Behavioural modelling and verification of real-time software product lines

Cordy, Maxime; Schobbens, Pierre-Yves; Heymans, Patrick; Legay, Axel

Published in:
Proceedings of the 16th International Software Product Line Conference (SPLC '12), Salvador, Brazil, September 2-7

DOI:
10.1145/2362536.2362549

Publication date:
2012

Document Version
Early version, also known as pre-print

Behavioural Modelling and Verification of Real-Time Software Product Lines

Maxime Cordy*,
Pierre-Yves Schobbens
PReCISE Research Center
University of Namur, Belgium.
{mcr,pys}@info.fundp.ac.be

Patrick Heymans
PReCISE Research Center,
University of Namur, Belgium.
INRIA Lille-Nord Europe –
Université Lille 1, France.
LIFL – CNRS , France.
phe@info.fundp.ac.be

Axel Legay
INRIA Rennes, France
Aalborg University, Denmark
University of Liège, Belgium.
axel.legay@inria.fr

ABSTRACT

In Software Product Line (SPL) engineering, software products are build in families rather than individually. Many critical software are nowadays build as SPLs and most of them obey hard real-time requirements. Formal methods for verifying SPLs are thus crucial and actively studied. The verification problem for SPL is, however, more complicated than for individual systems; the large number of different software products multiplies the complexity of SPL model-checking. Recently, promising model-checking approaches have been developed specifically for SPLs. They leverage the commonality between the products to reduce the verification effort. However, none of them considers real time.

In this paper, we combine existing SPL verification methods with established model-checking procedures for real-time systems. We introduce Featured Timed Automata (FTA), a formalism that extends the classical Timed Automata with constructs for modelling variability. We show that FTA model-checking can be achieved through a smart combination of real-time and SPL model checking.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Model checking*

General Terms

Theory, Verification

Keywords

Model Checking, Software Product Lines, Features, Real-Time.

*FNRS research fellow, Project FC 91490

SPLC '12, September 02 - 07 2012, Salvador, Brazil

1. INTRODUCTION

Software product line (SPL) engineering is an increasingly widespread development paradigm that aims at building *families* of software products instead of individual systems. Within a software family, a product can be represented by its set of *features*, which are units of difference between products. In the recent years, a large variety of systems have been developed as SPLs, including in critical areas like the aerospace, automobile and avionics industries. Providing SPL engineers with quality assurance techniques is thus of utmost importance since those must increase confidence that *all* the products they build work properly.

Model checking [4] is a well-known automated verification technique for complex systems. Given a behavioural model of a system under verification and a property expressed in some temporal logic, a model checker determines if there exists an execution of the model that violates the property. If such an execution exists, then the corresponding trace is returned and eventually used to improve the design.

The model checking problem for SPLs is much more complex than for single systems. Indeed, in addition to the inherent complexity of verification, the model checker has to deal with a potentially huge number of products. Conceptually, verifying an SPL comes to identifying *all* the software products that fail to meet their requirements and intended properties [9]. A simple but cumbersome method for addressing this problem is to reuse existing techniques to check each product individually. This "enumerative approach" is clearly impractical for designs with hundreds, or sometimes thousands, of product variants [9, 8].

In the recent years, we have observed the emergence of new model-checking techniques specific to product lines [19, 13, 9]. Some of these approaches have shown to significantly decrease the verification effort. In particular, we have proposed a new technique that relies on Feature Transition Systems (FTS) [8, 9], a compact representation for SPL behaviour. FTS are classical transition systems in which transitions are annotated with constraints on features. A transition is then executable only by the products that satisfy its associated constraints. The major contribution of our work is to exploit the structure of FTS in order to reduce the computational complexity of model-checking.

Unfortunately, dedicated SPL model-checking techniques still fail to handle SPLs whose behaviour depends on real-time constraints. The objective of this paper is to provide the first approach to model and verify real-time SPLs. Our

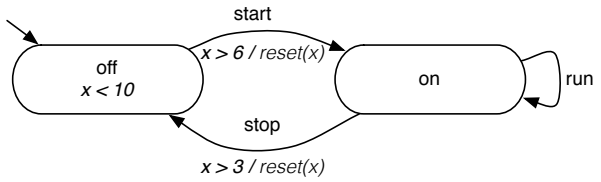

Figure 1: An example of Timed Automaton.

new technique relies on a smart combination of FTS model-checking with existing well-known real-time model-checking approaches. More precisely, we propose a featured extension of Timed Automata (TA) [12], the classical model for timed systems, that builds on FTS. An example of TA is given in Figure 1. Like in usual transition systems, the possible states of the modelled system are represented by a set of *locations* (`off` and `on`). Transitions between locations, modelled by arrows, describe how the state of the system can evolve. For example, if the system is in location `off` and triggers the action `start`, it will reach location `on`. In addition to these constructs, TA model time with a non-empty set of *clocks* used to characterize time passing. We use so-called *clock constraints* to restrict the intervals of time during which the system can idle in a location or execute a transition to leave this location. For instance, the system can remain in location `off` only when the value of clock `x` is less than 10. Similarly, it can move to location `on` when the value of `x` is greater than 6. Overall, it means that the system always remain in location `off` between 6 and 10 time units. Upon the execution of a transition, the value of clocks can be reset to zero (e.g., see transitions `start` and `stop`).

Contributions. TA have become a popular formalism for reasoning over the behaviour of continuous-time systems. The main reasons are that TA model-checking remains decidable for branching-time logics, and efficient solutions can be implemented. Moreover, renown implementations like UPPAAL [5] managed to catch the interest of both academics and practitioners. This increases our confidence that TA are suitable in the context of verifying properties of real-time SPLs as well. The first contribution of this paper is *Feature Timed Automata* (FTA), a new formalism that extends timed automata with variability just like FTS extends transition systems. In the subsequent sections, we formally define both the syntax and semantics of FTA, and study how features may influence both timed and discrete behaviours of such models. We also study several ways of modelling variability by exploiting the structure of timed automata. One of our major contributions is to combine the principles of TA and FTS model checking into new verification algorithms specifically designed for FTA. We illustrate our approach on a running example and compare, using a prototype tool, the efficiency of our algorithms with regard to a product-by-product verification procedure. *In fine*, we assess that our new modelling formalism opens the path to the rigorous design of real-time SPLs.

Structure of the paper. We first summarize the concepts of existing SPL model-checking approaches in Section 2. Next, we intuitively present our formalism through an introductory example in Section 3. FTA and their surrounding theory are properly introduced in Section 4. In Section 5, we discuss the verification algorithms. Finally, Section 6 presents a prototype tool and some experimental results.

2. RELATED WORK

Nowadays, SPL verification is a hot topic that brings together the formal methods and SPL engineering communities. In consequence, many different approaches for modelling and verifying SPLs have appeared. We may distinguish between compositional and annotative methods [16].

Compositional approaches model and reason over features in isolation. Li *et al.* [19] propose to model a base system and each feature as independent finite state machines that cling to each other. Then they can verify if a given combination of features preserves the properties satisfied by the base system alone. Although their approach is suitable for incremental feature-oriented analyses, it is not really convenient for identifying all the products violating a property because it would require trying every combination of features.

Contrary to compositional approaches, annotative methods represent the behaviour of all the software variants in one common model. Modal Transition Systems [13, 3] fall into this former category. Basically, these are transition systems with compulsory and optional transitions. Although they are suitable for representing optional behaviour, they lack an explicit notion of variability and are thus unable to keep track of the different products during a verification. Lauenroth *et al.* [18] model the behaviour of SPLs using automata labelled with features. Their formalism is limited because they do not allow to label transition with any combination of features. Also, their model-checking algorithms do not visit the state-space of the models efficiently. Gruler *et al.* [14] extend CCS with a new operator for modelling variability. They sketch a model-checking procedure for the multi-valued modal μ-calculus.

More recently, Apel *et al.* [2] propose to model features in a compositional manner but to model check them following an annotative approach. More precisely, each feature is modelled as an isolate unit. These separate models are combined to form a single FTS-like model. Then a tool detects if one product violates safety properties because of unexpected interactions between features. As our FTS-based algorithms, the model-checking procedure avoids instantiating a model of every product to perform the verification.

In our previous work, we proposed FTS [9] to model the behaviour of SPLs in an annotative fashion. The foundations of this formalism are increasingly extended and now offer a range of model-checking algorithms and tools [9, 8]. More recently, we generalized the definition of behavioural simulation to FTS [11].Thanks to their nice expressiveness combined with efficient algorithms, FTS have proven to be one of the most promising approaches towards the modelling and the verification of discrete SPLs.

3. INTRODUCTORY EXAMPLE

In real-time SPLs, features may influence the behaviour of a system in various ways. First, features may add or remove capabilities like in discrete SPLs. Second, they may also influence the timed constraints that determine the minimum and maximum delays for executing a given action or remaining in a given state. We have to take these possibilities into account in the definition of our formalism.

The objective of this section is an informal introduction

to modelling with FTA through a simplification of the well-known mine pump example [17]. For now, we consider only a closed model of the pump itself and ignore the other components of the system.

We consider that this SPL has three optional features *Button* (B), *FastStart* ($Fstart$), and *FastStop* ($Fstop$). *Button* defines that the pump can be equipped with a button used to start and stop the engine. If *FastStart* (resp. *FastStop*) is enabled, then less time is required for the pump to start (resp. stop). This small SPL has a total of eight products.

Figure 2 gives a graphical representation of an FTA modelling the behaviour of the pump SPL. As in TA, the observable states of the system are modelled by locations. There are only two locations, `on` and `off`, which models that the pump is started and stopped, respectively. The pump can transit between these locations if and only if it has the feature B. Initially, the pump is in the `off` location. At some point, the pump must be started, that is, the pump cannot remain in the `off` location indefinitely. As for TA, we model this restriction with a *location invariants*. In FTA, however, invariants may depend on variability. For example, we should be able to express the following requirement :

> "The maximum amount of time the pump may remain in `off` location depends on whether or not *FastStart* is enabled. With this feature, the maximum time is 7 time units, otherwise it is 10 time units."

In order to model the above requirement, we combine the notion of feature with location invariants. More precisely, we define an invariant as the conjunction of clock constraints annotated with feature expressions, henceforth named *featured clock constraint*. Then, only the products satisfying the feature expression have to satisfy the associated clock constraint. In our example, the invariant of location `off` is given by $x < 7$ for the products having the feature *FastStart*; for the others, the invariant is given by $x < 10$. Accordingly, we specify the invariant of location `off` as

$$[FStart](x < 7) \wedge \neg[FStart](x < 10).$$

As for invariants, variability influences the executability of transitions and their associated clock constraints. As mentioned above, the feature *Button* is needed to execute the transitions `start` and `stop`. Consequently, these transitions are annotated with that feature. On the contrary, the transition `run` does not require the feature *Button* to be executed and is thus annotated with the feature expression `True`, which is satisfied by every product. Another requirement for the pump is the need for a preheating time before it begins to run. This delay depends on the feature $FastStart$. We can express that constraint as follows :

> "The pump must remain at least in 6 time units in the `off` location before starting. With the feature *FastStart*, this delay is reduced to 4 time units."

As for before, we may model this requirement by guarding the transition with featured clock constraints. In such case, the constraint is

$$[FStart](x > 4) \wedge \neg[FStart](x > 6).$$

The transition `stop` is constrained similarly. We thus annotate that transition with another featured clock constraint.

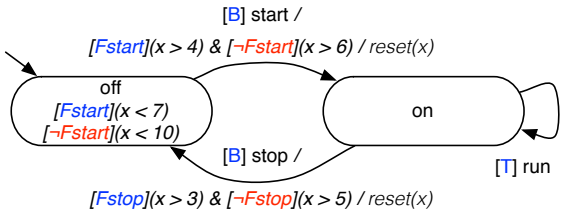

Figure 2: FTA modelling the pump SPL.

Overall this model defines that the pump always remains in location `off` between 4 and 7 time units if the feature $Fstart$ is enabled, and between 6 and 10 time units otherwise.

4. FEATURED TIMED AUTOMATA

This section formalizes the syntax and the semantics of Featured Timed Automata. The generalization from TA to FTA is similar to our previous work on Featured Transition Systems [9, 8]. However, a notable difference lies in the real-time nature of FTA. Not only features have an impact on the availability of transitions, but they may also influence the time constraints that drive the behaviour of the modelled system. Before we define FTA, we provide preliminary modelling constructs for expressing both discrete and real-time variability.

4.1 Encoding Real-Time Variability

As mentioned in the previous sections, we use *features* to capture the variability in SPLs and *feature models* to represent the set of their valid combinations. In this paper, we assume that products are represented by sets of Boolean features. This allows us to abstract from the complexity of feature models by defining them as a couple $(F, P \subseteq \mathcal{P}(F))$, where F is a set of features and P is the set of valid products. The semantics of a feature model d, noted $[\![d]\!]_{FD}$, is then its set of valid products.

We encode sets of products with *feature expressions* [8], which are boolean expressions defined over a set of features. Formally, a feature expression b is a total function $\{\top, \bot\}^{|F|} \rightarrow \{\top, \bot\}$ that associates every combination of feature with a truth value. For a feature model $d = (\{f_1, \ldots, f_{|F|}\}, P)$ and a feature expression b, a product p satisfies b iff

$$b(b_1, \ldots, b_{|F|}) = \top$$

where $b_i = (f_i \in p)$ for $i = 1..|F|$. Then, we denote by $[\![b]\!]$ the set of products that satisfy b. We also encode a set of products px by a feature expression noted $\mathbb{B}(px)$.

Now that we are able to encode sets of products as feature expressions, we may give a formal definition to the aforementioned featured clock constraints.

Definition 1 *A* Featured Clock Constraint *over a set F of features and a set C of clocks is a formula of the form*

$$g ::= \top \mid [\chi](c < n) \mid [\chi](c \leq n) \mid [\chi](c > n) \mid [\chi](c \geq n) \mid g \wedge g$$

where $c \in C, n \in \mathbb{N}$, and χ is a feature expression over F.

Intuitively, $g = [\chi]g'$ means that g' must hold for products in $[\![\chi]\!]$ – the others do not have to satisfy the constraint. Also note that we can use $[\chi](g_1 \wedge g_2)$ as a shortcut for $[\chi]g_1 \wedge [\chi]g_2$.

We denote by $FCC(N, C)$ the set of featured clock constraints over N and C and by $Eval(C)$ the set of clock evaluations, that is the set of functions $\eta : C \to \mathbb{R}^+$ that assign a positive real value to every clock. The satisfiability relation for featured clock constraints is a function $\models: \mathcal{P}(\mathcal{P}(F)) \times Eval(C) \times FCC(N, C) \to \{\top, \bot\}$ recursively defined as follows:

$$(p, \eta) \models g \Leftrightarrow \begin{cases} (p, \eta) \models g_1 \wedge (p, \eta) \models g_2, & g = g_1 \wedge g_2, \\ p \in [\![\neg\chi]\!] \vee \eta \models g', & g = [\chi]g'. \end{cases}$$

When $p \notin [\![\chi]\!]$, p is said to trivially satisfy any featured clock constraint of the form $[\chi]g$. Note that a classical (non-featured) clock constraint g can be regarded as a featured clock constraint where each feature expression is a tautology.

As an example, the featured clock constraint $[FastStart](x > 4)$ (see Figure 2) is satisfied by a product p and a clock valuation η iff p does not have feature $FastStart$ or clock `x` has a value higher than 4 in η.

4.2 Syntax and Semantics of FTA

The notions of feature expression and featured clock constraint allow us to model the impact of features on real-time models. By combining them with the usual definition of timed automaton [12], we obtain a new formalism to model the behaviour of real-time SPLs.

Definition 2 *A Featured Timed Automaton is a tuple* $(Loc, Act, C, trans, Loc_0, Inv, AP, L, d, \gamma)$ *such that* Loc *is a finite set of locations,* Act *a finite set of actions,* C *is a finite set of clocks,* $Loc_0 \in Loc$*, is the set of initial locations,* $trans \subseteq Loc \times FCC(N, C) \times Act \times C \times Loc$ *is a finite set of transitions,* $Inv : Loc \to FCC(N, C)$ *is a partial invariant function that associates featured clock constraints to locations,* AP *is a set of atomic propositions,* $L : Loc \to AP$ *is a total function labelling locations with atomic propositions,* $d = (F, P \subseteq \mathcal{P}(F))$ *a feature model,* $\gamma : trans \to \{\top, \bot\}^{|F|} \to \{\top, \bot\}$ *is a total function that labels transitions with feature expressions.*

According to this definition, we consider that a feature may modify transitions availability, transition guards, and location invariants. As for FTS [8], the function γ associates a transition f with a feature expression such that $[\![\gamma(t)]\!]$ is the set of products able to execute t. We model the variability of transition guards and location invariants with featured clock constraints, as opposed to classical clock constraints. For example, if $[\neg FastStart](x < 10)$ is an featured clock constraint occurring in the invariant of a location, then the system is forbidden to be in that location when the value of x is greater than 10. This prohibition, however, holds *only* for the products that do *not* have the feature $FastStart$.

One can easily relate the above definition with the example shown in Figure 2. However, be aware that we have omit to display the atomic propositions in order to increase the readability of the figure.

Thanks to the above constructs, FTA is a convenient formalism to model the behaviour of a set of products. Besides, from an FTA we can derive a TA that models the behaviour of a specific product. To achieve this, we define a so-called *projection* operator [9]. Basically, the projection of an FTA onto a product p is obtained by (1) removing the transitions unavailable to p, (2) replacing any featured clock constraint $[\chi]g$ by g such that $\chi(p) = \top$, and (3) discarding the others.

Definition 3 *The projection of an FTA* $fta = (Loc, Act, C, trans, Loc_0, Inv, AP, L, d, \gamma)$ *to a product* $p \in [\![d]\!]_{FD}$ *is the TA* $(Loc, Act, C, trans', Loc_0, Inv', AP, L)$ *with*

$$Inv'(l) = Inv(l)_{|p}, \forall l \in Loc$$
$$trans' = \{t = (l, g_{|p}, \alpha, R, l') \mid t \in trans \wedge p \in [\![\gamma(t)]\!]\}$$

where the projection of a featured clock constraint $[\chi]g$ *to a product* p *is recursively defined as*

$$g_{|p} = \begin{cases} (g_1)_{|p} \wedge (g_2)_{|p}, & g = g_1 \wedge g_2, \\ g', & g = [\chi]g' \wedge p \in [\![\chi]\!], \\ \top & \textit{otherwise.} \end{cases}$$

For example, Figure 1 is the TA resulting from the projection of the FTA shown in Figure 2 to the product $\{B, Fstop\}$.

Each TA resulting from the projection of an FTA onto a specific product exactly models the behaviour of that product. Given that an FTA models a set products, we define its semantics as a function that associates every valid product with the semantics of its projection.

Definition 4 *The semantics of an FTA* $fta = (Loc, Act, C, trans, Loc_0, Inv, AP, L, d, \gamma)$ *is a function* $[\![fta]\!]_{FTA}$ *such that*

$$\forall p \in [\![d]\!]_{FD} \bullet [\![fta]\!]_{FTA}(p) = [\![fta_{|p}]\!]_{TA}$$

where $[\![.]\!]_{TA}$ *denotes the semantics of TA.*

For instance, the FTA presented in Figure 2 associates the product $\{B, Fstop\}$ to the TA shown in Figure 1.

4.3 Alternate Definition of FTA

Consider again the FTA shown in Figure 2. We observe that the transition `start` from `off` to `on` could have been defined without the use of featured clock constraints. In this case, the transition would have to be split into two transitions, one for the products having the feature $FastStart$ with the guard $x > 4$, and one for the other products with the guard $x > 6$ (see Figure 3 for an illustration). The above reasoning thus rises the question of studying a possibly equivalent form of FTA where feature constraints have been removed. The contribution of this section is to show that we can transform every FTA into an equivalent FTA without featured clock constraint. Of course, such a transformation may be computationally costly. Depending on whether we consider transition guards or location invariants, the transformation procedure is different.

Let us first consider the former case (*e. g.* the transition `start` in Figure 2). Let $t = (l, g, \alpha, R, l')$ be a transition. The idea of the transformation is to create copies of the transitions such that a given product p can execute only one of the copies and the guard of this transition is a classical clock constraint equivalent to the projection of the guard of the original transition onto p. The original transition is then removed. The transformation is achieved as follows:

1. Separate products into groups such that two products p_1 and p_2 are in the same groups if and only if the projections of the transition guard onto p_1 and p_2 are equivalent. This results in a set of set of products $\{G_1, \ldots, G_k\} \subseteq \mathcal{P}(\mathcal{P}(F))$ and a set of classical clock constraints $\{c_1, \ldots, c_k\}$ such that c_i is the clock constraint resulting from the projection of the guard onto a product in G_i.

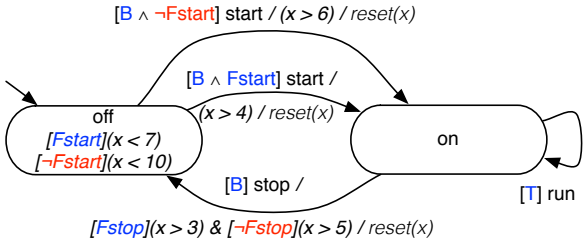

Figure 3: FTA with a transformed transition guard.

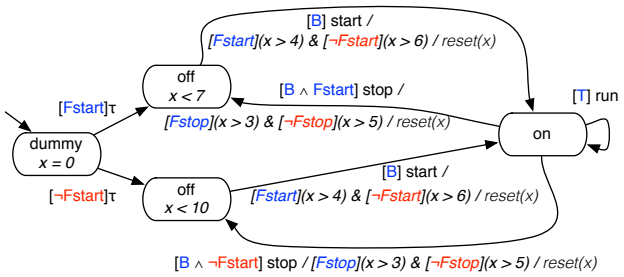

Figure 4: FTA with a transformed invariant.

2. Remove t from $trans$.

3. For each group G_i add $t_i = (l, c_i, \alpha, R, l')$ in $trans$ and set $\gamma(t_i) = \gamma(t) \wedge \mathbb{B}(G_i) \wedge \bigwedge_{j \neq i} \neg\mathbb{B}(G_j)$.

For the transition `start`, we would have two groups, namely the products that have the feature $Fstart$ and the ones that do not. The corresponding clock constraints are $x > 4$ and $x > 6$, respectively. As shown in Figure 3, this results in two new transitions : the first has the guard $x > 4$ and is executable by the products in $[\![B \wedge Fstart]\!]$; the second has the guard $x > 6$ and is executable by the products in $[\![B \wedge \neg Fstart]\!]$. This new FTA has the same semantics as the one shown in Figure 2.

The second transformation, which removes features constraints from a location invariant, also relies on duplication. However, we now have to duplicate both the location and its incoming transitions. Let l be this location. Like we did in the previous transformation, we separate the products into groups $G_1, \ldots, G_k$ such that there is a bijection between this set of groups and the distinct projections $c_1, \ldots, c_k$ of the location invariant. Then we perform the following steps:

1. If l was in Loc_0, add a location l_0 in Loc_0 with $\eta \models Inv(l_0) \Leftrightarrow \forall c \in C \bullet \eta(c) = 0$.

2. For each group G_i do:

   (a) Create a copy l_i of l, add it in Loc and remove l.

   (b) For every $t = (l, g, \alpha, R, l') \in trans$, remove it and add $t_i = (l_i, g, \alpha, R, l')$. Set $Inv(l_i) = c_i$ and $\gamma(t_i) = \gamma(t)$.

   (c) For every $t = (l', g, \alpha, R, l) \in trans$, remove it and add $t_i = (l', g, \alpha, R, l_i)$. Define $\gamma(t_i) = \gamma(t) \wedge \mathbb{B}(G_i) \wedge \bigwedge_{j \neq i} \neg\mathbb{B}(G_j)$.

   (d) If l was in Loc_0 add $t_0 = (l_0, \top, \tau, \emptyset, l_i)$ in $trans$ and set $\gamma(t_0) = \mathbb{B}(G_i)$.

The result of applying this transformation to location `off` is illustrated in Figure 4. In this new model, we have two copies of the location : the one above corresponds to products equipped with the feature $FastStart$; the one below corresponds to the other products. Since `off` was an initial location, a dummy location l_0 has been added. This is needed to ensure that any product starts in its corresponding location. Provided that no semantics is associated to the dummy location and its outgoing transitions, this FTA has the same semantics than the one shown in Figure 2. Note that the above two transformations are commutative.

From the above observations, we conclude that the sole purpose of featured clock constraint is to increase the conciseness of the model through a reduction in the number of transitions and locations. More precisely, both transformations multiply the number of transition by a factor of $\mathcal{O}(2^{|N|})$. The second transformation yields models where the number of locations is multiplied by the same factor. Although it considers but a small example, Figure 4 already gives account for that phenomenon.

4.4 An Infinite FTS Semantics for FTA

The model-checking algorithms for TA are based on the definition of its semantics in terms of infinite transition systems. Similarly, one of our model-checking algorithm reuse the efficient algorithms for FTS model-checking [9, 8, 11]. Consequently, we need to provide FTA with a new semantics, namely in terms of FTS. This alternate semantics is a generalization of the transformation from classical TA to transition systems [4].

In this section, we define a transformation of FTA into a so-called "induced" FTS. The initial step consists in removing all featured clock constraints as presented above. It results an FTA with classical clock constraints from which the induced FTS can be extracted just like an infinite-state transition system can be extracted from a TA.

First, a state of an induced FTS must refer to a precise point in time. Consequently, the whole state-space is defined as $Loc \times Eval(C)$ and is infinite. An initial state is then an initial location with every clock value set to 0 :

$$I = Loc_0 \times \{\eta \in Eval(C) \mid \forall c \in C \bullet \eta(c) = 0\}.$$

To be able to relate a state with the time constraints it satisfies, we augment the set of atomic propositions with the set of clock constraints occurring in the original FTA, which we denote by $AP' = AP \cup CC(C, FTA)$. We then update the function that associates each state with atomic propositions according to this definition of AP'. More formally, this new function is defined as $L' : Loc \times Eval(C) \to AP'$:

$$L'(l, \eta) = L(l) \cup \{cc \in CC(C, FTA) \mid \eta \models cc\}.$$

Finally, the set of transitions is composed of discrete and delay transitions. Discrete transitions are the result of a modification in the system location, namely an actual transition in the original FTA. The state reached by a discrete transition is then determined according to (1) the location that would be reached in the FTA and (2) the clock valuations of the state from which the transition is executed,

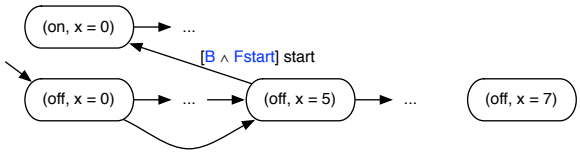

Figure 5: An infinite induced FTS.

the invariant of its location, the guard of the corresponding transition in the FTA, and the set of clocks to reset. Formally, $t' = ((l, \eta), \alpha, (l', \eta'))$ is a discrete transition iff

$$\exists t \in trans \bullet t = (l, g, \alpha, R, l') \wedge \\ \eta \models g \wedge \eta' = \text{reset } R \text{ in } \eta \wedge eta' \models Inv(l')$$

where reset R in η is a clock valuation function η' such that

$$\forall c \in C \bullet \eta'(c) = \begin{cases} 0, & c \in R, \\ \eta(c), & \text{otherwise.} \end{cases}$$

The feature expression labelling a discrete transition is equal to the one labelling the corresponding transition in the FTA, that is, $\gamma'(t') = \gamma(t)$.

Delay transitions models the passage of time when the system remains in a given location. Intuitively, a delay transition may occur if the duration of the delay does not lead to a violation of the current location invariant. Thus, $t' = ((l, \eta), \delta, (l, \eta+d))$ is a delay transition iff $\eta+d \models Inv(l)$, where $\eta + d$ is η with the value of every clock increased by d time units. Such a transition is executable by all the products : $\gamma(t') = \top$.

We partially illustrate the result of this translation in Figure 5, where it has been applied on the FTA shown in Figure 2. The initial state is composed of the location `off` and the clock valuation where `x` has the value 0. This state has an uncountable number of outgoing delay transitions (that is, to every state such that $\texttt{x} < 7$). When the value of `x` is greater than 4, a transition to the state `(on, x = 0)` is executable for the products that have the features B and $FastStart$. For obvious reasons, we do not provide the complete FTS resulting from this transformation.

So far, we have equipped FTA with two different semantics. Given that a TA is nothing more than an infinite transition system and that an FTS is a function from the set of products to transition systems [11], both end up describing FTA as a function that associates a product with the behaviour of an infinite transition system. These two semantics are equivalent, *i.e.* for every valid product p, they give rise to transition systems with equivalent behaviour.

5. MODEL CHECKING REAL-TIME SPL

We have presented the FTA formalism, its syntax and two forms of semantics, which end up being equivalent. The purpose of providing multiple definitions lies in the existence of two distinct methods for model-checking a real-time SPL modelled as an FTA. For recall, the objective of SPL model-checking is to pinpoint *all* the products that do not satisfy an intended property. Given that objective, it is natural to generalize the notion of satisfiability for FTA to a non-boolean form, *i.e.* as a function from products to truth values – as we did for FTS [10]. For a property Φ and an FTA fta defined over a feature model d, we define the F-satisfiability of fta with respect to Φ as the feature expression

$$(fta \models_{FTS} \Phi) \bullet (fta \models_{FTS} \Phi)(p) = (fta_{|p} \models \Phi)$$

In the following subsections, we propose two approaches for computing the F-satisfiability in FTA.

5.1 Enumerative Model-Checking

The first approach relies on an individual verification of each product. The idea is to compute the projection of the FTA onto every product and then model-check the resulting TA using standard algorithms. This approach offers the advantage of reusing existing model checking algorithms and tools that were developed for timed automata. We thus benefit from all their present and future optimizations. The ease of parallelization is another unquestionable strength of this enumerative method. It does not, however, take into account the commonality between the products to reduce the verification effort. Also, we have to build the feature expression encoding the set of bad products once the individual verifications are finished.

Without going into detail, we recap the principles of TA model-checking. As mentioned before, we may transform a TA into an infinite transition system. Should that model not be infinite, TA model-checking would boil down to model-checking a usual transition system. The infinite size comes from the definition of the state-space itself, that is, the cartesian product of the set of locations with the set of clock evaluations. The former is finite but the latter is not. That is the reason why all the existing TA model-checking algorithms make use of *time abstraction* to cope with the infinite number of clock valuations. One of the most popular form of time abstraction consists in representing a set of clock valuations with a *clock zone* [12]. Several TA model-checkers, like UPPAAL [5], rely on this representation.

Applying zone-based abstraction on the infinite transition system yields an abstract, finite transition system where states are couples of location and zone. We can thus compute the set of reachable states in this transition system and consequently determine if and when the system can reach a given location [12]. We do not provide algorithmic details for this exploration procedure because we also apply its principles in the second verification method.

We can improve the efficiency of the enumerative algorithm by purging every projection from unneeded clocks. It is well-known that the time complexity of TA model-checking is exponential in the number of clocks [12]. However, when projecting an FTA to a specific product, it may happen that some clocks become unused. This is the case when a clock does not occur in any location invariant, transition guard or reset of the considered projection. Discarding this clock might thus lead to significant improvements in the efficiency of the verification.

5.2 Variability-aware Model Checking

An important weakness of the aforementioned verification approach is its inability to take into account that multiple products can share common behaviour. To overcome this limitation, we suggest another method that combines the abstraction of time using zones with efficient algorithms we previously designed for discrete SPL model-checking [9]. In this section, we illustrate the use of this approach to

compute the *reachability relation*. The notion of reachability in FTA is different than that of TA and FTS, because it has to consider (1) the reachable locations, (2) the products able to reach them, and (3) the values of the clocks when these locations are reached. More formally, the reachability relation in an FTA fta is a set of triplets $\mathcal{R}_{FTA}(fta) \subseteq Loc \times (\{\top,\bot\}^{|F|} \to \{\top,\bot\}) \times \mathcal{P}(Eval(C))$ such that

$$(l,b,\varphi) \in \mathcal{R}_{FTA}(fta) \Leftrightarrow \forall p \in [\![b]\!] \bullet (l,\varphi) \in \mathcal{R}_{TA}(fta_{|p})$$

where $\mathcal{R}_{TA}$ denotes the reachability relation for TA. Given that the we expressed the semantics of fta as an infinite FTS, we can compute $\mathcal{R}_{FTA}(fta)$ by exploring the state space of this induced FTS, modulo a finite representation of sets of clock valuations. As before, we choose to use clock zones but any alternative finite representation is convenient as well. Once the FTS is computed, we begin the exploration with a depth-first search.

To perform the exploration, we make use of a new successor operator which takes both variability and time constraints into account. Since features do not occur in invariants and transition guards, the clock zone that is reached upon the execution of a transition does not depend on the features. Therefore, this successor zone is determined according to the current zone φ, the current location l, and the chosen transition $t = (l, g, \alpha, R, l')$. It is given by

$$next(\varphi, l, t) = \text{reset } \left((\varphi \wedge Inv(l))^{\Uparrow} \wedge Inv(l) \wedge g\right) \text{ in } R$$

where reset φ' in R is the zone φ' where every clock in R has been reset to 0 and $(\varphi')^{\Uparrow}$ is the zone φ' after an infinite elapsing of time [12]. The zone $next(\varphi, l, t)$ can be empty if φ, the invariant of l and the guard of t are incompatible. In this case, we must discard it and not pursue the exploration.

Similarly, since the set of features of a given product remains constant over time, the satisfiability of a feature expression annotating a transition does not depend on clocks values. These two properties show that Boolean features and real-time constraints are completely orthogonal constructs. *Ipso facto*, defining the successor operator comes down to determining, given a transition, which products are able to execute it and which zone is reached after the execution. Formally, the successors set of a state $(l,b,\varphi) \in Loc \times (\{\top,\bot\}^{|F|} \to \{\top,\bot\}) \times \mathcal{P}(Eval(C))$, noted $Post(l,b,\varphi)$, contains the state (l',b',φ') iff

$$\exists t \bullet t = (l,g,\alpha,R,l') \in trans \wedge \\ b' = b \wedge \gamma(t) \wedge [\![b']\!] \neq \emptyset \wedge \varphi' = next(\varphi,l,t)$$

The first condition requires the existence of a transition from l to l'. The second expresses that only the products in $[\![b]\!]$ compatible with the feature expressions $\gamma(t)$ can execute t. To derive this set of products, we compute the conjunction of b with $\gamma(t)$. We require that this set is not empty; otherwise, it would be pointless to include (l',b',φ') in the set of successors. Finally, the last condition is a direct application of the aforementioned zone successor operator.

We can compute the reachability relation $\mathcal{R}_{FTA}$ through successive applications of the successor operator – see Function Reachables. The initial locations are always reachable by all the valid products and for a clock zone φ_0 where every clock has its value set to zero (line 4). The procedure then explores the state-space of the induced FTS with a depth-first search. At each iteration, it computes the successor sets of the current state (line 9). For every successor $s = (l',b',\varphi')$, the procedure adds s in the relation of and only if there exist no element of the relation $(l',b''\varphi'')$ such that the products represented by b' are also represented by b'' and the clock zone φ' is a subset of φ''. Otherwise, it means that we already know that the products in $[\![b']\!]$ can reach l' in the clock zone φ'. In this case, it is useless to consider s and to pursue the search from this state. For a similar reason, when the procedure adds s in the relation it removes every element that has become redundant because of the addition of s (lines 10 – 16).

Input: $fta = (Loc, Act, C, trans, Loc_0, Inv, AP, L, d, \gamma)$.
Output: $\mathcal{R}_{FTA}(fta)$.

```
R ← ∅;
Stack ← [];
foreach s0 ∈ {(l0, B([[d]]_FD), φ0) | l0 ∈ Loc0} do
    R ← R ∪ {s0};
    push(s0, Stack);
end
while Stack ≠ [] do
    (l, b, φ) ← pop(Stack);
    foreach (l', b', φ') ∈ Post(l, b, φ) do
        if ∄(l', b'', φ'') ∈ R • [[b']] ⊂ [[b'']] ∧ φ' ⊆ φ'' then
            R ← R \ {(l', b'', φ'') • [[b'']] ⊆ [[b']] ∧ φ'' ⊆ φ'};
            R ← R ∪ {(l', b', φ')};
            push((l', b', φ), Stack);
        end
    end
end
return R_FTA(fta)
```

Function Reachables(fta)

As for TA, a forward reachability algorithm using zone-based abstraction does not always terminate without an extrapolation operator [6]. Since termination is a classical problem in TA model-checking and is independent of the addition of variability, we do not provide further discussions.

The above algorithm smartly combines existing model-checking approaches. The only new construct we introduced through the whole paper are featured clock constraints, which we could split up in feature expressions and classical clock constraints. This is reflected in the computation of the successor operator, where zones and products are considered in an orthogonal fashion. These nice characteristics make an implementation considerably easier.

6. IMPLEMENTATION

The material presented in this paper has been implemented in a new model checker for timed SPL. Our prototype tool takes as input the definition of a network of FTA and is able to compute the reachability relation defined in Section 5 as well as to verify a property expressed in temporal logic.

In our implementation, we consider a fragment of the Timed Computational Tree Logic (TCTL) [1]. More precisely, a formula Φ of this fragment has the form

$$\Phi ::= \exists\Diamond_J\phi \mid \exists\Box_J\phi \mid \forall\Diamond_J\phi \mid \forall\Box_J\phi \mid \forall\Box(\phi \to \exists\Diamond_J\psi)$$

where ϕ and ψ are conjunctions of atomic propositions, and J is a subinterval of $[0,\infty)$. Intuitively, the formula $\exists\Diamond_J\phi$ is

```
automaton pump
   synclabs start, stop, run;
   initially off;
   loc off :
      while [FastStart](x < 7) & [!FastStart](x < 10) wait
      if Button then
         when [FastStart](x > 4) & [!FastStart](x > 6) do
            sync start goto on;
   loc on : while True wait
      when True do sync run goto on;
      if Button then
         when [FastStop](x > 3) & [!FastStop](x > 5) do
            goto off;
end
```

Figure 6: A textual syntax for FTA.

satisfied iff "there exists a path where a state satisfying ϕ is reached, and the time needed to reach it is in J". Similarly, the formula $\exists\Box_J\phi$ expresses the requirement that "there exists a path where, during the interval of time J, the proposition ϕ is always satisfied". When the symbol $\forall$ replaces $\exists$, it means that the property that follows must hold for every path. Finally, $\forall\Box(\phi \rightarrow \exists\Diamond_J\psi)$ means that "whenever ϕ occurs, there must be a way to satisfy ψ within the interval of time J". This last formula is notably helpful to express real-time fault recovery requirements. One can check models against any of these formulae though reachability analyses.

In our tool, a network of FTA is specified in a textual language introduced hereafter. Given such a specification and a TCTL formula, the tool determines the exact set of valid products violating the formula and displays this set in the form of a feature expression. We have implemented the prototype from scratch and only reused libraries for representing clock zones and feature expressions.

6.1 An input language for FTA

Representing FTA graphically may be impractical for modelling large systems. To cope with this scalability issue, we propose a textual language for specifying FTA. We extended HyTEC [15], a well-acknowledged specification language for hybrid automata, with constructs to represent feature expressions and featured clock constraints.

More precisely, a textual specification is composed of a header, a core, and a footer. In the former part, clocks, actions and features are declared as variables. The core contains the description of a network of FTA. The FTA resulting from a specification is thus the parallel composition of several FTA. Parallel composition of FTA is a combination of parallel composition of TA and FTS.

Figure 6 shows the textual specification of the pump FTA (see Figure 2). Each FTA is identified with a unique id. The specification begins with the keyword `synclabs`, which is followed by the actions over which this automaton synchronizes with other FTA of the network. The initial location is declared after the keyword `initially`. Next, we declare the locations and their outgoing transitions. The keyword `loc` is followed by the id of the location. Then, the invariant is given in the form `while` fcc `wait` where fcc is a featured clock constraint. The textual description of featured clock constraints follows the same pattern than in the graphical representation. Once the invariant is given, we specify all the outgoing transitions. This specification begins with `if` f `then when` fcc `do`, where f and fcc are respectively the feature expression and the guard of the transition. Then, we write `sync` α if the transition has to be synchronized over the action α. Finally, the keyword `goto` is followed by the id of the location reached by the transition. Once all the locations are declared, we end the specification of the current FTA with the keyword `end`.

The footer is dedicated to the definition of the atomic propositions. Unlike for the graphical representation, we do not associate such propositions directly with the locations. Instead, we propose to describe them symbolically. More precisely, their definition is composed of disjunctions and conjunctions of atomic elements, `i.e.` clock constraints or locations. For instance, we could define the proposition

$$rapid\_on := loc[pump] = on \wedge x < 5$$

which is satisfied iff the pump FTA is in location `on` and the clock `x` has a value inferior to 5.

6.2 Data structures

In Section 4, we have shown that we can transform featured clock constraints into a combination of clock constraints and feature expressions labelling transitions. Thanks to this result, we can view a FTA as an FTS whose states are triplets composed of a location, a feature expression and a clock zone. This implies that in actual implementations, we may encode a state as a data structure with three fields, one for each member of the triplet.

The most important part of this data structure is the zone representation. In our implementation, we represent zones by its most popular dedicated data structure, i.e., *difference bound matrix* [12]. This allows us to reuse their efficient library implementation in the UPPAAL model-checker [5]. However, this implies that we cannot use symbolic encoding for locations and transitions, and that we need an additional data structure for representing feature expressions.

Since feature expressions are Boolean formulae where each variable is a feature, we can make use of existing data structures for those. In our prototype implementation, we decided to encode these formulae as *binary decision diagrams* (BDD), and included the CUDD package[1] implementation as part of our tool. As an alternative, we could have used the combination of a CNF representation with a SAT solver. However, we learned from past experiences in implementing feature expressions that the size of a CNF rapidly explodes when no minimization technique is applied. In order to avoid that burden, we settle for the BDD representation.

Using the above representation, we can also encode the set of valid products. This is needed to avoid false negatives that would be returned by the model-checking procedure. Remember that we want to identify all the *valid* products that violate a given TCTL formula. On the contrary, it can happen that a counter-example returned by the model-checker is not executable by any valid product, and thus must not be considered. A wieldy way to prevent that phenomenon is to initialize all the initial states with the feature expression encoding the set of valid products. By doing so, the successive applications of the successor operator (see Section 5) consider only the valid products.

In order to provide a high-level representation for feature models, we have equipped the tool with the Text-Based Variability Language (TVL) library[2] [7]. Given a TVL model,

[1] `http://visi.colorado.edu/~fabio/CUDD`
[2] `http://info.fundp.ac.be/~acs/tvl/`

Table 1: Verification times (in seconds, two decimals).

	6 features 16 products		10 features 72 products		13 features 512 products	
Property	**Enum.**	**Our.**	**Enum.**	**Our.**	**Enum.**	**Our.**
Reachability	0.35	0.58	3.33	0.98	10.52	1.04
$\forall\Box(methane \Rightarrow \exists\Diamond_{\leq 4}\ pumpOff)$	0.69	1.04	6.24	2.13	18.14	2.58
$\forall\Box(methane \Rightarrow \exists\Diamond_{\leq 2}\ pumpOff)$	0.44	0.71	5.17	2.67	13.13	2.72
$\forall\Box(pumpOn \Rightarrow \exists\Diamond_{\leq 15}\ lowWater)$	0.42	0.67	3.88	1.12	12.55	1.73
$\forall\Box(pumpOn \Rightarrow \exists\Diamond_{\leq 10}\ lowWater)$	0.39	0.66	4.07	1.01	9.75	1.04
$\forall\Box(pumpOn \Rightarrow \exists\Diamond_{\leq 5}\ lowWater)$	0.35	0.51	3.16	0.59	6.33	0.59
$\forall\Box(highWater \wedge \neg methane \Rightarrow \exists\Diamond_{\leq 10}\ pumpOn)$	0.66	0.93	5.57	1.97	18.86	2.15
$\forall\Box(highWater \wedge \neg methane \Rightarrow \exists\Diamond_{\leq 5}\ pumpOn)$	0.43	0.78	5.66	1.31	14.31	1.41
$\forall\Box(highWater \Rightarrow \exists\Diamond_{\leq 20}\ lowWater)$	1.06	1.48	9.05	3.59	31.99	3.93
$\forall\Box(highWater \Rightarrow \exists\Diamond_{\leq 15}\ lowWater)$	0.67	1.19	6.32	2.44	21.18	3.28
$\forall\Box(highWater \Rightarrow \exists\Diamond_{\leq 10}\ lowWater)$	0.69	3.67	7.55	10.98	14.60	11.69
$\forall\Box(highWater \Rightarrow \exists\Diamond_{\leq 5}\ lowWater)$	0.26	0.46	2.37	0.47	7.03	0.47
Total	**6.41**	**12.68**	**62.37**	**29.26**	**178.39**	**32.63**

the library returns the feature expression of the set of valid products, which we next encode as a BDD.

During the verification, the tool maintains the set of products already known to violate the checked formula. These products are then excluded from the subsequent searches. This optimisation grants a reduction in the verification time because it allows the checker to avoid exploring paths that only bad products can execute. In particular, the exploration is immediately stopped when all the valid products are known to violate the property.

6.3 Experiments

In Section 5, we have proposed two algorithms for FTA model-checking. In our previous work on FTS, we have shown that variability-aware procedures are generally more efficient than enumerative computations [9, 8, 10]. It means that the avoidance of redundant verification offsets the overhead of variability management. If we add another dimension, namely real-time, we can expect the latter overhead to become an even lower part of the total verification effort.

In this subsection, we provide preliminary experimental results regarding the relative performance of the aforementioned algorithms. For this purpose, we use our prototype tool to analyze a system modelled as an FTA. More precisely, the tool computes the reachability relation and verifies the system against several TCTL formulae. These formulae were declined in different variants such that two variants differ only from their time constraints.

First, we measure the time needed by the tool to perform the analysis using the variability-aware algorithm. Second, given the textual definition of the FTA, we used a separate script to produce each projection of the FTA into a valid product on which classical timed model checking techniques are applied. We do not measure the runtime of this script as we do not want to include the time for computing the projection to the total verification time.

For our experiments, we consider a real-time extension of the mine-pump controller exemplar [17], of which the toy example of the previous sections is inspired. The controller intends to clear a mine shaft from water by activating a pump. Meanwhile, it must avoid explosion danger by switching the pump off in presence of methane. We modelled the features of this system. Depending on the degree of refinement in the feature model, we obtained three distinct SPLs. The first has six features and 12 products, the second has 10 features and 72 products, and the third has 13 features and 512 products. In this SPL, both the capabilities of the controller and the efficiency of the pump are subject to variations between products. For instance, the feature $MethaneAlarm$ determines whether or not the system is equipped with an alarm that triggers once there is methane; the feature $FastPump$ increases the running speed of the pump. In our model, no clock becomes useless upon projection. Therefore the optimisation discussed in Subsection 5.1 could not be applied.

All benchmarks were run on a MacBook Pro with a 2.4 GHz Core 2 Duo processor and 4 Gb of RAM. To avoid the influence of other running processes, we repeated each experiment several times.

The results are shown in Table 1. For every variant of the SPL and every formula, we give the time needed by the enumerative algorithm (**Enum.**) and the variability-aware (**Our.**) algorithm to verify the FTA against the formula. For the SPL with the smallest set of features, it turns out that the enumerative method always outperforms our new algorithm. Altogether, we observe an increase of 97.82% in the verification. This seemingly weakness is not due to the exploration of the state-space itself, but rather to the time needed by the TVL library to analyze the feature models. According to additional benchmarks, this time amounts to 0.42 second on average. When the number of features and products increases, our variability-aware algorithm overcomes this overhead and completely outmatches the enumerative one. For the medium-sized and the big-sized SPL, it decreases the total verification time by 53.09% and 81.71%, respectively. These results are highly encouraging because they show that our approach seems to be more scalable with respect to the size of the SPL.

For one formula, the enumerative approach is a bit faster than ours in the small-sized and medium-sized cases. This formula is particular since all the features that have an impact on time lead to different path during the exploration. As a consequence, more zones have to be explored and compared during the verification process. We postulate that heuristics can improve this situation.

Threads to validity. Two characteristics of our experiments may harm the validity of our conclusions regarding relative efficiency of our algorithms. First, our evaluation is based on a sole small case study. To assess our approach at a greater scale, we should carry out experiments on additional examples. Second, the enumerative method is evaluated through the classical TA checking algorithms implemented in our tool. There exist alternative solutions like UPPAAL which implement optimisations that our tool does not. However, the overhead of real-time models verification is mostly due to the representation of time. Given that we reused the very efficient DBM library of UPPAAL, we should draw the same conclusions even if we mapped the enumerative algorithm with existing tools.

7. CONCLUSION

In this paper, we presented Featured Timed Automata, a mathematical formalism for reasoning on the behaviour of real-time SPLs. FTA is a compact representation for a set of TA through the insertion of feature-related information. By exploiting the particular structure of FTA representation, we designed an algorithm to efficiently model-check it. The work has been implemented and evaluated on a case study. Preliminary results show that our approach clearly outperforms verification of SPLs through the enumeration of individual products.

As a future work, we will improve the efficiency of our algorithms. For doing so, we plan to study additional data structures for representing states in FTA. In our current implementation, such states are represented by triplets composed of an explicit location, a binary decision diagram, and a difference bound matrix. As alternatives, we plan to use either Multi-Terminal BDD or a data structure that combines zone and binary variables representations. For the latter, Clock Restriction Diagrams [20] and appear as promising candidates.

Also, we plan to extend FTA modelling to non-Boolean features. This would allow to directly express real-time constraints in terms of a feature attributes. As an example, the running speed of a processor may drastically influence the execution time of a software. Considering this extension would definitively impact on our model-checking procedure. Indeed, in this context we would not be able to use binary decision diagrams or SAT solver to reason on sets of products. Second, we may not be able to translate the FTA into an infinite FTS anymore.

Acknowledgement. This work was funded by the FNRS and the IAP Programme of Belgium (MoVES project).

8. REFERENCES

[1] R. Alur, C. Courcoubetis, and D. Dill. Model-checking in dense real-time. *Inf. Comput.*, 104:2–34, May 1993.

[2] S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer. Feature-interaction detection using feature-aware verification. In *Proceedings of ASE'11*. IEEE Computer Society, 2011.

[3] P. Asirelli, M. H. T. Beek, A. Fantechi, and S. Gnesi. Formal description of variability in product families. In *Proceedings of SPLC'11*, pages 130–139. Springer-Verlag, 2011.

[4] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2007.

[5] J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Uppaal in 1995. In *Proceedings of TACAS'96*, pages 431–434, London, UK, 1996. Springer-Verlag.

[6] J. Bengtsson and W. Yi. Timed automata: Semantics, algorithms and tools. In *Lectures on Concurrency and Petri Nets*, pages 87–124, 2003.

[7] A. Classen, Q. Boucher, and P. Heymans. A text-based approach to feature modelling: Syntax and semantics of tvl. *Sci. Comput. Program.*, 76:1130–1143, December 2011.

[8] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. Symbolic model checking of software product lines. In *Proceedings of ICSE '11*, pages 321–330. ACM, 2011.

[9] 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 *Proceedings of ICSE '10*, ICSE '10, pages 335–344, New York, NY, USA, 2010. ACM.

[10] M. Cordy, A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. Managing evolution in software product lines : A model-checking perspective. In *Proceedings of VaMoS'12*, pages 183–191. ACM, 2012.

[11] M. Cordy, A. Classen, G. Perrouin, P. Heymans, P.-Y. Schobbens, and A. Legay. Simulation relation for software product lines. In *Proceedings of ICSE'12 (to appear)*. IEEE, 2012.

[12] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *Proceedings of the international workshop on Automatic verification methods for finite state systems*, pages 197–212, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

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

[14] A. Gruler, M. Leucker, and K. Scheidemann. Modeling and model checking software product lines. In *IFIP WG 6.1 FMOODS '08*, pages 113–131. Springer, 2008.

[15] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. Hytech: A model checker for hybrid systems. In *Proceedings of CAV'97*, pages 460–463, London, UK, 1997. Springer-Verlag.

[16] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *Proceedings of ICSE'08*, pages 311–320, New York, NY, USA, 2008. ACM.

[17] J. Kramer, J. Magee, M. Sloman, and A. Lister. Conic: an integrated approach to distributed computer control systems. *Computers and Digital Techniques, IEE Proceedings E*, 130(1):1–10, 1983.

[18] K. Lauenroth, S. Toehning, and K. Pohl. Model checking of domain artifacts in product line engineering. In *Proceedings of ASE '09*, pages 269–280, 2009.

[19] H. C. Li, S. Krishnamurthi, and K. Fisler. Interfaces for modular feature verification. In *Proceedings of ASE'02*, pages 195–204, 2002.

[20] F. Wang. Efficient model-checking of timed automata with clock-restriction diagram. In *APLAS*, pages 207–224, 2001.