A framework for the rigorous design of highly adaptive timed systems

Maxime Cordy [†], Axel Legay[*], Pierre-Yves Schobbens [†], and Louis-Marie Traonouez[*]
[*]INRIA Rennes, France, {axel.legay,louis-marie.traonouez}@inria.fr
[†]PreCISE Research Center, University of Namur, Belgium, {mcr,pys}@info.fundp.ac.be

Abstract—**Adaptive systems can be regarded as a set of static programs and transitions between these programs. These transitions allow the system to adapt its behaviour in response to unexpected changes in its environment. Modelling highly dynamic systems is cumbersome, as these may go through a large number of adaptations. Moreover, often they must also satisfy real-time requirements whereas adaptations may not complete instantaneously. In this paper, we propose to model highly adaptive systems as dynamic real-time software product lines, where software products are able to change their features at runtime. Adaptive features allow one to design systems equipped with runtime reconfiguration capabilities and to model changes in their environment, such has failure modes. We define Featured Timed Game Automata, a formalism that combines adaptive features with discrete and real-time behaviour. We also propose a novel logic to express real-time requirements on adaptive systems, as well as algorithms to check a system against them. We implemented our method as part of PyECDAR, a model checker for timed systems.**

Index Terms—**Software Product Lines, Features, Real-time systems, Model-checking, Timed Games**

I. Introduction

Computers play a central role in modern life and their errors can have dramatic consequences. Proving the correctness of computer systems is therefore an extremely relevant problem for which quality assurance techniques like model checking and testing provide efficient solutions. Testing consists in applying a finite series of test cases to the system. Although it can detect errors, it cannot guarantee their absence. Another of its limitations is that nowadays, systems are embedded and highly configurable, which makes it hard to specify relevant test cases. Model checking [1] is an automated technique for verifying systems against functional requirements. The approach relies on an exhaustive verification of a bevahioral model of the system against a property expressed in temporal logic. If the system fails to satisfy the property, then the model checking algorithm provides an example of violation. By nature, model checking guarantees the absence of errors. Albeit it suffers from the so-called state-space explosion, it has been widely used and applied on both academic and industry case studies. Model checking was initially intended for closed and static Boolean systems, but has been extended to target increasingly wider classes of systems, including real-time systems.

The recent advances in computer science pose new challenges to model checking. One of the major difficulties is that today's systems often run in open and potentially unsafe environments, which requires them to adapt their behavior in order to accomplish their tasks reliably. In case of highly evolving environment, these adaptations must be performed as quickly as possible, hence the need for *self-adaptive* systems. These are harder to verify than static, closed systems of which behaviour and environment are known a priori. Applying model checking to such systems requires to represent all its classes of behaviour as well as its capability to transit between them. Moreover, adaptive systems must satisfy multiple goals which may evolve over time and according to changes in the system or its environment [2].

One way to model an adaptive system is to view it as a set of static programs and transitions between these programs [3]. When the system has to adapt its behaviour, it triggers a transition to one of its other programs. The drawback of this approach is that all these programs must be modelled and verified individually. This approach has huge costs and is intractable. Another difficulty is the need to verify dynamic properties of adaptive systems. Classical logics cannot express them in a proper way. Alternatives to existing model checking techniques of adaptive systems are thus needed.

The static programs composing an adaptive system likely share commonality, as they also have proper parts. An alternative is to organize the variability between these programs into features, a concept borrowed from software product line engineering (SPLE) [4]. In the latter discipline, a feature is an added functionality that meet a requirement of customers. A product of the line is thus obtained by composing desired features together. In the context of adaptive systems, features model differences between the static programs composing the system. Modifications in its behaviour are therefore triggered by changing its features. We name this process *reconfiguration*. Features constitute an appropriate modelling artifact to reason on runtime variability. Moreover, transposing this concept to adaptive systems permits to benefit from the formal verification techniques currently developed in SPLE.

The behavior of adaptive systems often rely on real-time requirements such as matching deadline or reacting in real-time to fault. For example, a routing protocol must ensure that a data packet must reach the recipient within a certain amount of time (see more in Section II). Unexpected changes in the environment may prevent the satisfaction of these requirements, hence the need for the system to perform adaptations. The reconfiguration process is not always instantaneous, though. The system may require time to change its features, or can have to delay the reconfiguration until it reaches a stable state.

Unfortunately, most of existing model checking techniques for adaptive systems are not capable of handling such constraints.

In this paper, we propose a formal framework to model and verify adaptive systems that must satisfy evolving real-time requirements. We introduce *Featured Timed Game Automata* (FTGA), a formalism to represent adaptive behaviour, dynamic environment, and real-time. Our model results from the combination of (1) Adaptive Featured Transition Systems [4], a formalism to model dynamic reconfiguration, and evolving environment, and (2) timed automata [5], an established formalism for real-time systems. The semantics of a FTGA is defined as a timed game, where the system plays against the environment. Our formalism differs from existing game-based approaches [6] in that it concisely models reconfigurations of the systems and evolutions of the environment by exploiting the featured transition approach [7]. This latter provides even more flexibility to our method, which supports not only runtime configuration but also design-time variability. FTGA thus constitute an unified formalism to model the behaviour of *real-time adaptive software product lines*.

As a second contribution, we propose a new temporal logic to express requirements on FTGA. In [4], we introduced *Adaptive Configuration Time Logic* (AdaCTL), a variant of the *Computational Tree Logic* (CTL) to reason on features and reconfigurations. The main differences between AdaCTL and CTL are that (1) the existential and universal quantifiers have a game-based semantics similar to Alternating Tree Logic (ATL) [8], and (2) the satisfaction relation returns a set of configurations rather than a Boolean value. In this paper, we go one step further and introduce *Timed Adaptive Configuration Time Logic* (T-AdaCTL), a real-time extension of AdaCTL, the semantics of which is inspired from Timed-ATL [9].

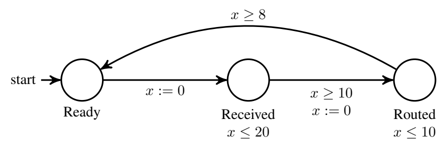
Finally, we design efficient model-checking algorithms to verify an adaptive system modelled as an FTGA against requirements expressed in T-AdaCTL. These algorithms extend efficient timed-game algorithms [10]. As a proof-of-concept, we implemented our method as part of PyECDAR, a model-checker for timed systems [11].

Structure of the paper. In Section II, we introduce our running example and the state of the art is presented in Section III. We define FTGA in Section IV, whereas we introduce T-AdaCTL and our model checking algorithms in Section V. We discuss our implementation in Section VI.

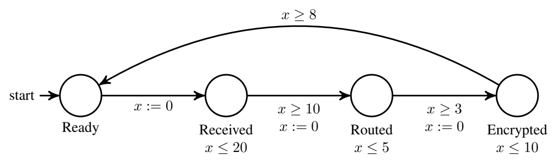
II. Introductory Example

We present an example inspired by the TCP routing protocol described in [3]. We consider a routing protocol that can work in two different environments: a safe environment, where all the nodes are fully trusted, and an unsafe environment, where some nodes might be corrupted. In an unsafe environment, a message must be encrypted before it is sent. Every operation (routing, sending and encryption) requires time to complete. The behavior of the protocol in the two types of environment are modelled as timed automata in Fig. 1.

The protocol must satisfy safety and liveness properties. When the environment is unsafe, all the messages must be encrypted before they are sent. When the environment is safe, the messages must be sent at most 20 time units after being received. In a changing environment, the protocol must switch between the two configurations in order to adapt itself. This reconfiguration is only possible in state received. We study two different implementations: in the first one, the reconfiguration can occur at most every 25 time units; in the second one, the reconfiguration can always be done but its application requires 5 time units. We want to determine in which implementations the system can satisfy its specifications.



(a) Safe environment



(b) Unsafe environment

Figure 1: Routing protocol in safe and unsafe environment.

III. State of the Art

This paper is at the intersection of adaptive systems verification and software product line model checking. We overview relevant related work in both fields.

A. Quality Assurance for Adaptive Systems

In their research roadmap for adaptive systems [2], Cheng *et al.* stated that in the context of adaptive systems, the objective of quality assurance is to provide evidence that the system is able to cope with changes in its objectives and its environment. They presented a framework for adaptive systems assurance, in which the system, the goals, and the context are subject to modifications. This results in a succession of models for the system and properties to verify. Based this idea, several verification methods model adaptive systems as a set of programs [12], [13], [14], [3]. To ensure the satisfaction of intended properties in an unstable environment, the system is able to make transitions between those programs, that is, to perform an adaptation to modify its future behaviour. One distinguishes between local properties that specific programs must satisfy, global properties that must be satisfied by any execution of the system, and transitional properties that must hold during an adaptation. To specify the transitional properties, Zhang *et al.* proposed a new logic called A-LTL [15] and algorithms to check a model against properties expressed in this logic [3]. As an alternative, Kulkarni *et al.* [14] consider that an adaptive system is a program able to add or remove components during runtime. Instead of traditional model checking, they use proof lattice to verify that all the adaptations satisfy the global properties. Instead of behavioural properties, Filieri *et al.* [16] are interested in verifying non-functional requirements on adaptive systems. They propose novel algorithms to check parametric Markov models. Combining our work with theirs is an interesting perspective, as it could allow us to quantify the impact of adding or removing features at runtime in terms of non-functional properties.

B. Software product line model checking

SPL model checking has become a hot topic that received a lot of attention in the last years. Li *et al.* [17] first proposed to model features as independent automata that are subsequently composed together to form a product. They provide incremental algorithms that can check whether the connection of a new feature preserves the properties of the system. Other work is based on modal transition systems where transition are either mandatory (included in all the configurations) or optional (only part of some products) [18], [19], [20]. Gruler *et al.* [21] extended the CCS process algebra with a variability operator and sketched an algorithm to verify a model against multi-valued μ-calculus formulae. In the last years, a family of approaches and tools based on featured transition systems were designed [7], [22], [23], [24], [25], [26]. In featured transition systems, transitions are constrained by Boolean formula over the features, such that the system can execute a transition only if its features satisfy the associated formula. This relation does not allow for reconfigurations and is thus a particular case of ours. Cordy *et al.* [4] extended this formalism with reconfiguration capabilities in order to model adaptive systems. Their work is closely related to this paper but there are also significant differences. First, the authors do not make the distinction between controlled and uncontrolled actions. They assume that non-deterministic transitions are executed by the environment, and are thus always uncontrolled. Second, they do not consider real-time, whereas adaptive systems are often require to adapt their behaviour within a given time frame and these adaptations may not be instantaneous. Third and last, no implementation is available.

IV. Featured Timed Games

This section introduces the mathematical model we propose to represent real-time adaptive systems. It includes a representation of an open environment with which the system interacts in real-time. This environment evolves over time, and the system must adapt its behavior to cope with these variations. To make our models concise and facilitate reasoning, we represent both the different functional modes of the system and the state of the environment with adaptive features, *i.e.,* features that can be enabled or disabled at runtime. In standard SPLE, features usually model design-time variability and are thus not meant to be modified at runtime. Our formalism considers these features as a particular case of adaptive features. Therefore, it is flexible enough to support product lines of real-time adaptive systems.

We first introduce the syntax of the model. Then we define its semantics as a timed game. We shall see that timed games are particularly suitable to reason on system's reconfigurations with regard to change in the environment. Beforehand, we recall basic concepts to formally represent runtime variability and real-time.

A. Encoding Variability and Real-Time Constraints

Variability. In SPLE, features usually designate units of difference between software products. We extend this notion to represent the possible adaptations of the system, as well as dynamic characteristics of the environment. Therefore, we distinguish between adaptive and static features, which may or may not change at runtime, respectively. Dependencies between features can be captured in a feature model. In this paper, we define a feature model as tuple $d = (F_s, F_a, F_e, [\![d]\!])$ where F_s is the set of features of the system, $F_a \subseteq F_s$ contains its adaptive features, and F_e denotes the features of the environment. We assume that F_s and F_e are disjoint and denote their union by F. A configuration of d is any subset of $F_s \cup F_e$. Therefore it denotes a particular variant of the system equipped with specific static and adaptive features, and deployed in a certain type of environment. Finally, $[\![d]\!] \subseteq \mathcal{P}(F)$, where $\mathcal{P}$ denotes the powerset, is the set of the valid configurations that satisfy the dependencies between the features.

To express that the possible behaviors of the system and the environment may depend on their features, we extend the notion of feature expressions borrowed from featured transition system (FTS) [7]. FTS extends labelled transition systems, such that a transition may only be triggered by a restricted set of configurations. Each transition is labelled with a feature expression, that is a Boolean function $exp : \mathcal{P}(F) \to \{\top, \bot\}$ such that $exp(p) = \top$ iff p can execute the transition. We denote by $[\![exp]\!] \subseteq \mathcal{P}(F)$ the set of configurations that satisfy exp and by $\top$ the feature expression such that $[\![\top]\!] = \mathcal{P}(F)$. Further in this section, we show how we generalize feature expressions to handle reconfiguration and how we combine them with time constraints.

Real-time. Timed Automata are an established formalism to represent real-time behavior. They extend labelled transition systems with real-time clocks of which value evolve as time passes. The clocks evolution and the discrete behaviors of the system are controlled by clock reset added to the transitions, and clock constraints. These constraints are either transitions *guards* that specifies when the system can execute a transition, or location *invariants* that defines when the system may remain in a given location. Examples of Timed Automata are shown in Fig. 1 to describe the models of the routing protocol.

Let C be a finite set of *clocks*. A *clock valuation* over C is a function $u : C \to \mathbb{R}_{\geq 0}$, that is, $u \in \mathbb{R}^C_{\geq 0}$. Given two valuations u and τ, we write $u+\tau$ for the valuation defined by $(u+\tau)(x) = u(x)+\tau(x)$. For $\lambda \in \mathcal{P}(C)$, we write $u[\lambda]$ for a valuation agreeing with u on clocks in $C\backslash\lambda$, and setting to 0 the clocks in λ. Let $\mathcal{B}(C)$ denote all *clock constraints* φ generated by the grammar $\varphi ::= x \prec k \mid x-y \prec k \mid \varphi\wedge\varphi$, where $k\in\mathbb{Q}$, $x, y \in C$ and $\prec \in \{<, \leq, >, \geq\}$. By $\mathcal{U}(C) \subset \mathcal{B}(C)$, we denote the set of constraints restricted to upper bounds and without clock differences. For $\varphi\in\mathcal{B}(C)$ and $u\in\mathbb{R}^C_{\geq 0}$, we write $u \models \varphi$ iff u satisfies φ. For $Z \subseteq \mathbb{R}^C_{\geq 0}$, we write $Z \models \varphi$ iff $u \models \varphi$

for all $u \in Z$. We write $[\![\varphi]\!]$ to denote the set of valuations that satisfy φ. Then $Z \subseteq \mathbb{R}^{C}_{\geq 0}$ is a *zone* iff $Z = [\![\varphi]\!]$ for some $\varphi \in \mathcal{B}(C)$.

To represent the behavior of system deployed in open environments, a model must distinguish between actions of the system from those of the environment. Timed Game Automata [6] are Timed Automata where actions are either controllable (actions of the system) or uncontrollable (actions of the environment). In this formalism, the satisfaction of properties is determined by solving a two-player timed game.

B. Featured Timed Game Automata

We are now ready to introduce *Featured Timed Game Automata* (FTGA) as a formalism to model product lines of real-time adaptive systems. FTGA result from the combination of the encodings presented above. It provides the following modelling facilities:

(1) **Open environment**. An FTGA distinguishes between controllable and uncontrollable transitions.
(2) **Real-time**. Clock constraints in invariants and transition guards model real-time constraints on the system and its environment.
(3) **Variability**. Each transition is constrained by a feature expression that defines in which configurations the system or its environment can execute it. It allows one to differentiate between the capabilities of every configuration.
(4) **Adaptations**. The transition relation also encodes which reconfigurations are possible upon the execution of an action by the system or its environment.

Formally, FTGA are defined as follows.

Definition 1 *An FTGA is a tuple* $\mathcal{G} = (Loc, l_0, C, Act, Inv, Trans, d, \gamma, AP, \mathcal{L})$ *where Loc is a finite set of locations,* $l_0 \in Loc$ *is the initial location, C is a finite set of clocks,* $Act = Act_c \uplus Act_e$*, is a finite set of actions partitioned between controllable actions in* Act_c *and uncontrollable actions in* Act_e*,* $Inv : Loc \to \mathcal{U}(C)$ *associates an invariant to each location,* $Trans \subseteq Loc \times Act \times \mathcal{B}(C) \times \mathcal{P}(C) \times Loc$ *is a set of transitions,* $d = (F_s, F_a, F_e, [\![d]\!])$ *is a feature model,* $\gamma : Trans \to (\mathcal{P}(F) \times \mathcal{P}(F) \to \{\bot, \top\})$*, specifies for each transition which configurations can execute it, and how the configuration of the system and the environment can evolve,* AP *is a finite set of atomic propositions,* $\mathcal{L} : Loc \to 2^{AP}$ *labels each location ith atomic propositions it satisfies.*

The adaptation process is encoded as part of function γ. This function is defined such that only adaptive features may only be changed by controllable transitions, and environment features may only be changed by uncontrollable transitions. Formally, let $\alpha = (l, a, \varphi, \lambda, l') \in Trans$. For any configurations c, c', e, e', if $a \in Act_c$, $\gamma(\alpha)(c \cup e, c' \cup e') \implies (c \setminus c') \cup (c' \setminus c) \subseteq F_a \wedge e' = e$ and if $a \in Act_e$, $\gamma(e_i)(c \cup e, c' \cup e') \implies c' = c$. Moreover, any reconfiguration of the system or the environment must ensure that the new configuration is valid, that is, $\gamma(\alpha)(c \cup e, c' \cup e') \implies c' \cup e' \in [\![d]\!]$. This function provides a flexible encoding to restrict the reconfiguration process. In particular, it is able to specify the minimum and maximum amount of time needed to transit from a given configuration to another one. To that aim, one may define a self-loop transition constrained by a given clock, and annotated with an action that represents the reconfiguration process.

C. Game semantics

An FTGA specifies the behavior of a set of systems, that is, one per valid configuration. The initial configuration of the system will determine how its behavior may evolve over time. Indeed, static features cannot be changed at runtime and thus fix parts of the system capabilities. Similarly, reconfiguration is not always doable; the initial value of adaptive features may thus impede the system to perform actions early in the execution, which may lead to unavoidable errors.

Accordingly, we define the semantics of an FTGA as a function $[\![.]\!] : \mathcal{P}(F_s) \to (Loc \times \mathbb{R}^{C}_{\geq 0} \times \mathcal{P}(F_s) \times \mathcal{P}(F_e))^*$ that associates an initial *system* configuration with its set of infinite executions. A *state* of the execution is a tuple $s = (l, u, c, e)$, where $l \in L$ is a location, $u \in \mathbb{R}^{C}_{\geq 0}$ is a clock valuation, $c \in \mathcal{P}(F_s)$ is a system configuration and $e \in \mathcal{P}(F_e)$ is an environment configuration such that $c \cup e \in [\![d]\!]$. An initial state is $(l_0, \mathbf{0}, c_0, e_0)$, where $\mathbf{0}$ is the valuation that initializes all clocks to zero, and c_0, e_0 are the initial configuration of the system and the environment, respectively. Whereas the configuration of the system is an input of the semantics function, the initial configuration of the environment is uncontrolled and is thus chosen non-deterministically. Since we consider timed systems, an execution includes two types of transitions:

- delay transitions: $(l, u, c, e) \xrightarrow{\tau} (l, u + \tau, c, e)$ if $\tau \in \mathbb{R}_{\geq 0}$ and $u + \tau \models Inv(l)$.
- discrete transitions: $(l, u, c, e) \xrightarrow{a} (l', u', c', e')$ if $a \in Act$ and $\exists \alpha = (l, a, \varphi, \lambda, l') \in Trans$, such that: $u \models \varphi$, $u' = u[\lambda]$ and $\gamma(\alpha)(c \cup e, c' \cup e') = \top$.

Finally, a *run* (or execution) in an FTGA is a sequence of states starting from an initial state and alternating delay and discrete transitions:

$$\rho = s_0 \xrightarrow{\tau_0} s'_0 \xrightarrow{a_1} s_1 \xrightarrow{\tau_1} s'_1 \xrightarrow{a_2} s_2 \ldots s_n \xrightarrow{\tau_n} s'_n \xrightarrow{a_{n+1}} s_{n+1} \ldots$$

Given that an FTGA considers continuous time, it specifies an infinite number of runs.

Among the transitions executed during a run, some are controlled by the system and others are uncontrolled, *i.e.* executed by the environment. Also, the system controls how it reconfigure itself, but has no control on the configuration of environment. The achievement of goals can thus be considered as a two-player games where the system plays against the environment. The strategy of one player prescribes a set of moves to perform according to the states previously visited. Each move consists of either delaying or executing an available action. A player can reconfigure itself only after executing an action. Formally, a strategy for the system is a function: $Str_C : (Loc \times \mathbb{R}^{C}_{\geq 0} \times \mathcal{P}(F_s) \times \mathcal{P}(F_e))^k \to (Act_c \times \mathcal{P}(F_s)) \cup \{\tau\}$ with $k \geq 0$. A strategy for the environment is defined symmetrically, except that the environment also selects its initial configuration. A strategy is valid iff (1) it complies with the transition relation and function γ, and (2) it does not lead to time-convergent or zeno runs [27]. From now on we consider valid strategies only.

The game proceeds as a concurrent game. In a given state, if one player chooses to delay while the other chooses an action, then this action is performed and the corresponding transition is triggered. If both players select an action then the transition to execute is chosen non-deterministically. Given a system strategy Str_C and an environment strategy Str_E, the possible outcomes of the game, noted $Outcome(Str_C, Str_E)$, are the set of infinite runs $\rho = s_0 \xrightarrow{\tau_0} s'_0 \xrightarrow{a_1} s_1 \xrightarrow{\tau_1} s'_1 \xrightarrow{a_2} s_2 \dots s_n \xrightarrow{\tau_n} s'_n \xrightarrow{a_{n+1}} s_{n+1} \dots$ such that

- if $a_i \in Act_c$ then $Str_C(s_0, \dots, s_i) = (a_i, c_{i+1})$.
- if $a_i \in Act_e$ then $Str_E(s_0, \dots, s_i) = (a_i, e_{i+1})$.
- if $\tau_i \in \mathbb{R}_{\geq 0}$ then $\forall \tau'_i \in [0, \tau_i[.\ s_i \xrightarrow{\tau} (l_i, u_i + \tau'_i, c_i, e_i)$ and $Str_C(s_0, \dots, s_i, (l_i, u_i + \tau'_i, c_i, e_i)) = Str_E(s_0, \dots, s_i, s_i + \tau'_i) = \{\tau\}$.

where $s_k = (l_k, u_k, c_k, e_k)$ for any $k \in \mathbb{N}$.

Example. Fig. 2 presents an FTGA modelling the routing protocol. The system actions are the plain transitions: *route, reconfig, t-reconfig*. The environment actions are the dashed transitions: *init, receive, encryption, sent*. The adaptive feature *encrypt* determines in which operation modes the system currently is. Two static features *p-reconf* and *t-reconf* specifies which of the two configuration methods the system can use (see Section II. Finally, the environment is described with a feature *safe* that specifies whether the current node in the network can be trusted or not. The function γ is defined in two steps. First, feature expressions are added in the graph to the guard of the transitions, in order to specify which set of features enables the transition. Second, we specify the possible reconfigurations:

- The system may only reconfigure the feature encrypt during the transitions labelled "*reconfig*".
- The environment may only reconfigure the feature safe during the transitions labelled "*sent*" and "*receive*".

In consequence, a possible strategy for the environment is to start in a *safe* configuration, do the *init* action at $y = 25$, then the *receive* action at $y = 30$, and disable the feature *safe* during this transition. In reaction, the system strategy can be to start with the system feature *p-reconf* while the adaptive feature *encrypt* is disabled, then wait until the environment reaches the location Received. At this point it can do a *reconfig* action immediately, and enable the feature *encrypt* during the transition. Finally, at $x = 10$ it performs the *route* action to reach the location RoutedUnsafe. The outcome produced by these two strategies is:

$$\left(\text{Init}, \begin{bmatrix} x = 0 \\ y = 0 \end{bmatrix}, \begin{Bmatrix} \text{p-reconf} \\ \text{safe} \end{Bmatrix}\right) \xrightarrow{25, init} \left(\text{Ready}, \begin{bmatrix} 25 \\ 25 \end{bmatrix}, \begin{Bmatrix} \text{p-reconf} \\ \text{safe} \end{Bmatrix}\right) \xrightarrow{5, receive}$$
$$\left(\text{Received}, \begin{bmatrix} 0 \\ 25 \end{bmatrix}, \{\text{p-reconf}\}\right) \xrightarrow{0, reconfig} \left(\text{Received}, \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{Bmatrix} \text{p-reconf} \\ \text{encrypt} \end{Bmatrix}\right) \xrightarrow{10, routed}$$
$$\left(\text{RoutedUnsafe}, \begin{bmatrix} 0 \\ 10 \end{bmatrix}, \begin{Bmatrix} \text{p-reconf} \\ \text{encrypt} \end{Bmatrix}\right)$$

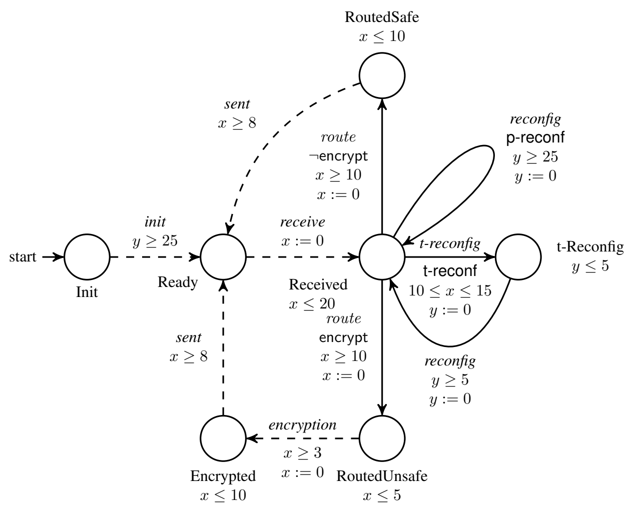


Figure 2: FTGA of the routing protocol.

V. Timed AdaCTL Model-Checking

To express requirements on real-time adaptive systems, we propose T-AdaCTL, a timed extension of the *Adaptive Configuration Time Logic* (AdaCTL), a logic we recently introduced to reason on reconfigurable systems. We first present its syntax and semantics, and then provide algorithms to check an FTGA against a T-AdaCTL formula.

A. Timed AdaCTL

The formulae of T-AdaCTL are embedded into three levels. The first level is the *feature formula*, which has the form $\Psi ::= [\chi]\Phi$ where χ is a feature expression and Φ is a *state formula*. Intuitively, $[\chi]\Phi$ defines that if the current configuration of the system and the environment satisfies χ, then the current state must satisfy Φ. Feature formulae can thus define requirements on specific configurations, or even forbid some others. A state formula has the form $\Phi ::= \top \mid a \mid \Psi_1 \wedge \Psi_2 \mid \neg\Psi \mid \mathcal{A}\varphi \mid \mathcal{E}\varphi$ where $a \in AP$, Ψ, Ψ_1 and Ψ_2 are feature formulae, and φ is a *path formula*. Intuitively, a state satisfies $\mathcal{A}\varphi$ (resp. $\mathcal{E}\varphi$) if from this state, the system can come up with a strategy of which the outcome will (resp. may) satisfy φ. The path formulae have the form $\varphi ::= \Psi_1\ \mathrm{U}_I\ \Psi_2 \mid \Psi_1\ \mathrm{W}\ \Psi_2$ where Ψ, Ψ_1 and Ψ_2 are feature formulae, I is an interval of $\mathbb{R}_{\geq 0}$ with integral bounds, U is called the *until* operator and W is called the *weak until* operator. T-AdaCTL extends AdaCTL with a time constraint attached to the until operator, in the same manner as TCTL [5] extends CTL. We omit the next operator of AdaCTL as there is no notion of direct successor in timed systems. Two path operators can be derived from U and W: *eventually* ($\Diamond$), such that $\Diamond_I\Psi = \top \mathrm{U}_I \Psi$, and *forever* ($\Box$), such that $\Box\Psi = \Psi \mathrm{W} \bot$. When it comes to state and path formulae, TATL [9] is a generalisation of T-AdaCTL, as it can express more general time constraints and requirements on the environment too. However, it does not include any notion of features, which makes it inappropriate for expressing properties on our feature-based formalism.

Example. Let us express the properties that the routing protocol must satisfy in T-AdaCTL. The property *"If the environment is unsafe, all the messages must be encrypted before they are sent."* can be expressed by the formula $A\Box([\neg\mathsf{safe}]\neg RoutedSafe)$. This formula specifies that the system can never reach the location $RoutedSafe$ if the environment is not *safe*. The property *"If the environment is safe, the messages must be sent at most 20 time units after being received."* can be expressed by the formula $A\Box([\mathsf{safe}]Received \Rightarrow A\Diamond[0,20]Ready)$. It specifies that whenever location $Received$ is reached in a safe environment, the location $Ready$ must be reached within 20 time units.

We provide T-AdaCTL with a formal semantics:

Definition 2 *Let $\mathcal{G}$ be an FTGA and $s = (l,u,c,e)$ one of its states. Then the satisfiability of a T-AdaCTL feature or state formula by $\mathcal{G}$ in state s is determined as follows:*

$$\begin{array}{lcl}
\mathcal{G},s \models [\chi]\Phi & \Leftrightarrow & c\cup e \notin [\![\chi]\!] \vee \mathcal{G},s\models\Phi \\
\mathcal{G},s \models \top & \Leftrightarrow & \top \\
\mathcal{G},s \models a & \Leftrightarrow & a \in \mathcal{L}(l) \\
\mathcal{G},s \models \Phi_1\wedge\Phi_2 & \Leftrightarrow & \mathcal{G},s\models\Phi_1 \wedge \mathcal{G},s\models\Phi_2 \\
\mathcal{G},s \models \neg\Phi & \Leftrightarrow & \mathcal{G},s\not\models\Phi \\
\mathcal{G},s \models \mathcal{E}\varphi & \Leftrightarrow & \exists Str_C\cdot\exists Str_E\cdot \\
\multicolumn{3}{l}{\quad\exists\rho\in Outcome(s,Str_C,Str_E)\cdot\mathcal{G},\rho\models\varphi} \\
\mathcal{G},s \models \mathcal{A}\varphi & \Leftrightarrow & \exists Str_C\cdot\forall Str_E\cdot \\
\multicolumn{3}{l}{\quad\forall\rho\in Outcome(s,Str_C,Str_E)\cdot\mathcal{G},\rho\models\varphi}
\end{array}$$

The semantics of path formulae is similar to that of TCTL path formulae:

$$\begin{array}{l}
\mathcal{G},\rho\models \Psi_1\ \mathrm{U}_I\ \Psi_2 \;\Leftrightarrow\; \exists r\in I\cdot\mathcal{G},\rho[r]\models\Psi_2 \wedge \\
\qquad\qquad\qquad\qquad \forall 0\le r'<r\cdot\mathcal{G},\rho[r']\models\Psi_1 \\
\mathcal{G},\rho\models \Psi_1\ \mathrm{W}\ \Psi_2 \;\Leftrightarrow\; (\forall r'\ge 0\cdot\mathcal{G},\rho[r']\models\Psi_1)\vee \\
(\exists r\ge 0\cdot\mathcal{G},\rho[r]\models\Psi_2 \;\wedge\; \forall 0\le r'<r\cdot\mathcal{G},\rho[r']\models\Psi_1)
\end{array}$$

where $\rho[r]$ the state reached in ρ at time r.

Note that we assume a continuous-time semantics for timed path operators [28]. We now define the satisfaction of a T-AdaCTL formula by an FTGA. Contrary to classical temporal logics, this relation, noted $\models_F$ is not Boolean: it is defined as the set of initial system configurations such that the FTGA satisfies the formula from its initial state.

Definition 3 *Let $\mathcal{G}$ be an FTGA and Ψ a T-AdaCTL formula.*

$$\begin{array}{l}
(\mathcal{G}\models_F\Psi) = \{c_0\in\mathcal{P}(F_s)\mid\exists e_0\in\mathcal{P}(F_e)\ \cdot\ c_0\cup e_0\in[\![d]\!]\ \wedge \\
\forall e_0\in\mathcal{P}(F_e)\ \cdot\ c_0\cup e_0\in[\![d]\!]\Rightarrow\mathcal{G},(l_0,\mathbf{0},c_0,e_0)\models\Psi\}
\end{array}$$

B. Model-Checking Algorithms

The semantics of T-AdaCTL is defined over execution paths, of which FTGA contain an infinite number. This means that a model checking procedure for T-AdaCTL must use a symbolic representation to capture this infinite number of runs in a finite data structure. To represent the time domain of symbolic states, we extend the grammar of clock constraints with the negation. Then for a clock constraint, φ, $[\![\varphi]\!]$ is a *federation*, *i.e.* a finite union of zones. In combination with federations, we use feature expressions to encode *sets* of configurations symbolically, as opposed to representing each configuration individually in separate states. Therefore, our algorithms manipulate *symbolic* states, *i.e.* tuples of the form (l,b,φ), where l is a location, b is a feature expression, φ is a clock constraint. A symbolic state is an abstraction of all the concrete states (l,u,c,e) such that $u\in[\![\varphi]\!]$ and $(c\cup e)\in[\![b]\!]$.

To model check a T-AdaCTL formula Ψ, we first decompose it into its *parse tree*, where each node is a subformula. The root is Ψ itself, whereas the leaves are atomic formulae. Then, starting from the leaves, we associate each subformula by the set of symbolic states that satisfy it. This method is similar to the one used to check CTL formulae [29].

We present how to compute the set of symbolic states that satisfy each form of T-AdaCTL formula. For feature and state formulae, the satisfaction rules are the following:

$$\begin{array}{rl}
Sat([\chi]\Phi) & = Sat(\Phi)\ \cup\{(l,\neg\chi,\top)\mid l\in Loc\} \\
Sat(\top) & = \{(l,\top,\top)\mid l\in Loc\} \\
Sat(a) & = \{(l,\top,\top)\mid a\in\mathcal{L}(l)\} \\
Sat(\Psi_1\wedge\Psi_2) & = \{(l,b_1\wedge b_2,\varphi_1\wedge\varphi_2)\mid \\
 & \quad (l,b_1,\varphi_1)\in Sat(\Psi_1)\ \wedge\ (l,b_2,\varphi_2)\in Sat(\Psi_2)\} \\
Sat(\neg\Psi) & = \overline{Sat(\Psi)}
\end{array}$$

where for any $S\in L\times\mathcal{P}(F)\times\mathbb{R}_{\ge 0}$, the complement of S is defined as $\overline{S} = \{(l,b,\varphi)\mid\nexists(l,b',\varphi')\in S\bullet[\![b]\!]\cap[\![b']\!]\neq\emptyset\wedge[\![\varphi]\!]\cap[\![\varphi']\!]=\emptyset\}$. Computing $Sat(\mathcal{E}\varphi)$ and $Sat(\mathcal{A}\varphi)$ comes down to solving a two-player game where the system is the verifier and the environment is the spoiler. To that aim, we perform a backward fixed-point computation as it is performed for solving timed games in [6]. The algorithms are based on *discrete predecessors* and *safe timed predecessors* operators. The definition of these operators in FTGA takes into account both variability and real-time, which makes it different from other game-based formalisms. It constitutes the cornerstone and the real novelty of our verification algorithms. Formally, let $\alpha=(l,a,\varphi_\alpha,\lambda_\alpha,l')\in Trans$ and (l',b',φ') be a symbolic state. We define the discrete predecessors $\mathrm{Pred}_\alpha(l',b',\varphi') = (l,b,\varphi)$ such that:

- $b=\{c\cup e\mid\exists(c'\cup e')\in b'\ \cdot\ \gamma(\alpha)(c\cup e,c'\cup e')=\top\}$
- $\varphi = free\big(\varphi'\wedge\{x=0\mid x\in\lambda_\alpha\},\lambda_\alpha\big)\wedge\varphi_\alpha\wedge Inv(l)$, where $free(\varphi,\lambda)=\{u\mid\exists v\in[\![\varphi]\!]\cdot\forall x\notin\lambda\cdot u(x)=v(x)\}$.

Observe that the distributivity law applies to this operator:

$$\mathrm{Pred}_\alpha\big(\bigcup_i s_i\big)=\bigcup_i\mathrm{Pred}_\alpha(s_i)$$

The discrete predecessors operator can be used to compute the controllable (resp. uncontrollable) moves that allow the system (resp. the environment) to reach (resp. to avoid) a winning state. However, these moves may not be safe as the other player may have concurrent moves. Formally, given a location l and the sets of winning states $Win[l']$ for each location l', these controllable moves are:

$$Next_c(l,Win)=\bigcup_{\alpha=(l,a\in Act_c,\varphi,\lambda,l')}\mathrm{Pred}_\alpha(Win[l'])$$

The uncontrollable moves of the environment are defined symmetrically. The winning moves are obtained through the *safe timed predecessors* operator. Let $s_1 = (l, b_1, \varphi_1)$ and $s_2 = (l, b_2, \varphi_2)$ be two symbolic states, the safe timed predecessors of s_1 wrt. s_2 are the states that can reach s_1 while avoiding any state from s_2. They are given by $\mathrm{Pred}_t(s_1, s_2) = \{(l, b_1 \wedge \neg b_2, \mathrm{Pred}_t(\varphi_1, \perp)), (l, b_1 \wedge b_2, \mathrm{Pred}_t(\varphi_1, \varphi_2))\}$ where $\mathrm{Pred}_t(\varphi_1, \varphi_2)$ is the safe timed predecessors operator for zones as defined in [6]. This operator has the following property:

$$\mathrm{Pred}_t\big(\bigcup_i g_i, \bigcup_j b_j\big) = \bigcup_i \bigcap_j \mathrm{Pred}_t(g_i, b_j).$$

In what follows, we denote by $\mathrm{Pred}(l)$ the locations from which there is a transition to l.

To enforce that the players' strategies are valid, we compute the deadlock states, which are the states beyond the locations invariant that should not be reached if one player had an urgent action to perform. We denote by $DL_c(l)$ (resp. $DL_e(l)$) the deadlock states in location l for which the system (resp. environment) is responsible.

Algorithm 1 computes $Sat(\mathcal{A}\Psi_1 \text{ U}_I \Psi_2)$. The algorithm starts with the winning symbolic states that satisfy Ψ_2 (Lines 3–7), and next performs a backward exploration (Lines 8–19) to discover predecessors of winning states that satisfy $\Psi 1$ and that the environment cannot impede the system to reach. To check that the time spent to reach the goal in Ψ_2 satisfies the interval constraint I, an additional clock, named $clock$, is added to the model; this is a standard way to handle timing constraints of logical formula. This extra clock is initialized in I (Line 4) and then decreases during the backward exploration. $Sat(\mathcal{A}\Psi_1 \text{ U}_I \Psi_2)$ is the set of winning states for which the value of the extra clock is zero (Lines 20–25).

Algorithm 2 executes a similar procedure to compute $Sat(\mathcal{A}\Psi_1 \text{ W } \Psi_2)$. In this case, however, it starts from the states that violate the formula (Line 3) and performs a backward exploration to compute the states from which the system cannot guarantee to avoid losing states (Lines 4–14). Then, the set of winning states for the system is the complement of the set of those states (Line 15).

The algorithms used to compute $Sat(\mathcal{E}\Psi_1 \text{ U}_I \Psi_2)$ and $Sat(\mathcal{E}\Psi_1 \text{ W } \Psi_2)$ also use similar procedures. The main differences are that the two players now cooperate in order to reach the goal expressed by the path formula. As an example, an algorithm for $Sat(\mathcal{E}\Psi_1 \text{ U}_I \Psi_2)$ can be obtained by adding $Next_e(l, Win)$ to set of states $Good$ in Line 11 of Algorithm 1, and by removing Lined 12-13 (hence setting the set of bad states to empty set in timed predecessors). Deadlock states must be avoided. Indeed, the two players cooperate. In case of a deadlock, they both lose.

Algorithm 1: $Sat(\mathcal{A}\Psi_1 \text{ U}_I \Psi_2)$

```
Input: $\mathcal{G}, \Psi_1, \Psi_2, I$
Output: $Sat(\mathcal{A}\Psi_1 \text{ U}_I \Psi_2)$
1  begin
       /* Initialisation */
2      $Wait \leftarrow \emptyset$;
3      for $s = (l, b, \varphi) \in Sat(\Psi_2)$ do
4          $\varphi \leftarrow \varphi \wedge clock \in I$;
5          $Win[l] \leftarrow s$;
6          $Wait \leftarrow Wait \cup \mathrm{Pred}(l)$;
7      end
       /* Backward exploration */
8      while ($Wait \neq \emptyset$) do
9          $l \leftarrow \mathsf{pop}(Wait)$;
10         $Good \leftarrow Win[l] \cup (DL_e(l) \cap \neg DL_c(l))$;
11         $Good \leftarrow Good \cup Next_c(l, Win)$;
12         $Bad \leftarrow Sat(\neg\Psi_1) \cap \neg Win[l]$;
13         $Bad \leftarrow Bad \cup Next_e(l, \overline{Win})$;
14         $NewWin \leftarrow \mathrm{Pred}_t(Good, Bad)$;
15         if $Win[l] \subsetneq NewWin$ then
16             $Win[l] \leftarrow NewWin$;
17             $Wait \leftarrow Wait \cup \mathrm{Pred}(l)$;
18         end
19     end
20     for $l \in Loc$ do
21         for $s = (l, b, \varphi) \in Win[l]$ do
22             $\varphi \leftarrow \varphi \wedge clock = 0$;
23             $\varphi \leftarrow free(\varphi, clock)$
24         end
25     end
26     return $Win$
27 end
```

Algorithm 2: $Sat(\mathcal{A}\Psi_1 \text{ W } \Psi_2)$

```
Input: $\mathcal{G}, \Psi_1, \Psi_2$
Output: $Sat(\mathcal{A}\Psi_1 \text{ W } \Psi_2)$
1  begin
2      $Wait \leftarrow \emptyset$;
3      $Win \leftarrow Sat(\neg\Psi_1) \cap Sat(\neg\Psi_2)$;
4      while ($Wait \neq \emptyset$) do
5          $l \leftarrow \mathsf{pop}(Wait)$;
6          $Bad \leftarrow Win[l] \cup DL_c(l)$;
7          $Bad \leftarrow Bad \cup Next_e(l, Win)$;
8          $Good \leftarrow \Psi_2 \cup Next_c(l, \overline{Win})$;
9          $NewWin \leftarrow \mathrm{Pred}_t(Bad, Good \setminus Bad)$;
10         if $Win[l] \subsetneq NewWin$ then
11             $Win[l] \leftarrow NewWin$;
12             $Wait \leftarrow Wait \cup \mathrm{Pred}(l)$;
13         end
14     end
15     return $\overline{Win}$
16 end
```

VI. Implementation

Our modelling formalism and the associated algorithms have been implemented on top of PyECDAR [11], a tool for the analysis of timed systems.

In PyECDAR, a model is written in an XML file that follows the format of UPPAAL tool set [30]. This allows us to reuse the intuitive user interface provided by UPPAAL. In our extension, we use two variables for each adaptive features that defines its value before and after the reconfiguration following the transition. This encoding is sufficient to entirely represent the function γ. Additional patterns can be used to facilitate the design of the system. For example, feature expressions can be used in the invariant of a location, which offer another way to specify possible reconfigurations. Similarly, assignments can be used to forbid the reconfiguration of an adaptive feature during a transition.

The original game algorithms of PyECDAR were limited to safety and reachability objectives specific to timed specifications. We have implemented the new game algorithms presented in this paper and the recursive procedure that checks a T-AdaCTL formula. To encode continuous time, we use federations, which are finite unions of DBMs, implemented in UPPAAL DBM Library. We encode feature expressions with BDDs, implemented using PyCUDD library (python bindings for CUDD library [31]). Using these two libraries we have implemented a new encoding for symbolic states with both features and time domains, as well as an encoding for finite unions of symbolic states. Therefore, the different operators (union, intersection, negation, discrete predecessors, timed predecessors) are implemented in PyECDAR for unions of symbolic states.

Example. We consider again the routing protocol modelled in Fig. 2. We first use PyECDAR to check it against the T-AdaCTL formula $\Psi_1 = A\Box([\neg\mathsf{safe}]\neg RoutedSafe)$. PyECDAR computes the satisfaction relation for the formula, which is given by *p-reconf* $\vee$ *t-reconf* $\vee$ *encrypt*. It means that the formula is satisfied iff any of the two reconfiguration features are enabled, or feature *encrypt* is initially enabled. Then, we verify the formula $\Psi_2 = A\Box([\mathsf{safe}]Received \Rightarrow A\Diamond[0,20]Ready)$, and we obtain the following result: *p-reconf* $\vee$ $\neg$*encrypt*. Finally, we consider the formula

$$\Psi_3 = A\Box\big(([\neg\mathsf{safe}]\neg RoutedSafe) \wedge ([\mathsf{safe}]Received \Rightarrow A\Diamond[0,20]Ready)\big)$$

that combines the previous ones. The satisfaction relation is now restrained to *p-reconf*, which proves that in order to satisfy both properties at the same time the system requires the *p-reconf* feature. Note that this result is not the same as the conjunction of the two previous results. Indeed, solving the formula $\Psi_1 \wedge \Psi_2$ comes down to finding configurations in which the system has either strategies to satisfy Ψ_1 or strategies to satisfy Ψ_2, but there may exist no strategy that satisfies both goals.

VII. Conclusion

This paper presents a new formal model for highly adaptive real-time systems. Our approach relies on a combination of adaptive feature transition systems with timed automata. The semantics of our model is given as a timed game, which views the system and the environments as concurrent entities. In our setting, requirements are expressed in a new logic called T-AdaCTL, for which we provide a model checking procedure. We have implemented our approach as an extension of the PyECDAR toolset. The new tool has been applied to academic case studies. Our next objective is to evaluate our approach on real-life case studies.

References

[1] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching-time temporal logic," in *Logic of Programs*, ser. LNCS, vol. 131. Springer, 1981.

[2] B. H. C. Cheng, R. Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. Marzo Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. M. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. A. Müller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. Whittle, "Software engineering for Self-Adaptive systems: A research roadmap," in *Software Engineering for Self-Adaptive Systems*, ser. LNCS. Springer Berlin / Heidelberg, 2009, ch. 1.

[3] J. Zhang, H. J. Goldsby, and B. H. Cheng, "Modular verification of dynamically adaptive systems," in *Proceedings of AOSD '09*. New York, NY, USA: ACM, 2009.

[4] M. Cordy, A. Classen, P. Heymans, A. Legay, and P.-Y. Schobbens, "Model checking adaptive software with featured transition systems," in *Assurances for Self-Adaptive Systems*, ser. LNCS. Springer, 2013.

[5] R. Alur and D. L. Dill, "A theory of timed automata," *Theor. Comput. Sci.*, vol. 126, no. 2, 1994.

[6] O. Maler, A. Pnueli, and J. Sifakis, "On the synthesis of discrete controllers for timed systems (an extended abstract)," in *STACS*, 1995.

[7] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin, "Model checking lots of systems: efficient verification of temporal properties in software product lines," in *ICSE'10*. ACM, 2010.

[8] R. Alur, T. A. Henzinger, and O. Kupferman, "Alternating-time temporal logic," *J. ACM*, vol. 49, no. 5, 2002.

[9] T. A. Henzinger and V. S. Prabhu, "Timed alternating-time temporal logic," in *FORMATS*, ser. LNCS, vol. 4202. Springer, 2006.

[10] F. Cassez, A. David, E. Fleury, K. G. Larsen, and D. Lime, "Efficient on-the-fly algorithms for the analysis of timed games," in *CONCUR*, ser. LNCS, vol. 3653. Springer, 2005.

[11] A tool for timed games and timed specifications, "PyECDAR," 2011, https://project.inria.fr/pyecdar.

[12] R. Allen, R. Douence, and D. Garlan, "Specifying and analyzing dynamic software architectures," in *Proceedings of FASE '98*, Lisbon, Portugal, March 1998, pp. 21–37.

[13] J. Kramer and J. Magee, "Analysing dynamic change in software architectures: A case study," in *CDS'98*. IEEE Computer Society, 1998, pp. 91–100.

[14] S. Kulkarni and K. Biyani, "Correctness of component-based adaptation," in *Proceedings of CBSE '04*, ser. Lecture Notes in Computer Science, I. Crnkovic, J. Stafford, H. Schmidt, and K. Wallnau, Eds., vol. 3054. Springer Berlin / Heidelberg, 2004, pp. 48–58.

[15] J. Zhang and B. H. Cheng, "Using temporal logic to specify adaptive program semantics," *Journal of Systems and Software*, vol. 79, no. 10, pp. 1361 – 1369, 2006.

[16] A. Filieri, C. Ghezzi, and G. Tamburrelli, "Run-time efficient probabilistic model checking," in *Proceedings of ICSE '11*, 2011, pp. 341–350.

[17] H. C. Li, S. Krishnamurthi, and K. Fisler, "Interfaces for modular feature verification," in *Proceedings of ASE'02*, 2002, pp. 195–204.

[18] D. Fischbein, S. Uchitel, and V. Braberman, "A foundation for behavioural conformance in software product line architectures," in *ROSATEA'06, ISSTA 2006 workshop*. ACM Press, 2006, pp. 39–48.

[19] A. Fantechi and S. Gnesi, "Formal modeling for product families engineering," in *SPLC*, 2008, pp. 193–202.

[20] P. Asirelli, M. H. ter Beek, A. Fantechi, and S. Gnesi, "A logical framework to deal with variability," in *Proceedings of the 8th international conference on Integrated formal methods*, ser. IFM'10. Berlin, Heidelberg: IEEE, 2010, pp. 43–58.

[21] A. Gruler, M. Leucker, and K. Scheidemann, "Modeling and model checking software product lines," in *FMOODS'08*. Springer, 2008, pp. 113–131.

[22] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay, "Symbolic model checking of software product lines," in *ICSE'11*. ACM, 2011, pp. 321–330.

[23] A. Classen, M. Cordy, P. Heymans, A. Legay, and P.-Y. Schobbens, "Model checking software product lines with SNIP," *STTT*, vol. 14, no. 5, pp. 589–612, 2012.

[24] M. Cordy, A. Classen, G. Perrouin, P. Heymans, P.-Y. Schobbens, and A. Legay, "Simulation-based abstractions for software product-line model checking," in *ICSE'12*. IEEE, 2012, pp. 672–682.

[25] M. Cordy, P. Heymans, P.-Y. Schobbens, and A. Legay, "Behavioural modelling and verification of real-time software product lines," in *SPLC'12*. ACM, 2012.

[26] M. Cordy, P.-Y. Schobbens, P. Heymans, and A. Legay, "Beyond boolean product-line model checking: Dealing with feature attributes and multi-features," in *ICSE'13 (to appear)*. IEEE, 2013.

[27] L. de Alfaro, M. Faella, T. A. Henzinger, R. Majumdar, and M. Stoelinga, "The element of surprise in timed games," in *CONCUR*, ser. LNCS, vol. 2761. Springer, 2003.

[28] P. Bouyer, "Model-checking timed temporal logics," *Electr. Notes Theor. Comput. Sci.*, vol. 231, 2009.

[29] A. Pnueli, "The temporal logic of programs," in *FOCS'77*, 1977.

[30] G. Behrmann, A. David, K. G. Larsen, P. Pettersson, and W. Yi, "Developing uppaal over 15 years," *Softw., Pract. Exper.*, 2011.

[31] Colorado University Decision Diagram Package, "CUDD," http://vlsi.colorado.edu/~fabio/CUDD/.