values which enable a LLHA to satisfy its required safety properties.

The following sections describe all the details of the proposed approach.

### 2.4.4 Symbolic BMC encoding

This section describes how the reachability problem on a LLHA can be symbolically represented by means of a parametric BMC formula. At first, an useful definition is provided.

**Definition 2.27 (Frame).** *A frame $F$ is a tuple $\langle K, c, t_1, t_2, t, q\rangle$ where $K = (k_1, k_2 \dots k_n)$ represents the variables evaluation; $c$ is the time elapsed since the last clock tick; $t_1$ is the actuation delay; $t_2$ is the sensing delay; $t$ is the time before transition to next frame; $q$ denotes the current control mode in $F$.*

Intuitively, a frame $F$ reports the state of the LLHA at the time instant $t$. Due to the finite-precision assumption of LLHA, each variable $k_i$ can be symbolically modeled by using a bit-vector [33] whose size depends on the legal range $B_i$ associated to the variable. The same considerations apply to symbolically represent the other quantities reported into a frame.

The semantics of a LLHA is described by a finite conjunction of predicates, modeling a symbolic transition relation, over the sequence of frames (i.e., states) $F_1$, $F_2$, $\dots$, $F_n$.

The initial state $Init(F_0)$ is defined as the predicate

$$Init(F_0) \equiv (q = q_{init}) \wedge \phi_0(K) \wedge (c = 0) \wedge (t_1 = 0) \wedge (t_2 = 0) \tag{2.3}$$

where $q_{init}$ denotes the initial control mode and $\phi_0(K)$ the initial constraint over continuous variables (e.g., variables initialization). At the initial step the values of the continuous variables are known, thus, no sensing delays can happen (i.e., $t_2 = 0$). Moreover, also the actuation delay $t_1$ is constrained to 0. In fact, no mode switches could have happened before the initial step. To make a mode switch possible, the $Init$ predicate constraints $c$ (i.e., time units before the next clock tick) to 0. This is due to the fact that, according to the LLHA semantics (Section 2.4.1), a mode switch can happen only at discrete time instants corresponding to clock ticks. Thus, the initial frame $F_0$ is being defined as $\langle K^0 \doteq \phi_0(K), c^0 \doteq 0, t_1^0 \doteq 0, t_2^0 \doteq 0, t^0 \doteq 0\rangle$.

On the contrary, each step of the symbolic transition relation is described by the predicate $\mathcal{TR}$ that defines constraints over the current frame $F_m$ by means of its previous frame $F_{m-1}$. Such a predicate consists of the disjunction of all possible control switches ($\mathcal{J}_{ij}$) and flow evolutions ($\mathcal{E}_i$) and ($\bar{\mathcal{E}}_i$) on such frames.

$$\mathcal{TR}(F_{m-1}, F_m) \equiv \bigvee_{(i,j)\in E} \mathcal{J}_{ij}(F_{m-1}, F_m) \vee \bigvee_{i\in Q} \mathcal{E}_i(F_{m-1}, F_m) \vee \bigvee_{i\in Q} \bar{\mathcal{E}}_i(F_{m-1}, F_m) \tag{2.4}$$

Intuitively, $\mathcal{J}_{ij}(F_{m-1}, F_m)$ specifies that a switch between modes $i$ and $j$ can happen. $\mathcal{E}_i(F_{m-1}, F_m)$ specifies that time can continue to elapse into the current mode $i$. Finally, $\bar{\mathcal{E}}_i(F_{m-1}, F_m)$ specifies that time can continue to elapse into the current mode $i$ despite of its invariant till the first occurring clock tick at which will be mandatory leave the control mode.

The formal definitions of predicates $\mathcal{J}_{ij}$, $\mathcal{E}_i$ and $\bar{\mathcal{E}}_i$, are specified in terms of quantities which, in turn, depend on two functions: *compensated for sensing delay* ($csd$) and *compensated for actuation delay* ($cad$).

These two functions *csd* (Equation 2.5) and *cad* (Equation 2.6) map a valuation $K$ of the continuous variables to a set of possible corresponding valuations obtained after compensating for sensing and actuation delay, respectively.

$$csd(K, i, t_2) \equiv \{(k_1 - \dot{k}_1 \cdot t_2, \ldots, k_n - \dot{k}_n \cdot t_2) \mid (\dot{k}_1, \ldots, \dot{k}_n) \models flow(i)\} \tag{2.5}$$

$$\begin{aligned} cad(K, h, i, t_1, t) \equiv \{(k_1 - (\dot{k}_{1h} - \dot{k}_{1i}) \cdot t_1 + \dot{k}_{1i} \cdot t, \ldots, k_n - (\dot{k}_{nh} - \dot{k}_{ni}) \cdot t_1 + \dot{k}_{ni} \cdot t) \mid \\ (\dot{k}_{1h}, \ldots, \dot{k}_{nh}) \models flow(h), (\dot{k}_{1i}, \ldots, \dot{k}_{ni}) \models flow(i)\} \end{aligned} \tag{2.6}$$

Now, let $F_m = (K^m, c^m, t_1^m, t_2^m, t^m, q^m)$ and $F_{m-1} = (K^{m-1}, c^{m-1}, t_1^{m-1}, t_2^{m-1}, t^{m-1}, q^{m-1})$ be the current and previous frames, respectively. The following quantities are needed for formalizing the aforementioned $\mathcal{J}_{ij}$, $\mathcal{E}_i$ and $\bar{\mathcal{E}}_i$ predicates.

The predicate $I_i$,

$$I_i(F_m) \equiv (i = q^m) \wedge (0 \leq t_2^m \leq T_{SD}) \wedge \exists K'[K' \in csd(K^m, l^m, t_2^m) \wedge inv_i(K')] \tag{2.7}$$

evaluated on the frame $F_m$, tests the satisfiability of the invariant at the control mode $i$ in $F_m$. In particular, the predicate checks the existence of a valuation $K'$, derived from $K^m$ by compensating for the sensing delays in $t_2^m$, that satisfies the invariant $inv_{q^m}$. Notice that the values in $t_2^m$ range over the interval $[0, T_{SD}]$ where the upper-bound $T_{SD}$ is a parameter whose maximum value is being synthesized during the verification. Intuitively, the maximum value of $T_{SD}$ will represent the slowest admissible sensing delay ensuring the correctness of the control strategy modeled by the LLHA.

The predicate $g_{ij}$,

$$g_{ij}(F_{m-1}, F_m) \equiv (i = q^{m-1} \wedge j = q^m) \wedge \exists K'[K' \in csd(K^{m-1}, q^{m-1}, t_2^{m-1}) \wedge \psi_{ij}(K')] \tag{2.8}$$

evaluated on the frames $F_{m-1}$ and $F_m$, tests the satisfiability of guard $\psi_{ij}$ labeling the edge $(q^{m-1}, q^m) \in E$ by checking the existence of a valuation $K'$, derived from $K^m$ by compensating for the sensing delays in $t_2^m$, that is able to satisfy the constraint $\psi_{ij}$.

The predicate $e_{hi}$,

$$e_{hi}(F_{m-1}, F_m) \equiv (i = q^m \wedge \ q^m = q^{m-1}) \ \wedge K^m \in cad(K^{m-1}, h, i, t_1^{m-1}, t^m) \tag{2.9}$$

evaluated on the frames $F_{m-1}$ and $F_m$, deals with the time evolution in control mode $i$ with predecessor mode $h$. Practically, $e_{hi}$ computes the values of the continuous variables in $K^m$ according to the time elapsing $t^m$. Such a computation takes into account the actuation delays specified by $t_1^{m-1}$ due to a possible previous mode switch (i.e., $h = q^{m-2}$ such that $q^{m-2} \neq q$).

The switch and evolution predicates can now be defined as follows:

$$\begin{aligned} \mathcal{J}_{ij}(F_{m-1}, F_m) \equiv g_{ij}(F_{m-1}, F_m) \wedge (c^{m-1} = 0) \wedge [(K^m, c^m) = \\ = update_{ij}(K^{m-1}, c^{m-1})] \wedge (0 \leq t_1^m \leq T_{AD}) \end{aligned} \tag{2.10}$$

The switch predicate $\mathcal{J}_{ij}$ is satisfied if, in correspondence of a clock tick (i.e., $c^{m-1} = 0$), the guard constraint, specified by $g_{ij}$, evaluates to true by taking into account the sensing delays. As a result, the continuous variables retain the values specified in $K^{m-1}$ and $c^{m-1}$ (i.e., $[(K^m, c^m) = update_{ij}(K^{m-1}, c^{m-1})]$) and actuation delays $t_1^m$ are set because of the mode change. Notice that these actuation delays range over $[0, T_{AD}]$ where the upper-bound $T_{AD}$ is a parameter whose maximum value is being synthesized during the verification. The synthesized value of $T_{AD}$ will represent the slowest admissible actuation delay ensuring the correctness of the control strategy modeled by the LLHA.

$$\begin{aligned}\mathcal{E}_i(F_{m-1}, F_m) \equiv I_i(F_{m-1}) \wedge I_j(F_m) \wedge [\bigvee_{h \in pre(i)} e_{hi}(F_{m-1}, F_m)] \wedge \\ \wedge\ (t^m \geq t_1^{m-1}) \wedge [c^m = (c^{m-1} + t^m)\ mod\ P]\end{aligned} \tag{2.11}$$

where $pre(i)$ denotes the set of predecessor modes of $i$. The predicate $\mathcal{E}_i$ is satisfied if, at the current control mode, the time can elapse till $t^m$ , i.e., the associated invariant evaluates to true in all the time instants of the interval $[t^{m-1}, t^m]$. As a result of the time elapsing, the value of the continuous variables evolves according to the specified flow constraints (i.e., $e_{hi}$) by also taking into account the actuation delays (i.e., $t_1^{m-1}$) due to a possible previous mode change. Moreover, the elapsing of clock ticks is tracked (i.e., $c^m$).

$$\begin{aligned}\bar{\mathcal{E}}_i(F_{m-1}, F_m) \equiv I_i(F_{m-1}) \wedge \neg I_j(F_m) \wedge \neg[\bigvee_{(i,j) \in E} g_{ij}(F_{m-1}, F_m) \wedge (c^{m-1} = 0)] \wedge \\ \wedge\ [\bigvee_{h \in pre(i)} e_{hi}(F_{m-1}, F_m)] \wedge (t^m = P - c^{m-1})\end{aligned} \tag{2.12}$$

where $pre(i)$ denotes the set of predecessor modes of $i$. The predicate $\bar{\mathcal{E}}_i$ is satisfied if neither the invariant at the current mode is satisfiable at any time instant following $t^{m-1}$ nor a mode switch can occur at $t^{m-1}$. This means that the mode should be left as soon as possible. For this reason the time can elapse only till the closest clock tick (i.e., $t^m = P - c^{m-1}$) at which it would be possible evaluate the guards $\psi_{ij}$ labeling the outgoing edges of the current control mode $i$.

This completes the definition of the transition predicate.

Now the parameterized BMC formula, used for synthesizing the maximum values of $P$, $T_{AD}$ and $T_{SD}$, can be defined. Let $\varphi_{safe}(F) = (\phi_c(q) \wedge \phi_v(K))$ represent a safety property for the LLHA. The predicate $\varphi_{safe}(F)$ specifies both a constraint $\phi_c$ on the current control mode in the frame $F$ (e.g., either $q = q_i$ or $q \neq q_i$) and a constraint $\phi_v(K)$ on the value of continuous variables in $F$ (e.g., $x_i \in [-10, 12]$).

If $d$ is the number of steps to which check the transition relation for safety, then the unsatisfiability of the formula

$$BMC^d(P, T_{AD}, T_{SD}) \equiv Init(F_0) \wedge \bigwedge_{n=1}^{d} \mathcal{TR}(F_{n-1}, F_n) \wedge \bigvee_{n=1}^{d} \neg\varphi_{safe}(F_n) \tag{2.13}$$

will guarantee the correctness of the LLHA w.r.t. the specified safety property. Notice that, if $BMC^d$ is satisfied, then there exists a sequence of frames $F_0, F_1, \dots,$ $F_j$ with $j \leq d$ falsifying the property $\varphi_{safe}$. Such a sequence of frames represents a counterexample trace.

Furthermore, BMC is complete with respect to reachability if $d$ is large enough to guarantee that all reachable states of $A$ are considered. The smallest $d$ that has this property is called the reachability diameter. Due to the fact that the set of states $S_A$ is finite [5], such a smallest $d$ exists, thus, if $BMC^j$ is unsatisfied for all $j \leq d$, then the safety property is satisfied by the LLHA.

## 2.4.5 Synthesis procedure

The definition of the BMC formula described in the previous section is based on a set of parameters whose values affect the correctness of the LLHA model, i.e., its capability of satisfying a safety property $\varphi_{safe}$ that represents a specification for the hybrid system. The synthesis engine aims at identifying the maximum values of such parameters for which the discrete-time and finite-precision behaviors specified by the LLHA are able to satisfy $\varphi_{safe}$.

In particular, the parameters reported into the formula $BMC^d$ are the following:

- *clock period* $P$. It specifies the periodicity of clock ticks at which it is possible to evaluate the guards for performing a mode switch;
- *actuation delay upper-bound* $T_{AD}$. It specifies the maximum delay admitted for effecting the change of rate due to a mode switch (i.e., actuator latency);
- *sensing delay upper-bound* $T_{SD}$. It specifies the maximum delay admitted for notifying the controller that a mode switch can be performed (i.e., sensor latency).

At the moment, the precision $\epsilon$ of the observed values is not explicitly modeled as a parameter. Instead, it is assumed that it is fixed at some suitable level of granularity and that the constant values reported into the predicates modeling guard and invariant conditions have been scaled accordingly to be represented as integers[1].

For reducing the time required in identifying the suitable values for the parameters summarize above, it is asked to the user to specify a desired clock period $P$ that the control strategy has to adopt. Then a synthesis procedure will automatically retrieve the maximum values for $T_{SD}$ and $T_{AD}$ that preserve the safety of the model according to the specified clock period.

The procedure identifies the intended values by using a bisection method on a finite interval of feasible values for the parameters. Due to the fact that the LLHA semantics constraints the actuation delay to be smaller than the sensing delay and the sum of sensing and actuation delays to be smaller than the clock period, the starting search space is given by the intervals $T_{AD} \in [a, b]$ and $T_{SD} \in [c, d]$ such that $b < d$ and $b + d < P$. Algorithm 2 reports the pseudo-code implementing the values synthesis procedure.

[1] Remember that the underlying structure used for the symbolic representation of variables is the bit-vector.

**Algorithm 2:** The values synthesis procedure for LLHA-based control strategies.

```
procedure find_values($BMC^d$, $P$, $a$, $b$, $c$, $d$, $N$)
input: the $BMC^d$ formula, the clock period $P$, initial intervals $[a,b]$ and $[c,d]$ of
       feasible values for $T_{AD}$ and $T_{SD}$, resp., and the maximum number $N$ of
       iterations
output: maximal values of $T_{AD}$ and $T_{SD}$ for which the control strategy satisfies
        $\varphi_{safe}$, $T_{SD} = 0$ and $T_{AD} = 0$ otherwise
1  $it = 0$;
2  $T_{SD} = 0$;
3  $T_{AD} = 0$;
4  $mid_1 = \lfloor (a+b)/2 \rfloor$;
5  $mid_2 = \lfloor (c+d)/2 \rfloor$;
6  while $(it < N) \wedge ((b-a > 1) \vee (d-c > 1))$ do
7      if $\mathcal{SMT}(BMC^d(P, mid_1, mid_2)) \dashrightarrow valid$ then
8          $a = mid_1$;
9          $T_{AD} = mid_1$;
10         $c = mid_2$;
11         $T_{SD} = mid_2$;
12     else
13         $b = mid_1$;
14         $mid_{1-tmp} = \lfloor (a+b)/2 \rfloor$;
15         if $\mathcal{SMT}(BMC^d(P, mid_{1-tmp}, mid_2)) \dashrightarrow valid$ then
16             $a = mid_{1-tmp}$;
17             $T_{AD} = mid_{1-tmp}$;
18             $c = mid_2$;
19             $T_{AD} = mid_2$;
20         else
21             $d = mid_2$;
22     $mid_1 = \lfloor (a+b)/2 \rfloor$;
23     $mid_2 = \lfloor (c+d)/2 \rfloor$;
24     $it = it + 1$;
25 return $(T_{SD}, T_{AD})$;
```

At each step, the procedure divides the current subintervals of $[a, b]$ and $[c, d]$ in two by computing their midpoints $mid_1$ and $mid_2$. Then it verifies whether the formula $BMC^d(P, mid_1, mid_2)$ is valid by fixing $T_{AD} = mid_1$ and $T_{SD} = mid_2$. Now, according to the verification results the method registers the current midpoints as candidate solutions and selects the subintervals to be used in the next step. In particular, if $mid_1$ and $mid_2$ let the formula be valid, they become *candidate* maximal values for $T_{AD}$ and $T_{SD}$, resp., and the new intervals of search will be $[mid_1, b]$ and $[mid_2, d]$, i.e., the procedure will check the validity of the formula on new values greater than the current midpoints. Otherwise, the procedure has to look for smaller ones. At first, it checks the formula validity by reducing only the current value for actuation delays. It computes the new candidate for $T_{AD}$ (i.e., $mid_{1-tmp}$) and, by preserving the previous candidate $mid_2$ for $T_{SD}$, verifies the validity of the formula $BMC^d(P, mid_{1-tmp}, mid_2)$. If the verification returns a positive answer, then $mid_{1-tmp}$ and $mid_2$ are recorded as new candidate maximal solutions for $T_{AD}$ and $T_{SD}$, resp., and the new intervals of search will be $T_{AD} \in [mid_{1-tmp}, mid_1]$ and $T_{SD} \in [mid_2, d]$. On the contrary, the non-validity of the formula underlines that also a smaller sensing delay is required. For this reason the search continues on the intervals $[a, mid_1]$ and $[c, mid_2]$. In this way the intervals that contains the final values of the parameters is reduced in width at least by 50% at each step. The process is continued until the maximum number $N$ of iteration is reached or each interval contains 1 element.

### 2.4.6 Experimental Results

This section reports the results obtained by applying the proposed LLHA parameter synthesis approach on three case studies. All experiments have been performed on a workstation with Intel Xeon 2.53 GHz processors and 16GB RAM. The hybrid models of the case studies have been described by means of CIF [129] language. The *cif2uclid* tool has been implemented to automatically derive, from such models, parametric-LLHA descriptions which have been automatically synthesized into equivalent SMT formulas by using the UCLID [99] modeling environment. Several SMT solvers have been used to verifying the models and identify the maximum values for the sensing and actuation delay parameters reported into the LLHA models. In particular, for each case study the performances of the following SMT solvers have been compared: Boolector[2] [32], Yices [55] and Beaver [90]. Notice that, any other SMT solver could alternatively be used as verification and parameter synthesis engine.

#### Train-Gate Controller

The train gate controller ensures that the gate is closed when the train is approaching it. The train is assumed to move at a constant speed $v$ on a circular track and the gate begins to close at a constant angular speed $u$ when the train is $d_{max}$ distance from the gate. Once the train has moved $d_{max}$ distance away from the gate, the gate begins to open again. The system is shown in Figure 2.11. The distance of the train is measured in meters, the angle of the gate in degrees and the time in seconds. The set of parameter values used in the running example is as follows: $v = 40$, $u = 15$, $d_{far-away} = 2000$, $d_{max} = 400$ and $d_{safe} = 160$.

The system is considered safe, i.e., the train is never closer to gate than $d_{safe}$ unless the gate is completely closed, only if the following safety property is satisfied: $-d_{safe} \leq d \leq d_{safe} \rightarrow a <= 0$.

Such a property is used during the synthesis phase for identifying the maximum values for the sensing and actuation delay parameters ($T_{SD}$ and $T_{AD}$, respectively) that are reported into the LLHA modeling the system. In particular, the parametric LLHA has been automatically generated by using a digitizing precision $\epsilon = 10^{-3}$ and a clock period $P = 10^{-2}$. Then, the parameter synthesis approach has determined that the coarse values for the sensing and actuation delays which

[2] MiniSat and PicoSat have been used as the underlying SAT engines.

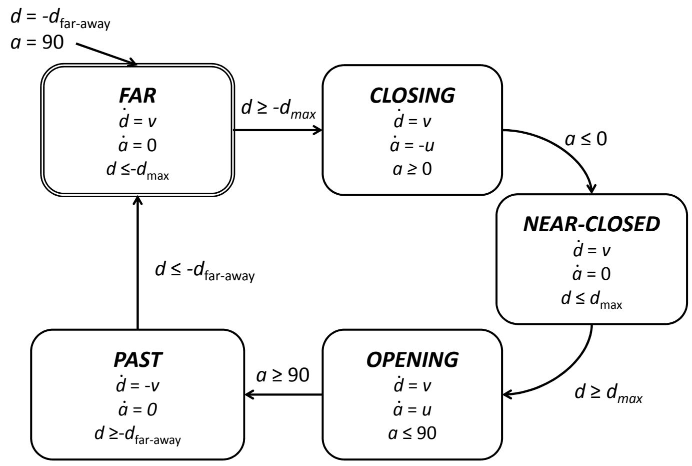


Fig. 2.11: LHA model of the Train-Gate controller.

ensure the correctness of such a LLHA are $T_{SD} = 4 \cdot 10^{-3}$ and $T_{AD} = 2 \cdot 10^{-3}$. The time required for synthesizing such values is reported in Table 2.3.

Table 2.3: Comparison of the synthesis times using different SMT solvers.

| $\mathcal{SMT}$ | $T_{SD}$ **s-Space** | $T_{AD}$ **s-Space** | **# Bisect.** | **Time (sec)** |
|---|---|---|---|---|
| *Beaver* | $[0,\ 5 \cdot 10^{-3}]$ | $[0,\ 4 \cdot 10^{-3}]$ | 15 | 123.336 |
| *Boolector* | $[0,\ 5 \cdot 10^{-3}]$ | $[0,\ 4 \cdot 10^{-3}]$ | 15 | 101.544 |
| *Yices* | $[0,\ 5 \cdot 10^{-3}]$ | $[0,\ 4 \cdot 10^{-3}]$ | 15 | 49121.940 |

In particular, column $\mathcal{SMT}$ reports the name of the SMT solvers compared; columns $T_{SD}$ *s-Space* and $T_{AD}$ *s-Space* report the initial search spaces used for identifying suitable values for the sensing and actuation delays, respectively. Column *# Bisect.* shows the maximum number of bisection iterations allowed for synthesizing the parameters values and, finally, column *Time* reports the total time (in seconds) spent for the synthesis process.

### Room Heating Controller

The room heating controller ensures that the temperature of a room is kept into a comfort interval by turning *on* and *off* the heater installed into the room. Figure 2.12 depicts the model of the system. Intuitively, the automaton is composed by two control modes *on* and *off*, representing the status of the heater. The variable $x$ denotes the room temperature (measured in $^\circ C$). When the heater is *off*, the temperature of the room falls according to any of rates specified by the rectangular constraint $\dot{x} \in [-b, -a]$. Instead, when the heater is *on* the temperature of the room rises following any of rates specified by the constraint $\dot{x} \in [b, c]$. The heater is turned on as soon as the falling temperature reaches $x_{low}$: the automaton moves to the control mode *on* and the temperature rises starting at a value $x <= x_{low}$. The heater is turned off as soon as the temperature reaches $x_{high}$: the automaton moves to the control mode *off* and the temperature starts falling again at a value $x >= x_{high}$. This control strategy guarantees the temperature of the room will remain at between $x_{min}$ and $x_{max}$ starting at the initial temperature $x_{init}$ such that $x_{low} < x_{init} < x_{high}$. The set of parameter values used in the running example is as follows: $a = 2$, $b = 3$, $c = 4$, $x_{min} = 17.8$, $x_{low} = 18$, $x_{high} = 22$ $x_{max} = 22.5$ and $x_{init} = 19$.

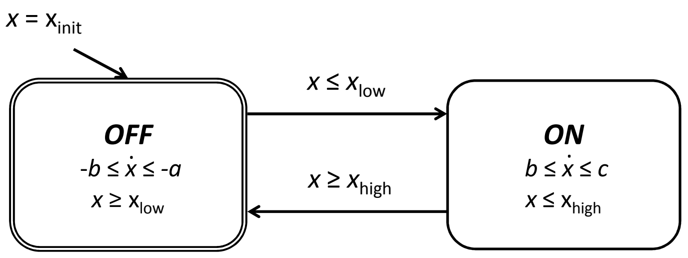


Fig. 2.12: The LHA model of the room heating controller.

The property $x_{min} < x < x_{max}$ is used for synthesizing the maximum values of the sensing delay $T_{SD}$ and the actuation delay $T_{AD}$ which are reported into the parametric LLHA modeling the finite-precision lazy heating controller. Such a model has been generated from the one depicted in Figure 2.12 by choosing a digitizing precision $\epsilon = 10^{-4}$ and a clock period $P = 10^{-2}$. The synthesis procedure has determined that the values $T_{SD} = 11 \cdot 8^{-3}$ and $T_{AD} = 10 \cdot 10^{-3}$ guarantee that the lazy controller keeps the temperature into the comfort bounds (i.e., $x_{min}$ and $x_{max}$). The time required for synthesizing such values is reported in Table 2.4.

Table 2.4: Comparison of the synthesis times using different SMT solvers.

| $\mathcal{SMT}$ | $T_{SD}$ **s-Space** | $T_{AD}$ **s-Space** | **# Bisect.** | **Time (sec)** |
|---|---|---|---|---|
| *Beaver* | $[0, 40 \cdot 10^{-3}]$ | $[0, 30 \cdot 10^{-3}]$ | 15 | 343.888 |
| *Boolector* | $[0, 40 \cdot 10^{-3}]$ | $[0, 30 \cdot 10^{-3}]$ | 15 | 1584.864 |
| *Yices* | $[0, 40 \cdot 10^{-3}]$ | $[0, 30 \cdot 10^{-3}]$ | 15 | 90903.836 |

### Watertank Controller

The watertank system is centered on a water tank, which is characterized by an uncontrolled outbound water flow, while the inbound water flow is controlled by the aperture of a valve. The controller acts on the aperture of the valve $a$ in order to keep the water level $x$ in a safe interval $x_{min} < x < x_{max}$. The system model is shown in Figure 2.13. The water level is measured in deciliters, the valve aperture in degrees and the time in seconds. The set of parameter values used in the running example is as follows: $a = 1$, $b = 2$, $c = 4$, $d = 7$, $v = 20$, $x_{high} = 850$, $x_{low} = 550$, $y_{min} = 0$, $y_{max} = 360$, $x_{max} = 870$ and $x_{min} = 540$.

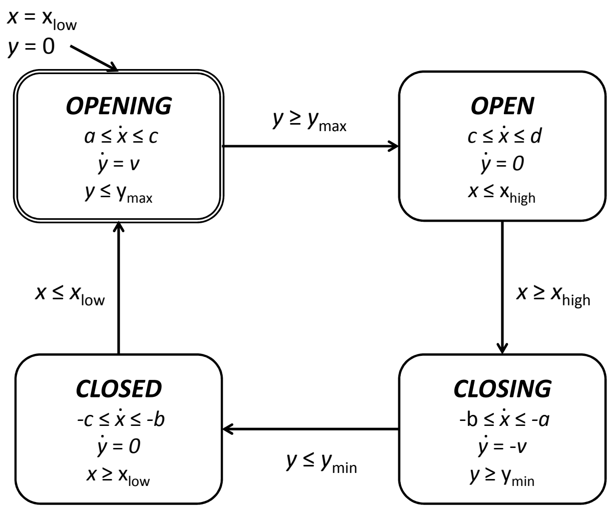


Fig. 2.13: LHA model of the watertank controller.

In the model, the water level begins to increase according to a rectangular constraint $\dot{x} = [a, c]$ when the valve starts to open at constant angular speed $v$. As soon as the valve reaches its full aperture, the incoming flow reaches its maximum value, filling faster the water tank. Once the water level crosses an upper threshold $x_{high}$, the valve starts to close in order to avoid a water overflow. Once the valve is completely closed, no inbound water flow is present and the water level keeps decreasing. When the water level reaches its lower threshold $x_{low}$, the valve begins to open again.

The property $x_{min} < x < x_{max}$ is used for synthesizing the maximum values of the sensing delay $T_{SD}$ and the actuation delay $T_{AD}$ which are reported into the parametric LLHA modeling the finite-precision lazy watertank controller. Such a model has been generated from the one shown in Figure 2.13 by choosing a digitizing precision $\epsilon = 10^{-3}$ and a clock period $P = 10^{-1}$. At the end of the synthesis phase, the verification has determined that the values $T_{SD} = 23 \cdot 10^{-3}$ and $T_{AD} = 11 \cdot 10^{-3}$ guarantee that the lazy controller keeps safely the water level into the $x_{min}$ and $x_{max}$ bounds. The time required for synthesizing such values is reported in Table 2.5.

## Automated Highway Control System

Automated Highway Control System (AHS) is an arbiter which ensures that there is no collision between cars running on a highway by imposing legal speed ranges.

Table 2.5: Comparison of the synthesis times using different SMT solvers.

| $\mathcal{SMT}$ | $T_{SD}$ **s-Space** | $T_{AD}$ **s-Space** | **# Bisect.** | **Time (sec)** |
|---|---|---|---|---|
| *Beaver* | $[0, 50 \cdot 10^{-3}]$ | $[0, 40 \cdot 10^{-3}]$ | 15 | 531.560 |
| *Boolector* | $[0, 50 \cdot 10^{-3}]$ | $[0, 40 \cdot 10^{-3}]$ | 15 | 2413.017 |
| *Yices* | $[0, 50 \cdot 10^{-3}]$ | $[0, 40 \cdot 10^{-3}]$ | 15 | 107236.980 |

The linear hybrid automata representing the case of four cars is shown in Figure 2.14. The distance between cars is measured in $km$, time in hours and speeds in $km/h$. The set of parameter values used in the running example is as follows: $rl = 20$, $b = 30$, $c = 40$, $d = 50$, $e = 60$, $ru = 70$, $f = 100$, $\alpha_{min} = 2 \cdot 10^{-3}$, $\alpha_{max} = 1$ and $\alpha'_{min} = 5 \cdot 10^{-4}$.

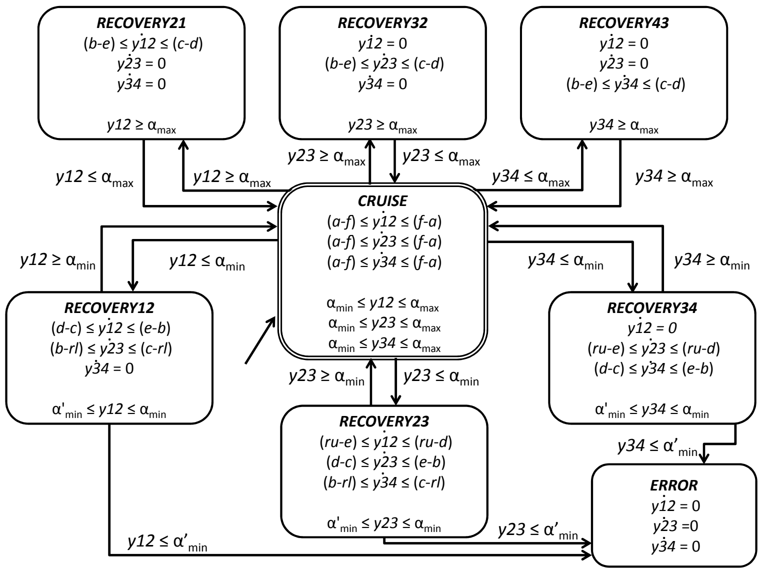


Fig. 2.14: LHA model of the Automated Highway Control System.

To avoid collisions, the arbiter specifies speed limits (i.e., $[a, f]$) for each vehicle. When two vehicles $i$ and $j$ come within a distance $y_{ij} \leq \alpha_{min}$ of each other, there exists a possible collision event. The arbiter asks the approaching car to slow down by reducing the speed into the interval $[b, c]$ and asks the leading car to speed up by keeping a speed into the interval $[d, e]$; it also requires that all other cars not involved in the possible collision slow down to a constant recovery mode velocity $rl$ for cars behind the critical region and $ru$ for cars in front of the critical region. When the distance between the two vehicles involved in the possible collision exceeds $\alpha$, the arbiter model goes back to the dynamics of the cruise mode. Moreover, the arbiter keeps all the vehicles below a maximal distance $\alpha_{max}$ of each other. When two vehicles $i$ and $j$ exceed such a distance (i.e., $y_{ij} \geq \alpha_{max}$), the arbiter asks the leading car to slow down by reducing the speed into the interval $[b, c]$ and asks the approaching car to speed up by keeping a speed into the interval $[d, e]$; it also requires that all other cars keep the current distance constant (i.e., speeding up for cars behind the critical region and slowing down for cars in front of the critical region). When the distance between the two vehicles decreases below $\alpha_{max}$, the arbiter model goes back to the dynamics of the cruise mode.

The only safety property to be satisfied by the model is that the control mode is never the *error* mode.

Again, once the parametric LLHA model has been extracted by choosing a digitizing precision $\epsilon = 10^{-5}$ and a clock period $P = 10^{-2}$, the safety property is used as constraint for identifying the coarse values of the sensing and actuation delay parameters ($T_{SD}$ and $T_{AD}$, respectively). On this case study, the synthesis phase has determined that the values $T_{SD} = 218 \cdot 10^{-5}$ and $T_{AD} = 112 \cdot 10^{-5}$ guarantee the safety of the system, i.e., the lazy controller is able to avoid cars collision. The time required for synthesizing such values is reported in Table 2.6.

Table 2.6: Comparison of the synthesis times using different SMT solvers.

| $\mathcal{SMT}$ | $T_{SD}$ **s-Space** | $T_{AD}$ **s-Space** | **# Bisect.** | **Time (sec)** |
|---|---|---|---|---|
| *Beaver* | $[0,\ 500 \cdot 10^{-5}]$ | $[0,\ 400 \cdot 10^{-5}]$ | 15 | 154.053 |
| *Boolector* | $[0,\ 500 \cdot 10^{-5}]$ | $[0,\ 400 \cdot 10^{-5}]$ | 15 | 147.753 |
| *Yices* | $[0,\ 500 \cdot 10^{-5}]$ | $[0,\ 400 \cdot 10^{-5}]$ | 15 | 7533.104 |

## 2.5 Conclusions

The development of methodologies for the synthesis of implementable control strategies for hybrid automaton-based models is a new and valuable research area. This chapter tackled this problem in two ways. At first, it proposed a new framework, including the *s-extract* manipulation tool, the *cif2ariadne* and *cif2phaver* translators, *Ariadne* and *PHAVer* model checkers and a model checker independent synthesis procedure, that makes practical the applicability of the AASAP synthesis approach on general classes of hybrid automata for which the reachability problem is not decidable. Afterwards, it focused on defining a new methodology that, supported by tools such as *cif2uclid*, *UCLID* and a SMT solver independent synthesis procedure, enables the synthesis of implementable control strategies for the interesting class of *lazy linear hybrid automata*. In both cases experimental results have been proposed to underline the applicability of the proposed approaches.

# 3

# Model-driven design and verification of embedded software

## 3.1 Introduction

With the increasing complexity of embedded controllers and, in particular, of the related software part, i.e., Embedded Software (ESW), the need arises to improve the design and the verification of ESW. As is well known, ESW is a specific-purpose software tightly integrated with the underlying execution platform and it constantly reacts to event occurrences and mixes control and data flows [102].

Typically, *implementation* and *verification* of ESW is a time consuming and error-prone process [119]. In fact, code implementation inevitably implies fixing and re-factoring: the designer needs to implement specifications that, described by means of natural language, may lead to misinterpretations.

To address the limitations of natural languages, the industries have realized that, whenever possible, requirement specifications should be given in formal notations, both graphical and language-based [59, 111].

Since the mid 1990s, the massive use of multiple graphical-modeling paradigms in embedded-system design leads to consider the usage of Model-Driven Design (MDD) methodologies [126]. MDD aims at raising the level of abstraction through an extensive use of generic models in all the phases of the development. It describes the system under development in terms of abstract characterization, attempting to be generic not only in the choice of implementation platforms, but even in the choice of execution and interaction semantics. Thus, MDD has emerged as the most suitable solution to develop complex systems and has been supported by academic [60] and industrial tools [1, 11, 68, 82, 83, 130].

The gain offered by the adoption of a MDD approach is the capability of generating the source code implementing the ESW in a systematic way, i.e., it avoids the need of manual writing, analyzing and modifying the code. However, even if the MDD simplifies the software implementation it does not prevent the designer to wrongly define the software behaviors. Therefore, the MDD gives full benefits if it integrates also functional verification.

Unfortunately, nowadays, only certain aspects concerning verification of ESW are automated, as for example the structural analysis of code, but specification conformance, i.e., functional verification, is still a human-based process [57, 71]. Indeed, the de-facto approach to guarantee the correct behavior of ESW is monitoring the system simulation: company verification teams are responsible of putting the system into appropriate states by generating the required stimuli, judging when stimuli should be executed, manually simulating environment and user interactions, and analyzing the results to identify unexpected behaviors.

Assertion-based Verification (ABV) [64] aims at providing verification engineers with a way for formally capturing the intended specifications and checking their compliance with the implemented ESW. In ABV, specifications are expressed by means of temporal assertions, that, to overcome the ambiguity of natural languages, are defined according to formal assertion languages, like, for example, the Property Specification Language (PSL) [87]. PSL assertions can be verified by model checking (static ABV) or by simulation (dynamic ABV). Dynamic ABV is preferred in case of large designs due to its scalability. In particular, in the hardware domain, dynamic ABV is affirming as a leading strategy in industry to guarantee fast and high-quality verification of hardware components [35, 108] and several verification approaches have been proposed [61, 64]. On the contrary, companies, which develop embedded software, still incur practical problems in adopting dynamic ABV in their design flows [128].

It is, therefore, evident that MDD and dynamic ABV *individually* suffer some limitations that prevent their integration in the ESW design and verification flow. But, if combined in a comprehensive framework, these approaches support and enable each other. On one side, MDD gains full benefits only if the MDD environment integrates automatic code generation with functional verification to ensure designer about the conformance between specification and implementation. That reduces the need of directly examining and verifying the generated code. In this way, the problem is abstracted and moved from the source-code level to the model level. On the other side, dynamic ABV needs to be supported by a MDD approach to be easily applied to ESW verification. Indeed, dynamic ABV, has been widely used in the hardware-design domain, since HW descriptions satisfy some simulation assumptions that generally do not hold in SW, as reported in Section 3.3. The use of a MDD approach for generation of ESW code provides a solution to such problems, as described in this paper.

In this context, extending the idea proposed in [49], this chapter will describe how MDD approaches and dynamic ABV can be integrated in an off-the-shelf framework for supporting the ESW development.

### 3.1.1 Framework overview and contributions

Our framework is composed of two environments: radCASE and radCHECK (Figure 3.1). radCASE is a UML-modeling and development environment for ESW supporting model-driven design. radCHECK is a dynamic ABV environment that supports assertion definition and automatic checkers and stimuli generation.

Starting from the informal specifications and requirements, the designer, with the Model Editor of radCASE, defines the ESW model using an UML-based approach. Concurrently, with the Property Editor of radCHECK, he/she defines a set of PSL assertions that the application must fulfil. Then, radCASE automatically translates the UML specifications in the ESW C-code implementation, and automatically extracts an Extended Finite State Machine (EFSM) [37] model to

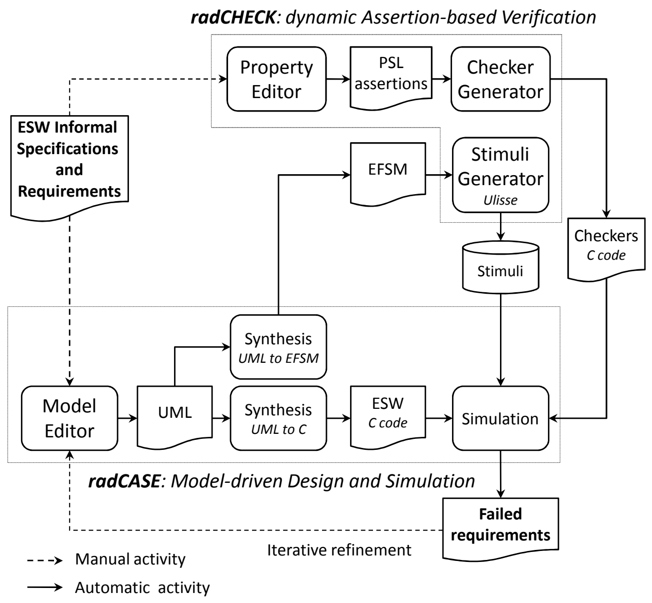


Fig. 3.1: The radCASE and radCHECK model-driven and verification framework.

support verification. At the same time, radCHECK can be used to automatically generate executable checkers from the defined PSL assertions. Checkers are executable components that monitor the evolution of the ESW during dynamic ABV. The dynamic ABV is guided by stimuli automatically generated by Ulisse, i.e., a corner-case-oriented concolic stimuli generator that exploits the EFSM model to explore the ESW state space. Thus, the verification phase is performed by using the generated stimuli on top of an ESW simulator or on the application running on the target hardware. The checkers execute within the ESW and monitor if the state of the application causes an assertion to falsify. The designer uses the resulting information, i.e., failed requirements, for refining the UML specifications incrementally and in an iterative fashion.

The main contributions of such a framework are summarized as follows:

- it presents an extension of the existing radCASE environment [133] to cover all possible aspects of ESW design taking care of verification requirements. In particular, we extended radCASE to automatically synthesize from the UML-like specifications of the ESW a corresponding EFSM model supporting the automatic stimuli generation required for the effective dynamic verification of the ESW implementation;
- it presents a new dynamic ABV environment for ESW, named radCHECK [131], which integrates a graphical assertion editor for assisted assertion definition, a checker generator, and a stimuli generator to simulate the ESW. In particular:
  – the assertion editor graphically supports the designer in defining formal assertions from the informal specifications of the ESW through a large set of parametric templates.

- the checker generator synthesizes the defined formal assertions into assertion checkers which are automatically integrated into the simulation environment.
- the stimuli generator provides the simulator with stimuli to efficiently cover corner cases when checkers are simulated to determine the correctness of the implementation w.r.t the specification;

The remainder of this chapter is structured as follows. Section 3.2 summarizes the state of the art concerning model-driven design and dynamic assertion-based verification for ESW; Section 3.3 summarizes the main limitation of current approaches and clarifies the goal of the proposed design and verification framework; Section 3.4 presents radCASE; Section 3.5 presents radCHECK; Section 3.6 is devoted to the stimuli generation engine included in radCHECK; Section 3.7 deals with experimental results and comparisons with other approaches. Finally, Section 3.8 is devoted to concluding remarks.

## 3.2 State of the art

This section provides an overview of existing formalisms and tools for ESW design, as well as state-of-the-art approaches for ESW verification.

### 3.2.1 Model-driven design of embedded software

The focus of MDD is to elevate software development to a higher level of abstraction than that provided by third-generation programming languages [126]. The development is based on *models*, which are abstract characterizations of requirements, behaviors and functionalities of the ESW and separate "what" the software should do, from "how" it needs to be implemented on an available technology platform.

Nowadays, several standardized visual languages have been proposed for the definition of these abstract models. Among all, due to the noticeably effort of the Object Management Group (OMG) [114], the Unified Modeling Language (UML) [116] has been adopted as the reference modeling language for describing ESW. UML provides general-purpose graphic elements to create visual models of object-oriented-software systems and attempts to be generic in both the integration and the execution semantics. Due to such a general-purpose semantics, more specific UML profiles have been introduced for dealing with specific domains or concerns. They extend subsets of the UML meta-model with new standard elements and refine the core UML semantics to cope with particular hardware/-software problems.

For example, the System Modeling Language (SysML) [134] profile extends UML to support the specification, the analysis and the design of complex systems which may include software and hardware. The main aim of the profile is to provide graphical representations with a semantic foundation for modeling system requirements, behavior, structure, and parametrics, which are used to integrate with other engineering analysis models. The Gaspard2 [74] profile, instead, extends the UML semantics to support the modeling of SoCs (System-on-Chip) by specifying the hardware/software system at different level of abstraction. The modeling process consists of three sequential steps: software (behavior of the SoC) specification, hardware architecture specification, and, finally, association of the software to the hardware architecture. The software is represented following a data flow model, but additional mechanisms permit the usage of control flow models. In addition to those notions, the profile introduces factorization mechanisms to enable the compact description of massively parallel and repetitive systems. The MARTE [112] profile adds capabilities to UML for Modeling and Analysis of Real-Time and Embedded (software) systems. It replaces the obsolete Schedulability, Performance and Time (SPT) [115] profile and redefines UML concepts for real-time modeling and analysis concerns. The modeling concepts provide support for representing time and time-related mechanisms, the use of concurrent resources and other embedded systems characteristics (such as memory capacity and power consumption). The analysis concepts, instead, provide model annotations for dealing with system properties analysis such as schedulability analysis and performance analysis. The Synchronous Reactive [45] profile, on the opposite, refines UML semantics to provide state diagrams (and to a certain extend a restrictive set of activity diagrams and sequence diagrams) with a clear and semantically sound way of generating valid execution sequences, thereby endowing them with a programming language quality. Nevertheless the specification style promoted by the synchronous reactive approach is still fully independent from any runtime system mechanism or execution platform. Notice that other UML profiles exist, but they target mainly hardware related aspects such as system level modeling and simulation rather than embedded software specification [109,123]. Other hardware-oriented profiles and a comparison of them is clearly described in [29]. Finally, some proprietary variants of the UML notations exist. The most famous ones are the MathWorks Stateflow and Simulink [137] formalisms. They use finite-state-machine-like and functional-block-diagram-like models, respectively, for specifying behaviors and structure of reactive hardware/software systems with the aim of rapid ESW prototyping and engineering analysis.

Several MDD tools on the market support UML and Model-Driven Architecture (MDA) for embedded software development. The underlying idea of MDA is the definition of models at different level of abstraction which linked together form an implementation. MDA distinguishes the conceptual aspects of an application from their representation on specific implementations technologies. For this reason, the MDA design approach uses Platform Independent Models (PIMs) to specify what an application does and Platform Specific Models (PSMs) to specify how the application is implemented and executed in the target technology. Basically, a PIM is described by means of UML (or one of its profiles), whereas a PSM may be represented in a variety of forms including executable code such as C, C++ or Java. The key element of a MDA approach is the capability to automatically transform models: transformation of PIMs into PSMs enables realizations, whereas transformations between PIMs enable integration features.

For example, Artisan Studio [11] is a MDD framework which provides support to UML and SysML models and generates C, C++, Ada, Java and C# code specifications. Enterprise Architect [130] is a MDD and analysis framework that supports a comprehensive set of UML models and is able to systematically transforms them into C, C++, Java, .NET code. IAR visualSTATE [82] is a MDD framework that supports a state machine subset of UML and generates C and C++ specifications. CoDeSys [1] is a MDD framework supporting PIMs described by means of IEC 61131-3 standard [91]. This standard combines graphical and textual programming languages, e.g., function-block diagram and structured text. The framework generates PSMs defined using the IEC 61131-3 Structured Text formalism. Rational Rhapsody [83] helps teams collaborate to understand and elaborate requirements, to abstract the complexities using industry standard languages, such as UML and automatically transforms them into C, C++ specifications. TOPCASED [3] is a MDD framework that supports ESW modeling and code synthesis. The framework allows to specify software requirements by means of UML models which adhere the SysML and MARTE profiles. Then, it is able to translate them into C, Java and Python code. Finally, Poseidon for UML Embedded Edition [68] is a MDD framework that, starting from UML diagrams, focuses on the development of C, C++ software for small systems with strong memory and performances constraints. A comparison of these tools is provided in the experimental result section.

### 3.2.2 Dynamic ABV of embedded software

Approaches based on ABV are traditionally classified in two main categories: *static* (i.e., *formal*) and *dynamic* (i.e., *simulation-based*).

In static ABV, assertions, representing design specifications, are exhaustively checked against a formal model of the design by exploiting, for example, a model checker. Such an exhaustive reasoning provides the verification engineers with high confidence in system reliability. However, the well-known state-space explosion problem limits the applicability of static ABV to small/medium-size, high-budget and safety-critical projects [93].

On the contrary, thanks to the scalability provided by simulation-based techniques, dynamic ABV approaches are preferred for verifying large designs, which have both reliability requirements and stringent development-cost/time-to-market constraints. In dynamic ABV, assertions defined by using formal languages (e.g., PSL) are compiled into *assertion checkers*, or simply *checkers*, i.e., modules that capture the behavior of the corresponding assertions and monitor if they hold with respect to the design [27] when this latter is simulated by using a set of (automatically generated) stimuli.

In the context of ESW verification, dynamic ABV provides an effective solution to the functional verification problem.

#### Assertion definition and checker generation

In software verification, software designers widely use *executable assertions* [81] for specifying conditions that apply to some states of a computation, e.g., "pre-conditions" and "post-conditions" of a procedural code block. A runtime error is released whenever execution reaches the location at which the executable assertion occurs and the related condition does not hold any more. This kind of executable assertions is limited to Boolean expressions, which are totally unaware of temporal aspects.

However, if the designers aims to check more complex requirements in which Boolean expressions are used for defining relations spanning over the time, they have to (i) define assertions in a formal language and (ii) synthesize them as executable modules, i.e., *checkers*. Checkers, integrated into the simulation environment, monitor the software execution for identifying violation of the intended requirement.

In hardware verification several solutions have already been proposed. These approaches can be classified in (i) library-based or (ii) language-based.

*Library-based* approaches rely on libraries of pre-defined checkers, e.g., the Open Verification Library (OVL) [65], which can be instantiated into the simulation environment for simplifying the checking of specific temporal behaviors. Unfortunately, due to their inflexibility of checking general situation, the predefined checkers limit the completeness of the verification.

*Language-based* approaches, instead, use declarative languages, such as PSL [87] and SystemVerilog Assertions (SVA) [86], for formalizing the temporal behaviors into well-defined mathematical formulas (i.e., assertions) that can be synthesized into executable checkers by using automatic tools named checker generators [2, 26, 27, 44]. These tools may generate checkers implementations at different levels of abstraction, from the register transfer level (RTL), e.g., MBAC [27] and FoCs [2], to the C-based electronic system level (ESL) [43], i.e., FoCs.

Although some attempts have been tried to extend hardware ABV to embedded software, still several problems persist. In [38], the authors present a Microsoft-proprietary approach for binding C language with PSL. They define a subset of PSL and use a simulator as an execution platform. In this case, only a relative small set of temporal assertions can be defined, since only equality operator is supported for Boolean expressions, and the simulator limits the type of embedded-software applications.

Another extension of PSL is proposed in [146], where the authors unify assertion definition for hardware and software by translating their semantics to a common formal semantic basis. In [145], the authors use temporal expressions of *e* hardware verification language to define checkers. In both these cases, temporal expressions are similar, but not compatible with PSL standards.

Finally, in [103] the authors propose two approaches based on SystemC checkers. In the first case, embedded software is executed on top of an emulated SystemC processor and, every clock cycle, the checkers monitor the variables and functions stored in the memory model. In the second approach, embedded software is translated in SystemC modules which run against checkers. In this case, timing reference is imposed by introducing an event notified after each statement and the SystemC process is suspended on additional `wait()` statements. In both cases, there are several limitations. First, the approaches are not general enough to support real-life embedded software: the SystemC processor cannot reasonably emulate real embedded-system processors, as well, the translation of embedded-software applications in SystemC may be not flexible enough. Secondly, in both the cases the SystemC (co-)simulation and the chosen timing references introduce significant overhead. In particular, clock cycle or statement step may be an excessive fine granularity for efficiently evaluating a sufficient number of temporal assertions; moreover, on the one hand, defining assertions which consider absolute time may generate significantly large checkers to address the high number of intermediate steps; on the other, it is difficult to define temporal assertions at source-code level, i.e., C applications, in terms of clock cycles.

## Stimuli generation

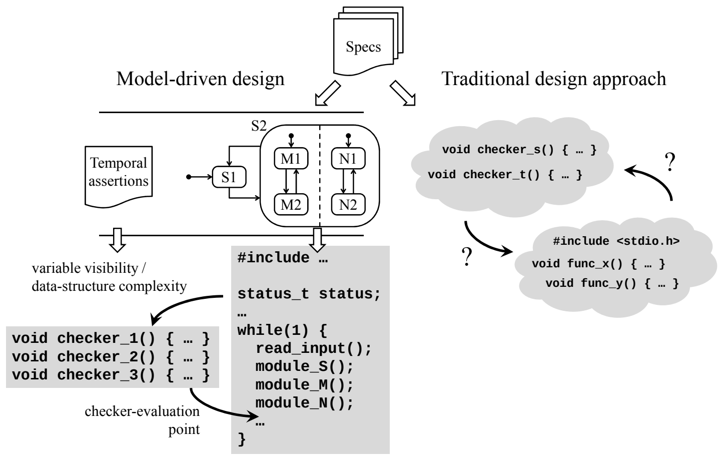
Actual-value inputs may be either automatically generated or developed by engineers as stimuli suites. In both the cases, the main purpose of dynamic verification is increasing the confidence of designers in the ESW behavior by creating stimuli and evaluating them in terms of adequacy criteria, e.g., coverage metrics.

In particular, there are three main categories for classifying stimuli-generation techniques for (embedded) software: *concrete execution*, *symbolic execution*, and *concolic execution*. The concrete execution is based on random, probabilistic, or genetic techniques [107]. It is considered a narrow-width and long-range exploration method, since it reaches deep states of the system space by executing a large number of long paths, but it is not an exhaustive approach. The symbolic execution [94] represents an alternative for overcoming concrete execution limitations, where an executable specification is simulated using symbolic variables and a decision procedure is used to obtain concrete values for inputs. However, such approaches suffer the solver limitations in handling the complexity of either the formulas or the data structures or the surrounding-execution environment.

Such limitation have been recently addressed by proposing *concolic execution* [105, 127]. In concolic approaches, concrete and symbolic executions run together, and, when necessary, symbolic constraints are simplified by using the corresponding concrete values. However, a concolic engine still represents the module execution as a symbolic-execution tree. Thus, state space explosion can still be a problem, since the size of the execution tree grows exponentially in the number of the maintained paths, states, and conditions.

Several tools on the market adopt these approaches and provide the user with automatic stimuli generation addressing coverage metrics in embedded software [69, 72, 143, 144].

In particular, DART [70] is a tool for generating stimuli for C programs that are able to trigger errors such as crashes. It combines random stimuli generation with symbolic reasoning to keep track of constraints for executed control-flow paths. CUTE [127] is a variation on the DART approach addressing complex data structures and pointer arithmetic. KLEE [34] is a framework for symbolically executing LLVM [101] byte-code. PEX [139] is an automated structural-testing generation tool for .NET code developed at Microsoft Research. In [105] the authors describe a hybrid-concolic stimuli generation approach for C programs. It interleaves random stimuli generation with bounded exhaustive symbolic exploration to achieve better coverage. However, it cannot selectively and concolically execute symbolic paths in a neighborhood of the corner cases. Nevertheless stressing structural entities of an application does not guarantee to observe and achieve specification conformance and vice versa [75].



```
#include …

status_t status;
…
while(1) {
  read_input();
  module_S();
  module_M();
  module_N();
  …
}
```

```
void checker_1() { … }
void checker_2() { … }
void checker_3() { … }
```

```
void checker_s() { … }
void checker_t() { … }
```

```
#include <stdio.h>
void func_x() { … }
  void func_y() { … }
```

Fig. 3.2: Integration of dynamic ABV and MDD approach vs. a traditional (manual) approach. MDD permits to address ESW critical aspects, i.e., timing references, complex data-structures, and variable visibility.

## 3.3 Joining MDD and dynamic ABV: what is missing?

As reported in the previous section, many tools and methodologies have been proposed for MDD as well as dynamic ABV. However, ESW designers and verification engineers still encounter many practical problems in the application of such approaches, and particularly in the set up of an efficient and effective dynamic ABV environment [128]. This is mainly due to the absence of a single framework that provides a tight integration between MDD and dynamic ABV. In fact, even if, for example, Simulink is widely used in industry together with other verification tools, the lack of a tight integration of a dynamic ABV environment for verifying the generated code induces the verification engineers to perform a tedious and error-prone manual set up of the verification phases.

For example, in the traditional (manual) ESW design approach, the integration between the ESW implementation and the checkers derived from temporal assertions to check the correctness of such an implementation is not clear (Figure 3.2, right). Both the checkers and the ESW code are represented by a set of subprograms without clear rules concerning when the former should interact with the latter. Moreover, existing checker generators are limited to the hardware domain and creates checkers that cannot be easily integrated in an ESW design flow [2, 28]. This is mainly due to the characteristics of ESW, which generally differs from HW descriptions in several aspects. In particular, HW descriptions satisfy some simulation assumptions, which make very easy their integration with checkers to perform dynamic ABV. On the other hand, ESW is generally far from guaranteeing the same assumptions, and in particular:

- *Timing references:* simulation of HW descriptions exploits either a cycle-based or an event-driven scheduling strategy which makes extremely clear how syn-

chronization between modules happens, and exactly fixes the time when computation reaches a stable condition. Such timing reference provides designers with the exact time when checkers must be activated to verify the behavior of the design. The execution of ESW lacks such a timing reference, thus the checkers are not aware of the right time when ESW computation reaches a stable condition, nor they can control ESW concurrency.
- *Presence of complex data structures:* existing HW checker generators support the creation of assertion checkers including expressions conforming the Boolean layer of PSL. Reasoning in terms of Boolean data type is straightforward in the HW domain, especially at RTL and gate level. On the contrary, ESW may involve more complex data structures, like pointers, unions, etc., which cannot be easily handled by existing checker generators.
- *Variable visibility:* assertion checkers can monitor the behavior of variables which are made visible at the boundary of the design. For HW descriptions this means primary inputs and outputs and signals, which is enough to predicate about the functionality of the design. On the other hand, the scope of ESW variables is more complex and in some cases, relevant variables can be invisible to the checkers.

Such problems are avoided by our proposed approach (Figure 3.2, left). First, radCASE generates the ESW code from graphical and textual formalisms which adhere to specific, well-affirmed, execution semantics (i.e., synchronous reactive models [45]). This makes clear when checkers must be invoked to monitor the correctness of the implementation, thus solving the problem of timing reference, as described in Section 3.4.3. Moreover, thanks to the possibility of analyzing this ESW abstract model, radCHECK can extract fundamental information suited for supporting the checker synthesis overcoming problems related to variable visibility and management of complex data structures during the set up of the verification phase, as described, respectively, in Section 3.5.1 and Section 3.5.2. In this way, thanks to the adoption of a well-defined coding style for implementing the synchronous reactive execution semantics, the ESW code generated by radCASE can be verified by means of the dynamic ABV approach implemented in radCHECK.

Another problem which discourages the application of ABV for ESW verification is represented by the high skills required to define effective assertions. Indeed, project managers encourage software architects and developers to write assertions, but they have problems with learning PSL and translating informal specifications into assertions. PSL allows to formalize complex temporal behaviors in a precise and concise manner, but the complexity of its semantics requires qualified verification engineers for correctly defining the intended assertions. To overcome such difficulties, predefined libraries of assertion checkers have been proposed in the HW domain, like, for example, the Open Verification Library (OVL) [65], but, to the best of our knowledge, nothing similar is available for ESW. For this reason, radCHECK integrates an assertion editor that guides the user during the formalization of assertions, as described in Section 3.5.1.

Besides allowing a strict integration between MDD and dynamic ABV, radCASE and radCHECK provide unique features that are not present in the existing tools. In particular, radCASE is the only MDD tool providing:

- an integrated engine which extracts an EFSM-based model from the UML-based specifications of the ESW that supports the dynamic ABV tool (i.e., radCHECK) in analyzing the ESW behaviors for generating effective stimuli;

Moreover, radCHECK unique features are:

- a Checker Generator which synthesizes the defined formal assertions into executable checkers implemented in C-code. Beside the checker implementation, the tool generates automatically also the supporting code that enables the smooth integration of the checkers into the simulation environment, avoiding any manual set up.
- a stimuli generation engine (i.e., Ulisse) which exploits a weight-based dependency analysis and multi-level backjumping over EFSM to efficiently cover corner cases when checkers are simulated to check the correctness of the implementation w.r.t the specification.

Next sections will describe details of both radCASE and radCHECK. Moreover, they will show how practical problems concerning the application of ABV to embedded SW can be solved by applying a MDD-based approach.

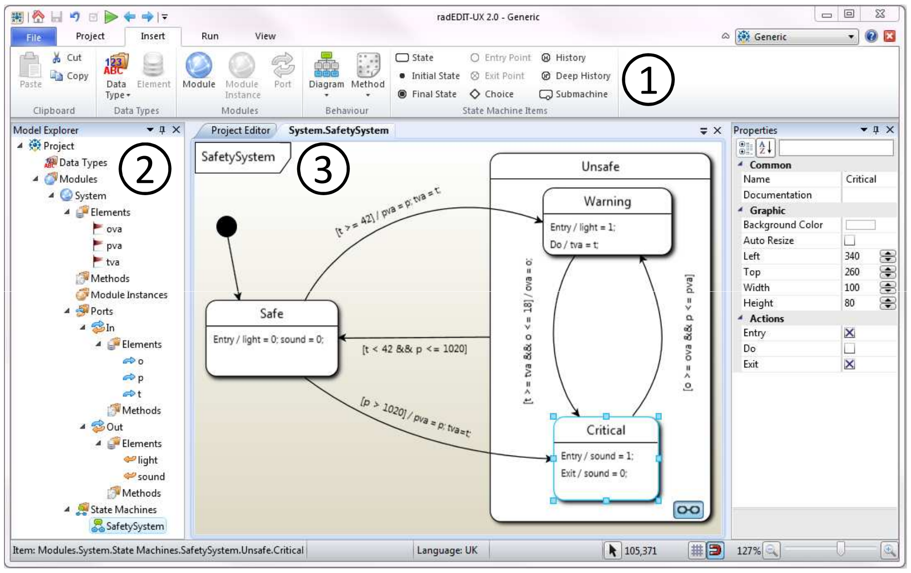
## 3.4 Model-driven design for ABV of embedded software

The radCASE environment has been extended for supporting the dynamic ABV of ESW. In particular, the adoption of graphical and textual formalisms which adhere to the synchronous reactive modeling paradigm [45] allowed us to reach two important results. First, they enabled us to establish a well-defined semantics for ESW execution in such a way the identification of the timing references at which verify assertions is clear. Second, although their heterogeneity, the synchronous reactive models enabled us to define a model-to-model transformation for supporting the automatic generation of stimuli. Indeed, radCASE integrates a synthesis engine of ESW specifications into an EFSM-based model, i.e., a model that efficiently represents the execution flow of the ESW and enables the stimuli-generation approach implemented in radCHECK.

In what follows, Section 3.4.1 describes the environment for the graphical definition of the ESW model. Section 3.4.2 provides the details of the MDA approach implemented in radCASE for the synthesis of the ESW model into executable C-code. Section 3.4.3 shows how MDD based on synchronous reactive models permits to solve the problem of identifying the timing references in ESW. Finally, Section 3.4.4 provides an example of the transformation of an UML-based model of the ESW in an EFSM-based model.

### 3.4.1 Model-driven design environment for embedded software

radCASE is a MDD framework that supports the most important software-engineering standards, such as UML and IEC 61131-3, and MDA for the embedded software development. The radCASE design process is conforming to the Overall Object-Oriented [63] (also called $O^3$ or *Ozon*) approach that provides



| | |
|---|---|
| **Common** | |
| Name | Critical |
| Documentation | |
| **Graphic** | |
| Background Color | |
| Auto Resize | ☐ |
| Left | 340 |
| Top | 260 |
| Width | 100 |
| Height | 80 |
| **Actions** | |
| Entry | ☒ |
| Do | ☐ |
| Exit | ☒ |

Fig. 3.3: radCASE Model Editor window: the GUI provides the designer with all the means for graphically defining ESW structure, functionality and meta-data, as well as requirements, testing scenarios and documentation.

a more integral approach with respect to traditional UML Object-Oriented Design (OOD) [100]. In fact, unlike the other Computer Aided Software Engineering (CASE) tools, radCASE unifies structural information, functionalities and meta-data inside the component (named *module*) they belongs to and not at separate positions. Thus, there is no redundancy of information in the complete model and no manual adjustment are required to align different portions of the model in case of changes on a part of them. Besides, the radCASE project description contains all the information, as requirements specification, underlying architecture annotations, testing scenarios and documentation, to allow a full automatic generation of all necessary outcomes.

In the radCASE Model Editor (Figure 3.3), UML diagrams, IEC 61131-3 standards and C-code are used to model different aspects of ESW (Figure 3.3, *Insert Ribbon Page*, i.e., box 1). As usual, high level requirements are described by means of *Use Case diagrams*. They provide the simplified and graphical representation of what the ESW has actually to do and specify the various ways the actors (either a human or an external system) interact with it. Notice that, a use case diagram is only meant to provide the business reasoning and outcomes of the ESW, while the technical outline of the functionality of the system, is captured using structural and behavioral views.

The *structural views* (Figure 3.3, box 2) define the ESW structure using objects, attributes, operations (i.e., methods) and relationships. radCASE offers a variety of different structural views including:

- *Class diagrams*, to specify components by describing their modeling classes, attributes and methods, and relationships between classes;
- *Composite structure diagrams*, to specify components by describing the internal structure of the modeling classes and which relationships they make possible;
- *Object diagrams*, sometimes referred to as instance diagrams, to specify class instances and relationships between them;
- *Component diagrams*, to specify components by describing their interfaces and the allowed relationships to define larger software systems.

The *behavioral views* (Figure 3.3, box 3) specify ESW behavior by showing collaborations among the components and changes to their internal states. In radCASE, the ESW behavior is specified choosing the best suited formalism among the following:

- *Statecharts*, to represent the series of events and actions that could occur in one or more possible states of a component. Notice that Statecharts may contain superstates, i.e., hierarchically nested states. This formalism has become standard in the CASE tool area;
- *Sequence diagrams*, to represent the flow of methods invocations between different components, where a method invocation corresponds to a message in UML terminology;
- *Activity diagrams*, to model a process control flow specifying stepwise activities, choices, iterations and concurrency;

In radCASE, besides the previous diagrams, the behavioral view of an ESW can be specified using also a subset of the formalisms included into the IEC 61131-3 standard [91]. In particular:

- *Functional block diagrams*, to represent a function between input and output variables of a component. A function is described as a set of elementary blocks. Input and output variables are connected to blocks by connection lines. An output of a block may also be connected to an input of another block.
- *Structured text*, to specify the input/output relations of a component by means of a programming language that is block structured and syntactically resembles Pascal. The language is suited for conditional and iterative coding style.

Finally, radCASE allows to specify components functionality by using C-code. This solution can integrate the graphical elements with the full expressiveness of this procedural language allowing to cover all possible aspects of ESW behavior specification.

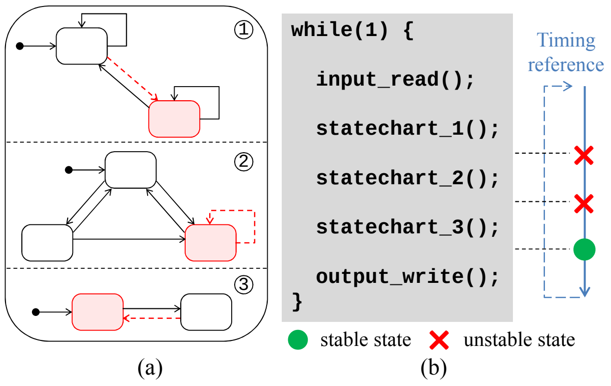
### 3.4.2 Synchronous reactive model synthesis to embedded software

The synchronous reactive paradigm, adopted by radCASE and other state-of-the-practice tools such as IAR VisualState, MathWorks Simulink and SCADE Suite of Esterel Technologies [135], has affirmed for ESW design of safety critical avionics and automotive applications [12], which are reactive systems. A reactive system monitors continuously the environment, at the speed determined by the latter [21]. Moreover, synchronous approaches differ from the more general class of modeling languages that includes support for asynchronous and concurrent execution of components and message passing [20].

In particular, reactive systems are called synchronous when reactions take place within a logical instant, which is shared by all components of the system. Thus, in synchronous reactive systems time is represented as an ordered sequence of instants. All components will complete their behavior for the current instant (or *stabilize*) before the next instant starts. For this reason, in a synchronous reactive modeling environment, components can be designed as part of a single application embedded inside a periodic-execution loop: the application acquires its inputs at the start of the loop, computes using these values and the current states to produce the system next state and related outputs as a single atomic computation step. Communication between components is performed using dataflow signals. Signals consist of data values that are aligned with the global logical instant that is a complete loop execution.

The main advantage of this synchronous reactive paradigm, is the capability of handling input stimuli which trigger internal and output computations, with a simultaneous change of state in a number of concurrent components. All the computations occur in a single instant (atomic computation step), through a deterministic interleaving of local computations including possibly local signaling (i.e., data values propagation) which implement the components concurrency. The local computation ordering is established either by the designer (i.e., designer-defined), who specifies priorities between components execution, or by the model itself (i.e., model-dependent), due to the fact each component involved into the model must be executed (local computation) before any of the components whose input ports it drives. As a consequence, the *run-to-completion* semantics, i.e., the assumption that an event is not processed before the processing of the previous event is fully completed, requires the simultaneous consistent evolution of all components in the system, while propagating signals until a global stable state is found (and the global logical instant is terminated).

For example, let us consider the well-known Statechart model that extends traditional finite state machines with concepts of hierarchy, concurrency, and priority. By using Statecharts, designers can describe ESW functionalities in a concurrent manner. To satisfy the synchronous reactive execution behavior, any MDA approach adopted to transform such a model into an executable PSM has to compile away the concurrency, ordering the Statecharts executions in an opportune way (e.g., designer-defined or model-dependent execution order). For example, let us consider Figure 3.4(a) which reports an example of a system composed of three concurrent Statecharts. The synchronous reactive execution semantic of the model requires to traverse a transition at-a-time for each Statechart, i.e., the dashed transitions in the figure, according to the Statecharts priority. When the last scheduled Statechart advances, the overall system moves in a new *stable* state, i.e., the filled states in the figure, and it is ready for the next step. Thus, an effective way to transform the model into an executable C-code PSM consists of synthesizing the different Statecharts in different C-code functions scheduled according to the designer-defined priorities and execute them in a loop, as shown in Figure 3.4(b): first, the inputs are read; then, each of the machines evolves; the *stable state* is reached after the last function returns; finally, outputs are written.



```
while(1) {
  input_read();
  statechart_1();
  statechart_2();
  statechart_3();
  output_write();
}
```

Fig. 3.4: (a) Example of Statecharts, where three concurrent modules execute according to user-defined priority. (b) The corresponding synthesized code: each function `statechart_X()` implements the corresponding module of the Statecharts in (a); the execution loop monitor continuously the environment; when the last module advances, the overall system moves in a new stable state.

By following this idea, radCASE transforms the models for structural and behavioral specifications (Section 3.4.1) into C-code, implementing all what is specified in the diagrams: static structure, dynamic behavior, I/O mapping, data storage, communications, etc. Indeed, one of the main goals of the MDA approach of radCASE is to avoid the designer to manually complete the generated code, allowing him/her to include procedural code during the modeling phase without breaking the MDD.

The main steps performed by the MDA approach implemented in radCASE to transform PIMs into executable PSMs (i.e., ANSI C-code) are:

1. *PIM to PIM transformation.* In radCASE, a PIM description is composed of UML diagrams and methods (i.e., portions of procedural code) that define different aspects of the ESW model, e.g., structure, functionalities, I/O mapping, parameters, and so forth. Optionally, a set of platform-specific annotations may be specified on the elements of this PIM. Notice that, in the PIM, elements are defined once, but may be instantiated zero, one or many times to specify the whole software system. For such a reason, as first step, radCASE parses the PIM to identify all the instantiated elements, and then it generates a new *intermediate PIM* which specifies only the required parts of the model.
2. *Sanity checks.* Syntax and completeness checks are performed on the intermediate PIM to identify missing elements definition or syntactical errors into the methods code defined by the designer.
3. *PIM to PSM transformation.* Finally, the radCASE code-synthesis engine reads the intermediate PIM and applies to it a pre-defined transformation map for the target implementation platform. The transformation map consists of a set of templates and a run-time mechanisms library. The templates specify the rules to transform the PIM elements into executable code. The optional elements annotations guides the choice of such rules and related optimizations.

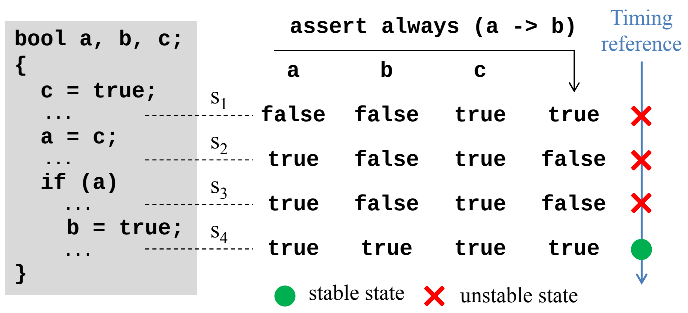
The run-time mechanisms library, instead, implements a set of target platform utilities required by the generated code. Thus, the engine generates the code representing the PSM from the (annotated) PIM, templates and the target platform run-time mechanisms library. The obtained code consists of a *single-thread* application described by means of C language conforming the ANSI standard.

As summarized above, the templates establish how a PIM element is transformed into code. In particular, radCASE includes the following templates:

- *Statecharts* are realized with a hierarchical switch-case construction, where each state of the statechart is described using a switch-case modeling all its entry/during/exit transitions. Superstates (i.e., hierarchical states) are realized as special case-statements including a call to a function implementing the corresponding superstate behavior (i.e., separate switch-case statements);
- *Activity diagrams* are realized as functions implementing the flow between the activities using a label-goto construction. Each activity is implemented as a separate function identified by a label, whereas transitions between activities are modeled with goto statements. Decision points in the diagram are simply realized using an if-then-else statement;
- *Sequence diagrams* are realized as functions implementing the synchronous message passing between objects as an ordered sequence of functions calls. In particular, each object of the diagram is realized as a call to the function implementing the correspondent functionality. Each message between objects is realized as assignments used to model the message exchanging between objects. The order in which the messages are exchanged specifies the order in which function calls are invoked.
- *Functional block diagrams* are realized as functions containing an ordered sequence of functions calls. In particular, each block of the diagram is realized as a call to the function implementing the correspondent block functionality. Each connection between blocks is realized as an assignment to variables used to model the source-destination value passing between blocks. The order of function calls depends on the dependency of blocks inputs on outputs of the other blocks: any function must be executed before any other whose input ports depend on its outputs. Furthermore, the usage of nested functional-block-diagrams generates a hierarchical call of functions.
- *Structured text* code is realized following a linear translation process that maps each PASCAL-like conditional and iterative statement into the corresponding one in C language.

Besides, PIM annotations are exploited to take care of particular hardware aspects and optimize the generated code. In particular, annotations support:

- object-oriented concepts in the ANSI-conformant C code, i.e., inheritance, (multiple) instantiation, and virtualization, for avoiding byte-code redundancy and addressing the strict memory constraints of embedded hardware;
- separated *compilation units*, for limiting the size of compiled objects and data segments for supporting performances- and memory-constrained processors (e.g., Atmel ATtiny);



```
bool a, b, c;
{
  c = true;
   ...
  a = c;
   ...
  if (a)
    ...
    b = true;
    ...
}
```

| | a | b | c | | |
|---|---|---|---|---|---|
| $s_1$ | false | false | true | true | ✗ |
| $s_2$ | true | false | true | false | ✗ |
| $s_3$ | true | false | true | false | ✗ |
| $s_4$ | true | true | true | true | ● |

Fig. 3.5: Is the assertion really violated in these states?

- different binary-data format, e.g., *big* or *little endian* alignment in a single 8-, 16-, 32-bit word, for supporting different processor families and architectures.

### 3.4.3 The problem of timing references

As previously described, MDD refers to the use of graphical modeling languages that allow to create an abstract model of the intended design. Such a model can be simulated, verified and refined in a iterative manner, very often with the use of automatic tools, till specifications are satisfied. Then, the model is automatically synthesized into the final implementation for the target platform. As a drawback, the designer has limited control on the synthesis process: software coding choices, e.g., functional partitioning and variable scope assignment, are delegated to the synthesis tool, which has only to preserve the semantic imposed by the abstract model. However, from the point of view of dynamic ABV, what appears to be a drawback becomes an advantage. In fact, automatic synthesis of ESW prevents designers from implementing code that escapes from a pre-defined template. Thus, it enables the definition of a dynamic ABV approach that overcomes the practical issue of identifying the timing references at which evaluating assertions.

Let us consider the piece of code and the assertion reported in Figure 3.5. By adopting the "single statement" as timing reference [103], the checker corresponding to the assertion is evaluated in states $s_1, \ldots, s_4$. For states $s_2$ and $s_3$, `a` is set to `true` and `b` not yet, thus the assertion is violated, but does this *really* violate the intent of the verification engineer? Indeed, states $s_1, \ldots, s_3$ should be considered unstable, because the computation is still on-going. Unstable states characterize (embedded) software and may provide incorrect responses in terms of verification. Such unstable states are similar to the intermediate configurations, which characterize cycle-based or event-driven simulation of hardware descriptions. To overcome this issue, in the hardware domain, dynamic ABV evaluates temporal assertions when the system reaches a synchronization event, e.g., a clock event.

Thus, a similar strategy is adopted to identify the exact time at which checkers (whose implementation details are reported in Section 3.5.2) must be activated to verify ESW. In particular, the proposed approach exploits the timing reference that characterizes synchronous-embedded applications generated by model-driven approaches.

Thus, considering the example shown in Figure 3.4, checkers are activated only *at the end of the execution loop of the application*, when each Statechart has executed at most one transition and the stable state is reached.

Activation of checkers at a higher or lower rate may correspond to their non-correct evaluation. Figure 3.6 shows different timing references for ESW.

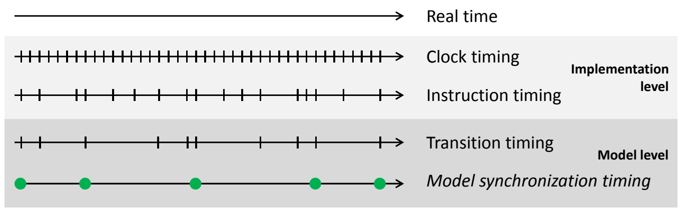


Fig. 3.6: Timing references for ESW.

At implementation level, i.e., source code, it is possible to distinguish clock and instruction timing references. Checkers evaluation at each clock cycle or after each statement in the application implies a high evaluation rate, which may trigger erroneous violation of assertions, as shown in the example of Figure 3.5. At model level, instead, transition and model-synchronization timing references can be distinguished. Transition-synchronization timing reference is the instant at which each single transition inside a UML diagram (e.g., Statecharts, Activity diagrams, Sequence diagrams, . . . ) completes. The example in Figure 3.4 shows that unstable states occur and may affect assertion evaluation. Finally, model-synchronization timing reference is the instant in which each diagram has executed at most one transition and all are waiting for the new iteration of the main loop. It is worth noting that only the model-synchronization timing reference provides the right time instants for checkers evaluation, i.e., when the system has reached a stable state.

### 3.4.4 Model transformation to EFSM

Because of the mixed nature of radCASE specifications, which combines different formalisms, the EFSM synthesis is required for providing the stimuli-generation phase (Section 3.6) with an abstract, uniform, and efficient model for describing the ESW behavior.

An EFSM is a Mealy finite state machine augmented with a finite number of *internal variables*, which are not part of the set of explicit states [37]. Each EFSM transition, noted as $t$, is labeled with an *enabling function* and an *update function*. The former is a triggering condition over internal and input variables. The latter is a sequence of assignments to internal and output variables.

A pair $\langle s, \overline{x} \rangle$, where $s$ represents the state and $\overline{x}$ represents the values of the internal and output variables, is called *configuration* of the EFSM. The *reset con-*

*figuration* is the pair $\langle s_0, \overline{x}_0 \rangle$, where $s_0$ is the *reset state*, and $\overline{x}_0$ represents the reset values of the internal and output variables. Traversing a transition $t$ changes the EFSM configuration. A *reset transition* reverts the system to the reset configuration $\langle s_0, \overline{x}_0 \rangle$. Notice that, there is a reset transition outgoing from each state of the EFSM.

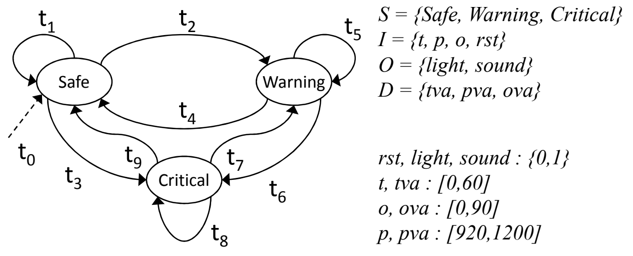
A *path* of the EFSM is a sequence of adjacent transitions starting from the reset transition. Intuitively, it describes one possible behavior of the EFSM, i.e., sequence of EFSM configurations. In this work we consider *deterministic* EFSMs. In a deterministic EFSM, for every state, outgoing transitions have mutually exclusive enabling functions. Such a condition also fits the semantic of imperative languages, e.g. C.

In the following, we show which is the EFSM model extracted from the ESW specification depicted in Figure 3.3. For the EFSM extraction, radCASE analyzes the intermediate PIM of the ESW generated during the PSM synthesis from the initial user-defined PIM (Section 3.4.2), and not the PSM itself. This is due to the fact that code target-dependent optimizations, which are required by the underlying hardware platform, limit the applicability of the approach for symbolic generation of stimuli. Moreover, this choice presents a further advantage, because both the graphical and textual models are synthesized in the intermediate PIM removing the unused diagrams and code (e.g., non-instantiated classes, un-used methods, etc.). It is worth noting that rules for generating EFSM models are thoroughly described in [50], for this reason we do not detail them.

Figure 3.3 reports the specification of a simplified in-flight safety system that monitors the temperature, the pressure and the oxygen status of the cabin and notifies alarms by turning on and off emergency light and buzzer, according to the danger level of the situation. The ESW structure (Figure 3.3, box 2) is specified as a *Class* characterized by several attributes that model input, output and internal variables of the ESW. In particular, the variables $\{o, p, t\}$ are the input variables and represent the temperature ($°C$), the pressure ($hPA$), the oxygen rate ($\%V/V$) of the cabin. The variables $\{sound, light\}$ are the output variables and represent the emergency light and sound (i.e., buzzer) controls. $\{ova, pva, tva\}$, instead, are the internal variables which store the values of the cabin parameters. For each input, output and internal variable, the domain is specified as follows: $light$ and $sound$ are Boolean variables, i.e., their domain is $\{0, 1\}$; $t$, $tva$ assume values in the interval of integers $[0, 60]$; $p$, $pva$ assume values in the interval of integers $[920, 1200]$; finally, $o$, $ova$ assume values in the interval of integers $[0, 90]$. Moreover, internal and output variables are all initialized to 0.

The ESW behavior is specified by using a hierarchical Statechart (Figure 3.3, box 3). The Statechart is characterized by an initial state, i.e., $Safe$, and a superstate, i.e., $Unsafe$, which consists of two different states, i.e., $Warning$ and $Critical$.

According to the model, when the temperature exceeds the threshold of $42°C$ degrees or the pressure is higher than 1020 $hPA$, the system moves from the $Safe$ to the $Unsafe$ state. In the former case, it enters the $Warning$ state. In the latter, it enters immediately the $Critical$ state. In the $Warning$ state, the system continuously monitors if the cabin temperature raises whereas the oxygen rate falls below a threshold of 18% $V/V$. If this is the case, it moves to the $Critical$



| T | En. Func. | Up. Func. |
|---|---|---|
| $t_0$ | $rst = 1$ | $tva = 0;\ ova = 0;\ pva = 0;$ $light = 0;\ sound = 0;$ |
| $t_1$ | $t < 42 \wedge p \geq 980$ | $-$ |
| $t_2$ | $t \geq 42$ | $tva = t;\ pva = p;\ light = 1;$ |
| $t_3$ | $t < 42 \wedge p > 1020$ | $tva = t;\ pva = p;\ sound = 1;$ |
| $t_4$ | $t < 42 \wedge p \leq 1020$ | $light = 0;\ sound = 0;$ |
| $t_5$ | $t \geq 42 \wedge p > 1020 \wedge$ $t \leq tva \wedge o \geq 18$ | $tva = t;$ |
| $t_6$ | $t \geq tva \wedge o < 18$ | $ova = o;\ sound = 1;$ |
| $t_7$ | $o > ova \wedge p < pva$ | $sound = 0;$ |
| $t_8$ | $o \leq ova \wedge p \geq pva \wedge$ $t \geq 42 \wedge p > 1020$ | $-$ |
| $t_9$ | $t < 42 \wedge p \leq 1020$ | $light = 0;\ sound = 0;$ |

Fig. 3.7: An EFSM specification of a simplified in-flight safety system.

state. In the $Critical$ state, the system constantly monitors the oxygen rate and the pressure of the cabin. Only when oxygen rate starts rising and the pressure decreases, the system moves back to the $Warning$ state. Finally, only when the temperature, pressure values return into the safe interval (i.e., $t < 42$, $p \leq 1020$) the system returns in the $Safe$ state.

Figure 3.7 reports the corresponding EFSM specification of the simplified in-flight safety system. Unlike the original Statechart, the states of the EFSM are $S = \{Safe, Warning, Critical\}$, where $Safe$ is the reset state, and the superstate $Unsafe$ is removed. The input variables are $I = \{t, p, o, rst\}$ and represent the corresponding temperature, pressure and oxygen variables, whereas $rst$ represents the EFSM reset signal. $O = \{light, sound\}$ is the set of output variables corresponding to light and sound controls. Finally, $D = \{tva, pva, ova\}$, is the set of internal variables of the EFSM. For each transition, the enabling function and update function are reported in the table. For readability, only a reset transition $t_0$ is depicted with a dotted arrow and represents each of the reset transitions outgoing from the states of the EFSM.

Notice that the EFSM model in Figure 3.7 is flatten and reports explicitly all the transition of the original Statechart (including the implicit ones). Enabling functions are derived from transition conditions of the Statechart diagram and defined in such a way they are mutually exclusive. Again, update functions are derived from transition action of the Statechart diagram and are extended by analyzing the entry/exit actions of the states that represent the source and the target of the transition. In particular, the entry and exit actions of a state are reported in the update conditions of the ingoing and outgoing transitions of the state, respectively. Moreover, new transitions are introduced to specify the *during actions* defined into the Statechart states. They are realized as self loops on the corresponding states. Finally, due to the fact that superstates are flatten, their outgoing transitions are reported as outgoing transitions of each internal state of the superstate.

## 3.5 ABV for model-driven embedded software

The radCHECK environment has been defined and implemented to overcome the typical problems of adopting dynamic ABV techniques for verifying ESW. First, the Property Editor (PE) graphically guides the designer in defining assertions by providing him/her with all the information (e.g., software structure, variables identifier and data type, parametric assertion templates, etc.) required to formalize the informal ESW specifications. In this way, the PE allows to easily exceed the challenge of teaching the designer formal assertion languages making them master quickly assertion definition. Moreover, the PE can update the ESW structural specifications by modifying variables scopes (and handling the related problem of name clashing) to avoids that variables, which need to be accessed by checkers during the simulation, will be hidden by the ESW C-code synthesis from the UML diagrams.

The integrated Checker Generator (CG) engine, unlike the existing checker generators, is effectively suited for synthesizing checkers able to verify the ESW C-code implementation. In fact, because of the C language is rich of complex data-types (e.g., floats, pointers, user-defined types), the CG exploits the information reported in the UML specifications of the ESW for extracting all it needs for defining support functions that retrieve the value of variables declared by using such complex data-types. Thus, all the possible variables of the ESW can be checked by generated checkers.

Finally, to avoid the tedious manual generation of input stimuli for executing the ESW and checkers, radCHECK provides the designer with a stimuli generation engine, i.e. Ulisse. Ulisse is based on a corner-case-oriented concolic approach, that generates effective stimuli by exploiting the EFSM-based representation of the ESW. In this way, also the simulation phase is automated.

In what follows, each component of radCHECK is analyzed in details. At first, the assertion definition infrastructure, i.e., PE, is described in Section 3.5.1. Section 3.5.2 is devoted to the CG engine that synthesizes the assertions into C-code checkers enabling the functional verification of the ESW. Finally, the algorithms implemented by the stimuli generation engine integrated in radCHECK are summarized in Section 3.6.

### 3.5.1 Assertion definition for model-driven embedded software

The PE (Figure 3.8) is a graphical tool that provides the designer with a means for extracting automatically the information he/she needs for defining assertions.

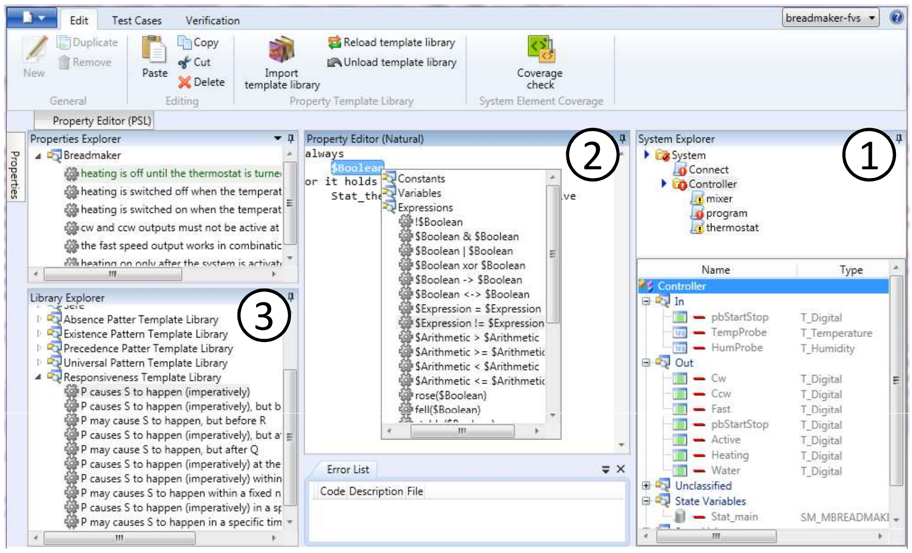

Edit
Test Cases
Verification
breadmaker-fvs
New
Duplicate
Remove
General
Paste
Copy
Cut
Delete
Editing
Import template library
Reload template library
Unload template library
Property Template Library
Coverage check
System Element Coverage
Property Editor (PSL)
Properties
Properties Explorer
Breadmaker
heating is off until the thermostat is turne
heating is switched off when the temperat
heating is switched on when the temperat
cw and ccw outputs must not be active at
the fast speed output works in combinatic
heating on only after the system is activat
Library Explorer
3
Absence Patter Template Library
Existence Pattern Template Library
Precedence Patter Template Library
Universal Pattern Template Library
Responsiveness Template Library
P causes S to happen (imperatively)
P causes S to happen (imperatively), but b
P may cause S to happen, but before R
P causes S to happen (imperatively), but a
P may cause S to happen, but after Q
P causes S to happen (imperatively) at the
P causes S to happen (imperatively) within
P may causes S to happen within a fixed n
P causes S to happen (imperatively) in a sp
P may causes S to happen in a specific tim
Property Editor (Natural)
2
always
$Boolean
or it holds
Stat_the
ve
Constants
Variables
Expressions
!$Boolean
$Boolean & $Boolean
$Boolean | $Boolean
$Boolean xor $Boolean
$Boolean -> $Boolean
$Boolean <-> $Boolean
$Expression = $Expression
$Expression != $Expression
$Arithmetic > $Arithmetic
$Arithmetic >= $Arithmetic
$Arithmetic < $Arithmetic
$Arithmetic <= $Arithmetic
rose($Boolean)
fell($Boolean)
Error List
Code Description File
System Explorer
1
System
Connect
Controller
mixer
program
thermostat
Name
Type
Controller
In
pbStartStop T_Digital
TempProbe T_Temperature
HumProbe T_Humidity
Out
Cw T_Digital
Ccw T_Digital
Fast T_Digital
pbStartStop T_Digital
Active T_Digital
Heating T_Digital
Water T_Digital
Unclassified
State Variables
Stat_main SM_MBREADMAKI

Fig. 3.8: Property Editor main window: the GUI provides the user with information about the ESW structure for guiding, with an effective point-and-click mechanism, the assertion definition based on parametric templates.

In particular, by analyzing the radCASE intermediate UML specification of the ESW, the PE automatically extract a list of component signatures which are being implemented into the ESW. This list is shown by using a tree view (Figure 3.8, box 1) that gives a well-organized description of the ESW. Notice that, by selecting a node of the tree view, i.e., choosing a particular component, the GUI shows the component parameters and internal variables as well as the list of internally referred components. Both the parameters and variables information are shown by reporting their own identifier and the associated data type. These information are also stored into a specific data-dump successively exploited by the CG (Section 3.5.2).

Moreover, for formalizing the informal specifications, the PE provides the designer with parametric assertion templates [52] which specify different property patterns [56] (i.e, functional behaviors that a verification engineer usually verifies on ESW implementations). These assertion templates ensure a clean separation between assertion semantics and its formal definition. They are characterized by (i) an *interface* (Figure 3.9a) extended with placeholders (i.e., *$P*, *$Q* and *$R*) that provide an intuitive idea of the property pattern meaning and (ii) a *formal parametric PSL definition* (Figure 3.9b). In this way, for defining an assertion, the designer needs only to understand the semantic of the interface and graphically replace the placeholders with the intended expressions by exploiting drag-and-drop as well as point-and-click mechanisms.

The parameter replacement is guided by the tool: the placeholders are strongly typed parameters and the designer can replace them only with legal elements according to specific semantic checks included into the PE. Practically, when the

**$P** holds at least once in between **$Q** and **$R**.

(a) The interface

next_event! (**$Q**)(**$P** before!_ **$R**) & eventually! **$R**)

(b) The PSL parametric definition

Fig. 3.9: Example of a parametric assertion template: the interface simply explains to the user the assertion meaning whereas the PSL parametric definition is used by the tool for automatically generating the formal PSL assertion.

designer selects a specific placeholder the PE blocks the drag-and-drop of all the illegal expressions for that placeholder, whereas, if the designer uses the point-and-click substitution mechanism, he/she is provided immediately with a list of all the legal expressions (Figure 3.8, box 2). In this way the PE avoids the definition of syntactically erroneous assertions.

It is clear that the more the set of parametric assertion templates is complete (Figure 3.8, box 3) the easier is the definition of semantically correct assertions. At the moment, more than 60 templates have been defined and have been organized into 5 libraries (e.g., a selection of templates is reported in Table 3.1) each one focuses on a specific category of patterns: universality, existence, absence, responsiveness and precedence. The *universality* library describes behaviors that must hold continuously during the software execution (e.g., a condition that must be preserved for the whole execution, a condition that has to hold continuously after that the software reaches a particular configuration, etc). The *existence* library describes behaviors in which the occurrence of particular conditions is mandatory for the software execution (e.g., a condition must be observed at least once during the whole execution or after that a particular configuration is reached, etc). The *absence* library describes behaviors that must not occur during the software execution or under certain conditions. The *responsiveness* library, instead, describes behaviors that specifies cause-effect relations (e.g., a particular condition implies a particular configuration of the software variables, etc). Finally, the *precedence* library describes behaviors that require a precise ordering between conditions during the software execution (e.g., a variable has to assume specific values in an exact order).

It is worth noting that this initial set of templates provided with the PE can be extended according to the designer needs. The new parametric assertion templates can be defined by specializing the existing ones or starting from scratch by the most confident designers. Moreover, the new parametric assertion templates can be organized into new libraries or added to the existing ones, sorted according to the different functional aspects they aims to check. In this way, the so defined libraries of templates can be shared between designers allowing also the inexpert ones to get confident in defining formal assertions.

Table 3.1: Selection of assertion templates.

| **Library** | **Parametric Interface** | **Parametric PSL definition** |
|---|---|---|
| Universality | P holds continuously | `always $P` |
| | P holds continuously since Q | `next_event! ($Q)( always $P ))` |
| | P holds continuously until R | `$P until! $R` |
| | P holds continuously in between Q and R | `next_event! ($Q)($P until!_ $R))` |
| Existence | P holds at least once | `eventually! ($P)` |
| | P holds at least once since Q | `next_event! ($Q)(eventually! $P)` |
| | P holds at least once before R | `(eventually! $R) & ($P before! $R)` |
| | P holds at least once in between Q and R | `next_event! ($Q)($P before!_ $R) & eventually! $R)` |
| Absence | P never holds | `never $P` |
| | P never holds since Q | `next_event! ($Q)(never $P)` |
| | P never holds until R | `(never $P) until! $R` |
| | P never holds in between Q and R | `next_event! ($Q)((never $P) until!_ $R))` |
| Responsiveness | P causes S to happen at the same time | `always ($P -> $S)` |
| | P causes S to happen eventually | `always ($P -> eventually! ($S))` |
| | P causes S to happen, but before R | `always ($P -> (($S before! $R) & eventually!( $R )))` |
| | P causes S to happen, but after Q | `always ($P -> ((eventually! $S) & ($Q before! $S)))` |
| Precedence | P precedes S | `($P before! $S) & (eventually! $S)` |
| | P precedes S after Q | `next_event!($Q)(($P before! $S) & (eventually! $S))` |
| | P precedes S before R | `((!$S & !$R) until!_ $P) & eventually! ($R) &`<br>`next_event! ($P)(next($S before! $R))` |
| | P precedes S in between Q and R | `((!$P & !$S & !$R) until!_ $Q) & (eventually! $R) &`<br>`next_event!($Q)(($P before! $S) & ($S before! $R))` |
| | P precedes S repeatedly | `($P before! $S) & always ($P -> next ($S before $P)) &`<br>`always ($S -> next ($P before $S))` |

### The problem of variable visibility

C and C++ are the main languages used in the industry for the implementation of ESW applications [104]. Variables in C/C++ can be either of *global* or *local scope* and this affects the variable accessibility in the defined assertions: global variables, declared in the main body of the source code, outside all functions, can be referred from anywhere in the code, and also by the checkers (i.e., C functions) implementing the assertions semantics. Local variables, instead, are declared within a function body or a block, thus, their visibility is limited to the block where they are declared, preventing the checkers from any variable reference. As a consequence verification engineers who define assertions starting from system specifications and, in first instance, ignoring some of the implementation details, may be forced to do code re-factoring. This is a time-consuming and error-prone activity (e.g., variable name clashing).

The PE plays an important role in ensuring the right variable visibility to both the designer, during assertion definition, and the CG engine, during checker synthesis. As previously described (Section 3.5.1), the PE is able to identify all the parameters and the variables of the ESW by analyzing the UML diagrams describing the ESW model. Then the tool provides the designer with such variables organized by scopes. Once the designer has defined the intended assertions, the PE updates the UML diagrams by marking as *visible* all the variables referenced by such assertions. As a consequence, these variables are being synthesized in the ESW implementation in such a way they can be accessed by the checkers that will be generated.

## 3.5.2 Checker generation for model-driven embedded software

The CG synthesizes the assertions defined by the designer into compact procedural C-code functions that can be distinguished in: *logic*, *wrapper* and *checking* functions.

The *logic* functions (one for each assertion) implement the semantics of the checker. The logic function definition differs according to the assertion, but their signature is always characterized by the following set of Boolean parameters (Figure 3.10):

- the *enable* parameter is used to start the assertion evaluation in any instant of the simulation, and, in particular, to prevent a checker from scanning variables strictly before they have been initialized (i.e., in the unstable initial states);
- the *reset* parameter is used to (re-)initialize the checker to its initial configuration every time a new assertion evaluation has to be started;
- the *fail* parameter is set by the checker to notify an assertion violation;
- the *fail_if_last_cycle* parameter is set by the checker when an unfulfilled obligation has been detected. In this way it is possible to distinguish a property failure due to a too short simulation execution from a failure due to the presence of an error into the ESW implementation (i.e., fail).
- finally, the Boolean parameters $v_0 \dots v_n$ represent the minimal subformulas of the assertion. The values of these minimal subformulas are retrieved by the wrapper function associated to each logic function.

For completeness sake, Figure 3.10 reports the logic function implementation, but notice that the handling of complex data structures is independent from it, as explained in the following.

```
//assert
// always (((thermo.Heating = On) &
//   (thermo.Temperature) >
//      (thermo.Setpoint - thermo.Hysteresis)) ->
//                  eventually! (thermo.Heating = Off))
// assert always v0 & v1 -> eventually! v2
#include <stdint.h>

#define CONVERT_BOOL(b) ((b) ? 0x3 : 0x0)

typedef int bool;

void checker_logic( bool const enable, bool const reset,
                    bool* fail, bool* fail_if_last_cycle,
                    bool v0, bool v1, bool v2)
{
// 1. Define registers.
  static int unsigned var12;
  static int unsigned var13;
  static int unsigned enable;
  if (reset) {
    var13 = CONVERT_BOOL(false);
    var12 = CONVERT_BOOL(false);
  }
  else {
    // 4. Compute intermediate values.
    int unsigned var0 = /* lastCycle */ 0x2;
    int unsigned var1 = enable;
    int unsigned var2 = CONVERT_BOOL(v0);
    int unsigned var3 = CONVERT_BOOL(v1);
    int unsigned var4 = CONVERT_BOOL(v2);
    int unsigned var5 = ~var4;
    int unsigned var6 = var1 | var12;
    int unsigned var7 = var2 & var3;
    int unsigned var8 = var6 & var7;
    int unsigned var9 = var8 | var13;
    int unsigned var10 = var5 & var9;
    int unsigned var11 = var0 & var10;
    int unsigned tmp_var12 = var1 | var12;
    int unsigned tmp_var13 = var5 & var9;
    // 5. Checker outputs.
    fail_if_last_cycle = (var11 & 0x2) != 0;
    fail = (var11 & 0x1) != 0;
    // 6. Update registers.
    var12 = tmp_var12;
    var13 = tmp_var13;
  }
}
```

Fig. 3.10: The C-code implementation of a typical logic function.

The *wrapper* function (Figure 3.11) contains the satellite expressions that handle the use of ESW complex data structures inside PSL assertions (Section 3.5.2). The satellite expressions implement the minimal subformulas of a PSL assertion (e.g., `thermo.Heating = On`, `thermo.Temperature` > `(thermo.Setpoint - thermo.Hysteresis)`, `thermo.Heating = Off`) and are automatically ex-

```
1  void get_var(
2      struct System const* sys,
3      char const* name,
4      void* var,
5      long unsigned var_sizeof);
6
7  #define GET_VAR(T,N,S) \
8    T N; get_var(sys, S, &N, sizeof(T));
9
10 void checker_wrapper(bool const enable,
11     bool const reset,
12     bool* fail,
13     struct System const* sys) {
14
15   // 1. Retrieve system-variable values
16   GET_VAR(int8_t, thermo_Heating,"thermo.Heating");
17   GET_VAR(int16_t, thermo_Hysteresis,"thermo.Hysteresis");
18   GET_VAR(double, thermo_Setpoint,"thermo.Setpoint");
19   GET_VAR(double, thermo_Temperature,"thermo.Temperature");
20
21   // 2. Evaluate complex Boolean expressions
22   bool v0 = (thermo_Heating == /* On */ 1);
23   bool v1 = (thermo_Temperature >
24       (thermo_Setpoint - thermo_Hysteresis));
25   bool v2 = (thermo_Heating == /* OFF */ 0);
26
27   // 3. Invoke the checker over Boolean variables
28   checker_logic(enable, reset, fail, v0, v1, v2);
29 }
```

Fig. 3.11: The C-code implementation of a typical wrapper function.

```
1  typedef void (*checker_binding_func)(bool enable,
2      bool reset, bool* fail,
3      struct System const* sys);
4
5  void checker_run_check(bool enable, bool reset,
6      struct System const* sys,
7      checker_binding_func* checkers) {
8    bool fail;
9    int unsigned i;
10   update_log();
11   for (i=0; checkers[i]; ++i) {
12    checkers[i](enable, reset, &fail, sys);
13    if (fail)
14      update_fail_log(i);
15   }
16 }
```

Fig. 3.12: The C-code implementation of a typical checking function.

tracted by the CG to correctly evaluate the corresponding logic function (line 28).

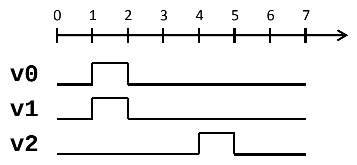
The *checking* function (Figure 3.12) is used to perform a verification step when a stable configuration of the ESW is reached. It executes all the synthesized checkers by calling the corresponding wrapper functions (e.g., `checker_wrapper`, line 11). Moreover, it manages a log file in which it stores the simulation traces (i.e., variable evolution). At each invocation, labeled with a time stamp inside the log file, the current values of variables are recorded (line 9) and, error notification is reported whenever an assertion violation occurs(lines `12-13`). In this way, simulation traces can be used for debugging purposes. In fact, the trace portion that precedes an error message into the log file represents a counterexample to the satisfiability of the assertion.

## The problem of complex data-types handling

PSL, being an extension of LTL and CTL temporal logics towards HW design, natively supports the specification of temporal assertions based on Boolean expressions. It supports also relational expressions on a restricted set of operands (integers, Booleans, bit vectors). Indeed, RTL and gate-level HW descriptions can be easily brought down to the Boolean level, for example see Figure 3.13(a). As a consequence, all the existing checker generators adhere to such a restriction. On the contrary, temporal assertions for ESW may involve more complex relational expressions, since C language allows to declare variables using complex data types, e.g., floats, pointers, structures, unions, etc., which are not supported by PSL.

(a)

```
always (v0 & v1 → eventually! (v2))
```



(b)

```
always ((thermo.Heating = ON) &
        (thermo.Temperature >
          (thermo.Setpoint - thermo.Hysteresis)) ->
              (eventually! (thermo.Heating = OFF) )
```

Fig. 3.13: Example of a PSL assertion for HW (a) and ESW (b).

Figure 3.13(b) reports a temporal assertion defined for an ESW application. Its meaning is the same as in Figure 3.13(a), but the Boolean expression is built on top of more complex types and data-structures.

To take care of such a complexity, the CG environment extracts, from each assertion defined by using the PE, all the minimal subformulas (i.e., the relational expressions) replacing them with fresh Boolean variables. In this way, it generates (i) pure PSL assertions and (ii) mappings between the several fresh Boolean variables and the corresponding minimal subformulas which may contain variables with complex data-types (e.g., floats, pointers, etc.).

Thus, the CG engine can easily synthesize the PSL assertions into correct-by-construction executable checkers by adopting a *satellite*-based approach. A satellite is an arbitrary complex propositional expression wrapped into a C function that returns a Boolean value representing the evaluation of the complex expression

(Figure 3.11). In particular, the data-dump created by the PE stores the name of the ESW variables, as well as their type. In this way, the CG can correctly specify a unique function (lines `1-5`) to retrieve from a structure containing all the visible variables defined in the ESW (i.e., `sys`, line `2`), the value of variables required to evaluate the satellite expressions (lines `16-19`). Such an evaluation is stored into support Boolean variables (lines `22-25`) that can be used as parameters for the checker evaluation function (i.e., `checker_logic`, line `23`) implementing the related PSL assertion semantics.

## 3.6 EFSM-based stimuli generation

Let us consider a generic EFSM and let $t$ be a transition of this EFSM. Traversing the transition $t$ depends on the values of both input and internal variables within the *enabling function* of $t$. Moreover, the internal variable values depend on the *update functions* in the path leading from the reset state to $t$. In the following, a *target transition* $t$ is supposed to be a *not-yet-traversed* transition.

This section presents the concolic stimuli generation approach for ESW that has been implemented into Ulisse. The approach is based on the EFSM model of the ESW and leads to traverse a *target transition* by integrating concrete execution, which reaches deep states of the system, and a symbolic technique, that is weight-oriented and ensures exhaustiveness along specific paths.

**Algorithm 3:** The EFSM-based concolic algorithm for stimuli generation; it alternates concrete approach, i.e, *LongRangeSearch*, and weight-oriented symbolic approach, i.e., *GuidedWideWidthSearch*.

```
procedure EfsmStimuliGen(Efsm, MaxTime, InaTime)
input: embedded-application model Efsm,           overall timeout
       MaxTime,                          inactivity timeout InaTime
output: set of stimuli Stimuli
1 Stimuli $\leftarrow \emptyset$; RInf $\leftarrow \emptyset$;
2 DInf $\leftarrow$ DependencyAnalysis(Efsm);
3 while elapsed time $<$ MaxTime do
4     while inactivity timeout InaTime not expired do
5         (stimulus, reach) $\leftarrow$ LongRangeSearch(Efsm, RInf);
6         Stimuli $\leftarrow$ Stimuli $\cup$ {stimulus}; RInf $\leftarrow$ RInf $\cup$ {reach};
7     (stimulus, reach) $\leftarrow$ GuidedWideWidthSearch(Efsm, RInf, DInf);
8     Stimuli $\leftarrow$ Stimuli $\cup$ {stimulus}; RInf $\leftarrow$ RInf $\cup$ {reach};
9 return Stimuli
```

Algorithm 3 is a high-level description of the proposed concolic approach. It takes as inputs the EFSM model and two timeout thresholds: overall timeout and inactivity timeout, i.e., *MaxTime* and *InaTime* respectively, which are measured in milli-seconds (*real CPU time*). The overall timeout is the maximum execution time of the algorithm; the inactivity timeout is the maximum execution time the long-range concrete technique can spend without improving the transition coverage. At the beginning, the stimuli set *Stimuli* is empty, and no reachability information, i.e., *RInf*, is available (line 1). In particular, *RInf* keeps track of the EFSM configurations which are used for (re-)storing the system status when the algorithm switches between the symbolic and concrete techniques. The algorithm identifies the dependencies between internal and input variables and EFSM paths. At first, it statically analyzes the *EFSM transitions*, and it generates initial dependency information, i.e., *DInf* (line 2), which is used in the following corner-case-oriented symbolic phases, when a further dynamic analysis between *EFSM paths* is performed. Such a dependency analysis permits to selectively choose a path for the symbolic execution when the concrete technique fails in improving the transition coverage of the EFSM. The stimuli generation runs until the specified overall timeout expires (line 3). First, the algorithm executes a long-range concrete technique (line 5), then a symbolic wide-width technique, which exploits the multi-level backjumping (MLBJ) to cover corner cases (line 7). The latter starts when the transition coverage remains steady for the user specified inactivity timeout (line 4). The algorithm reverts back to the long-range search as soon as the wide-width search traverses a target transition. Finally, the output of the algorithm is the generated stimuli set (line 9). The adopted long-range search (line 5) exploits constraint-based heuristics [51], that focus on the traversal of just one transition at a time. Such approaches scale well with design size and, significantly improve the bare pure-random approach.

The following sections describe, respectively, the proposed dependency analysis over the EFSM model, the snapshot-and-restoring mechanism required during the switch between the concrete and symbolic execution and, finally, the MLBJ that addresses the selective symbolic execution of EFSM paths which are terminating with the target transition and have a high dependence on the inputs.

## Dependency analysis

Without a proper dependency analysis, the stimuli-generation engine wastes considerable effort in the exploration of uninteresting parts of the design. Thus, the proposed approach focuses on dependencies of enabling functions, i.e., control part, on internal variables. As a further motivating example let us consider the EFSM in Figure 3.7. Let $t_8$ be the target transition. Let us compare paths $\pi_1 = t_2 :: t_6$ and $\pi_2 = t_3$. ($t :: t'$ denotes the concatenation of transitions $t$ and $t'$.)

The enabling function of $t_8$ involves the variables '*ova*' and '*pva*'. Both are defined along $\pi_1$ by means of primary inputs. Along $\pi_2$ only '*pva*' is defined by means of primary inputs. Thus, to traverse $t_8$, MLBJ will select $\pi_1$ instead of $\pi_2$ since $t_8$ enabling function is more likely to be satisfied by the symbolic execution of $\pi_1$ rather than $\pi_2$. We will consider again this example at the end of the section.

Dependencies are approximated as *weights*. Indirect dependencies, as in the sequence of assignments '$d_1 := i_1 + i_2;\ d_2 := d_1 + i_3$' the dependency of $d_2$ on $i_1$, are approximated as *flows* of weights between assignments. Given a target transition $\bar{t}$, each path ending in $\bar{t}$ is mapped to a non-negative weight. Intuitively, the higher is this weight, the greater is the dependency of the enabling function of $\bar{t}$, i.e., $e_{\bar{t}}$, on inputs read along such a path, and the higher is the likelihood that its symbolic execution leads to the satisfaction of $e_{\bar{t}}$. This section formalizes how a path weight is computed.

An initial weight is assigned to $e_{\bar{t}}$, then it "percolates" backward along paths. Each transition lets a fraction of the received weight percolate to the preceding nodes and retains the remaining fraction. The weight associated with a path $\pi$ is defined as *the sum of weights retained by each transition of* $\pi$. The ratio of the weight retained by a transition is defined by its update function. In the following, these concepts are formally defined.

**Definition 3.1 (Weight tuple).** *Let $W \subseteq \mathbb{R}^k$. A* weight tuple *$w \in W$ is a tuple of the form $w = \langle w_1, \dots, w_k \rangle$, i.e., a $k$-tuple of non-negative real values. $w_i$ denotes the $i$-th element of $w$ where $i \in \{1, \dots, k\}$.*

Given $w$, let $w_i$ be the weight associated with $d_i \in D$, that is the $i$-th internal variable of the design. Given a target transition $\bar{t}$, the computation of the weight associated with a path ending in $\bar{t}$ starts considering an *initial weight tuple* $w^0$ that assigns higher weight to variables that influence the satisfaction of $e_{\bar{t}}$, i.e., $w_i^0 = 1$ if $d_i$ occurs in $e_{\bar{t}}$, $w_i^0 = 0$ otherwise. For example, the initial weight tuple of transition $t_8$ (Figure 3.7) is $w^0 = \langle 1, 1, 0 \rangle$ for $D = \langle \mathrm{ova}, \mathrm{pva}, \mathrm{tva} \rangle$ since only 'ova' and 'pva' occur in the enabling function. To define how a transition retains and percolates the weight, let us consider the assignments that compose its update function. All assignments to output variables are ignored since they do not affect the execution flow.

**Definition 3.2.** *Let $A$ be the set of assignments which occur in the EFSM. The function* Mod: $A \to D$ *returns the internal variable updated by the assignment $a$, i.e., the* left-hand-side *(lhs) of $a$. The function* Ref: $A \to \wp(D \cup I)$ *returns the set of internal and input variables referenced by $a$, i.e., in the* right-hand-side *(rhs) of $a$. ($\wp(S)$ denotes the powerset of the set $S$.)*

For example, said $a \in A$ the assignment '$d_3 := i_2 + d_4 - d_3$', $\mathrm{Mod}(a) = d_3$ and $\mathrm{Ref}(a) = \{i_2, d_3, d_4\}$. The next definition shows how the lhs depends on other variables. In other words, it shows how the weight percolates from the lhs back to the rhs of an assignment. The weight associated with $a$'s lhs is equally split among the variables in $a$'s rhs. By splitting weight in *equal parts*, all variables in $a$'s rhs are considered *equally relevant* in the definition of the lhs.

**Definition 3.3 (Percolation coefficient).** *Let $\Delta$: $A \to W$ and $a \in A$. $\Delta_i(a)$ denotes the $i$-th element of the weight tuple $\Delta(a)$. Then:*

$$\Delta_i(a) = \begin{cases} \frac{1}{|\mathrm{Ref}(a)|} & \text{if } d_i \in \mathrm{Ref}(a), \\ 0 & \text{otherwise.} \end{cases}$$

*($|S|$ denotes the cardinality of the set $S$.)*

Now, $\Upsilon(a)$ is being defined as the ratio of weight that, coming from $a$'s lhs, is retained by $a$. Once the weight has been equally split among variables in $a$'s rhs, $a$ retains the weight that is gone to input variables. The retention of $a$ models the dependency of $e_t$ on $a$'s lhs. Intuitively, the more weight is retained, the more inputs are in $a$'s rhs, the more the solver is able to constrain $a$'s lhs, and the higher is the likelihood that the symbolic execution of a path that passes through $a$ satisfies $e_t$.

**Definition 3.4 (Retention coefficient).** *Let $\Upsilon\colon A \to \mathbb{R}^+$ be:*

$$\Upsilon(a) = \begin{cases} 0 & \textit{if } \mathrm{Ref}(a) = \emptyset, \\ \frac{|\,\mathrm{Ref}(a) \cap I|}{|\,\mathrm{Ref}(a)|} & \textit{otherwise.} \end{cases}$$

The special case $\mathrm{Ref}(a) = \emptyset$ happens when $a$'s rhs is a literal constant expression, for example, '$d_3 := 0$'. Coefficients $\Delta$ and $\Upsilon$ are statically computed at line 2 of Algorithm 3 in time proportional to the number of internal variables, i.e., $|D|$ and to the number of assignments in the EFSM.

The previously defined coefficients are used in the next definition to compute how an assignment dynamically percolates and retains a weight tuple $w$.

**Definition 3.5.** *Let $a \in A$, $d_i = \mathrm{Mod}(a)$, $w \in W$, $w_i$ be the weight associated with $d_i$, and '$w[i/0]$' be the weight tuple $w$ where the $i$-th element has been replaced by 0. The* percolation function $P\colon A \times W \to W$ *and the* retention function $R\colon A \times W \to \mathbb{R}^+$ *are defined as:*

$$P(a, w) = w_i \cdot \Delta(a) + w[i/0],$$
$$R(a, w) = w_i \Upsilon(a).$$

The definition states that (1) the weight tuple $P(a, w)$ that percolates through the assignment $a$ is obtained by removing from the original weight $w$ the weight associated with $a$'s lhs (i.e., $w[i/0]$) and by distributing it (i.e., $w_i$) in equal parts to the internal variables that occur in $a$'s rhs (i.e., $w_i \cdot \Delta(a)$), and (2) the overall weight $R(a, w)$ retained by the assignment $a$ is the fraction of the original weight according to the retention coefficient ("it splits through input variables").

An EFSM update function is expressed as a possibly empty sequence of assignments, i.e. $\alpha \in A^*$. ($S^*$ denotes the *Kleene closure* of the set $S$, that is, the set of all finite sequences of elements of $S$). The next definition shows how update functions percolate and retain weight. It extends the previous definition to sequences of assignments. It states that the given weight tuple is first applied to the lhs of the last assignment. Then, it is iteratively percolated from the lhs back to the rhs of the current assignment and then applied to the lhs of the previous assignment. The weight retained by the sequence is the sum of weights retained at each step.

**Definition 3.6.** *Let the* percolation function over sequences $P^*\colon A^* \times W \to W$, *and the* retention function over sequences $R^*\colon A^* \times W \to \mathbb{R}^+$ *be defined by induction on the length of $\alpha \in A^*$. The empty sequence of assignments $\epsilon$ does not affect the weight propagation, i.e., $P^*(\epsilon, w) = w$, nor it retains weight, i.e., $R^*(\epsilon, w) = 0$. Let $\alpha = a :: \beta$ be the concatenation of the assignment $a$ with the sequence of assignments $\beta$. Then:*

$$P^*(\alpha, w) = P\big(a, P^*(\beta, w)\big),$$
$$R^*(\alpha, w) = R\big(a, P^*(\beta, w)\big) + R^*(\beta, w).$$

Finally, given the target transition $\bar{t}$, its associated initial weight tuple $w^0$, and the concatenation $\alpha$ of all assignments of the update functions along a path ending in $\bar{t}$, $R^*(\alpha, w^0)$ denotes the weight associated with such a path. From Definition 3.5, a *weight conservation* property can be derived: $R(a, w) + \sum_i P(a, w)_i = \sum_i w_i$. Intuitively, this means that the repeated application of $P$ in the definition of $P^*$ and the consequent repeated retention of a fraction of $w$ causes a progressive attenuation of the percolated weight $P^*(\alpha, w)$. This models the assumption that the dependence of enabling function $e_{\bar{t}}$ on assignments dims as the distance from $\bar{t}$ increases. Thus, when searching for a path for the symbolic execution, the MLBJ will select paths with higher weight first (the details are described in the following sections).

To evaluate these definitions, let us consider again the example from Figure 3.7. Let $t_8$ be the target transition. We want to show that on paths $\pi_1 = t_2{::}t_6$ and $\pi_2 = t_3$ we have $R^*(\pi_1, w^0) > R^*(\pi_2, w^0)$. Precisely, said $D = \langle \text{ova}, \text{pva}, \text{tva} \rangle$, according to Equations 3.6, the percolation along path $\pi_1 = t_2 :: t_6$ is $P^*(\pi_1, w) = \langle 0, 0, 0 \rangle$. The retained weight is $R^*(\pi_1, w) = w_{\text{ova}} + w_{\text{pva}} + w_{\text{tva}}$. The percolation along path $\pi_2 = t_3$, is $P^*(\pi_2, w) = \langle w_{\text{ova}}, 0, 0 \rangle$. The retained weight is $R^*(\pi_2, w) = w_{\text{tva}} + w_{\text{pva}}$. The initial weight tuple of $t_8$ is $w^0 = \langle 1, 1, 0 \rangle$ as its enabling function contains occurrences of 'ova' and 'pva' but not 'tva'. Thus, the retained weight of $\pi_1$ w.r.t. $t_8$ is $R^*(\pi_1, w) = w_{\text{ova}} + w_{\text{pva}} + w_{\text{tva}} = 2$. The retained weight of $\pi_2$ w.r.t $t_8$ is $R^*(\pi_2, w) = w_{\text{tva}} + w_{\text{pva}} = 1$. In such a case, MLBJ will select $\pi_1$ first.

## Snapshots of The Concrete Execution

The ability of saving the EFSM configurations allows the system to be restored during the switches between concrete and symbolic phases. This avoids the time consuming re-execution of stimuli. Algorithm 3 keeps trace of the reachability information, i.e. *RInf*, and, in particular, a cache of snapshots of the concrete execution is maintained. Each time a stimulus is added to the set of stimuli, the resulting configuration is stored in memory and explicitly linked to the reached state. The wide-width technique searches feasible paths that both start from an intermediate state of the execution and lead to the target transition. Moreover, during the MLBJ, for a given configuration and target transition, many paths are checked for feasibility, as described in the following section. Thus, caching avoids the cost of recomputing configurations for each checked path. Both the time and memory requirements of each snapshot are proportional to the size of $D$.

## Multi-level Backjumping

When the long-range concrete technique reaches the inactivity-timeout threshold, the concolic algorithm switches to the weight-oriented symbolic approach, see line 7 in Algorithm 3. Typically, some hard-to-traverse transitions, whose enabling functions involve internal variables, prevent the concrete technique going further in the exploration, as depicted in Figure 3.14. In this case, the MLBJ technique is able to selectively address paths, with high dependency on inputs, i.e., high retained weight, for symbolically executing them. Such paths are leading from an intermediate state of the execution to the target transition, thus the approach is exhaustive in a neighborhood of the corner case.

Algorithm 4 presents a description of the core of the MLBJ procedure.

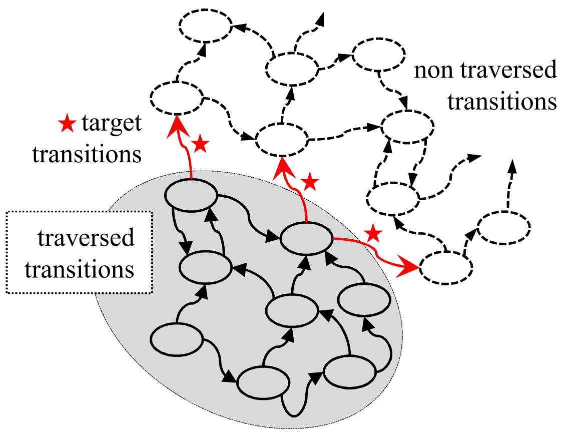


Fig. 3.14: When the approach changes from concrete to symbolic, the EFSM transitions are classified as traversed and non traversed: the target transitions (non traversed) are the frontier between these two partitions and prevent further exploration.

**Algorithm 4:** The core of the MLBJ technique.

```
   procedure MLBJ($\bar{t}$, timeout)
   input: target transition $\bar{t} \in T$, timeout
   output: stimuli for $\bar{t}$, in case empty
1  Let $w^0$ be the initial weight tuple such that
2  $\forall d_i \in D \,.\, w^0_i = \begin{cases} 1 & \text{if } d_i \text{ occurs in EnablingFunction}(\bar{t}), \\ 0 & \text{otherwise.} \end{cases}$
3  $p \leftarrow \{(\bar{t}, w^0, 0, 0)\}$;
4  while elapsed time < timeout do
5      $(t :: \pi, w, r', r) \leftarrow \text{remove\_top}(p)$
6      // $w = P^*(t :: \pi, w^0)$, $r' = R^*(t :: \pi, w^0)$,
7      // $r = R^*(\pi, w^0)$
8      if $t :: \pi$ is satisfiable then
9          if $r < r'$ then
10             foreach configuration $\langle src(t), k \rangle$ do
11                 if $k \wedge t :: \pi$ is satisfiable then
12                     return stimuli for $\bar{t}$;
13         foreach $\{t' \in T \mid dst(t') = src(t)\}$ do
14             $w' = P^*(t' :: t :: \pi, w^0)$;
15             $r'' = R^*(t' :: t :: \pi, w^0)$;
16             $\text{push}\big(p, (t' :: t :: \pi, w', r'', r')\big)$;
```

A transition $\bar{t}$ is selected in the set of target transition, and then a progressively increasing neighborhood of $\bar{t}$ is searched for paths $\pi$ leading to $\bar{t}$ and having maximal retained weight, i.e., $R^*(\pi, w^0)$. If the approach fails, another target transition is selected in the set and the procedure is repeated.

More in details, a visit is started from $\bar{t}$ that proceeds backward in the EFSM graph. The visit uses a priority queue $p$, whose elements are paths that end in $\bar{t}$. In particular, each path of $p$ is accompanied by its weight tuple, i.e., $w = P^*(\pi, w^0)$ and retained weights, i.e., $r = R^*(\pi, w^0)$ and $r' = R^*(t :: \pi, w^0)$. At the beginning, the queue $p$ contains only $\bar{t}$ and the associated initial weight $w^0$ (lines 1-2); no weight is initially retained (line 3). At each iteration, a path $t :: \pi$ with maximal retained weight is removed from $p$ (line 5). The decision procedure is used to check if the path $t :: \pi$ can be proved unsatisfiable in advance (line 8), e.g., it contains clause conflicts. In this case $t :: \pi$ is discarded so the sub-tree preceding $t :: \pi$ will not be explored. Otherwise, if the transition $t$ has yielded a positive retained weight (line 9), then for each configuration associated with the source state of $t$ the decision procedure checks the existence of a sequence of stimuli that leads to the traversal of $t :: \pi$ and thus of $\bar{t}$ (lines 10-12). In particular, the path constraint is obtained by the identified EFSM path, i.e. $t :: \pi$, and the concrete values of the internal variables, i.e., $k$, (line 11). In the case a valid sequence of stimuli has not been identified, for each transition $t'$, that precedes $t$, the path $t' :: t :: \pi$ is added to the priority queue $p$ (lines 13-16). The values of the associated weight tuple and retained weights are computed according the Definition 3.6. Notice that notation has been relaxed for readability. In particular, $P^*$ and $R^*$ functions are intended to be applied to the concatenation of transition update functions along paths $\pi$, $t :: \pi$, and $t' :: t :: \pi$.

## 3.7 Experimental results

In this section we show that even if many MDD tools already exist, the integration of a dynamic ABV environment for verifying the generated code has been an open problem. In fact, only few of them integrate natively or by means of third party toolboxes a verification environment, but, as shown in Section 3.7.1, they are limited only to specific kind of checking and do not support functional verification by means of dynamic ABV of the ESW code. For this reason, in Section 3.7.2 we thoroughly commented the characteristics of radCHECK by providing empirical results showing that the adoption of our proposed integration approach of MDD and dynamic ABV makes possible the functional verification of ESW.

In the following, Section 3.7.1 analyzes the radCASE design and synthesis environment comparing it with similar commercial tools, whereas Section 3.7.2 provides a thorough evaluation of the radCHECK verification environment.

### 3.7.1 RadCase

Table 3.2 presents a comparison between radCASE features and the most-used model-driven tools in the field of ESW design. In particular, the table reports features organized by categories. *Use Case* diagrams are used for writing high-level requirements of the ESW. *Class Diagrams*, *Composite Diagrams*, *Object Diagrams* and *Component Diagrams* are meant for describing the ESW structure. *Statecharts*, *Sequence Diagrams*, *Activity Diagrams* are meant for describing the the ESW behavior, and, in some tools, also IEC 61131-3 *FBDs* (Functional Block Diagrams), *IEC 61131-3 ST* or *C* languages can be used to describe the functionalities. The comparison shows that radCASE supports all the most used UML features and IEC 61131-3 formalisms. Another advantage of radCASE is that it supports a *HMI Graphical editor* for designing human-machine interfaces (from text display to full color video and touch screen). Besides, the reported minimum memory requirements for the target embedded system (*Minimum HW*) highlight that radCASE can be used to develop applications for embedded systems with very high hardware constraints.

A particular comment is mandatory about the Mathworks Simulink MDD framework. This framework supports proprietary formalisms, such as Stateflow, Simulink and Matlab languages, for defining the ESW specifications, but their expressivity can be considered comparable to the one of Statecharts, FBDs and C-language, respectively.

Although all the tools but one (i.e., *Poseidon embedded*) integrate a simulator with graphical capability (*Visualizer*), only radCASE, Rational Rhapsody and Simulink are supported by environments for automatic stimuli generation (i.e., radCHECK, Rational Rhapsody Automatic Test Generation [84] and Simulink Verification and Validation toolbox [138], respectively) which relieve the designers from manually putting the system into appropriate states and simulating platform and user interactions. However, through the Visualizer, they still have to analyze the simulation to identify unexpected behaviors. On the contrary, radCASE is the only tool that, supported by a dynamic ABV environment, i.e., radCHECK, allows the designers to perform functional verification of the ESW simulation in batch mode: in fact, stimuli are automatically provided to the simulator, and, at the end of the simulation, all the falsified requirements are graphically noticed to the designer. This is a useful feedback to ease and speed-up the ESW model fixing. It is worth noting that also *IAR VisualState*, *TOPCASED* and *Simulink* integrate a verification environment, i.e., *Verificator*™, Topcased OCL tooling and Simulink Verification and Validation toolbox, respectively, but they present some limitations. The first is limited to structural checks on VisualState models, such as conflicting transitions between states, unreachable states, unused variables, parameters and constants, and so forth. The second is able to check OCL [113] constraints on the UML models supported by TOPCASED. In particular, such constraints are suited for specifying invariants on classes and types in the classes, as well as pre- and post-conditions on Operations and Methods. However, the OCL language lacks of means to specify temporal constraints. The third environment, i.e., Simulink Verification and Validation toolbox, supports partially the dynamic ABV of ESW. The toolbox provides the designer with a limited number of blocks implementing restricted temporal checking: the designer has to manually combine them and graphically model the temporal behavior they want to check during simulation. It is worth noting that these temporal checks are limited w.r.t. the ones allowed by PSL assertions. Moreover, by using the checker generation engine integrated in radCHECK, assertion checkers are automatically embedded into the simulation environment for performing dynamic ABV avoiding any manual set up.

Table 3.2: Model-driven tool comparison.

| *Category* | *Feature* | radCASE | ARTiSAN studio | Enterprise Architect | Rational Rhapsody | IAR visualSTATE | CoDeSys | Poseidon embedded | Topcased | Matlab Simulink |
|---|---|---|---|---|---|---|---|---|---|---|
| Requirements | Use Case | ✓ | ✓ | ✓ | ✓ | - | - | - | ✓ | - |
| Structure | Class diagrams | ✓ | ✓ | ✓ | ✓ | - | - | ✓ | ✓ | - |
| | Composite diagrams | ✓ | ✓ | ✓ | ✓ | - | - | - | ✓ | - |
| | Object diagrams | ✓ | ✓ | ✓ | ✓ | - | ✓ | - | ✓ | - |
| | Component diagrams | ✓ | ✓ | ✓ | ✓ | - | - | - | ✓ | - |
| Behavior | Statecharts | ✓ | ✓ | ✓ | ✓ | ✓ | - | ✓ | ✓ | ✓* |
| | Sequence diagrams | ✓ | ✓ | ✓ | ✓ | - | - | - | ✓ | - |
| | Activity diagrams | ✓ | ✓ | ✓ | ✓ | - | - | ✓ | ✓ | - |
| | IEC FBDs | ✓ | - | - | - | - | ✓ | - | - | ✓* |
| | IEC ST | ✓ | - | - | - | - | ✓ | - | - | - |
| | C | ✓ | ✓ | ✓ | ✓ | - | - | ✓ | ✓ | ✓* |
| HMI | Graphical Editor | ✓ | - | - | - | - | ✓ | - | - | - |
| Minimum HW | ROM/RAM | 1kB / 100B | 32kB / 8kB | 32kB / 8kB | 32kB / 8kB | 64kb / 1kb | 128kB / 32kB | N.A. | N.A. | N.A. |
| Simulation | Stimuli generation | ✓ | - | - | ✓ | - | - | - | - | ✓ |
| | Visualizer | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | - | ✓ | ✓ |
| Verification | ABV | ✓ | - | - | - | (✓) | - | - | (✓) | (✓) |

### 3.7.2 RadCheck

radCHECK implements the dynamic-ABV approach proposed in Section 3.5 by integrating an assertion editor (i.e., PE), a checker-generation engine (i.e., CG) and a stimuli-generation engine (i.e., Ulisse). In the following, Section 3.7.2 provides a comparison of CG with other tools developed in the hardware-verification domain (since they are the only available). Section 3.7.2 provides a comparison of Ulisse with other approaches in literature. Finally, Section 3.7.2 analyzes the efficiency and effectiveness of the adopted timing reference for assertion evaluation, i.e., the model synchronization timing reference, with respect to other timing resolutions.

#### Checker generation

While the assertion templates included in radCHECK help the designers during the assertion definition, their automatic synthesis into executable checkers improve notably the quality of the simulation-based verification avoiding the manual and error prone definition of the monitors for checking the correctness of the software. In the following, the characteristics of CG have been compared with the ones of FoCs and MBAC tools. In particular, Table 3.3 reports the set of PSL operators supported by CG w.r.t. FoCs and MBAC generators.

It is worth noting that CG supports more operators w.r.t. FoCs but, unlike MBAC, it introduces some *restrictions on the SERE operators*: the “or”, the “length-matching and” and the “non-length matching and” operators cannot appear on the right-hand-side (RHS) of a suffix implication. These operators require further investigations for guaranteeing the correctness of the procedural code implementing the checker. On the contrary, MBAC seems to not suffer of this problem (but it is not possible to test it, since the MBAC is not accessible). However, it is worth noting that MBAC has been explicitly developed for supporting the generation of checker in the hardware-context, while, CG generates C-code checkers to be used in the embedded-software context.

To provide a clear idea of the synthesis capabilities of CG, the three checker generation tools have been compared on a set of interesting assertions (Table 3.4) proposed in [26, 27] (p1-p15) or manually written (p16, p17, p18). In these assertions, the identifiers “a”,“b”, “c”, etc, represent Boolean expressions whose complexity does not affect the capability of the tools to generate the checkers. Columns of the Table 3.4 report the PSL assertion definition (*PSL assertion*) and for each of the considered tools if the assertion has been correctly synthesized into a checker (✓) or not (−).

Table 3.3: Set of supported PSL operators.

| **PSL operator** | **CG** | **FoCs** | **MBAC** |
|---|---|---|---|
| **next**, **next_a**, **next_e** *family* | ✓ | ✓ | ✓ |
| **next!**, **next_a!**, **next_e!** *family* | ✓ | - | ✓ |
| **next_event** *family* | ✓ | ✓ | ✓ |
| **next_event!** *family* | ✓ | - | ✓ |
| **next_event_a** *family* | ✓ | - | ✓ |
| **next_event_a!** *family* | ✓ | - | ✓ |
| **next_event_e** *family* | ✓ | - | ✓ |
| **next_event_e!** *family* | ✓ | - | ✓ |
| **before** *family* | ✓ | ✓ | ✓ |
| **before!** *family* | ✓ | - | ✓ |
| **until** *family* | ✓ | ✓ | ✓ |
| **until!** *family* | ✓ | ✓ | ✓ |
| **eventually!** | ✓ | ✓ | ✓ |
| **always** | ✓ | ✓ | ✓ |
| logical *FL operators* | ✓ | ✓ | ✓ |
| *SERE suffix implication* | ✓ | ✓ | ✓ |
| *SERE consecutive repetition* | ✓ | ✓ | ✓ |
| *SERE non-consecutive repetition* | ✓ | ✓ | ✓ |
| *SERE goto repetition* | ✓ | ✓ | ✓ |
| *SERE or* | ✓* | ✓* | ✓ |
| *SERE non-length-matching and* | ✓* | ✓* | ✓ |
| *SERE length-matching and* | ✓* | ✓* | ✓ |
| *SERE within* | ✓ | - | ✓ |

* not supported on the RHS of the suffix implication operator

Table 3.4: Benchmarking Assertions

| **PSL assertion** | **CG** | **FoCs** | **MBAC** |
|---|---|---|---|
| p1: always(a->next(next_a[2:10](next_event(b)[10](next_e[1:5](d)) until (c)))) | ✓ | ✓ | ✓ |
| p2: always((a->next(next[10](next_event(b)((next_e[1:5](d)) until (c)))))\|\| e) | ✓ | ✓ | ✓ |
| p3: always(a->(next_event_a(b)[1:4](next((d before e) until (c))))) | ✓ | ✓ | ✓ |
| p4: always(a->next_event_e(b)[1:6](c)) | ✓ | ✓ | ✓ |
| p5: always(a->next_a![2:4](b)) | ✓ | – | ✓ |
| p6: always(a->next_e![2:4](b)) | ✓ | – | ✓ |
| p7: always(a->next_event_a!(b)[5:10](c)) | ✓ | – | ✓ |
| p8: always(a->next_event_e!(b)[2:4](c)) | ✓ | – | ✓ |
| p9: always(a->(b until! c)) | ✓ | ✓ | ✓ |
| p10: always(a->({b;c} until! d)) | ✓ | – | ✓ |
| p11: never {a;[*];b;c[+]} | ✓ | ✓ | ✓ |
| p12: always({a;d}\|->next_e[2:4](b)) until c | ✓ | – | ✓ |
| p13: always {a;b[*0:2];c}\| =>(d[*2]\|-> next(!e)) | ✓ | – | ✓ |
| p14: always (e \|\| (a->({b;c} until d))) | ✓ | – | ✓ |
| p15: always({a;b}\|->eventually! c;d) | ✓ | – | ✓ |
| p16: always({a;b;c} \|->{{d;e}\|{f;g}}) | – | – | N.A. |
| p17: always({a;b;c} \|->{{d;e}&{f;g}}) | – | – | N.A. |
| p18: always({a;b;c} \|->{{d;e}&&{f;g}}) | – | – | N.A. |

While it has been possible to directly test the synthesis results by using CG and FoCs, the MBAC results are retrieved from [27] because it is not possible to access the tool. It is worth noting that, on all the assertions reported in [26, 27], the proposed CG has behaved exactly as MBAC. Finally, let us consider assertions p16, p17 and p18 which escape the aforementioned restrictions on the PSL SERE operators. In this case, CG and FoCs has not been able to generate the checkers for them. As well, it is not possible to desume from [27] if MBAC is able to synthesize them.

## Stimuli generation for ESW

Ulisse implements the EFSM-based concolic approach for stimuli generation described in Section 3.6. Ulisse is developed in C and uses MiniSAT [58] as decision procedure. It has been tested on eight application modeled using radCASE: five applications, *Ciitp*, *Inres*, *Lift*, *Ifss*, and *Atm*, have been deduced from the specifications proposed in [92], while, the others, *Thermox*, *Filter*, and *Elevator*, have been implemented starting from the industrial specifications of the embedded applications [132]. In particular, *Ciitp* is based on a module of a class II transport protocol; *Inres* is a connection-oriented protocol controller; *Lift* is an elevator system; *Ifss* is a in-flight safety system that monitors some craft-cabinet parameters (Figure 3.7 is a very simplified version of it); *Atm* is the model of cash-point with different menu and services; *Thermox* is a controller for an industrial oven which monitors parameters like temperature, time, air humidity, and air circulation; the *Filter* controller is part of a device which produces water highly purified from salts and chemical pollution; finally, *Elevator* (different from the *Lift* module) has several functionalities, in particular the controller provides an interface both in the cabin and on the floors, moreover the speed, the acceleration, the position, the service direction, and door status are monitored.

Table 3.5: Characteristics of the designs.

| DUT | LoC | I | O | V | T | S |
|---|---|---|---|---|---|---|
| Ciitp | 324 | 55 | 274 | 80 | 28 | 6 |
| Inres | 158 | 41 | 4 | 32 | 21 | 4 |
| Lift | 297 | 82 | 16 | 48 | 30 | 4 |
| Ifss | 282 | 77 | 6 | 65 | 36 | 3 |
| Atm | 282 | 168 | 48 | 80 | 42 | 10 |
| Thermox | 142 | 58 | 8 | 156 | 168 | 2 |
| Filter | 256 | 66 | 32 | 293 | 237 | 21 |
| Elevator | 3391 | 8 | 32 | 5037 | 775 | 382 |

The characteristics of the considered benchmarks are described in Table 3.5. Column *LoC* shows the number of lines of the C code generated by radCASE; columns *I*, *O*, and *V* respectively show the bit size of the inputs, outputs, and internal variables; columns *T* and *S* show the number of EFSM transitions and states. Benchmarks are ordered according to column *V*, since the number of variables is significant in the case of decision-procedure-based (symbolic) approaches.

In what follows two experiments are proposed. The first one (Table 3.6) compares Ulisse with alternative model-based techniques for stimuli generation. The second experiment (Table 3.7) compares Ulisse with KLEE [34], which works at bytecode level.

Table 3.6: Transition-coverage comparisons.

| | **Rand** | | **Cons** | | **Symb** | | **Ulisse** | |
|---|---|---|---|---|---|---|---|---|
| **DUT** | **TC%** | **Time** | **TC%** | **Time** | **TC%** | **Time** | **TC%** | **Time** |
| Ciitp | 96.1 | 6.8 | 96.1 | 18.7 | 100.0 | 180.3 | 100.0 | 23.8 |
| Inres | 89.4 | 24.1 | 92.7 | 18.4 | 100.0 | 186.8 | 100.0 | 24.0 |
| Lift | 11.1 | 0.7 | 81.4 | 63.5 | 100.0 | 1224.5 | 100.0 | 13.9 |
| Ifss | 32.3 | 25.6 | 55.8 | 59.1 | 100.0 | 1024.1 | 100.0 | 12.7 |
| Atm | 12.5 | 0.2 | 71.1 | 76.0 | 100.0 | 934.1 | 100.0 | 51.0 |
| Thermox | 33.8 | 8.0 | 47.0 | 170.5 | 47.0 | 3402.3 | 60.2 | 37.6 |
| Filter | 62.8 | 0.8 | 62.8 | 2.7 | 68.5 | 3503.1 | 100.0 | 30.5 |
| Elevator | 69.3 | 30.2 | 69.3 | 190.9 | 1.4 | 4914.1 | 81.5 | 4459.6 |

*Time* is expressed in seconds.

Table 3.6 compares Ulisse with a pure random approach (Rand), a constraint-based heuristic (Cons) [51], and a pure symbolic approach (Symb). The Cons approach focuses on the traversal of just one transition at a time by submitting each enabling function formula to a constraint solver. Even if Cons uses a solver for guiding the stimuli generation, it can be considered a long-range concrete technique. On the contrary, the Symb extends a deep-first-search procedure to EFSMs [127] and is intended to be an exhaustive and wide-width approach.

The table reports the maximum achieved transition coverage (TC%) and the execution time in seconds (Time). Each experiment has been carried out with a time threshold of 5000 seconds, and the reported execution time refers to the instant the engine achieved the last improvement in transition coverage. It is worth noting that:

- for designs with a smaller number of internal variables and transitions, Ulisse always outperforms Rand and Cons in terms of transition coverage; while in cases Symb achieves the same coverage as Ulisse, Symb is one or two orders of magnitude slower;
- for designs with a larger number of internal variables and transitions, Ulisse outperforms both Cons and Symb in terms of transition coverage; in particular, the coverage achieved by Symb is poor because of the high number of decision paths and the complexity of the decision problems, which would require an execution time higher than the selected time threshold.

Figure 3.15 presents the trend of the four approaches on the *Filter* and *Atm* testcases, in terms of transition coverage versus stimuli generation time. For readability the transition coverage progress is depicted up to 6 seconds. Notice that Rand and Cons very rapidly increase the coverage, but they achieve steady values, as they are unable to cover corner cases. On the contrary, Symb slowly increases the

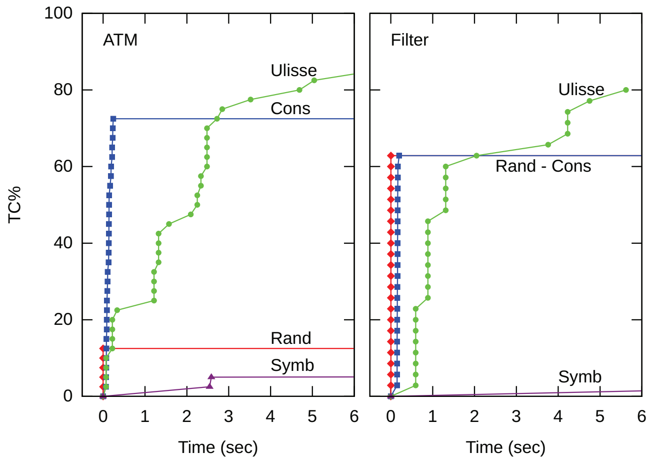


Fig. 3.15: The transition coverage versus the stimuli generation time for the *Atm* and *Filter* designs.

coverage because of both high execution time required by the decision procedure and non corner-case-oriented nature. Indeed, SYMB tends to waste a lot of time trying to compute irrelevant stimuli data, i.e., stimuli exercising no new transitions. On the contrary, Ulisse spends a certain amount of time for the initial dependency analysis (Section 3.6). Then, the achieved coverage presents two different trends. It very rapidly increases as the long-range concrete approach is executed. As soon as it is no more able to traverse new transitions, the wide-width search (based on the *MLBJ* algorithm proposed in Section 3.6) is started. It is slower, because it requires more time for solving path constraints, but it allows to exit from steady conditions achieving a higher transition coverage.

Table 3.7: Instruction-coverage comparisons.

| **DUT** | **KLEE-IC%** | **Ulisse-IC%** |
|---|---|---|
| Thermox | 23.15 | 54.30 |
| Filter | 69.18 | 94.74 |
| Elevator | 30.48 | 74.61 |

The second experiment (Table 3.7), considers the three designs where the pure symbolic model-based approach *Symb* has not achieved 100% transition coverage, and compares the quality of the stimuli sequences generated by Ulisse and KLEE, which symbolically analyzes the LLVM byte-code generated from the C specification.

In particular, the KLEE infrastructure is used for comparing the instruction coverage on the LLVM byte-code. At first, KLEE has symbolically executed the C code with a maximum execution time set to 5000 seconds and its instruction coverage is collected. Then, KLEE has executed the C code using the stimuli sequences generated by Ulisse considering the same maximum execution time (i.e., 5000 seconds).

Table 3.7 reports the results of such experiment. Columns *KLEE-IC%* shows the KLEE instruction coverage; column *Ulisse-IC%* shows the instruction coverage achieved by simulating the stimuli sequences generated by Ulisse with KLEE.

Notice that KLEE, which works at bytecode level and implements an deep-first-search approach, has not been able to reach the same instruction coverage achieved by Ulisse, which exploits the model information to explore more efficiently the system state space.

## Dynamic ABV for ESW

This final set of experiments validates the adopted timing reference for assertion evaluation. For the purpose, some industrial ESW for control and automation systems have been developed with radCASE. Table 3.8 reports their characteristics.

Table 3.8: Case-studies characteristics.

| **Design** | **Specs** | **Modules** | **LoC** | **Asserts** |
|---|---:|---:|---:|---:|
| DSC | 25 | 93 | 26000 | 211 |
| Tridomix | 12 | 18 | 10782 | 163 |
| Desal | 10 | 28 | 5319 | 113 |
| HV | 7 | 18 | 2626 | 88 |
| Sewing | 40 | 164 | 120638 | 258 |

Column *Design* reports the design name; *Specs* is the number of pages in the specification and requirement document (in natural language); *Modules* is the number of the Statecharts modules in the radCASE model; *LoC* is line of code of the synthesized ESW; finally, *Asserts* is the number of assertions which engineers defined for the ESW verification.

Table 3.9 compares the efficiency and effectiveness of the proposed dynamic ABV for ESW with respect to different timing references. The efficiency is measured in terms of execution time of the ESW running with checkers, whereas the effectiveness of the adopted timing reference is measured by counting the number of false negatives and false positives which arise during the checker simulations. A false negative (positive) occurs when the checker indicates that an assertion does not hold (hold) when it really does (does not).

In particular, columns *Design* and *Asserts* report the design name and number of associated assertions, respectively; column *MSync* reports the results of the proposed approach, which evaluates checkers according to the model synchronization timing reference; *Trans* refers to the transition timing reference; finally, *Instr* refers to instruction timing reference, as described in [103]. For each approach, *FN/P* is the number of false negatives or false positives which occur, and *Time* is the execution time in seconds.

Table 3.9: ABV efficiency and effectiveness.

| | | **MSync** | | **Trans** | | **Instr** | |
|---|---|---|---|---|---|---|---|
| **Design** | **Asserts** | **FN/P** | **Time** | **FN/P** | **Time** | **FN/P** | **Time** |
| DSC | 211 | 0 | 50 | 27 | 2088 | 184 | 99105 |
| Tridomix | 163 | 0 | 36 | 42 | 327 | 139 | 11759 |
| Desal | 113 | 0 | 17 | 16 | 337 | 102 | 13294 |
| HV | 88 | 0 | 11 | 29 | 16.8 | 72 | 2358 |
| Sewing | 258 | 0 | 161 | 52 | 4577 | 213 | 226146 |

*Time* is expressed in seconds.

As expected, since all designs meet the specifications, the use of the model synchronization timing reference does not lead to false negatives/positives. On the contrary, false negatives/positives occur adopting the other timing references: typically this is due to assertions that predicate over variables before a stable condition is reached (e.g. Figure 3.5). In particular, the number of false negatives/positives is greater when assertions are evaluated according to the instruction timing reference. Whereas, in case of transition timing reference, only assertions whose variables span over different modules may be affected by false negatives/positives.

Finally, increasing the checker-evaluation rate significantly impacts the overall execution time, as shown in columns *Time*.

## 3.8 Conclusions

This chapter has described how a suitable combination of MDD and dynamic ABV is an effective solution for ESW development. On one hand, a particular adoption of MDD-based techniques exceeds both the problem of manual defining the ESW C-code and practical issues concerning integration of dynamic ABV in ESW design. In fact, MDD approaches allow to identify the right instants for checkers evaluation, as well as they provide support for handling of variable visibility inside ESW and complex data-types inside assertion checkers. Besides, also automatic generation of effective stimuli for simulation becomes practical. On the other hand, the adoption of ABV-based techniques exceeds the problem of manually debugging the ESW C-code. In particular, dynamic ABV approaches allow to refine the ESW model iteratively and incrementally. Indeed, each assertion falsification identified by the corresponding checker implies a violation of a specific ESW requirement, thus, it is a useful information for guiding the model fixing. Besides, by adopting a guided assertion definition strategy based on property patterns (i.e., parametric assertion templates) and aided by graphical tools, no effort is required to teach the designers new languages for specifying assertions.

For these reasons, a comprehensive design and verification framework for ESW has been proposed. It integrates radCASE, which is a model-driven design, code generation and simulation environment, and radCHECK, which is a dynamic ABV environment that includes guided assertion definition, automatic generation of checkers, and stimuli generation.

# 4

# Vacuity analysis for assertion qualification in dynamic ABV

## 4.1 Introduction

Assertion qualification aims at evaluating the quality of assertions defined to check the correctness of a design implementation, either hardware or software. It focuses on measuring assertion coverage [140], identifying redundant assertions [30], searching for vacuous assertions [16], etc. In particular, the presence of vacuous assertions (i.e., assertions that are trivially satisfied) in assertion-based verification (ABV), in both static [42] and dynamic [66] contexts, can lead designers to a false sense of safety because the design implementation could be erroneous even if all the defined assertions are satisfied.

For this reason vacuity analysis is a mandatory process that looks for formulas[1] that pass vacuously and points out problems which require a refinement process in the design under verification (DUV), or in its specification (i.e., in the assertions themselves), or, in case of dynamic ABV, in the stimuli sequences (Figure 4.1).

Current approaches for vacuity checking are suited for static ABV, where temporal assertions are formally checked against the DUV model. Basically they exploit formal methods to search for *interesting witnesses* proving that assertions do not pass vacuously [16, 98]. Such approaches are computationally as complex as model checking, and they require to define and model check further assertions obtained from the original ones by substituting their sub-formulas in some way, sensibly increasing the verification time. Moreover, these approaches are not able to identify vacuity when it is due to the presence of tautologies. Only in [10] authors propose a technique to identify vacuity caused by errors in the DUV model as well as by tautologies, but the designer has to analyze the produced vacuity alerts to decide which should be ignored.

This chapter, instead, describes an alternative methodology for vacuity checking that extends the current approaches and makes vacuity detection feasible also in dynamic ABV, where assertion checkers[2], synthesized from temporal assertions, are used to check the DUV behaviors simulated using a set of (automatically generated) stimuli.

[1] By standard conventions, the term formula is used to identify a temporal assertion.

[2] The notion of checker and its structure have been summarized in Section 3.2.2 and Section 3.5.2, respectively. We will recall them in Section 4.5

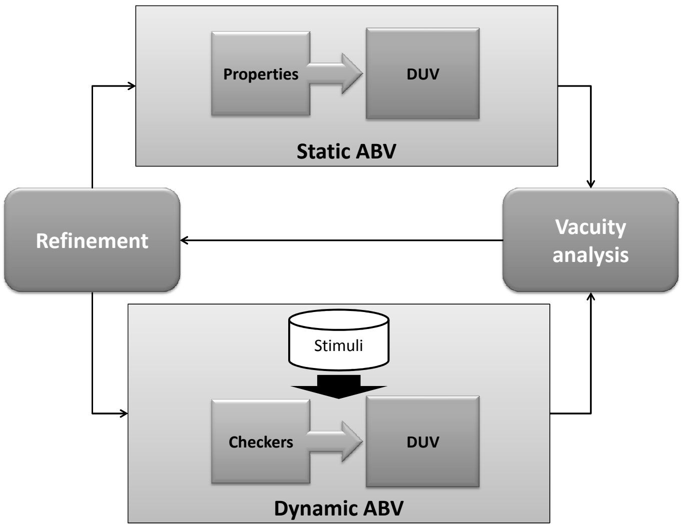


Fig. 4.1: Vacuity detection in model checking and ABV.

The proposed methodology is based on *mutation analysis* [117]. Usually, mutation analysis relies on the perturbation of a DUV by introducing new statements or modifying existing ones in small ways. As a consequence, many versions of the model are created, each containing one mutation and representing a *mutant* of the original DUV. Stimuli sequences are used to simulate mutants with the goal of distinguishing their outputs from the original DUV ones. In fact, the presence of not-distinguished mutants points out inadequacies in the stimuli sequences or in the DUV model. Thus, the main purpose of mutation analysis consists of helping the verification engineers to develop effective stimuli able to activate all DUV sections.

On the contrary, we use mutation analysis with the purpose of detecting vacuous assertions: instead of perturbing the DUV, we perturb the assertion checkers by a small set of mutations, named *interesting faults*. These interesting faults enable us to create mutants of the original checker which have to be distinguished during simulation. Stimuli sequences enabling to distinguish such mutants represent the interesting witnesses proving that the assertions do not pass vacuously. If the interesting witnesses are not identified, the methodology provides an alert to force the designers to investigate the stimuli sequences, the assertions and the DUV model to understand which is the cause of the alert.

### 4.1.1 Contributions

The main contributions of this work can be summarized as follows:

- it redefines the notion of vacuity checking in such a way it can be applied in the context of dynamic assertion-based verification of temporal assertions defined according to the *Simple Subset* of PSL [87];

- it proposes a mutation-based approach for vacuity checking that does not require the definition and the verification of *witness formulas* to generate the *interesting witnesses*, reducing the computational complexity of the analysis;
- it extends the current vacuity checking methodologies to identify tautological vacuity in a more accurate way than [10];
- it implements the proposed methodology into radCHECK;
- it shows the applicability of the proposed methodology in the context of dynamic ABV for both embedded software and electronic system level (ESL) models.

The remind of this chapter is organized as follows. Section 4.2 summarizes related works. Section 4.3 reports preliminary definitions. Section 4.4 presents an overview of the proposed vacuity analysis methodology. Section 4.5 is devoted to checker structural analysis and describes how subformulas of a temporal assertion can be identified in different parts of a checker. Section 4.6 defines the theoretical basis of the proposed vacuity analysis. Section 4.7 describes the problem of detecting tautological vacuity. Section 4.8 deals with prons and cons of the proposed methodology. Finally, Section 4.9 and Section 4.10 report, respectively, experimental results and concluding remarks.

## 4.2 State of the art

Vacuity analysis is a mandatory process looking for assertions that, passing vacuously, lead designers to a false sense of safety.

According to Beer et al. [16], a formula $\varphi$ passes vacuously in a model $M$ if it passes in $M$, and there is a sub-formula $\psi$ of $\varphi$ that can be changed arbitrarily without affecting the outcome of model checking.

Automatic techniques to detect trivial passes are proposed in [10,13,16,19,39,40,73,98] and, generally, define and model check new assertions obtained from the original ones by substituting their sub-formulas in some way. Comparisons and discussions about such techniques have been proposed in [41,97,124].

The work in [98] proposes a methodology for vacuity detection applicable to CTL* formulas. The authors argue that vacuity detection consists of checking whether all the sub-formulas of $\varphi$ affect its truth value in the system. This is obtained by replacing each sub-formula $\psi$ by either **true** or **false** depending on whether $\psi$ occurs in $\varphi$ with negative polarity (i.e., under an odd number of negations) or positive polarity (i.e., under an even number of negation). This method is for vacuity with respect to sub-formula occurrences. While a sub-formula occurrence has a *pure* polarity (exclusively negative or positive), a sub-formula with several occurrences may have *mixed* polarity (both positive and negative occurrences). It is shown that the method is no longer valid by considering sub-formulas instead of sub-formulas occurrences.

In [16], Beer et al. prove a result similar to the one in [98], that holds for all logics with polarity. First, the authors show that vacuity can be checked by examining only a subset of the sub-formulas, i.e., sub-formulas which are minimal with respect of the subformula pre-order. Secondly, the authors define logics with polarity for which it is enough to check the replacement of a sub-formula by either

**true** or **false**. These two results allow authors to check vacuity of a formula in a logic with polarity by defining and checking a relatively small number of other formulas, whose size is not greater than the size of $\varphi$. However, the authors show a practical solution only for a subset of ACTL, and they check vacuity only with respect to the smallest important sub-formula.

In [10], the aim of the authors is to remove the restriction of [16,98] concerning sub-formula occurrences of pure polarity. To keep things simple, the authors stick to LTL. Strategies presented in [16, 98] consider only syntactic perturbation on a formula $\varphi$, instead, in [10] the authors consider the notion of semantic perturbation which is modeled by universal quantifier (i.e., UQLTL). Such a semantic can be interpreted with respect either to the model (structure semantics) or to its set of computation (trace semantics). While the authors argue that structure semantics yields notions of vacuity that are computationally intractable, they argue that, instead, trace semantics is considered intuitive, robust and it is shown that it can be checked easily by a model checker. According to these semantics, the authors give new definitions of vacuity which do not restrict a sub-formula to occur once and it can be of mixed polarity.

In [73] the authors present the notion of mutual vacuity that is a kind of vacuity with respect to literal occurrences. Mutual vacuity focuses on finding the maximal set of literal occurrences that can be simultaneously replaced by **false** without causing the property to fail. In this way, the authors want to identify all sub-formulas of a given CTL formula that cause its vacuity, or better, identify all maximal such sub-formulas. Moreover, they propose an exponential-time multi-valued model checking algorithm to detect mutual vacuity.

In [40] Chockler et al. propose a definition of vacuity very similar to the one of [73]. The authors underlined that checking for mutual vacuity can lead to stronger properties than those obtained by vacuity checks with respect to sub-formula occurrences, but it is also harder because it has to consider subset of literals.

In [19], Ben-David et al. underline that vacuity detection for certain logics can be complex and time consuming and indicate that not all vacuities detected in practical applications are considered a problem by the system verifier. As a consequence, the authors limit their analysis to the problem of detecting antecedent failure. They define Temporal Antecedent Failure, a refinement of vacuity that occurs when some pre-condition in the formula is never fulfilled in the model.

The work in [13] defines vacuity in the context of testing. In particular, the authors distinguish strong and weak vacuity. Strong vacuity coincides with the common definition of vacuity in model checking (e.g., the one proposed in [98]) and points out that some behavior that the specifier expect cannot happen. Weak vacuity depends on test sets and it suggests either there is strong vacuity (missing behaviors) or more tests are needed. Although the authors state that the proposed vacuity analysis approach can lead to better specifications and test suits, no experimental results are provided.

In [39], the authors examine the possibility of finding a property stronger than the original LTL formula by either vacuity or mutual vacuity, that is sill satisfied by the model. In particular, the approach determines vacuity values over paths of the model in order to compute the strongest formula that is satisfied by the model and lies in the Boolean closure of the strengthening of the original property. Even if the approach finds formulas stronger than mutual vacuity does, its complexity seems impractical for large formulas.

## 4.3 Background

The following definitions and theorems report relevant results related to the notion of vacuity according to the work proposed by Beer et al. in [16]. The same concepts are exploited in the majority of the other papers summarized in Section 4.2, and they are fundamental to better understand the methodology proposed in this chapter.

In the following, the term formula denotes a temporal assertion $\varphi$, and the notation $\varphi[\psi \leftarrow \psi']$ denotes the formula obtained from $\varphi$ by replacing the sub-formula $\psi$ with $\psi'$.

**Definition 1 (Affect)** *A sub-formula $\psi$ of a formula $\varphi$ affects $\varphi$ in a model $M$ if there is a formula $\psi'$ such that the truth values of $\varphi$ and $\varphi[\psi \leftarrow \psi']$ are different in $M$.*

**Definition 2 (Vacuous pass)** *A formula $\varphi$ passes vacuously in a model $M$ if $M \models \varphi$ and $\varphi$ includes a sub-formula $\psi$ that does not affect $\varphi$ in $M$. In this case, we say that $\varphi$ is $\psi$-vacuous in $M$.*

While the previous definitions capture the intuitive notion of vacuity, they are not very useful from the practical point of view, because they require an infinite number of checking to determine the vacuity of a formula. For this reason, in [16] the following definitions and theorems have proposed to make the analysis feasible.

**Definition 3 (Minimal sub-formulas)** *Let $S$ be a set of sub-formulas. The minimal sub-formulas of $S$ is defined as:*

$$min(S) = \{\psi \in S \mid \nexists\ \psi' \in S \text{ such that } \psi' \leq \psi\},$$

*where $\leq$ denotes the pre-order relation among sub-formulas. That is $\psi' \leq \psi$ means $\psi'$ is a sub-formula of $\psi$.*

Definition 3 assumes that each sub-formula is unique, i.e., two separate occurrences of the same sub-formula are considered to be different sub-formulas. For example, let us consider the formula $\varphi = G(\neg\alpha \vee \neg\beta \vee X(\gamma \vee \neg\alpha))$. The set of sub-formulas of $\varphi$ is $S = \{\neg\alpha_1 \vee \neg\beta_1 \vee X(\gamma_1 \vee \neg\alpha_2), \neg\alpha_1 \vee \neg\beta_1, X(\gamma_1 \vee \neg\alpha_2), \neg\alpha_1, \neg\beta_1, \gamma_1 \vee \neg\alpha_2, \alpha_1, \beta_1, \gamma_1, \neg\alpha_2, \alpha_2\}$, where the subscript denotes the occurrence number of each sub-formula in $\varphi$ (in particular, $\alpha_1$ and $\alpha_2$ denote two distinct occurrences of $\alpha$, since $\alpha$ occurs twice in $\varphi$), and thus the minimal sub-formulas of $S$ are $min(S) = \{\alpha_1, \beta_1, \gamma_1, \alpha_2\}$.

**Definition 4 (Vacuity w.r.t. a set of sub-formulas)** *Let $M \vDash \varphi$ and $S$ be the set of sub-formulas of $\varphi$. Formula $\varphi$ is $S$-vacuous in $M$ if there exists $\psi \in S$ such that $\varphi$ is $\psi$-vacuous in $M$.*

**Theorem 4.1.** *Let $S$ be the set of sub-formulas of $\varphi$, true in model $M$. Formula $\varphi$ is $S$-vacuous iff $\varphi$ is $min(S)$-vacuous.*

**Theorem 4.2.** *In a logic with polarity*[3]*, for a formula $\varphi$ , and a set $S$ of sub-formulas of $\varphi$, for every model $M$, $\varphi$ is $S$-vacuous in $M$ iff there is $\psi \in min(S)$ such that $M \models \varphi[\psi \leftarrow X]$, where $X =$* **false** *if $M \models \varphi$ ($M \not\models \varphi$) and $\psi$ is of positive (negative) polarity*[4]*, otherwise, $X =$* **true**.

The full proofs of these theorems are available in [16]. While Theorem 4.1 states that to check vacuity of $\varphi$ it is enough to check for vacuity with respect to only the minimal sub-formulas of $\varphi$, Theorem 4.2 allows the authors to focus only on model checking $\varphi$ where the minimal sub-formulas are substituted with either **true** or **false**, thus reducing the total number of checks required to search for vacuity. The formulas obtained by such substitutions are called *witness formulas* for $\varphi$. As a consequence of Theorem 4.2, $\varphi$ is not $\psi$-vacuous if $M \not\models \varphi[\psi \leftarrow X]$, and in such a case the counterexample provided by the model checker is the desired *interesting witness* proving the non-vacuity of $\varphi$ with respect to $\psi$.

## 4.4 Methodology overview

All the techniques summarized in Section 4.2 check the vacuity of a formula $\varphi$ by using formal methods. This work, instead, considers the notion of vacuity specified by Definition 2 and presents an alternative strategy to reason about it to make practical vacuity checking in the context of dynamic ABV.

The proposed methodology is based on mutation analysis and works as follows (Figure 4.2):

1. Given the set of formulas satisfied during the dynamic ABV of the DUV, the corresponding assertion checkers are collected and *interesting faults* are injected on them. Each interesting fault is implemented as a *stuck-at 0/1* fault on a checker variable modeling an assertion minimal sub-formula. In particular, for each assertion $\varphi$, we identify how sub-formulas must be perturbed, i.e., if a stuck-at fault injected in the corresponding checker has to affect one or many minimal sub-formulas of $\varphi$. This allows us to accurately address the vacuity alerts. Intuitively, an interesting fault perturbs the checker behavior similarly to what happen when a sub-formula $\psi$ is substituted by **true** or **false** in $\varphi$.
2. The faulty checkers are integrated into the simulation environment. Then, stimuli sequences are used to simulate the DUV. The vacuity analysis relies on the observation of the faulty checkers status. A checker failure due to the effect of an interesting fault $f$ corresponds to prove that the sub-formulas $\psi$ associated to (perturbed by) $f$ affect the truth value of $\varphi$. Consequently, the specific stimuli sequence that causes the checker failure (i.e., the stimuli sequence of $f$) is an interesting witness proving that $\varphi$ is not $\psi$-vacuous (Section 4.6). On the contrary, faults that do not cause checker failures (i.e., not detected

[3] For a definition of logics with polarity please refer to [16].

[4] Intuitively, in a logic with polarity, a formula has positive polarity if it is preceded by an even number of not operators, has negative polarity otherwise [16].

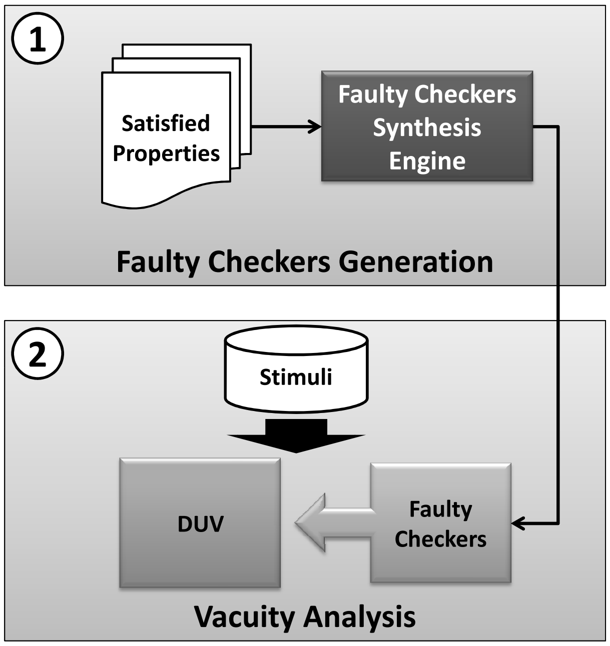


Fig. 4.2: Methodology overview.

faults) have to be analyzed to determine if either the property is vacuous or the vacuity alert generated is due to the inefficiency of the stimuli sequences used during the simulation (Section 4.8).

Notice that the proposed methodology works on LTL formulas defined according to the *Simple Subset* of PSL, that conforms to the notion of monotonic advancement of time. This is a mandatory restriction that guarantees that formulas defined within such a subset can be automatically translated into executable checkers and simulated.

## 4.5 Checker analysis

As reported also in Chapter 3, both academic and industrial tools are available to automatically generate checkers from temporal assertions. For example, radCHECK, MBAC and FoCs allow a designer to convert a PSL formula into either C-code or HDL-code implementing a state machine which will enter an error state in a simulation run if the corresponding assertion is falsified.

By analyzing the structure of a checker corresponding to a formula $\varphi$ it is possible to identify which parts of the checker are related to the minimal sub-formulas of $\varphi$. For example, let us consider the following formula:

$$\varphi = always((state = START) \rightarrow next[2](state = FETCH1))$$

By construction, checkers generated using radCHECK separates the temporal semantics of the assertion from the involved minimal Boolean sub-formulas in

```
1  void get_var(
2      struct System const* sys, char const* name,
3      void* var, long unsigned var_sizeof);
4
5  #define GET_VAR(T,N,S) \
6    T N; get_var(sys, S, &N, sizeof(T));
7
8  void checker_wrapper(bool const enable, bool const reset,
9                       bool* fail, struct System const* sys){
10
11   // 1. Retrieve system-variable values
12   GET_VAR(int8_t, design_State,"design.State");
13
14   // 2. Evaluate minimal sub-formulas
15   bool v0 = (design_State == /* Start */ 0);
16   bool v1 = (design_State == /* Fetch1 */ 1);
17
18   // 3. Invoke the checker over Boolean variables
19   checker_logic(enable, reset, fail, v0, v1);
20 }
```

Fig. 4.3: The C-code wrapper function of the checker for assertion $\varphi$.

different portions of C-code, i.e., the *checker logic* and the *checker wrapper* functions, respectively (Section 3.5.2). In particular, the checkers template adopted by radCHECK models minimal sub-formulas using satellites and explicitly reports them into the checker wrapper function. Figure 4.3 shows, as an example, the checker wrapper function for the assertion $\varphi$. The minimal sub-formulas (i.e., $state = START$ and $state = FETCH1$) are reported at lines 19 – 20.

Now, let us consider a VHDL checker corresponding to $\varphi$ generated, for example, by FoCs. Its structure is shown in Figure 4.4. The template used by FoCs for implementing checkers distinguishes the code modeling the minimal sub-formulas from the temporal semantics of the checker as follows. The truth values of minimal sub-formulas of $\varphi$ are captured by concurrent assignments introduced at the beginning of the checker architecture (lines 20 – 21), whereas, the temporal behavior of the checker is implemented by two synchronous processes, i.e., $p1$ and $pp1$. The first process (lines 23 – 31) handles the signal **focs_ok_checker_psl_1** which stores in each time instant $t$ (depending on the clock period) the truth value of the formula $\varphi$. According to the values reported into the signal **focs_v_checker_psl_1** (line 26) the process evaluates the truth value of the temporal assertion and *asserts* its violation when **focs_ok_checker_psl_1** is equal to *false* (line 32 – 36). The second process (lines 38 – 55) manages the temporal aspects of the formula. It sets the values in **focs_v_checker_psl_1** by evaluating the property sub-formulas during the DUV simulation. Such a signal is represented as a vector, whose length depends on the temporal characteristics of the formula. For example, the truth value of a sub-formula of $\varphi$ may be stored at time instant $t$ as first element of the vector, and in the following time instants $t + i$ with $i > 0$ it moves forward, position by position, modeling the elapsing of time. At time instant $t + i$, such a value can be retrieved at the $i$-th position of the vector and can be used to establish the satisfaction or the falsification of the formula $\varphi$.

As shown in Figure 4.3 and Figure 4.4, the identification of variables/signals associated to minimal sub-formulas is straightforward and, thus, the injection of

```
library ieee;
use ieee.std_logic_1164.all;
use work.sim_support_downto.all;
ENTITY checkers_psl IS
    PORT (
      clock         : IN std_logic;
      reset         : IN std_logic;
      state         : IN std_logic_vector(1 DOWNTO 0));
END checkers_psl ;

ARCHITECTURE checker OF checkers_psl IS
    CONSTANT start                  : std_logic_vector(1 DOWNTO 0) := "00";
    CONSTANT fetch1                 : std_logic_vector(1 DOWNTO 0) := "01";
    SIGNAL   state_start            : std_logic;
    SIGNAL   state_fetch1           : std_logic;
    SIGNAL   focs_ok_checkers_psl_1  : std_logic;
    SIGNAL   focs_v_checkers_psl_1   : std_logic_vector(3 DOWNTO 0);

 BEGIN
    state_start <= b2l((state(1 DOWNTO 0) = start(1 DOWNTO 0)));
    state_fetch1 <= b2l((state(1 DOWNTO 0) = fetch1(1 DOWNTO 0)));

  p1: PROCESS (clock)
     BEGIN
        IF ((clock =  '1')) THEN
           focs_ok_checkers_psl_1 <= NOT((focs_v_checkers_psl_1(3)
                                                AND NOT(state_fetch1)));
        ELSE
           focs_ok_checkers_psl_1 <= '1';
        END IF;
  END PROCESS p1;
  ASSERT (NOT(( focs_ok_checkers_psl_1 = '0')))
      REPORT "FAILURE EVENT rule: CHECKERS_PSL,
            formula: 1 in file checkers_psl.vhd: Assertion Failed;
            vunit: checkers_psl property 1"
      SEVERITY NOTE;

 pp1: PROCESS (clock)
        VARIABLE focs_vout_checkers_psl_1 : std_logic_vector(3 DOWNTO 0);
     BEGIN
        IF (l2b(reset)) THEN
           focs_v_checkers_psl_1(3 DOWNTO 0) <= "0011";
        ELSIF (clock'EVENT AND clock = '1') THEN
           focs_vout_checkers_psl_1(3 DOWNTO 0) :=
              reverse((((( focs_v_checkers_psl_1(0) AND  '1')) &
              (( focs_v_checkers_psl_1(1) AND state_start))) &
              (( focs_v_checkers_psl_1(2) AND  '1'))) &
              (( focs_v_checkers_psl_1(3) AND NOT(state_fetch1)))));
           focs_v_checkers_psl_1(3 DOWNTO 0) <=
              reverse(((( focs_vout_checkers_psl_1(0)) &
              (focs_vout_checkers_psl_1(0))) &
              (focs_vout_checkers_psl_1(1))) &
              (focs_vout_checkers_psl_1(2))));
        END IF;
 END PROCESS pp1;
END checker;
```

Fig. 4.4: A VHDL checker for $always((state = START) \rightarrow next[2](state = FETCH1))$.

interesting faults into the checkers can be automatized as described in the next section. It is worth noting that the idea of storing (and perturbing) the values of sub-formulas in explicit variables assigned by the checker statements can be exploited whatever is the structure of the checker. Thus, templates adopted by automatic tools to model the checkers do not affect the proposed methodology.

## 4.6 Mutation-based vacuity analysis

Let us consider the effect of faults on variables (or signals) storing the values of minimal sub-formulas into the checker. Faults are implemented as saboteurs that stuck at **true** or **false** the variable corresponding to a sub-formula $\psi$ of $\varphi$ inside the checker. We say that such a fault is detectable according to the following definition.

**Definition 5 (Detectable fault)** *A fault $f$, injected in the checker $C$ corresponding to a formula $\varphi$ satisfied in model $M$, is detectable if there exists a sequence of values (test sequence) that, once assigned to the primary inputs of $M$, causes the output of the faulty checker to become* **false**, *while the output of the fault-free checker remains* **true**.

According to the previous definition, and considering $\varphi = G(\neg\alpha \vee X\beta)$ as an example, if $\alpha$ never happens, independently from the value of $\beta$, the output of the corresponding checker will remain always **true**. Thus, a stuck-at **false** on the variable storing the value of sub-formula $\beta$ cannot be detected, since it cannot affect the result of the faulty checker because $\alpha$ never happens. On the contrary, if $\alpha$ eventually happens, the faulty checker will eventually become **false** due to the effect of the injected fault. In this case, the fault is detected, and the related stimuli sequence represents an *interesting witness* proving that the formula does not pass vacuously. Analogous considerations can be done for a stuck-at **true** on the variable storing the value of sub-formula $\alpha$.

Thus, injecting a fault of type stuck-at **false** (**true**) on the variable storing the value of sub-formula $\beta$ ($\alpha$) provides us the same result obtained by model checking $\varphi[\beta \leftarrow \mathbf{false}]$ ($\varphi[\alpha \leftarrow \mathbf{true}]$).

The previous considerations can be formalized by defining a correspondence between sub-formulas and interesting faults. In particular, we distinguish minimal sub-formulas as:

- *contemporaneous*: occurrences of the same sub-formula that affect the *same* temporal instants;
- *opposite*: contemporaneous occurrences of opposite polarity that are in the *same* clause.

A fault of type stuck-at **true**/**false** on a sub-formula perturbs not only the specific occurrence, but all the ones that are opposite. On the contrary, non-opposite occurrences are not perturbed.

The definitions of contemporaneous and opposite sub-formulas are formalized in Section 4.7 and they allow us to opportunely address identification of tautologies and increase the accuracy of vacuity alerts.

**Definition 6 (Interesting faults)** *Let $\varphi$ be a formula expressed by using a logic with polarity, $C$ the corresponding checker, $S$ the set of minimal sub-formulas of $\varphi$,* *and $A$ the set of assignments of $C$ setting the values of sub-formulas in $S$ during simulation. The set of interesting faults for $C$ is defined as follows:*

$$F = \{\textit{stuck-at } X \textit{ on } a | a \in A\}$$

*where $X =$ **false** if the sub-formula associated to assignment $a$ has positive polarity, $X =$ **true** otherwise.*

From the previous considerations the following theorems follow.

**Theorem 4.3.** *Let $\psi$ be a sub-formula of $\varphi$ expressed by using a logic with polarity. The checker of $\varphi[\psi \leftarrow X]$ (where $X$ can be either **true** or **false** depending on the polarity of $\psi$) is equivalent to the checker of $\varphi$ where the variable storing the value of $\psi$ is stuck at $X$.*

**Proof:** Let $C$ be a checker for $\varphi$ and $s_\psi$ be the variable of $C$ storing the truth value of the sub-formula $\psi$ of $\varphi$, as shown in Section 4.5. Ab absurdo, let us suppose that the checker $C'$, which captures by construction the behavior of $\varphi[\psi \leftarrow X]$, is not equivalent to $C$ in which the variable $s_\psi$ is stack at $X$.

When $s_\psi$ is stack at $X$ in $C$, the structure of $C$ becomes syntactically equal to the one of $C'$ modeling $\varphi[\psi \leftarrow X]$. Thus, they always behave in the same way. But, by assumptions, $C$ and $C'$ are not equivalent, so their behaviors must differ. Contradiction. □

**Theorem 4.4.** *Let $\varphi$ be a formula expressed by using a logic with polarity, $C$ the corresponding checker, $S$ the set of minimal sub-formulas of $\varphi$, and $F$ the set of interesting faults of $C$ associated to $S$. The fault $f \in F$, such that $f$ is associated to sub-formula $\psi \in S$ as defined in Definition 6, is detectable, iff $\varphi$ is not $\psi$-vacuous.*

**Proof:** Let us assume that $f$ of kind stuck at $X$ (where $X$ can be either **true** or **false** depending on the polarity of the associated sub-formula) is detectable. For Definition 5 there exists a stimuli sequence that causes a failure on the checker $C$ perturbed by $f$ once it is applied to the primary inputs of the model. Thus, for Theorem 4.3 the same sequence causes a failure also on the checker $C'$ corresponding to $\varphi[\psi \leftarrow X]$. However, if a checker fails, the corresponding formula cannot be satisfied in the model, by construction of the checker. From this observation and from Definition 1 and Definition 2 it derives that $\varphi$ is not $\psi$-vacuous.

On the contrary, let us assume that $\varphi$ is not $\psi$-vacuous. From Theorem 4.2 this means that $\varphi[\psi \leftarrow X]$ is not satisfied in the model, and thus a counterexample of $\varphi[\psi \leftarrow X]$ can be generated by model checking. However, the checker $C'$ corresponding to $\varphi[\psi \leftarrow X]$ must fail, by construction of the checker, when such a counterexample is simulated in the DUV model. Therefore, from Theorem 4.3, it derives that the checker $C$ of $\varphi$ perturbed by $f$ also fails when it is stimulated by the same counterexample. Thus, from Definition 5 $f$ is detectable. □

The previous theorem allows us to reason about vacuity by fault simulating interesting faults in the checker of the formula instead of model checking its witness formulas as proposed in [16]. Since the number of interesting faults to be considered amounts to $|min(S)|$ (it coincides with the number of witness formulas to be checked according to [16]), the approach results to be efficient in determining that a property *does not pass vacuously.*

```
 1 void faulty_checker_wrapper(bool const enable, bool const reset,
 2                             bool* fail, struct System const* sys,
 3                             int8_t fault_id) {
 4
 5   // 1. Retrieve system-variable values
 6   GET_VAR(int8_t, design_State,"design.State");
 7
 8   // 2. Retrieve sub-formulas polarity
 9   bool fault_v0 = GET_POLARITY("v0");
10   bool fault_v1 = GET_POLARITY("v1");
11
12   // 3. Evaluate minimal sub-formulas
13   bool v0 = fault_id == 1 ? fault_v0 : (design_State == /* Start */ 0);
14   bool v1 = fault_id == 2 ? fault_v1 : (design_State == /* Fetch1 */ 1);
15
16   // 4. Invoke the checker over Boolean variables
17   checker_logic(enable, reset, fail, v0, v1);
18 }
```

Fig. 4.5: The C-code wrapper function of the faulty checker for the property $always((state = START) \rightarrow next[2](state = FETCH1))$.

Moreover, the injection of interesting faults can be performed in a systematic way. For example, the choice made in Section 3.5.2 of modeling minimal sub-formulas using satellites makes easy the injection of interesting faults. In particular, in radCHECK, a *faulty checker* definition is automatically synthesized by generating the *faulty checker wrapper function* shown in Figure 4.5. It is clear that such a function is obtained by applying few modifications to the standard checker wrappers generator which manipulates the data dump containing the temporal assertions defined by means of the PE (i.e., Property Editor, Section 3.5.1). In particular, unlike the standard wrapper function, the faulty checker wrapper function is characterized by:

- a *fault_id* parameter (line 3). This parameter is used to activate the interesting faults injected into the faulty checker. Each fault is identified by an integer ID number. They are activated sequentially, one at a time, to check the non-vacuity of the assertion with respect to a specific minimal sub-formula. The *fail* parameter, in this case, notifies a violation of the property in presence of an interesting fault;
- a set of automatic variables $\{ \ fault\_v0, \ldots, fault\_vN \ \}$ (lines $9 - 10$). These variables model the interesting faults associated to the different minimal sub-formulas $\{ \ v0, \ \ldots, \ vN \ \}$ of the assertion. The function $GET\_POLARITY()$ elaborates the assertion structure reported into the data dump generated by the PE to establish the polarity of each sub-formula in $\{ \ v0, \ \ldots, \ vN \ \}$. Intuitively, a sub-formula has positive polarity if it is preceded by an even number of not operators, has negative polarity otherwise.
- a set of conditional expression statements for applying the interesting faults to minimal sub-formulas (lines $13 - 14$). These expressions are used to activate the effect of an interesting fault only when required.

## 4.7 Tautological vacuity

From Definition 2 and Theorem 4.2, a formula $\varphi$ is considered to pass vacuously if there exists a sub-formula $\psi$ that does not affect $\varphi$, i.e., if we can indifferently change the value of $\psi$ without affecting the value of $\varphi$. According to such a notion, tautologies are, indeed, not marked as vacuous formulas when two separate occurrences of the same sub-formula are considered to be different sub-formulas. Let us consider, for example, the formula $(\alpha \vee \neg\alpha)$. According to Theorem 4.2, the formula is non-vacuous, since the replacement of each occurrence of $\alpha$ with either **true** or **false**, depending on its polarity, leads to the following formulas, which do not have the same evaluation of $(\alpha \vee \neg\alpha)$:

- $(false \vee \neg\alpha)$, which is $\neg\alpha$;
- $(\alpha \vee \neg true)$, which is $\alpha$.

However, $(\alpha \vee \neg\alpha)$ is commonly considered vacuous, indeed. This problem has been highlighted also in paper [10] by Armoni et al. The solution proposed in [10] to correctly address $(\alpha \vee \neg\alpha)$ consists of considering *sub-formulas* instead of *occurrences* of sub-formulas. In this case, the two occurrences of $\alpha$, substituted by either **true** or **false**, lead to the following formulas:

- $(true \vee \neg true)$, which has the same evaluation of $(\alpha \ \vee \ \neg\alpha)$;
- $(false \ \vee \ \neg false)$, which still have the same evaluation of $(\alpha \ \vee \ \neg\alpha)$.

By considering sub-formulas instead of occurrences, $(\alpha \vee \neg\alpha)$ would be correctly marked as vacuous. However, the same authors, in [10], show that considering sub-formulas instead of occurrences of sub-formulas may lead to wrongly mark, as non-vacuous, formulas that are vacuous. For example, let us consider the formula $(\alpha \wedge G(\neg\beta \vee \alpha))$ and assume that $\beta$ never occurs. Then the formula is vacuous. Such a formula will be incorrectly marked as non-vacuous using the sub-formulas substitution:

- $(false \wedge G(\neg\beta \vee false))$, which is $false$;
- $(\alpha \wedge G(\neg true \vee \alpha))$, which is $\alpha \wedge G(\alpha)$ and it can be falsified.

Thus, they conclude by saying that "a thorough vacuity detection algorithm should detect both sub-formulas and occurrences that do not affect the examined formula. It is up to the user to decide which vacuity alerts to ignore". Therefore, considering the notion of vacuity with respect to sub-formulas or occurrences of sub-formulas is still an open problem. In this section, we propose an approach that addresses the problems highlighted in [10] to perform vacuity analysis adopting both sub-formulas and occurrences substitutions.

In the following we describe that the substitution of a sub-formula $\psi$ by true or false influences either many occurrences or a single occurrence of $\psi$ depending on clauses and the temporal scope of such occurrences. In particular, *opposite* occurrences are substituted simultaneously. On the contrary *non-opposite* occurrences are substituted individually.

**Definition 7 (Contemporaneous occurrences)** *Let $\psi_1$ and $\psi_2$ two occurrences of the same sub-formula $\psi$. $\psi_1$ and $\psi_2$ are contemporaneous if they are under the scope of the same temporal operators.*

**Definition 8 (Opposite occurrences)** *Let $\psi_1$ and $\psi_2$ two contemporaneous occurrences of $\psi$ in the same clause. $\psi_1$ and $\psi_2$ are opposite if they have opposite polarities.*

Notice that to correctly apply Definition 7 and Definition 8 to a formula $\varphi$, it is mandatory to solve the temporal overlapping between the occurrences of the same sub-formula into a clause. To do this it is possible to use the unwinding laws for the temporal operators $G$ (i.e., always), $F$ (i.e., eventually) and $U$ (i.e., until):

- $G(\alpha) \equiv \alpha \wedge X(G(\alpha))$;
- $F(\alpha) \equiv \alpha \vee X(G(\alpha))$;
- $\alpha\ U\ \beta\ \equiv \beta \vee (\alpha \wedge X(\alpha\ U\ \beta))$;

Due to the fact that the unwinding is performed only within clauses and the simple subset of PSL restricts one operand of the logical "or" to be a Boolean, the unwinding requires only a finite number of steps (one for each overlapping occurrence).

For example, let us consider the formula $\varphi = G(\neg\alpha \vee F(\alpha))$. The set of sub-formulas of $\varphi$ is $S = \{\neg\alpha_1, \alpha_1, \alpha_2\}$ where the subscript denotes the occurrence number of each sub-formula in $\varphi$. Due to the fact that the time interval of $\alpha_2$ overlaps the one of $\alpha_1$ (and it does not coincide with it), we have to introduce a new occurrence $\alpha_3$ to resolve the temporal overlap obtaining the following formula: $\varphi = G(\neg\alpha \vee \alpha \vee X(F(\alpha)))$. Now $S = \{\neg\alpha_1, \alpha_1, \alpha_3, \alpha_2\}$ and the minimal sub-formulas of $S$ are $min(S) = \{\alpha_1, \alpha_3, \alpha_2\}$. Notice that $\alpha$ occurs three times in $\varphi$ and its occurrences are all in the same clause but in the scope of different temporal operators. In particular $\alpha_1$ and $\alpha_3$ are opposite, they have the same temporal scope (i.e., $G$), while $\alpha_3$ has a scope (i.e., $GXF$) that is different from the previous one.

For a formula $\varphi$ and a subformula $\psi$ of $\varphi$, let $\varphi[\psi \leftarrow \bot]$ denote the formula obtained from $\varphi$ by replacing all opposite occurrences of $\psi$ by **false**. Dually, $\varphi[\psi \leftarrow \top]$ replaces the opposite occurrences by **true**.

Considering Definition 8 and Definition 1, the following theorem proposed by Kupferman et al. in [98] is still valid and it allows us to detect tautological vacuity.

**Theorem 4.5.** *Let $\varphi$ a formula in CNF form within the simple subset of PSL. For a minimal sub-formula $\psi$ of $\varphi$, let us distinguish the* opposite occurrences *of $\psi$ using the unwinding rules. Now, for every system $M$, if $M \models \varphi[\psi \leftarrow \bot]$, then for every formula $\xi$, we have $M \models \varphi[\psi \leftarrow \xi]$. In addition, if $M \not\models \varphi[\psi \leftarrow \top]$, then for every formula $\xi$, we have $M \not\models \varphi[\psi \leftarrow \xi]$.*

**Proof:** We prove the implications

$$\varphi[\psi \leftarrow \bot] \rightarrow \varphi[\psi \leftarrow \xi], \text{ and } \varphi[\psi \leftarrow \xi] \rightarrow \varphi[\psi \leftarrow \top]$$

together, by induction on the structure of $\varphi$.

- If $\varphi = \psi = p$, and $p$ is an atomic proposition, then for all $\xi$, we have $\varphi[\psi \leftarrow \bot] =$ **false**, $\varphi[\psi \leftarrow \xi] = \xi$ and $\varphi[\psi \leftarrow \top] =$ **true**. Hence, as for all $\xi$, we have **false** $\rightarrow \xi \rightarrow$ **true**, we are done.
- If $\varphi = \psi \vee \neg\psi$, and $\psi = p$. Notice that $\psi$ occurs one time with positive polarity and one time with negative polarity. Since the occurrences are contemporaneous

and belong to the same clause, they are opposite. Thus, they should be replaced simultaneously. So, for all $\xi$, we have
$\varphi[\psi \leftarrow \bot] = false \vee \neg false = \mathbf{true}$, $\varphi[\psi \leftarrow \xi] = \xi \vee \neg\xi = \mathbf{true}$ and $\varphi[\psi \leftarrow \top] = true \vee \neg true = \mathbf{true}$. We are done.

- Consider the case $\varphi = f(\theta, \theta')$, with $f \in \{\wedge, \vee\}$. Notice that $\varphi$ is into conjunctive normal form. By the semantics of LTL, the operator $f$ is *positively monotonic*, in the sense that for every $\theta_1$ and $\theta_2$, with $\theta_1 \rightarrow \theta_2$, for all $\theta'$ we have $f(\theta_1, \theta') \rightarrow f(\theta_2, \theta')$ and $f(\theta', \theta_1) \rightarrow f(\theta', \theta_2)$. Now since for all $\theta$ and $\theta'$, the polarities of sub-formulas of $\theta$ are equal to their polarities in $f(\theta, \theta')$ and $f(\theta', \theta)$, the claim follows immediately form the induction hypothesis.
- Consider the case $\varphi = g(\lambda)$ or $\varphi = h(\lambda, \lambda')$ with $g \in \{X\}$ and $h \in \{U\}$ and $\varphi$ is into CNF. By the semantics of LTL, the operators $g$ and $h$ are *positively monotonic*, in the sense that for every $\lambda_1$ and $\lambda_2$, with $\lambda_1 \rightarrow \lambda_2$, we have $g(\lambda_1) \rightarrow g(\lambda_2)$, and for all $\lambda'$ we have $h(\lambda_1, \lambda') \rightarrow h(\lambda_2, \lambda')$ and $h(\lambda', \lambda_1) \rightarrow h(\lambda', \lambda_2)$. So for all $\lambda$ and $\lambda'$, the polarities of subformulas of $\lambda$ are equal to their polarities in $g(\lambda)$, $h(\lambda, \lambda')$ and $h(\lambda', \lambda)$. Then, the claim follows from the induction hypothesis.

□

Now, let us consider properties $\varphi_1 = (\alpha \vee \neg\alpha)$ and $\varphi_2 = (\alpha \wedge G(\neg\beta \vee \alpha))$. By the adoption of the Definition 8 and Theorem 4.5 it is possible to correctly analyze formulas $\varphi_1$ and $\varphi_2$. In particular, considering $\varphi_1$, the set of its minimal sub-formulas is $min(S) = \{\alpha_1, \alpha_2\}$, and $\alpha_1$, $\alpha_2$ are opposite sub-formulas. It is worth noting that, the substitution of false on a sub-formula should influence all opposite sub-formulas simultaneously. So we have to check $\varphi_1[\alpha_{1,2} \leftarrow false]$. Performing the substitution and considering the presence of opposite sub-formulas, we obtain:

- $\varphi_1[\alpha_{1,2} \leftarrow false] = false \vee \neg false = \mathbf{true}$

So, the substitution underlines that property $\varphi_1$ is always true, thus it is vacuous and, in particular, it is a tautology. Notice that tautologies cannot be identified in a syntactic way. Indeed, the only things we identify syntactically are sub-formulas polarity and temporal scope of occurrences.

Now let us consider property $\varphi_2$ satisfied by the model and let us assume that $\beta$ never occurs. The set of its minimal sub-formulas is $min(S) = \{\alpha_1, \beta_1, \alpha_2\}$, but none of the minimal sub-formulas are opposite. According to sub-formulas polarity, we perform the following substitution:

- $\varphi_2[\alpha_1 \leftarrow false] = (false \wedge G(\neg\beta \vee \alpha)) = false$
- $\varphi_2[\beta_1 \leftarrow true] = (\alpha \wedge G(false \vee \alpha)) = (\alpha \wedge G(\alpha))$
- $\varphi_2[\alpha_2 \leftarrow false] = (\alpha \wedge G(\neg\beta \vee false)) = (\alpha \wedge G(\neg\beta))$

Due to the fact that $\varphi_2$ is satisfied by the model and $\beta$ never occurs, $\varphi_2[\alpha_2 \leftarrow false] = \alpha$ is satisfied, thus the property $\varphi_2$ is marked as vacuous. Such a vacuous pass is not revealed by adopting the approach proposed in [10].

## 4.8 Pros and cons

There are two reasons that may cause a formula to pass vacuously when it is model checked: (i) an error in the model of the DUV that does not implement correctly the specification and (ii) an error in the formula itself that does not capture the real intent of the specification. The same reasons may prevent also the detection of interesting faults. However, missing to detect a fault may be also due to the inefficiency of the stimuli sequences used during fault simulation. It happens when there exists a test sequence (interesting witness) for detecting the fault, but the stimuli sequences do not include it. In this case, the vacuity of a formula is due to the lack of exhaustiveness of the adopted stimuli, rather than a real problem on either the formula or the DUV model. Thus, from the practical point of view, according to Theorem 4.4, we can conclude that a formula $\varphi$ is not $\psi$-vacuous, when stimuli used to simulate the system (composed of the DUV and the faulty checker) detect the interesting fault associated to the sub-formula $\psi$ of $\varphi$. On the contrary, if stimuli sequences are not able to detect the fault, we cannot conclude that $\varphi$ is $\psi$-vacuous in the meaning commonly adopted in the model checking context, since it could depend on the incapability of generating the right stimuli sequence. In this case, an ineffective stimuli sequence may lead designers to mark a formula as vacuous even if it is not. Thus, the described methodology provides a semi-decidable strategy. On the other hand, in dynamic ABV, it is absolutely reasonable stating that a formula passes vacuously when the defined stimuli sequences cannot generate the interesting witness related to an interesting fault. In fact, stimuli sequences are a fundamental ingredient and they must be as effective as possible to guarantee an high-quality verification. Thus, refining stimuli sequences that allow a formula to pass vacuously is an objective. In this context, the proposed vacuity analysis is very satisfactory because in presence of low-quality stimuli sequences, it alerts the designers to search the cause that prevents the detection of a interesting fault associated to a particular sub-formula leading in investigating also simulation stimuli.

## 4.9 Experimental results

In what follows two set of experiments are proposed. The first set (Section 4.9.1) applies the proposed vacuity analysis methodology on assertions verified using radCHECK on eight designs modeled with radCASE. The second set (Section 4.9.2), instead, applies the methodology on assertions used to check the correctness of seven ESL designs within a dynamic ABV environment implemented in SystemC. In particular, on such designs and assertions, the proposed methodology for vacuity analysis is compared with the formal vacuity checking approach described in [16] in terms of effectiveness and efficiency.

### 4.9.1 Vacuity analysis in dynamic ABV of model-driven embedded software

In this section we considered eight application developed using radCASE and verified using radCHECK (Chapter 3), and we applied the proposed vacuity analysis approach to identify the presence of potential vacuously satisfied assertions.

In particular, the *Ciitp* application implements a module of a class II transport protocol; the *Inres* application implements a connection-oriented protocol controller; the *Lift* application models an elevator system; the *Ifss* application implements an in-flight safety system that monitors some craft-cabinet parameters; the *Atm* application is the model of cash-point with different menu and services; the *Thermox* application implements a controller for an industrial oven which monitors parameters like temperature, time, air humidity, and air circulation; the *Filter* application models the control software of a device which produces water highly purified from salts and chemical pollution; finally, the *Elevator* application (that is different from *Lift*) implements a controller which handles elevator requests from both the cabin and the floors, and monitors the speed, the acceleration, the position, the service direction, and door status of the elevator.

The formal assertions used to verify such applications have been defined using radCHECK and in particular, for the applications *Ciitp*, *Inres*, *Lift*, *Ifss*, and *Atm*, they have been deduced from the specifications proposed in [92], while, for the others, i.e., *Thermox*, *Filter*, and *Elevator*, assertions have been defined starting from the industrial specifications of the embedded applications [132].

Table 4.1: Characteristics of the designs and vacuity analysis results.

| | | *Stimuli* | | *Vacuity analysis* | | |
|---|---|---|---|---|---|---|
| **DUV** | **LoC** | **T** | **TC%** | **Assertions** | **Vacuous** | **Time (sec)** |
| Ciitp | 324 | 28 | 100.0% | 20 | 0 | 34.192 |
| Inres | 158 | 21 | 100.0% | 10 | 0 | 22.64 |
| Lift | 297 | 30 | 100.0% | 12 | 0 | 28.904 |
| Ifss | 282 | 36 | 100.0% | 12 | 0 | 30.23 |
| Atm | 282 | 42 | 100.0% | 10 | 0 | 29.812 |
| Thermox | 142 | 168 | 60.2% | 18 | 9 | 21.984 |
| Filter | 256 | 237 | 100.0% | 19 | 2 | 27.897 |
| Elevator | 3391 | 775 | 81.5% | 30 | 11 | 356.72 |

The characteristics of the considered benchmarks, the quality of the stimuli sequences used for the dynamic verification and the number of verified assertions are shown in Table 4.1. Column *DUV* reports the name of the design under verification; column *LoC* shows the number of lines of the C-code implementation generated by radCASE; column *T* and *TC%* show the number of EFSM transitions extracted by Ulisse (Section 3.6) from the designs model and the percentage of transition covered by the stimuli sequences generated by its stimuli generation engine (more information are reported in Section 3.7.2). Such stimuli sequences have been used for the vacuity analysis phase. Besides, column *Assertions* reports the number of formal assertions that, satisfied by the applications, need to be checked for possible vacuous satisfactions. The right-most columns of Table 4.1, instead, report, information related to the vacuity analysis results. In particular, *Vacuous* reports the total number of vacuous assertions identified by our methodology, and *Time* denotes the total execution time (in seconds) required for the vacuity analysis. It is worth noting that such time does not include the time required to generate stimuli sequences, because they have been generated for the dynamic ABV phase that preceded the vacuity analysis phase. Moreover, the *faulty* checkers required for the vacuity analysis have been automatically generated using radCHECK (Section 4.6).

As is shown by the table, only three designs present vacuously satisfied formulas. In particular, for *Termox* and *Elevator* applications, the presence of vacuously satisfied assertions has been due to the partial coverage of the stimuli sequences. In fact, in all the identified cases of vacuity, the assertions predicate over variables configurations (i.e., specific value assignments on variables) that have not been achieved by using the available stimuli sequences. In particular, the assertions activation conditions have not been satisfied causing the assertions to be trivially valid on the models. Such vacuity alerts have been solved by manually creating the stimuli sequences using the radCASE simulator. By interacting with the simulator, we have led the application to generate the desired configurations for activating the assertions and, then, proving their non-vacuity.

For the *Filter* application, instead, the methodology has identified real errors in two assertions definitions. These assertions are of the form $\varphi = always(\alpha \rightarrow next(\beta \rightarrow next(\gamma)))$, i.e., they specify that every time that a variables configuration $\alpha$ occurs during the application execution, this may be followed immediately by a configuration $\beta$, and, if this is the case, the configuration $\gamma$ has to occur immediately after. By analyzing the model and also the informal specifications of the application in [132], we have realized that while $\beta$ may eventually occur, it is not required to occur immediately after $\alpha$. For this reason, both the sub-formulas $\alpha$ and $\gamma$ have not affected $\varphi$ during the vacuity analysis and then $\varphi$ has been correctly marked as vacuous. As a consequence, we have rewritten the two assertions as $\varphi = always(\alpha \rightarrow eventually!(\beta \rightarrow next(\gamma)))$ and they have been satisfied non-vacuously by the *Filter* application.

### 4.9.2 Vacuity analysis in dynamic ABV of ESL designs

In this case, the proposed methodology has been evaluated by using some of the well-known ITC-99 benchmarks [118], the control unit of an 8-bit *CPU* with an instruction set architecture composed of 13 instructions and a real industrial case provided by STMicroelectronics that is a *square-root* calculator included into a face recognition module. Their characteristics are described in Table 4.2. Assertions for ITC-99 benchmarks have been taken from the *vis-verilog-models-1.2* package [142] provided to formal verify the correctness of the considered benchmarks with VIS model checker [142]. Assertions for the *square-root* model have been provided with its implementation. On the contrary, assertions for the *CPU* model have been defined by analyzing its specifications. The dynamic ABV environment has been modeled using SystemC [85] language. Assertion checkers have been generated by using FoCs, while interesting faults have been manually injected into the checkers. Stimuli sequences have been generated by using Laerte++ [62], an automatic test pattern generator for SystemC designs. The achieved results have been compared with the formal vacuity analysis performed according to the approach described in [16] by using VIS to model check the assertions and the witness formulas.

Columns of Table 4.2 report, respectively, design names ($DUV$), the number of primary inputs ($PIs$), primary outputs ($POs$), gates ($Gates$), flip flops ($FF$) and analyzed assertions ($\Psi$). Table 4.2 also reports information related to the

Table 4.2: Vacuity analysis results.

| | | | | | | Fault simulation | | | | Formal technique [16] | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **DUV** | **PIs** | **POs** | **Gates** | **FF** | $\Psi$ | **IF** | **DF** | **SS** | **Time (sec.)** | **WF** | **UWF** | **Time (sec.)** | **Sav.** |
| b02 | 3 | 1 | 71 | 4 | 3 | 6 | 6 | 50 | 0.040 | 6 | 6 | 0.140 | 71% |
| b04 | 13 | 8 | 1815 | 66 | 8 | 23 | 23 | 320 | 0.144 | 23 | 23 | 6.452 | 98% |
| b06 | 4 | 6 | 135 | 8 | 3 | 3 | 3 | 30 | 0.020 | 3 | 3 | 0.140 | 86% |
| b09 | 3 | 1 | 491 | 28 | 2 | 6 | 6 | 44 | 0.048 | 6 | 6 | 0.156 | 69% |
| b10 | 13 | 6 | 560 | 17 | 3 | 13 | 13 | 340 | 0.112 | 13 | 13 | 0.492 | 77% |
| cpu | 10 | 2 | 52 | 10 | 8 | 20 | 16 | 480 | 0.227 | 20 | 17 | 0.313 | 27% |
| sqrt | 66 | 64 | 6383 | 163 | 10 | 32 | 32 | 120 | 1.632 | 32 | 32 | 31.564 | 94% |

vacuity analysis methodology proposed in this chapter (*Fault simulation*), i.e., the number of injected interesting faults (*IF*), detected interesting faults (*DF*), stimuli sequences (*SS*), and the time required for automatic stimuli generation and fault simulation of interesting faults (*time*). On the contrary, the right-most columns report, information related to the vacuity analysis performed according to the technique proposed in [16] (*Formal technique*), i.e., the number of witness formulas (*WF*) to be model checked, the number of witness formulas that fail (*UWF*), and the time required for reasoning about vacuity by model checking the witness formula (*time*). Finally, column *Sav.* reports the percentage of computational time saved by using fault simulation of interesting faults instead of model checking witness formulas.

Both approaches have shown that the assertions included in the *vis-verilog-models-1.2* archive for *ITC-99* benchmarks and the ones provided with the *square-root* module do not pass vacuously, since all interesting faults have been detected and all witness formulas have been proved to be false. On the contrary, while our methodology has generated vacuity alerts for two assertions $\varphi_1$, $\varphi_2$ defined for the control unit of the *CPU* (16 *DF* vs. 20 *IF*), the formal approach has produced an alert only for $\varphi_1$ (17 *UWF* vs. 20 *WF*).

In particular, both methodologies have underlined that the vacuity of the property $\varphi_1$ has been due to an error in the property definition. $\varphi_1$ is of the form $\varphi_1 = \alpha \wedge \beta \rightarrow \mathit{eventually}!(\gamma)$, i.e., it specifies that an occurrence of the variables configurations $\alpha$ and $\beta$ at the same time, requires the configuration $\gamma$ to occur eventually. Notice that according to the assertion formalization, $\alpha$ and $\beta$ are expected to occur at the first step of the simulation (or, in the context of formal verification, in the first state of the trace). On the contrary, in such a step (state, resp.), these configurations cannot occur according to the design implementation and, thus, $\varphi_1$ has turned out to be vacuous (in particular, neither $\alpha$, or $\beta$ or $\gamma$ affects $\varphi_1$). To be consistent with the informal specifications, the assertion should have been defined as $\varphi_1 = \mathit{always}(\alpha \wedge \beta \rightarrow \mathit{eventually}!(\gamma))$, i.e., every time that the configurations $\alpha$ and $\beta$ occur at the same time, then the configuration $\gamma$ has to occur eventually. Indeed, such a new defined assertion has been classified as non-vacuous by both our dynamic approach and the formal one in [16].

As regards $\varphi_2$, the assertion is of the form $\varphi_2 = \alpha \rightarrow \mathit{eventually}!(\alpha)$, i.e., it is a temporal tautology. The vacuity alert generated by our vacuity analysis approach has detected that such an assertion has been vacuously satisfied (a discussion on such a property has been proposed in Section 4.7). On the contrary the methodology proposed in [16] marked the tautological property as non-vacuous due to the fact that opposite occurrences are not substituted simultaneously.

Regarding the comparison of computation times, Table 4.2 highlights that our approach is, in average, 74% faster than the one proposed in [16]. This is achieved by detecting the interesting witnesses to the non-vacuity of assertions using fault simulation instead of formally verifying witness formulas. As a drawback, our methodology depends strongly on the quality of the stimuli sequences. The more they are exhaustive, the more the proposed vacuity analysis approach is effective and efficient w.r.t. the current existing formal approaches.

## 4.10 Conclusions

In this chapter a mutation-based methodology for vacuity analysis suited for dynamic ABV has been proposed. Current vacuity checking strategies in literature are suited only for static ABV and look for interesting witnesses to the non-vacuity of assertions by verifying witness formulas. On the contrary the proposed approach establishes the vacuity of a formula $\varphi$ by checking, during simulation, the status of checkers perturbed by faults associated to the sub-formulas of $\varphi$. The stimuli sequences which detect such faults represent the interesting witnesses.

Moreover, by proposing the definition of opposite occurrences of a sub-formula (Definition 8), the approach allows us to detect also tautological vacuity and improve the accuracy of vacuity alerts with respect to [10, 16].

As regards the experimental results, we have proved that the proposed vacuity analysis approach can be effectively applied in the context of dynamic ABV for model-driven embedded software as well as ESL designs. In the first case, the methodology has allowed us to detect assertions trivially satisfied due to both errors in assertions definition and low-quality stimuli sequences. In particular, results have highlighted the need of improving the quality of stimuli to avoid erroneous vacuity alerts. In fact, the proposed approach depends strongly on stimuli sequences and the more they are exhaustive, the more the proposed approach is effective and efficient. In the second case, experiments have shown that the proposed approach is more effective than the one presented in [16], and, in average, 74% faster. Such a speed-up is guaranteed by the use of simulation to rapidly identify non-vacuous properties. It is worth noting that the experimental results provided in Section 4.9.2 are mainly based on academic benchmarks. This is due to the fact that retrieving real-industrial cases accompanied with the set of properties used to check their correctness is not easy. Indeed, only one of the papers related to vacuity we cited in our work reports experimental results. The complexity of such benchmarks and properties seem comparable with ours.

# 5 Conclusions and future works

This thesis has proposed a complete methodology and tools that combined together return a framework to derive from hybrid automata specifications proved correct in the hybrid domain, correct realizable models of embedded controllers and the related discrete implementations. The need of such a framework is mainly due to the fact that formal verification results performed using the traditional semantics of hybrid automata rely on un-implementable assumptions which cannot be used in practice for deriving a correct hardware/software implementation in a systematic way. For example, one of these assumptions is the synchrony hypothesis, i.e., the capability of performing any computation in zero time units and forcing a change in the dynamics of the hybrid system without delays, that is clearly an unrealistic assumption.

To overcome this problem, we have identified that several steps are required to obtain a correct implementation of a hybrid automaton-based model $M$ of a hybrid system.

At first, given the model $M$, it is mandatory to retrieve, if it exists, a realizable model $M_{\mathcal{R}}$ which includes a controller that is *implementable* and is still able to handle the surrounding physical environment satisfying all the safety properties $\varphi_{sp}$ used to check the correctness of $M$ in the hybrid domain. In other words, this step aims at automating the synthesis of an $AASAP$ or *lazy* control strategy for $M$ which takes into account the digital and imprecise aspects of the hardware device on which the actual control strategy is being executed. The goal of this synthesis consists of establishing the *performance bounds* that any conservative concrete hardware/software device implementing the embedded controller has to satisfy. As a consequence, if a realizable model $M_{\mathcal{R}}$ is identified, then its control strategy can be translated into an embedded software that has to be executed on a hardware device satisfying the synthesized performance bounds. In this context, our main contributions can be summarized as follows:

- We developed tools and a synthesis procedure which make *practical* the applicability of the Almost-ASAP semantics for synthesizing implementable control strategies for relevant classes of hybrid automata. In particular, such tools manipulates CIF descriptions for generating the $M_{\mathcal{R}}$ model (i.e., *s-extract*) and translating it in suitable formalisms required for the formal verification (i.e., *cif2phaver* and *cif2ariadne*). The synthesis procedure, instead, identifies the performance bounds which enable the relaxed control strategy to satisfy its safety properties. Notice that this procedure is model checker independent, and, thus, different model checkers can be used according to the complexity of the hybrid model dynamics to analyze.

- We defined a new methodology, supported by tools, for the synthesis of implementable control strategies for the interesting class of *lazy linear hybrid automata*. In particular, we defined a symbolic encoding of the set of reachable states of a LLHA that reduces the synthesis of implementable control strategies for LLHA to the reachability problem on LLHA, that is decidable. In fact, by verifying the safety properties as reachability queries, the proposed representation makes possible to establish if there exists a lazy control strategy able to handle the continuous plant by following finite-precision and discrete-time behaviors. Such a representation is automatically generated by combining our *cif2uclid* tool and the UCLID modeling environment. Then, our proposed synthesis procedure can exploit different SMT solvers to identify the performance bounds which enable the lazy control strategy to satisfy its safety properties.

The following step consists of modeling the identified implementable control strategy specified by $M_{\mathcal{R}}$ as an embedded software. Instead of manually defining the code implementation, we proposed to adopt a *model-driven design* approach. The control strategy behaviors are specified by means of graphic formalisms and are synthesized into code implementing the embedded software in a systematic way. This avoids the need of manual writing and fixing the code, because the embedded software implementation is correct by construction, that is, it implements correctly the specified model behaviors. However, even if the model-driven design simplifies the generation of the software, it does not prevent the designer to wrongly define the software behaviors using the graphic formalisms. For this reason, we supported the design phase with functional verification. We proposed to perform functional verification by means of *dynamic assertion-based verification*. Such a kind of verification uses formal temporal assertions for checking the functional and temporal correctness of the embedded software model and, thus, of its implementation. In this context, our main contributions can be summarized as follows:

- We extended radCASE, an existing environment for model-driven design, code generation, and simulation of embedded software, for supporting a dynamic assertion-based verification approach. In particular, the internal engine of the tool has been modified for generating from the UML-like description of the specification it uses to synthesize the final C code, a corresponding EFSM model to be used for automatic stimuli generation during verification.
- We defined and implemented a dynamic ABV environment for embedded software, named radCHECK, which integrates a graphical assertion editor for assisted assertion definition, a checker generator, and a stimuli generator to simulate the embedded software. In particular:
  - the assertion editor graphically supports the designer in defining formal assertions from the informal specifications of the embedded software through a large set of parametric templates.

- the checker generator synthesizes the defined formal assertions into assertion checkers which are automatically integrated into the simulation environment.
- the stimuli generator provides the simulator with stimuli to efficiently cover corner cases when checkers are simulated to determine the correctness of the implementation w.r.t the specification;

The quality of the verification, strongly depends on the quality of the checked assertions. For this reason, we introduced a further step in the refinement process. Such a step aims at identifying the presence of vacuous assertions, i.e., assertions that are trivially satisfied by the embedded software during the dynamic verification and hide errors into either the software or the assertions themselves. Thus, this kind of assertions can lead designers to a false sense of safety because they may be meaningless or the embedded software implementation may be erroneous even if all the defined assertions are satisfied. In this context, our main contributions can be summarized as follows:

- We redefined the traditional notion of *vacuity checking* in such a way it can be applied in the context of dynamic assertion-based verification of temporal assertions defined according to the *Simple Subset* of PSL. In particular, unlike the existing approaches in literature, we proposed a mutation-based approach that does not require the definition and the verification of support assertions (i.e, *witness formulas*) to generate the *interesting witnesses* to assertions vacuity, reducing the computational complexity of the analysis;
- We extended the current vacuity checking methodologies to identify tautological vacuity in a more accurate way.
- Finally, we implemented the proposed methodology into radCHECK.

## 5.1 Future works

Although we have achieved encouraging results, there are still possible extensions and future works:

- The proposed methodology for synthesis of implementable control strategies based on the AASAP semantics uses a reduction for Elastic controllers that returns a description that is exponentially larger than the initial specification. Indeed, the number of modes of the obtained automaton is larger by an exponential factor in the number of inputs events of the Elastic controller. This state explosion is not a problem from a theoretical point of view, but in practice it does not allow to handle interesting industrial examples. The work in [47] proposes an alternative reduction that is compositional. Instead of modeling the Elastic controller with an unique timed automaton, the authors generate several (smaller) timed automata whose composition behaves as the initial Elastic controller interpreted using the AASAP semantics. However, the exponential behavior can still appear during the verification phase but only in the worst case. This solution is very interesting but it is model checker dependent: the authors have defined transformation rules which exploit ad hoc constructs typical of HyTech or Uppaal model checkers. Thus, more generic rules could be investigated to avoid the model blow up and keep it compatible with different model checkers, such as PHAVer and Ariadne.

- The proposed methodology requires that once the implementable control strategy has been identified, it is manually refined into a discrete graphic formalism to enable the embedded software generation. A very valuable extension consists of automatically synthesize the control strategy into another model suited for MDD of embedded software to enable the generation of a correct-by-construction generation of the embedded code.
- Another valuable extension could be the refinement of the whole implementable model $M_{\mathcal{R}}$ into a Stateflow-Simulink model. In this way, the integrated Simulink *embedded code generator* can be used to obtain the embedded software implementing the control strategy. Moreover, the Simulink environment can be used for further refinements to achieve a more detailed embedded software specification definition.
- As a consequence, also the radCHECK environment should be extended to support Simulink models verification. This requires the investigation of how to automatically generate effective stimuli sequences for Simulink designs and integrate assertion checkers into the Simulink simulation environment. Although the guided assertion definition infrastructure does not need modifications, future works can be related to the investigation of property patterns suited for Simulink designs.
- The proposed vacuity analysis approach could be extended by introducing a preliminary check that analyzes the Büchi automata derived from assertions to identify trivially valid formulas such as tautologies. Intuitively, the analysis of the automaton structure allows to identify satisfiable and un-satisfiable assertions. Thus, the un-satisfiability of $\neg\varphi$, i.e., the negation of an assertion $\varphi$, proves the validity of $\varphi$. This preliminary analysis speeds up the identification of vacuous assertions, and can be easily implemented into radCHECK. More investigations can be performed to identify if other cases of vacuity can be structurally determined avoiding the simulation of *faulty* checkers.

# Part II

## Articles

# 6

# Published contributions

## 6.1 Journal Publications

Giuseppe Di Guglielmo, Luigi Di Guglielmo, Franco Fummi, and Graziano Pravadelli
*Efficient Generation of Stimuli for Functional Verification by Backjumping Across Extended FSMs*
Journal of Electronic Testing: Theory and Application
Volume 27 , Issue 2 , 2011 , pp. 137-162

## 6.2 International Conferences

Giuseppe Di Guglielmo, Luigi Di Guglielmo, Franco Fummi, and Graziano Pravadelli
*Enabling Dynamic Assertion-based Verification of Embedded Software through Model-driven Design*
In the proceedings of ACM/IEEE Design, Automation and Test in Europe (DATE)
Dresden, Germany, 2012

Graziano Pravadelli, Luigi Di Guglielmo, Franco Fummi, Sara Vinco and Francesco Stefanni
*UNIVERCM: the UNIversal VERsatile Computational Model for heterogeneous embedded system design*
In the proceedings of IEEE International High Level Design Validation and Test Workshop (HLDVT)
Napa Valley, CA, USA, November 10-11, 2011

Giuseppe Di Guglielmo, Luigi Di Guglielmo, Franco Fummi, and Graziano Pravadelli
*IPA: Assertion-based Verification in Embedded-Software Design*
In the proceedings of IEEE International High Level Design Validation and Test Workshop (HLDVT)
Napa Valley, CA, USA, November 10-11, 2011

Luigi Di Guglielmo, Franco Fummi, and Graziano Pravadelli
*IPA: Reusing of Properties after Discretization of Hybrid Automata*

In the proceedings of IEEE International High Level Design Validation and Test Workshop (HLDVT)
Napa Valley, CA, USA, November 10-11, 2011

Davide Bresolin, Luigi Di Guglielmo, Luca Geretti, and Tiziano Villa
*Correct-by-Construction Code Generation from Hybrid Automata Specification*
In the proceedings of IEEE Workshop on Design, Modeling and Evaluation of Cyber Physical Systems (CyPhy)
Istanbul, Turkey , July 5-8, 2011

Giuseppe Di Guglielmo, Masahiro Fujita, Luigi Di Guglielmo, Franco Fummi, Graziano Pravadelli, Cristina Marconcini and Andreas Foltinek
*Model-Driven Design and Validation of Embedded Software*
In the proceedings of IEEE/ACM International Workshop on Automation of Software Testing (ICSE Workshop)
Waikiki, Honoulu, Hawaii , May 23-24, 2011

Luigi Di Guglielmo, Franco Fummi, Nicola Orlandi, and Graziano Pravadelli
*DDPSL: an Easy Way of Defining Properties*
In the proceedings of IEEE International Conference on Computer Design (ICCD)
Amsterdam, the Netherlands, October 3-6, 2010

Nicola Bombieri, Giuseppe Di Guglielmo, Luigi Di Guglielmo, Michele Ferrari, Franco Fummi, Graziano Pravadelli, Francesco Stefanni, and Alessandro Venturelli
*HIFSuite: Tools for HDL Code Conversion and Manipulation*
In the proceedings of IEEE International High Level Design Validation and Test Workshop (HLDVT)
Anaheim, CA, USA, June 10-12, 2010

Luigi Di Guglielmo, Franco Fummi, and Graziano Pravadelli
*Vacuity Analysis for Property Qualification by Mutation of Checkers*
In the proceedings of ACM/IEEE Design, Automation and Test in Europe (DATE)
Dresden, Germany, May 8-12, 2010

Luigi Di Guglielmo, Franco Fummi, and Graziano Pravadelli
*The Role of Mutation Analysis for Property Qualification*
In the proceedings of ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)
Cambridge, MA, USA, July 13-15, 2009

Luigi Di Guglielmo, Franco Fummi, and Graziano Pravadelli
*Vacuity Analysis by Fault Simulation*
In the proceedings of ACM/IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE)
Anaheim, CA, USA, 5-7 June, 2008

# References

1. 3S Software. CoDeSys, 2012. `http://www.3s-software.com`.
2. Yael Abarbanel, Ilan Beer, Leonid Gluhovsky, Sharon Keidar, and Yaron Wolfsthal. FoCs: Automatic Generation of Simulation Checkers from Formal Specifications. In *Proc. of International Conference on Computer Aided Verification (CAV)*, pages 538–542, 2000.
3. Aerospace Valley. TOPCASED project, 2012. `http://www.topcased.org`.
4. M. Agrawal, F. Stephan, P.S. Thiagarajan, and S. Yang. Behavioural Approximations For Restricted Linear Differential Hybrid Automata. In *Proc. of International Conference on Hybrid Systems: Computation and Control (HSCC)*, pages 4–18, 2006.
5. M. Agrawal and P. Thiagarajan. The Discrete Time Behavior of Lazy Linear Hybrid Automata. In *Proc. of International Conference on Hybrid Systems: Computation and Control (HSCC)*, pages 55–69, 2005.
6. M. Agrawal and P.S. Thiagarajan. Lazy Rectangular Hybrid Automata. In *Proc. of International Conference on Hybrid Systems: Computation and Control (HSCC)*, pages 249–257, 2004.
7. K. Altisen and S. Tripakis. Implementation of Timed Automata: an Issue of Semantics or Modeling? In *Proc. of International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS)*, pages 273–288, 2005.
8. R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P. H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The Algorithmic Analysis of Hybrid Systems. *Theoretical Computer Science*, 138(1):3–34, 1995.
9. R. Alur, T. Henzinger, G. Lafferriere, and G. J. Pappas. Discrete Abstractions of Hybrid Systems. *Proceedings of the IEEE*, 88(7):971–984, 2000.
10. R. Armoni, L. Fix, A. Flaisher, O. Grumberg, N. Piterman, A. Tiemeyer, and M.Y. Vardi. Enhanced Vacuity Detection in Linear Temporal Logic. In *Proc. of International Conference on Computer Aided Verification (CAV)*, volume 2725, pages 368–380, 2003.
11. Atego. ARTiSAN, 2012. `http://www.atego.com/products/artisan-studio`.
12. M. Baleani, A. Ferrari, L. Mangeruca, and A. Sangiovanni-Vincentelli. Efficient Embedded Software Design with Synchronous Models. In *Proc. of ACM International Conference on Embedded Software (EMSOFT)*, pages 187–190, 2005.
13. T. Ball and O. Kupferman. Vacuity in Testing. In *Proc. of ACM International Conference on Tests and Proofs (TAP)*, volume 4966, pages 4–17, 2008.
14. A. Balluchi, A. Casagrande, P. Collins, A. Ferrari, T. Villa, and A.L. Sangiovanni-Vincentelli. Ariadne: a Framework for Reachability Analysis of Hybrid Automata. In

*Proc. of International Symposium on Mathematical Theory of Networks and Systems (MTNS)*, 2006.

15. Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability Modulo Theories. In Armin Biere, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 4, chapter 8. IOS Press, 2009.
16. I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient Detection of Vacuity in Temporal Model Checking. *Formal Methods in System Design*, pages 141–163, 2001.
17. Gerd Behrmann, Alexandre David, Kim Guldstrand Larsen, Paul Pettersson, and Wang Yi. Developing UPPAAL over 15 years. *Software: Practice and Experience*, 41(2):133–142, 2011.
18. E. Beigné, F. Clermidy, S. Miermont, P. Vivet, and G. MINATEC. Dynamic Voltage and Frequency Scaling Architecture for Units Integration within a GALS NoC. In *Proc. of ACM/IEEE International Symposium on Networks-on-Chip (NoCS)*, pages 129–138, 2008.
19. S. Ben-David, D. Fisman, and S. Ruah. Temporal Antecedent Failure: Refining Vacuity. In *Proc. of International Conference on Concurrency Theory (CONCUR)*, volume 4703, pages 492–506, 2007.
20. A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282, 1991.
21. A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. De Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.
22. L. Benvenuti, A. Ferrari, E. Mazzi, and A. Vincentelli. Contract-based Design for Computation and Verification of a Closed-loop Hybrid System. In *Proc. of International Conference on Hybrid Systems: Computation and Control (HSCC)*, pages 58–71, 2008.
23. A. Biere, A. Cimatti, E.M. Clarke, O. Strichman, and Y. Zhu. Bounded Model Checking. *Advances in Computers*, 58:117–148, 2003.
24. V. D. Blondel, O. Bournez, P. Koiran, C. H. Papadimitriou, and J. N. Tsitsiklis. Deciding Stability and Mortality of Piecewise-Affine Dynamical Systems. *Theoretical Computer Science*, 255(1-2):687–696, 2001.
25. V. D. Blondel and J. N. Tsitsiklis. Complexity of Stability and Controllability of Elementary Hybrid Systems. *Automatica*, 35:479–490, 1999.
26. D. Borrione, Miao Liu, K. Morin-Allory, P. Ostier, and L. Fesquet. On-line Assertion-based Verification with Proven Correct Monitors. In *Proc. of International Conference on Information and Communications Technology (ICICT)*, pages 125–143, 2005.
27. M. Boulé and Z. Zilic. Automata-based Assertion-Checker Synthesis of PSL Properties. *ACM Transactions on Design Automation of Electronic Systems*, 13:1–21, February 2008.
28. M. Boulé and Z. Zilic. *Generating hardware assertion checkers: for hardware verification, emulation, post-fabrication debugging and on-line monitoring*. Springer, 2008.
29. F. Boutekkouk, M. Benmohammed, S. Bilavarn, M. Auguin, et al. UML 2.0 Profiles for Embedded Systems and Systems on a Chip (SoCs). *Journal of Object Technology*, 2009.
30. S. Brait, F. Fummi, and G. Pravadelli. On the Use of a High-Level Fault Model to Analyze Logical Consequence of Properties. In *Proc. of ACM/IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, pages 221–230, 2005.

31. D. Bresolin, L. Di Guglielmo, L. Geretti, and T. Villa. Correct-by-construction code generation from hybrid automata specification. In *Proc. of International Wireless Communications and Mobile Computing Conference (IWCMC)*, pages 1660 – 1665, 2011.
32. R. Brummayer and A. Biere. Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. In *Proc. of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 174–177, 2009.
33. R. Bryant, S. Lahiri, and S. Seshia. Modeling and Verifying Systems Using a Logic of Counter Arithmetic With Lambda Expressions and Uninterpreted Functions. In *Proc. of International Conference on Computer Aided Verification (CAV)*, pages 106–122. Springer, 2002.
34. C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *Proc. of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.
35. Cadence. Assertion-based Verification, 2012. `http://www.cadence.com/products/fv/pages/abv_flow.aspx`.
36. F. Cassez, T. Henzinger, and J. F. Raskin. A Comparison of Control Problems for Timed and Hybrid Systems. In *Proc. of International Conference on Hybrid Systems: Computation and Control (HSCC)*, pages 134–148, 2002.
37. K. T. Cheng and A. S. Krishnakumar. Automatic Generation of Functional Vectors Using the Extended Finite State Machine Model. *ACM Transactions on Design Automation of Electronic Systems*, 1(1):79, 1996.
38. P. Cheung and A. Forin. A C-language Binding for PSL. In *Proc. of International Conference on Embedded Software and Systems (ICESS)*, pages 584–591. Springer, 2007.
39. H. Chockler, A. Gurfinkel, and O. Strichman. Beyond Vacuity: Towards the Strongest Passing Formula. In *Proc. of International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 1–8, 2008.
40. H. Chockler and O. Strichman. Easier and More Informative Vacuity Checks. In *Proc. of ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 189–198, 2007.
41. H. Chockler and O. Strichman. Before and After Vacuity. *Formal Methods in System Design*, 34(1):37–58, 2009.
42. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.
43. A. Dahan, D. Geist, L. Gluhovsky, D. Pidan, G. Shapir, Y. Wolfsthal, L. Benalycherif, R. Kamidem, and Y. Lahbib. Combining System Level Modeling with Assertion-based Verification. In *Proc. of International Symposium on Quality of Electronic Design (ISQED)*, pages 310–315, 2005.
44. S. Das, R. Mohanty, P. Dasgupta, and P.P. Chakrabarti. Synthesis of System Verilog Assertions. In *Proc. of Design, Automation & Test in Europe Conference & Exhibition (DATE)*, volume 2, pages 1–6, 2006.
45. R. De Simone and C. André. Towards a "Synchronous Reactive" UML profile? *International Journal on Software Tools for Technology Transfer*, 8(2):146–155, 2006.
46. M. De Wulf, L. Doyen, and J. F. Raskin. Almost ASAP Semantics: from Timed Models to Timed Implementations. *Formal Aspects of Computing*, 17(3):319 – 341, 2005.
47. M. De Wulf, L. Doyen, and J.F. Raskin. Systematic Implementation of Real-Time Models. *FM 2005: Formal Methods*, pages 596–596, 2005.
48. G. Di Guglielmo, L. Di Guglielmo, F. Fummi, and G. Pravadelli. Efficient generation of stimuli for functional verification by backjumping across extended fsms. *Journal of Electronic Testing*, pages 1–26, 2011.

49. G. Di Guglielmo, M. Fujita, L. Di Guglielmo, F. Fummi, G. Pravadelli, C. Marconcini, and A. Foltinek. Model-driven Design and Validation of Embedded Software. In *Proc. of ACM International Workshop on Automation of Software Test (AST)*, pages 98–104, 2011.
50. G. Di Guglielmo, F. Fummi, C. Marconcini, and G. Pravadelli. EFSM Manipulation to Increase High-Level ATPG Effectiveness. In *Proc. of International Symposium on Quality of Electronic Design (ISQED)*, pages 57–62, 2006.
51. G. Di Guglielmo, F. Fummi, G. Pravadelli, S. Soffia, and M. Roveri. Semi-formal functional verification by EFSM traversing via NuSMV. In *Proc. of IEEE International High Level Design Validation and Test Workshop (HLDVT)*, pages 58–65, 2010.
52. L. Di Guglielmo, F. Fummi, N. Orlandi, and G. Pravadelli. DDPSL: An Easy Way of Defining Properties. In *Proc. of IEEE International Conference on Computer Design (ICCD)*, pages 468–473, 2010.
53. L. Di Guglielmo, F. Fummi, and G. Pravadelli. The Role of Mutation Analysis for Property Qualification. In *Proc. of IEEE/ACM International Conference on Formal Methods and Models for Co-Design (MEMOCODE)*, pages 28–35, 2009.
54. L. Di Guglielmo, F. Fummi, and G. Pravadelli. Vacuity analysis for property qualification by mutation of checkers. In *Proc. of Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 478–483, March 2010.
55. B. Dutertre and L. De Moura. The Yices SMT Solver, 2006. http://yices.csl.sri.com/tool-paper.pdf.
56. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proc. of IEEE International Conference on Software Engineering (ICSE)*, pages 411–420, 1999.
57. C. Ebert and C. Jones. Embedded software: Facts, figures, and future. *Computer*, 42(4):42–52, 2009.
58. N. Eén and N. Sörensson. An Extensible SAT-solver. In *Proc. of International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 502–518, 2003.
59. E.A. Emerson. Temporal and Modal Logic. *Handbook of theoretical computer science*, 8:995–1072, 1990.
60. A. Ferrari, G. Gaviani, G. Gentile, G. Stara, G. Romagnoli, and T. Thomsen. From conception to implementation: a model based design approach. In *Proc. of IFAC Symposium on Advances in Automotive Control*, 2004.
61. L. Ferro and L. Pierre. ISIS: Runtime Verification of TLM Platforms. *Advances in Design Methods from Modeling Languages for Embedded Systems and SoCs*, 63:213–226, 2010.
62. A. Fin and F. Fummi. Laerte++: an Object Oriented High-Level TPG for SystemC Designs. *Languages for System Specification*, pages 105–117, 2004.
63. Foltinek, A. UML in practice ... a practical guideline for C/C++ developers, 2011. http://www.imacs-gmbh.de/download.php?id=26&site=13&lang=EN.
64. H. Foster, A. Krolnik, and D. Lacey. *Assertion-based design*. Springer Netherlands, 2004.
65. H. Foster, K. Larsen, and M. Turpin. Introducing the New Accellera Open Verification Library Standard. In *Proc. of Design and Verification Conference (DVCON)*, 2006.
66. Harry D. Foster, Adam C. Krolnik, and David J. Lacey. *Assertion-based Design*. Springer Academic Publishers Group, The Netherlands, 2004.
67. Goran Frehse. PHAVer: Algorithmic Verification of Hybrid Systems Past HyTech. In *Proc. of International Conference on Hybrid Systems: Computation and Control (HSCC)*, pages 258–273, 2005.

68. Gentleware. Poseidon for UML embedded edition, 2012. http://www.gentleware.com/uml-software-embedded-edition.html.
69. Razorcat Development GmbH. Classification Tree Editor (CTE) for Embedded Systems, 2012. www.razorcat.com.
70. P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proc. of ACM SIGPLAN Conference on Programming Language, Design, and Implementation (PLDI)*, pages 213–223, 2005.
71. B. Graaf, M. Lormans, and H. Toetenel. Embedded Software Engineering: The State of the Practice. *IEEE Software*, 20(6):61–69, 2003.
72. M. Grochtmann and K. Grimm. Classification Trees for Partition Testing. *Software: Testing, Verification and Reliability*, 3(2):63–82, 1993.
73. A. Gurfinkel and M. Chechik. How Vacuous is Vacuous? In *Proc. of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988, pages 451–466. Springer, 2004.
74. HAL - Inria. Gaspard2 UML profile documentation, 2012. http://hal.inria.fr/inria-00171137/en.
75. M. P. E. Heimdahl, D. George, and R. Weber. Specification Test Coverage Adequacy Criteria = Specification Test Generation Inadequacy Criteria? In *Proc. of IEEE International High Assurance Systems Engineering Symposium (HASE)*, pages 178–186, 2004.
76. T. Henzinger. Hybrid Automata With Finite Bisimulations. In *Proc. of International Colloquium on Automata, Languages and Programming (ICALP)*, pages 324–335, 1995.
77. T. Henzinger. The Theory of Hybrid Automata. In *Proc. of IEEE Symposium on Logic in Computer Science (LICS)*, pages 278 – 292, 1996.
78. T. Henzinger, P. H. Ho, and H. Wong-Toi. HyTech: A model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):110–122, 1997.
79. T. Henzinger and P. Kopke. Discrete-Time Control For Rectangular Hybrid Automata. In *Proc. of International Colloquium on Automata, Languages and Programming (ICALP)*, pages 582–593. Springer, 1997.
80. T. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya. What's Decidable About Hybrid Automata? In *Proc. of ACM Symposium on Theory of Computing (STOC)*, pages 373–382, 1995.
81. M. Hiller. Executable Assertions for Detecting Data Errors in Embedded Control Systems. In *Proc. of IEEE International Conference on Dependable Systems and Networks (DSN)*, pages 24–33, 2000.
82. IAR Systems. IAR visualSTATE, 2012. http://www.iar.com/Products/IAR-visualSTATE/.
83. IBM. Rational Rhapsody, 2012. http://www.ibm.com/software/awdtools/rhapsody.
84. IBM. Rational Rhapsody Automatic Test Generation Add On, 2012. http://www.ibm.com/software/rational/products/rhapsody/developer/features/test.html.
85. IEEE Computer Society. IEEE Standard for SystemC (IEEE Std 1666-2005), 2006.
86. IEEE Computer Society. IEEE Standard for System Verilog-Unified Hardware Design, Specification, and Verification Language (IEEE Std 1800-2009), 2009.
87. IEEE Computer Society. IEEE Standard for Property Specification Language (PSL) (IEEE Std 1850-2010), 2010.
88. Interdisciplinary Center for Scientific Computing. MUSCOD II, 2012. http://www.iwr.uni-heidelberg.de/~agbock/RESEARCH/muscod.php.
89. S. K. Jha, B. A. Brady, and S. A. Seshia. Symbolic Reachability Analysis of Lazy Linear Hybrid Automata. In *Proc. of International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS)*, pages 241–256, 2007.

90. S. K. Jha, R. Limaye, and S. A. Seshia. Beaver: Engineering An Efficient SMT Solver for Bit-Vector Arithmetic. In *Proc. of International Conference on Computer Aided Verification (CAV)*, pages 668–674, 2009.
91. K. H. John and M. Tiegelkamp. *IEC 61131-3: Programming Industrial Automation Systems: Concepts and Programming Languages, Requirements for Programming Systems, Decision-Making Aids*. Springer, 2010.
92. A. S. Kalaji, R. M. Hierons, and S. Swift. A Search-Based Approach for Automatic Test Generation from Extended Finite State Machine. In *Proc. of The Testing: Academic & Industrial Conference - Practice and Research Techniques (TAICPART)*, pages 131–132, 2009.
93. M. Kim, Y. Kim, and H. Kim. A Comparative Study of Software Model Checkers as Unit Testing Tools: An Industrial Case Study. *IEEE Transactions on Software Engineering*, 37(2):146–160, 2011.
94. J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, 1976.
95. P. Krčál, L. Mokrushin, P. Thiagarajan, and W. Yi. Timed vs. time-triggered automata. In *Proc. of International Conference on Concurrency Theory (CONCUR)*, pages 340–354, 2004.
96. D. Kroening and O. Strichman. Efficient Computation of Recurrence Diameters. In *Proc. of International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 298–309, 2003.
97. O. Kupferman. Sanity Checks in Formal Verification. *Concurrency Theory*, 4137:37–51, 2006.
98. O. Kupferman and M.Y. Vardi. Vacuity Detection in Temporal Model Checking. *International Journal on Software Tools for Technology Transfer*, pages 224–233, 2003.
99. S. Lahiri and S. A. Seshia. The UCLID Decision Procedure. In *Proc. of International Conference on Computer Aided Verification (CAV)*, pages 259–262, 2004.
100. C. Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice Hall PTR, 2004.
101. C. Lattner and V. Adve. The LLVM Compiler Framework and Infrastructure Tutorial. In *Proc. of International Workshop on Languages and Compilers for High Performance Computing (LCPC)*, pages 15–16. Springer, 2005.
102. E.A. Lee. Embedded software. *Advances in computers*, 56:55–95, 2002.
103. D. Lettnin, P.K. Nalla, J. Ruf, T. Kropf, W. Rosenstiel, T. Kirsten, V. Schonknecht, and S. Reitemeyer. Verification of Temporal Properties in Automotive Embedded Software. In *Proc. of Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 164–169. ACM, 2008.
104. D.W. Lewis. *Fundamentals of embedded software: where C and Assembly meet*. Prentice-Hall, 2002.
105. R. Majumdar and K. Sen. Hybrid Concolic Testing. In *Proc. of IEEE International Conference on Software Engineering (ICSE)*, pages 416–426, 2007.
106. J. McManis and P. Varaiya. Suspension Automata: A Decidable Class of Hybrid Automata. In *Proc. of International Conference on Computer Aided Verification (CAV)*, pages 105–117, 1994.
107. P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.
108. Mentor Graphics. Assertion-based Verification, 2012. `http://www.mentor.com/products/fv/methodologies/abv`.
109. F. Mischkalla, D. He, and W. Mueller. A UML Profile for SysML-Based Comodeling for Embedded Systems Simulation and Synthesis. In *Proc. of Workshop on Model Based Engineering for Embedded System Design (MBED)*, 2010.

110. Modelica Association. Modelica, 2012. https://modelica.org.
111. K.D. Muller-Glaser, G. Frick, E. Sax, and M. Kuhl. Multiparadigm Modeling in Embedded Systems Design. *IEEE Transactions on Control Systems Technology*, 12(2):279–292, 2004.
112. Object Management Group, Inc. MARTE Resource Page, 2012. http://www.omgmarte.org/.
113. Object Management Group, Inc. OMG Object Constraint Language (OCL), 2012. http://www.omg.org/spec/OCL/.
114. Object Management Group, Inc. OMG Specifications, 2012. http://www.omg.org.
115. Object Management Group, Inc. UML Profile For Schedulability,Performance, And Time, 2012. http://www.omg.org/spec/SPTP/.
116. Object Management Group, Inc. UML Resource Page, 2012. http://www.uml.org.
117. A. Jefferson Offutt and Ronald H. Untch. Mutation 2000: Uniting the Orthogonal. *Mutation Testing for the New Century*, pages 34–44, 2001.
118. Politecnico di Torino. ITC-99 Benchmarks, 1999. http://www.cad.polito.it/tools/itc99.html.
119. R. S. Pressman. *Software Engineering - A Practitioners Approach*. McGraw Hill, 1992.
120. Process Systems Enterprise Limited. gPROMS, 2012. http://www.psenterprise.com/gproms/.
121. A. Puri and P. Varaiya. Decidability of Hybrid Systems With Rectangular Differential Inclusions. In *Proc. of International Conference on Computer Aided Verification (CAV)*, pages 95–104, 1994.
122. J. F. Raskin. Reachability Problems for Hybrid Automata. In *Proc. of International Workshop on Reachability Problems (RP)*, pages 28–30, 2011.
123. E. Riccobene, P. Scandurra, S. Bocchio, A. Rosti, L. Lavazza, and L. Mantellini. SystemC/C-based Model-driven Design for Embedded Systems. *ACM Transaction on Embedded Computing Systems*, 8(4):1–37, 2009.
124. M. Samer and H. Veith. On the Notion of Vacuous Truth. In *Proc. of International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, volume 4790, pages 2–14. Springer, 2007.
125. O. Sankur, P. Bouyer, and N. Markey. Shrinking Timed Automata. In *Leibniz International Proceedings in Informatics*, 2011.
126. B. Selic. The Pragmatics of Model-driven Development. *IEEE Software*, 20(5):19–25, 2003.
127. K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *Proc. of International Conference on Computer Aided Verification (CAV)*, pages 419–423. Springer, 2006.
128. S. Shukla, A. Hu, J. Abrahams, P. Ashar, H. Foster, A. Landver, and C. Pixley. Panel: Assertion-Based Verification - What's the Big Deal? In *Proc. of IEEE International High Level Design Validation and Test Workshop (HLDVT)*, page 189, 2006.
129. C. Sonntag, R. R. H. Schiffelers, D. A. van Beek, J. E. Rooda, and S. Engell. Modeling and Simulation using the Compositional Interchange Format for Hybrid Systems. In *International Conference on Mathematical Modelling (MATHMOD)*, pages 640–650, 2009.
130. Sparx Systems. Enterprise Architet, 2012. http://www.sparxsystems.com.au.
131. STM Products. radCHECK. http://www.verificationsuite.com.
132. STM Products. STM Embedded Applications. http://www.stm-products.com.
133. STMProducts, IMACS. radCASE. http://www.stm-case.com.
134. SysML Partners. SysML Resource Page, 2012. http://www.sysml.org.
135. Esterel Technologies. Scade suite, 2012. http://www.esterel-technologies.com/products/scade-suite.

136. The MathWorks. Matlab, 2012. `http://www.mathworks.it/products/matlab/`.
137. The MathWorks, Inc. Simulink, 2012. `http://www.mathworks.com/products/simulink/`.
138. The MathWorks, Inc. Simulink Verification and Validation toolbox, 2012. `http://www.mathworks.it/help/toolbox/slvnv/ug/bs_ftl2.html`.
139. N. Tillmann and J. De Halleux. Pex: White Box Test Generation for. NET. In *Proc. of ACM International Conference on Tests and Proofs (TAP)*, pages 134–153, 2008.
140. J. Tong, M. Boulé, and Z. Zilic. Defining and Providing Coverage for Assertion-Based Dynamic Verification. *Journal of Electronic Testing*, 26:211–225, 2010.
141. UC Berkeley EECS Dept. Ptolemy II, 2012. `http://ptolemy.berkeley.edu/ptolemyII/`.
142. University of Colorado. VIS, 1999. `http://vlsi.colorado.edu/~vis`.
143. J. Wegener, R. Pitschinetz, K. Grimm, and M. Grochtmann. TESSY - Yet Another Computer-Aided Software Testing Tool. In *Proc. of European International Conference on Software Testing, Analysis and Review (EuroSTAR)*, 1994.
144. G. Williams, M. Karlesky, and M. VanderVoord. Unity - Compact Test Framework for C, 2012. `http://sourceforge.net/projects/unity`.
145. M. Winterholer. Transaction-based Hardware Software Co-Verification. In *In Proc. of Forum on Specification & Design Languages (FDL)*, 2006.
146. F. Xie and H. Liu. Unified Property Specification for Hardware/Software Co-Verification. In *Proc. of International Computer Software and Applications Conference (COMSAC)*, pages 483–490, 2007.