Noname manuscript No.
(will be inserted by the editor)

Verification and Abstraction of Real-Time Variability-Intensive Systems

Maxime Cordy · Axel Legay

the date of receipt and acceptance should be inserted later

Abstract Featured Timed Automaton (FTA) is a concise formalism to model the real-time behaviour of variability-intensive systems. FTA extends the timed automaton by allowing optional transitions and clock constraints that are relevant only for a subset of the system variants. Then, one can verify a variant individually by deriving the corresponding TA from the FTA and using established tools like UPPAAL, or apply family-based algorithms to verify all variants at once. These latter algorithms consist of computing the reachability relation in FTA as an antichain. Yet, they suffer from a three-source complexity: the number of states, the number of time clocks and the number of variants. This motivates the design of abstraction-refinement heuristics to reduce verification effort. In this paper, we present the syntax and semantics of FTA, efficient algorithms to compute their reachability relations, and discuss how abstraction methods can be applied.

Keywords Model checking · variability · real time · abstraction

1 Introduction

Nowadays, Variability-Intensive Systems (VIS), in particular Software Product Lines [17] (SPLs), are deployed in a wide variety of domains, including in critical areas such as avionics and telecommunications. These systems interact with their environment in real time, and therefore have to cope with hard real-time requirements. Providing VIS engineers with quality assurance techniques is of utmost importance since those must increase confidence that *all* product variants 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 VIS is, however, 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 a VIS comes to identifying *all* the software products that fail to meet their requirements and intended properties [15]. A simple but cumbersome method for addressing this problem is to reuse existing techniques to check each product individually. This product-based approach is clearly impractical for designs with hundreds, or sometimes thousands, of product variants [15,14].

Many model-checking techniques were designed for verifying VIS and SPL efficiently, e.g. those based on featured transition systems [13], modal transition systems [5] and multi-valued model checking [11]. These typically exploit the commonalities between different products to reduce the verification effort, that is, they check only once behaviours common to multiple products. Unfortunately, they suffer from two drawbacks that hinder their applicability in practice.

This work was partly done when Maxime Cordy was working at the University of Namur.

This work was partly done when Axel Legay was working at Inria.

University of Luxembourg, Luxembourg
E-mail: maxime.cordy@uni.lu
· Université Catholique de Louvain
E-mail: axel.legay@ucl.be

First, most of these techniques cannot handle VIS whose behaviour depends on real-time constraints. As an answer to these limitations, we proposed *Featured Timed Automata (FTA)* as the first approach to model and verify real-time VIS [22]. Our technique relies on a smart combination of variability-aware model-checking with established real-time model-checking approaches. More precisely, FTA extends Timed Automata (TA) [26], the classical model for timed systems, with variable timed constraints and variable transitions, thereby allowing the modelling of multiple real-time systems in a concise formalism.

Second, variability exacerbates the state-explosion problem inherent to model checking by adding a second source of complexity, *viz.* the number of products. In single-system verification, one of the most effective answers to state explosion is *model abstraction*, which creates more concise – therefore easier to verify – models of the system, typically by merging similar states. A fundamental challenge to address is the design of abstraction techniques that can reduce the two-dimensional complexity of VIS behavioural models. [21]

In this paper, we aim to address these two limitations by extending our previous work [22]. Accordingly, our first contribution is the formal definition of FTA syntax and semantics. We also study several ways of modelling variability by exploiting the structure of TA. Then, we combine the principles of TA and VIS model checking into new verification algorithms specifically designed for FTA. Our second contribution is to provide a formal definition of FTA abstraction and to discuss how such an abstraction integrates within a verification procedure based on an abstraction refinement loop. A peculiarity of this abstraction, which distinguishes it from our previous work [21] is that in TA (and thus in FTA), state changes are subjected to timed constraints. Therefore, state abstraction techniques for FTA should lean on state abstractions that deal with real-time behaviour [25,34].

Overall, this paper brings the following improvements and additional content over [22]:

- Improved text and formalism (all over the paper);
- Reworked introduction (Section 1);
- The definition of antichain in FTA reachability relation and an algorithm to compute it (from Definition 15 to the end of Section 5, including Algorithm 2);
- A new Section discussing how to design abstraction refinement methods for FTA (Section 6);
- Up-to-date related work section reporting the latest publications on real-time variability-intensive system verification and abstraction refinement methods for such systems (Section 7).

This paper is structured as follows. Section 2 recapitulates TA syntax and semantics. Section 3 defines the FTA formalism and the concept of featured clock constraints. Section 4 presents a transformation of FTA that is a prerequisite for the verification algorithms based on the reachability relation in FTA. These algorithms are presented in Section 5. In Section 6, we discuss how we can design abstraction refinement techniques for FTA. Section 7 presents related work.

2 Timed Automata

Timed Automata (TA) [26] have become a popular formalism for reasoning over the behaviour of continuous-time systems. The main reasons are that model checking of TA remains decidable for branching-time logics such as CTL and that efficient solutions can be implemented. Moreover, renown implementations like UPPAAL [7] managed to catch the interest of academics and practitioners alike. This increases confidence that TA are suitable for verifying properties of real-time VIS as well. Fundamentally, TA are Transition Systems (TS) that can remain in a given state only a certain amount of time, and can execute a transition within a certain time interval. As a reminder, TS are defined as follows.

Definition 1 (Transition system) [4] A TS is a tuple $(S, Act, Trans, I, AP, L)$ where

- S is a set of states named the state space;
- Act is a set of actions;
- $Trans \subseteq S \times Act \times S$ is the transition relation, where $(s, \alpha, s') \in Trans$ (also noted $s \xrightarrow{\alpha} s'$) means that there is a transition from state s to state s' labelled with action α;
- $I \subseteq S$ is a set of initial states;
- AP is a set of atomic propositions representing basic system properties that are satisfied or not in a given state;
- $L : S \to 2^{AP}$ is a function that associates every state with the set of atomic propositions satisfied by this state.

We assume that our TS have no terminal state, i.e. a state without outgoing transition: $\forall s \in S \bullet \exists \alpha \in Act, s' \in S \bullet (s, \alpha, s') \in Trans$.

Definition 2 (Transition system semantics) [4] Let $ts = (S, Act, Trans, I, AP, L)$ be a TS. The semantics of ts, noted $[\![ts]\!]$, is its set of traces:

$$[\![ts]\!] = \{L(s_0), L(s_1), \cdots \in (2^{AP})^\omega \mid s_0 \in I \wedge \forall i \in \mathbb{N} \bullet \exists \alpha_i \in Act \bullet (s_i \xrightarrow{\alpha_i} s_{i+1})\}.$$

In TA, time is represented by clocks whose values evolve continuously. Clocks can be regarded as chronometers: their value can be inspected and reset, but not modified arbitrarily. Conditions over clock values are named *clock constraints* and take the following form.

Definition 3 (Clock constraint) [4] A clock constraint over a set C of clocks is formed according to the following grammar:

$$g ::= \top \mid n \sim c \mid g \wedge g$$

with $n \in \mathbb{N}$, $c \in C$, and $\sim \in \{<, \leq, \geq, >\}$.

We denote by $CC(C)$ the set of clock constraints over C. In TA, a clock constraint can label a state or a transition. In the first case, the constraint is an *invariant*, which defines the interval of time in which the system can be in the state. In the second case, it is a *transition guard* specifying the interval of time during which the system can execute the transition. Note that the value domain of the numeric member of clock constraints is limited to natural numbers. This is, however, without loss of generality, as one could use real numbers instead. Using natural numbers offer the benefit of facilitating the implementation of clock constraint, by allowing the use of efficient data structure like difference bound matrices [7].

This leads us to the definition of TA.

Definition 4 (Timed automaton) A TA is a tuple $(Loc, Act, C, Trans, Loc_0, Inv, AP, L)$ where

- Loc is a set of locations;
- Act is a set of actions;
- C is a set of clocks;
- $Trans \subseteq Loc \times CC(C) \times Act \times 2^C \times Loc$ is a transition relation. For a transition (l, g, α, Res, l'), l is the starting location, g is the transition guard, α is the action triggering the transition, Res is the subset of clocks to reset, and l' is the target location.
- $Loc_0 \subseteq Loc$ is a set of initial locations;
- $Inv : Loc \to CC(C)$ is a total function associating a location with an invariant;
- AP is a set of atomic propositions;
- $L : Loc \to 2^{AP}$ is a total function associating a location with the atomic propositions satisfied in that location.

where $CC(C)$ denotes the set of clock constraints over C.

Example 1 An example of TA is given in Figure 1. Similarly to TS, 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`. We use 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`).

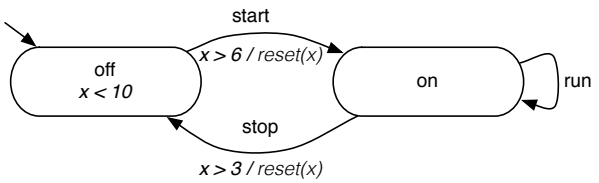

Fig. 1 An example of a Timed Automaton.

The semantics of a TA is commonly defined as an infinite TS whose states consist of a location and a valuation of the clocks. The transitions in this TS can be categorized into two types. *Delay transitions* do not change the location of the system; they only represent the passage of time. A delay transition may occur only if the invariant of the current location is still satisfied after the delay modelled by the transition. *Discrete transitions* occur when the system moves from one location to another. These are executable only when the current clock values satisfy both the guard of the executed transition and the invariant of the target location. After the execution of such transitions, clock values can be reset.

Definition 5 (TA Semantics [4]) Let $ta = (Loc, Act, C, Trans, Loc_0, Inv, AP, L)$ be a TA. The semantics of ta is the semantics of the TS $(S, Act', Trans', I, AP', L')$, noted $TS(ta)$, and such that

- $S = Loc \times Val(C)$;
- $Act' = Act \cup \mathbb{R}_{\geq 0}$;
- $I = Loc_0 \times \eta_0$;
- $AP' = AP \cup CC(C)$;
- $L'(l, \eta) = L(l) \cup \{cc \in CC(C) \mid \eta \models cc\}$.

where $Val(C)$ the set of clock evaluations, that is, the set of total functions $\eta : C \to \mathbb{R}^+$ that assign a non-negative real value to every clock, and η_0 is such that $\forall c \in C \bullet \eta_0(c) = 0$.

Temporal logics such as LTL and CTL cannot express real-time properties. Instead, one can specify them in Timed Computation Tree Logic [2] (TCTL), a timed variant of CTL.

Definition 6 (TCTL [2,4]) A TCTL formula Φ over a set AP of atomic propositions and a set C of clocks is formed according to the following grammar:

$$\Phi ::= \top \mid a \mid cc \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi_1 \mid \exists(\Phi_1 \mathrm{U}_J \Phi_2) \mid \forall(\Phi_1 \mathrm{U}_J \Phi_2)$$

where Φ_1 and Φ_2 are TCTL formulae, $a \in AP$, $cc \in CC(C)$, and J is a subinterval of $[0, \infty)$. When $J = [0, \infty)$, it can be omitted.

A TCLT formula thus defines time-sensitive requirements over the atomic propositions that should be satisfied or unsatisfied during during the executions of the TA. In this work, we consider a specific fragment of TCTL that can be checked through reachability analysis, as we deal with this type of analysis in the subsequent sections. Our approach is, however, not limited to this fragment and can straightforwardly be extended to fully support TCTL. A formula of this fragment has the form

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

where ϕ and ψ are Boolean formulae over the set of atomic propositions, $\Diamond_J \phi = \top \mathrm{U}_J \phi$, and $\forall\Box_J \varphi = \neg\exists_J \neg\varphi$. Intuitively, the formula $\exists\Diamond_J \phi$ is satisfied if and only if *"there exists a path where a state satisfying ϕ is reached, and this state can be reached in a time that lies 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.

Model checking a TA against a TCTL formula is achieved by computing the reachability relation in the underlying infinite TS. Should that model not be infinite, this would come down to model-checking a standard TS. 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 valuations. The former is finite but the latter is not. This is the reason why all the existing TA model-checking algorithms make use of *time abstraction*. One of the most popular mechanism for abstracting time consists in representing a set of clock valuations with a *clock zone* [26]. Given a clock constraint g, a clock zone for g is the maximal set of clock valuations satisfying g. In practice, zones can be represented as the conjunction of constraints of the form $x - y < n$ with $n \in \mathbb{Z}$ and $x, y \in C \cup \{0\}$. Several TA model-checkers, notably UPPAAL [7], rely on this representation. Applying zone-based abstraction on the infinite TS yields a finite TS where states are couples of location and zone. One can thus compute the set of reachable states in this TS and consequently determine if and when the system can reach a given location [26]. We apply the principles of this exploration procedure in our own verification algorithms for real-time VIS.

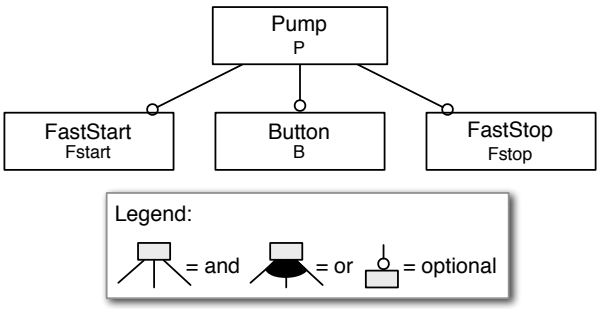

Fig. 2 Feature diagram of the pump SPL.

3 Modelling Real-Time VIS Behaviour

In real-time VIS, features may influence the behaviour of a system in two ways. First, features may add or remove capabilities like in discrete VIS. Second, they may also influence the time constraints that determine the minimum and maximum delays for executing a given action or remaining in a given state. Accordingly, we extend TA with variability in the same way as TS were extended to Featured Transition Systems [13]. The resulting formalism is named Featured Timed Automaton (FTA). Before giving its formal definition, we first present it through an introductory example.

Example 2 We consider a simplification of the well-known mine pump example [30] extended with real-time. We focus on a closed model of the software controlling the pump itself and ignore the other components of the system. We assume that the software is developed as a VIS with three optional features: *Button* (B), *FastStart* ($Fstart$), and *FastStop* ($Fstop$). The corresponding feature diagram is shown in Figure 2. *Button* defines that the pump is 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 VIS has a total of eight products.

Figure 3 shows a graphical representation of an FTA modelling the real-time behaviour of the pump VIS. As in TA, we model the locations of the system. Here, there are only two locations, `on` and `off`, which models that the pump is started and stopped, respectively. The

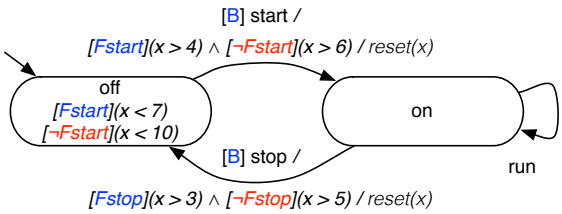

Fig. 3 FTA modelling the mine pump VIS.

pump can transit between these locations if and only if it has 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 invariant. 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 the presence of $FastStart$ 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 features with location invariants. More precisely, we define an invariant as the conjunction of clock constraints annotated with a feature expression. We name this construct *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 the feature expression B. On the contrary, the transition `run` does not require the feature $Button$ to be executed and is thus annotated with the feature expression $\top$, 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 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. If the pump is equipped with feature $FastStart$, this delay is reduced to 4 time units."

As before, we may model this requirement by guarding the transition with a featured clock constraint. In this 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. Overall this model defines that the pump always remains in location `off` between 4 and 7 time units if feature $FastStart$ is enabled, and between 6 and 10 time units otherwise.

3.1 Syntax and Semantics of FTA

As our introductory example suggests, the encoding of variability within FTA relies on a combination of feature expressions and clock constraints. The resulting formalisms is defined as follows.

Definition 7 (Featured clock constraint) A featured clock constraint over a set F of features and a set C of clocks is a formula of the form

$$g ::= \top \mid [\chi](c \sim n) \mid g \wedge g$$

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

Intuitively, $g = [\chi]g'$ means that g' must hold for products satisfying χ – the others do not have to satisfy the constraint. We denote by $FCC(F, C)$ the set of featured clock constraints over F and C. The satisfiability relation for featured clock constraints by a couple $(p, \eta) \in 2^F \times Val(C)$ is recursively defined as follows:

$$(p,\eta) \models g \Leftrightarrow \begin{cases} (p,\eta) \models g_1 \wedge (p,\eta) \models g_2, & g = g_1 \wedge g_2, \\ p \not\models \chi \vee \eta \models g', & g = [\chi]g'. \end{cases}$$

When $p \not\models \chi$, 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. Also, we allow the use of $[\chi](g_1 \wedge g_2)$ as a shortcut for $[\chi]g_1 \wedge [\chi]g_2$. As an example, the featured clock constraint $[FastStart]\ (x > 4)$ (see Figure 3) is satisfied by a product p and a clock valuation η if and only if p does not have feature $FastStart$ or clock `x` has a value higher than 4 in η.

Feature expressions and featured clock constraints allow us to model the impact of features on real-time models. By combining them with timed automata, we obtain a new formalism to model the behaviour of real-time VIS.

Definition 8 (Featured timed automaton (syntax)) A featured timed automaton is a tuple $(Loc, Act, C, trans, Loc_0, Inv, AP, L, fm, \gamma)$ 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,
- $Trans \subseteq Loc \times FCC(F,C) \times Act \times 2^C \times Loc$ is a finite set of transitions,
- $Inv : Loc \to FCC(F,C)$ is a total 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,
- fm is a feature model over a set F of features,
- $\gamma : Trans \to 2^{(2^F)}$ is a total function that labels transitions with feature expressions.

According to this definition, we consider that a feature may modify the availability of transitions, transition guards, and location invariants. As for FTS, the function γ associates a transition t with a feature expression such that $\gamma(t)$ encodes 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 a 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 3. 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 real-time 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 *projection* operator over FTA. 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 if $\chi(p) = \top$ holds, or by $\bot$ otherwise.

Definition 9 (Projection of FTA) The projection of an FTA $fta = (Loc, Act, C, Trans, Loc_0, Inv, AP, L, d, \gamma)$ onto a valid product $p \in [\![fm]\!]$ is the TA $fta_{|p} = (Loc, Act, C, Trans', Loc_0, Inv', AP, L)$ with

$$\forall l \in Loc \bullet Inv'(l) = Inv(l)_{|p}$$

$$Trans' = \{t = (l, g_{|p}, \alpha, R, l') \mid t \in Trans \wedge p \models \gamma(t)\}$$

where the projection of a featured clock constraint g onto a product p is recursively defined as

$$g_{|p} = \begin{cases} (g_1)_{|p} \wedge (g_2)_{|p}, & g = g_1 \wedge g_2, \\ g', & (g = [\chi]g') \wedge p \models \chi, \\ \top & (g = [\chi]g') \wedge p \not\models \chi. \end{cases}$$

For example, Figure 1 is the TA resulting from the projection of the FTA shown in Figure 3 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 10 (Featured timed automaton (semantics)) The semantics of an FTA $fta = (Loc, Act, C, trans, Loc_0, Inv, AP, L, d, \gamma)$ is the function $[\![fta]\!]$ such that

$$\forall p \in [\![fm]\!] \bullet [\![fta]\!](p) = [\![TS(fta_{|p})]\!].$$

For instance, the FTA presented in Figure 3 associates the product with features $\{B, Fstop\}$ to the TA shown in Figure 1.

4 An Alternative Definition of FTA

Definition 10 expresses the semantics of FTA in terms of TA, the semantics of the latter being itself defined as an infinite induced TS. In the end, the above semantics can be seen as a function from FTA to TS where variability is first resolved (through the definition of projection) before real-time is unfolded. This induces a model checking algorithm that can check each variant separately using single-TA verification algorithms. While this algorithm is straightforward to engineer, it suffers from the exponential blow-up induced by variability (that is, the number of variant grows exponentially with the number of features).

This calls for an alternative approach that can better exploit the commonalities between the variants to reduce. To achieve this, one can invert the aforementioned two processes and first transform an FTA into an infinite FTS, whose semantics is defined as the projection onto a set of products.

Definition 11 (Featured transition systems [13]) An FTS is a tuple $(S, Act, Trans, I, AP, L, fm, \gamma)$, where

- $S, Act, Trans, I, AP, L$ are defined as in Definition 1;
- fm is a FM over a set of features F;
- $\gamma : Trans \to 2^{(2^F)}$ is a total function that associates transition with a feature expression over F.

An FTS can be seen as the merging of the TS of all the products that compose the VIS. The TS modelling a specific product is obtained from the FTS by applying a *projection* function. In simple terms, this function prunes the FTS of all the transitions whose feature expression is not satisfied by the considered product [13], and then removes all feature expressions.

Definition 12 (Projection of FTS) [13] Let $fts = (S, Act, Trans, I, AP, L, fm, \gamma)$ be an FTS and $p \in [\![fm]\!]$ be a product. The projection of fts onto p, noted $fts_{|p}$, is the TS $(S, Act, Trans', I, AP, L)$ where

$$Trans' = \{t \in Trans \mid p \models \gamma(t)\}.$$

In this section, we investigate this alternative semantics and prove that it is equivalent to the previous one. Before giving the actual definition, it is first required to perform some transformations on FTA. We formalise these transformations and show that they preserve FTA semantics.

4.1 Separating Variability from Real-Time

Consider again the FTA shown in Figure 3. 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 is equivalent to two alternative transitions, i.e. 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 4 for an illustration). The above reasoning thus raises the question of studying a possibly equivalent form of FTA where feature expressions have been removed from featured clock constraints. In what follows, we show that we can transform every FTA into an equivalent FTA without featured clock constraints. Depending on whether we consider transition guards or location invariants, the transformation procedure is different.

Let us first consider the first case (e.g. the transition `start` in Figure 3). Let $t = (l, g, \alpha, Res, l')$ be a transition. The idea of the transformation is to create copies of the transition such that a given product p can execute only one of the copies and the guard of this copy 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 $\{G_1, \ldots, G_k\} \subseteq 2^{(2^F)}$ such that G_i is the largest set satisfying
$$\frac{c_{|p} = c_i}{p \in G_i}$$
where c_i is the clock constraint resulting from the projection of the guard onto any product in G_i.
2. Remove t from $Trans$.
3. For each group G_i add $t_i = (l, c_i, \alpha, Res, l')$ in $Trans$ and define $\gamma(t_i) = \gamma(t) \wedge \mathbb{B}(G_i) \wedge \bigwedge_{j \neq i} \neg\mathbb{B}(G_j)$.

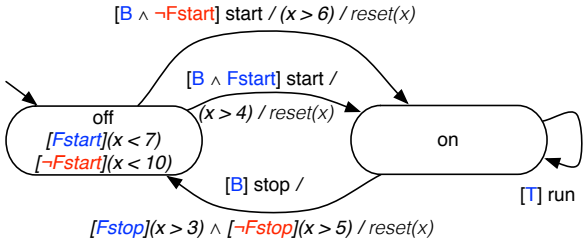

Fig. 4 FTA with a transformed transition guard.

For the transition `start`, we have two groups, namely the products that have feature $Fstart$ and the ones that do not. The corresponding clock constraints are $x > 4$ and $x > 6$, respectively. As shown in Figure 4, this results in two new transitions: the first has the guard $x > 4$ and is executable by the products satisfying $B \wedge Fstart$; the second has the guard $x > 6$ and is executable by the products satisfying $B \wedge \neg Fstart$. This new FTA has the same semantics as the one shown in Figure 3. This transformation preserves the semantics of the model by definition of the grouping criteria and of the projection for transition guards.

The second transformation, which removes features 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 $G_1, \ldots, G_k$ with the corresponding projections $c_1, \ldots, c_k$ of the location invariant. Then we perform the following steps:

1. If l is in Loc_0, add a new location l_0 in Loc_0 with
$$Inv(l_0) = \bigwedge_{c \in C} c \leq 0.$$
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 $t = (l, g, \alpha, Res, l') \in Trans$, remove it and add $t_i = (l_i, g, \alpha, Res, l')$. Set $Inv(l_i) = c_i$ and $\gamma(t_i) = \gamma(t)$.

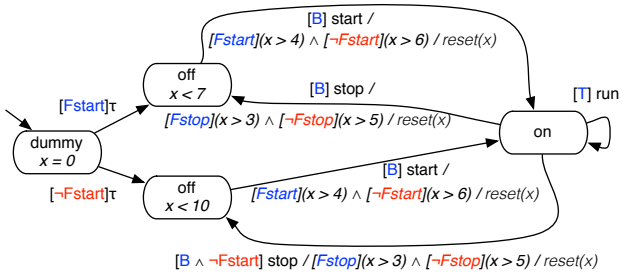

Fig. 5 FTA with a transformed invariant.

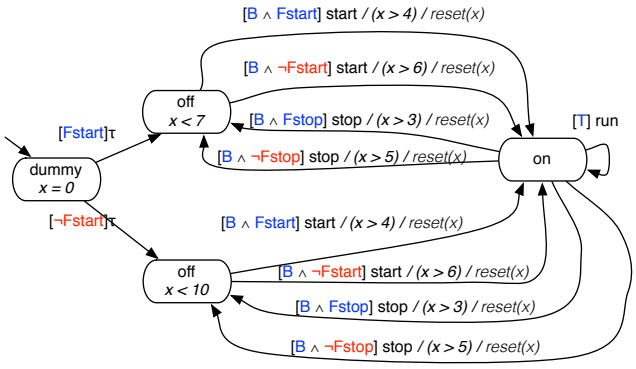

Fig. 6 FTA with transformed invariants and guards.

(c) For every $t = (l', g, \alpha, Res, l) \in Trans$, remove it and add $t_i = (l', g, \alpha, Res, l_i)$. Define $\gamma(t_i) = \gamma(t) \wedge \mathbb{B}(G_i)$.
(d) If l was in Loc_0, add τ in Act and $t_i = (l_0, \top, \tau, \emptyset, l_i)$ in $trans$, and set $\gamma(t_i) = \mathbb{B}(G_i)$.

The result of applying this transformation to location *off* is illustrated in Figure 5. In this new model, we have two copies of the location: the upper one corresponds to products having the feature $FastStart$; the lower one 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 a location with an appropriate invariant. Provided that no semantics is associated to the dummy location and its outgoing transitions, this FTA is equivalent to the one shown in Figure 3 modulo weak timed bisimulation of the respective projections [37]. Note that the above two transformations are commutative; their successful application yields the FTA shown in Figure 6.

From the above observations, we conclude that the sole purpose of featured clock constraints 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 transitions by a factor of $\mathcal{O}(2^{|F|})$, even though they decrease the size of the guards and invariants by the same factor. Additionally, The second transformation yields models where the number of locations is also multiplied by the same factor. Although it considers but a small example, Figure 6 already gives account for that phenomenon.

4.2 An Infinite FTS Semantics

We can finally provide FTA with a new semantics, namely in terms of FTS. This alternate semantics is a generalization of the transformation from TA to TS [4] combined with the aforementioned two FTA transformations. More precisely, we define a transformation of FTA into an *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 is extracted just like an infinite-state TS is produced from a TA.

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

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

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 $L' : Loc \times Val(C) \to 2^{AP'}$:

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

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 the execution of an actual transition in the original FTA. The state reached by a discrete transition is then determined according to:

1. the clock valuations of the state from which the transition is executed;
2. the guard of the corresponding transition in the FTA;
3. the set of clocks to reset;
4. the invariant of the target location.

Formally, $t' = ((l, \eta), \alpha, (l', \eta'))$ is a discrete transition if and only if

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

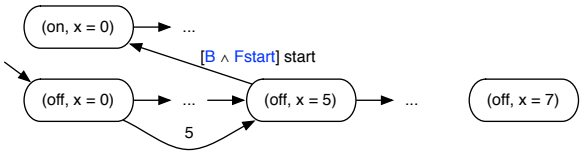

Fig. 7 An infinite induced FTS.

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

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

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, $t' = ((l, \eta), d, (l, \eta + d))$ is a delay transition if and only if $\eta + d \models Inv(l)$, 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: $\gamma(t') = \top$.

Example 3 We partially illustrate the result of this translation in Figure 7, where it has been applied on the FTA shown in Figure 3. 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 `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 *Button* and *FastStart*.

So far, we have equipped FTA with two different semantics. Given that a TA is nothing more than an infinite TS and that an FTS is a function from the set of products to TS [20], both end up describing FTA as a function that associates a product with the behaviour of an infinite TS. These two semantics are equivalent, i.e. for every valid product p, they give rise to TS with equivalent behaviour.

Theorem 1 *Let fta be an FTA defined over an FM fm and $FTS(fta)$ its induced infinite FTS. Let $TS(ta)$ denote the infinite TS induced by a TA ta. Then,*

$$\forall p \in [\![fm]\!] \bullet [\![FTS(fta)_{|p}]\!] = [\![TS(fta_{|p})]\!].$$

Proof Immediately follows from the definition of FTS projection [13], Definition 9, and the rules of the transformation from FTA to FTS.

These different formulations of FTA semantics reflects the existence of two distinct approaches for FTA model checking.

5 Reachability Relation in FTA

We have presented the FTA formalism, its syntax and two 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 a formula Φ and an FTA fta defined over an FM fm, we define the F-satisfiability of fta with respect to ϕ as the feature expression noted $fta \models \phi$ and defined such that

$$(fta \models \phi)(p) = (fta_{|p} \models \phi).$$

A first, product-based method to compute this consists in computing the projection of the FTA onto each valid product, thereby producing a set of TA, and checking all these TA against the property. An important weakness of this verification approach is its inability to take into account that multiple products can share common behaviour. To overcome this limitation, we propose another method that combines the abstraction of time using zones with the efficient algorithms previously designed for FTS. We illustrate the use of this approach to compute reachability relations. 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.

To compute reachability, we first define a new successor operator which takes both variability and time constraints into account. Since features do not occur in invariants and transition guards in the separated form (see Section 4.1), 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 ζ, the target location l', and the chosen transition $t = (l, g, \alpha, R, l')$. It is given by

$$next(l, \zeta, t) = \big((\text{reset } R \text{ in } \zeta \wedge g) \wedge Inv(l')\big)^{\Uparrow} \wedge Inv(l')$$

where reset R in ζ' is the zone ζ' where every clock in R has been reset to 0 and $(\zeta')^{\Uparrow}$ is the zone ζ' after any elapsing of time [26]. The zone $next(l, \zeta, t)$ can be empty if $\zeta^{\Uparrow}$, 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 the value of clocks. 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.

Definition 13 (Successor relation in FTA) Let fta Let $fta = (Loc, Act, C, Trans, Loc_0, Inv, AP, L, fm, \gamma)$ be an FTA, fts its induced FTS. The successor relation of fts is a relation $Post \subseteq (Loc \times 2^{Val(C)}) \times (Loc \times 2^{Val(C)}) \times 2^{(2^F)}$ defined as

$$((l,\zeta),(l',\zeta'),\gamma) \in Post \Leftrightarrow \forall p \in [\![\gamma]\!] \exists t \bullet$$
$$t = (l,g,\alpha,R,l') \in Trans \bullet \gamma(t)(p) \wedge \zeta' = next(\zeta,l,t).$$

The successor relation as defined above is not necessarily minimal with respect to the inclusion of zones and sets of products. This is an issue we have to deal with during the exploration of the state space of the induced FTS. We next derive the definition of reachability in FTS from the successor relation.

Definition 14 (Reachability relation in FTA) Let fta be an FTA. The reachability relation in fta is a set of triplets $R(fta) \subseteq Loc \times 2^{Val(C)} \times 2^{(2^F)}$ such that $(l_k,\zeta_k,b_k) \in R(fta) \Leftrightarrow \exists (l_0,\zeta_0,b_0)\dots(l_k,\zeta_k,b_k)$ where $l_0 \in Loc_0$, $\zeta_0 = \{\eta_0\}$, $b_0 = \top$, and for all $i < k$ and $p \in [\![fm]\!]$ we have

$$b_{i+1}(p) \Leftrightarrow b_i(p) \wedge \exists (l_{i+1},\zeta_{i+1},\gamma) \in Post(l_i,\zeta_i) \bullet \gamma(p).$$

We can thus compute the reachability relation through successive computations of the successor relation – see Algorithm 1. The initial locations are always reachable by all the valid products and for a clock zone ζ_0 where every clock has its value first set to zero and where time can elapse as long as l_0's invariant is satisfied (Line 4–8). 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 $s = (l',\zeta',b')$, the procedure adds s in the relation if and only if there exists no element of the relation (l',ζ'',b'') such that the products represented by b' are also represented by b'' and the clock zone ζ' is a subset of ζ''. Otherwise, it means that we already know that the products satisfying b' can reach l' in the clock zone ζ'. 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 a triplet in the relation it removes every triplet that has become redundant because of the addition of s (Line 11). The redundancies originate from the fact that, like the successor relation, the reachability relation is not always minimal. To overcome this, we can use a particular union operator we discuss further in this section.

As for TA, a forward reachability algorithm using zone-based abstraction does not always terminate without an extrapolation operator [8]. Since termination is a classical problem already solved in TA model-checking and is independent of the addition of variability, we do not deal with it.

Input: $fta = (Loc, Act, C, trans, Loc_0, Inv, AP, L, fm, \gamma)$.
Output: $R(fta)$.

```
1  R ← ∅;
2  Stack ← [];
3  foreach s0 ∈ {(l0, ζ0, fm) | l0 ∈ Loc0} do
4  |   R ← R ∪ {s0};
5  |   R ← R ∪ {l0, (ζ0)^⇑ ∧ Inv(l0), fm};
6  |   push({l0, (ζ0)^⇑ ∧ Inv(l0), fm}, Stack);
7  end
8  while Stack ≠ [] do
9  |   (l, ζ, b) ← pop(Stack);
10 |   foreach (l', ζ', b') ∈ Post(l, ζ, b) do
11 |   |   if ∄(l', ζ'', b'') ∈ R • [[b']] ⊆ [[b'']] ∧ ζ' ⊆ ζ'' then
12 |   |   |   R ← R ⊔ {(l', ζ', b')};
13 |   |   |   push((l', b', ζ'), Stack);
14 |   |   end
15 |   end
16 end
17 return R
```

Algorithm 1: Reachables(fta)

To increase the efficiency of this algorithm, we have to deal with a doubly-partial order, though. This issue is the inheritance of the model-checking algorithms for TA and FTS. For instance, let us suppose that the system already reached the state (l,ζ) for the products satisfying b and that it reaches a state (l,ζ') for the same products in the future. If ζ' is a subset of ζ, it is useless to continue the search from the latter state because the set of states reachable from (l,ζ') would also be a subset of the states reachable from (l,ζ). Similarly, if the system first reaches (l,ζ) for products satisfying b, and then reaches this state for products satisfying b' with $b' \Rightarrow b$, we should not pursue the exploration: since any products satisfying b' also satisfies b, any information obtained by further exploration would be redundant. This doubly-partial order may lead to trickier phenomena, where both the zones and the feature expressions have to be compared. To cope with such cases, we define formal rules that determine with which zone and feature expression we must resume the exploration. Applying these rules reduces the number of explored states during a verification. More precisely, they guarantee that the reachability relation is an *antichain* [18], thereby eliminating redundant elements from the reachability relation.

Definition 15 (Antichain in FTA reachability relation) Let $R \subseteq Loc \times 2^{Val(C)} \times 2^{(2^F)}$. R is an antichain if and only if

$$\forall r, r' \in R \bullet r \leq r' \Rightarrow r = r'$$

where

$$(l,\zeta,b) \leq (l',\zeta',b') \Leftrightarrow (l=l') \wedge (\zeta \subseteq \zeta') \wedge ([\![b]\!] \subseteq [\![b']\!]).$$

To preserve the antichain upon union, a specific operator is needed. Algorithm 2 shows the procedure to add a given triplet (l,ζ,b) to an antichain R. The idea is to iterate over the elements in R to remove any redundancy contained in (l,ζ,b) with respect to the relation. Let (l,ζ',b') be an element of R. If $\zeta \subset \zeta'$ then the triplet is redundant for any product satisfying both b and b'; indeed, we already know at what time these product can reach location l. To remove this redundancy, we transform b into $b \wedge \neg b'$ (see Lines 6–9). If $\zeta \supset \zeta'$ then the redundancy lies in R. In this case, we change b' into $b' \wedge \neg b$ and we remove the element from R if b' does not encode any valid product (Lines 10 – 15). If $\zeta = \zeta'$, we retain the products in $[\![b']\!]$ (Lines 16–18) and at the end of the algorithm, add to this set the products satisfying b that are not redundant (Line 28). Finally, any element (l',ζ',b') with $l \neq l'$ is added to R' (Lines 19–21).

In the end, our algorithm combines existing model-checking approaches. The only new construct (i.e. which exists neither in TA nor in FTS) we introduced 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 relation, where zones and products are considered orthogonally. These nice characteristics make an implementation considerably easier, as they allow one to reuse efficient data structures like binary decision diagrams and decision bounded matrices to encode feature expression and clock zones, respectively.

Input: $R \subseteq \{l\} \times 2^{Val(C)} \times 2^{(2^F)}$,
$(l,\zeta,b) \in Loc \times 2^{Val(C)} \times 2^{(2^F)}$.
Output: R', an antichain such that

$$\forall r = (l,\{\eta\},\mathbb{B}(\{p\}) \bullet (\exists r' \in R' \bullet r \leq r') \Leftrightarrow (\exists r'' \in R \cup \{l,\zeta,b\} \bullet r \leq r'').$$

```
1  R' ← ∅;
2  equal ← ⊥;
3  while (b ∧ fm ⊭ ⊥) ∧ (R ≠ ∅) do
4      Take (l', ζ', b') from R;
5      if l = l' then
6          if ζ ⊂ ζ' then
7              b ← b ∧ ¬b';
8              R' ← R' ∪ {(l, ζ', b')};
9          end
10         else if ζ ⊃ ζ' then
11             b' ← b' ∧ ¬b;
12             if b' ∧ fm ⊭ ⊥ then
13                 R' ← R' ∪ {(l, ζ', b')};
14             end
15         end
16         else if ζ = ζ' then
17             equal ← b';
18         end
19         else
20             R' ← R' ∪ {(l, ζ', b')};
21         end
22     end
23     else
24         R' ← R' ∪ {(l', ζ', b')};
25     end
26 end
27 if equal ≠ ⊥ then
28     R' ← R' ∪ {(l, ζ, equal ∨ b)};
29 end
30 if R ≠ ∅ then
31     R' ← R' ∪ R;
32 end
33 return R'
```

Algorithm 2: $R \sqcup \{(l,\zeta,b)\}$

6 Abstraction Refinement for FTA

Abstraction methods for TA [25,34] were proposed as a means of overcoming the two-source complexity of TA verification: the state-explosion problem inherent to any model-checking approach, and the complexity of TA algorithms which is exponential in the number of clocks [26]. By introducing features in FTA, we add a third source of complexity: indeed, the worst-case time complexity of verifying a variability-aware model is exponential in the number of features, that is, linear in the number of products [13]. FTA abstraction methods thus have to also cope with all this. Accordingly, we discuss two kinds of abstractions: one that abstracts the variability (feature abstraction), and the other that leans on TA abstraction to reduce the number of states in the induced FTS. These kinds of abstractions can be used separately or jointly, as shown in our previous work on FTS abstraction [21]. In whichever case, any abstract FTA yielded by those will be an *existential abstraction* of the original FTA [12], in the sense that any behaviour of the original FTA (in terms of timed execution trace) will also be a behaviour of the abstract FTA. On the contrary, the abstract FTA may contain behaviours that were not present in the original FTA, due to, e.g., the merging of similar states. Formally, an existential abstraction for FTA is defined as follows.

Definition 16 (Existential Abstraction for FTA) An existential abstraction function for FTA is a surjection $h : Loc \to \widehat{Loc}$ such that $h(l) = h(l') \Rightarrow L(l) = L(l')$. An abstraction of an FTA fta under h is an FTA $fta_h = (\widehat{Loc}, Act, Trans_h, (Loc_0)_h, Inv_h, AP, L_h, d, \gamma_h)$ where:

- $\widehat{Loc} = \{h(l) | l \in Loc\}$,
- $(Loc_0)_h = \{h(l_0) | l_0 \in Loc_0\}$,

- $L_h(h(l)) = L(l)$, $Inv_h(h(l)) \sqsupset \bigcup_{l':h(l')=h(l)} Inv(l')$,
- $Trans_h$ is defined such that $(\mathrm{h}(\mathrm{l}_1), g, \alpha, R, h(l_2)) \in Trans_h$ if and only if

$$\exists l'_1, l'_2 : (l'_1, g, \alpha, R, l'_2) \in Trans_h \wedge h(l_1) = h(l'_1) \wedge h(l_2) = h(l'_2)$$

- γ_h is such that

$$\Big(\bigvee_{l \in h^{-1}(\widehat{l}), l' \in h^{-1}(\widehat{l'}) \bullet t=(l,g,\alpha,R,l') \in Trans} \gamma(t)\Big) \Rightarrow \gamma_h(\widehat{l}, g, \alpha, R, \widehat{l'}) \quad (1)$$

with $h^{-1}(\widehat{l}) = \{l \in Loc \mid h(l) = \widehat{l}\}$.

Remember that any FTA with featured clock constraints corresponds to an FTA with standard clock constraints and additional feature expressions on the transition (see Section 4.1). In the above definition, we assume that this transformation has first been applied and thus that our FTA contains no featured clock constraint.

Intuitively, an existential abstraction can merge multiple locations that have the same labels (through the definition of h) and can generalize the γ function to make a transition executable by at least the same (if not more) products. To preserve the real-time behaviour of the original FTA, (i) the clock constraints on transition guards are not changed and (ii) the invariant of an abstract location $\widehat{l}$ (i.e. a clock constraint) must subsume the invariant of its corresponding concrete locations. Ideally, the invariant of the abstract location should be equal to the union of the invariants of the corresponding concrete locations, in order not to engender too many new timed behaviours. However, clock constraints are not closed under union; the resulting abstract FTA could thus be syntactically incorrect. In practice, one could choose an abstract invariant as close possible as the union of the concrete invariants so as to avoid creating to many new behaviours.

Given the above definition, the reachability relation in the abstract FTA fta_h will subsume the reachability relation of the concrete FTA fta. Indeed, merging locations does not remove any path present in the original FTA. Moreover, by making γ_h more general than γ, we ensure that any product can execute in the abstract FTA all the transitions it could execute in the original FTA. A consequence of this is that one can check a property on fta_h (which is easier to verify since it has fewer locations and a more general γ function) rather than on fta, and then infer whether this property also holds for fta. For instance, let $\Phi = \forall\Box_J\phi$. Then two cases may occur:

1. fta_h satisfies Φ, that is, all of its paths satisfy ϕ during the interval of time J. Since fta has fewer behaviours (that is, a smaller reachability relation), this implies that all of its paths also satisfy ϕ during J.
2. fta_h does not satisfy Φ that is, it has at least one path that does not satisfy ϕ during J. This path can either also be a valid path of fta or a path that was created because of the abstraction. In this latter case, we can detect that this execution trace – the counterexample – is *spurious* by replaying it in fta, following the principles of the well-known CEGAR loop [12].

Similarly, consider $\Phi = \exists\Diamond_J\phi$. Again, we distinguish between two cases:

1. fta_h does not satisfy Φ, that is, it has no reachable location that satisfies ϕ within J. Since fta has fewer behaviours (that is, a smaller reachability relation), this implies that fta does not contain any path where ϕ is satisfied either.
2. fta_h satisfies Φ, that is, it has a path to a location satisfying ϕ that can be reached within J. This path can either also be a valid path of fta or a path that was created because of the abstraction function. Replaying this (positive) example on fta again allows one to check whether or not it is spurious.

When a spurious execution trace is found, the abstraction must be refined in order to make it disappear from fta_h. Then we verify the resulting abstract FTA from scratch, with the hope of not finding another spurious execution trace.

Another source for spuriousness lies in that an execution trace may be executable by more products in fta_h than in fta. This is because the γ_h function can be more general than γ so as to reduce the complexity due to variability. In such a case, the execution trace provides guarantees only for the product that can really execute it; the others must be verified again after modifying γ_h such that these products cannot execute the execution trace anymore. The products actually able to execute the transition are symbolically encoded by the conjunction of all $\gamma(t)$, for any transition t that belongs to the spurious trace. Thus, to limit the execution of the trace in fta_h, only to those products, it suffices to replace $\gamma_h(t)$ by $\gamma(t)$ for all transition of the trace.

In what follows, we discuss each kind of spuriousness and give insights into how an FTA abstraction can be refined to eliminate spurious execution trace. Following the principles depicted in our previous work on FTS abstraction [21], location abstraction and feature abstraction can be combined in a straightforward and orthogonal way. However, previous empirical evaluations

tend to show that one should instead use one or the other exclusively [21].

6.1 Feature Abstraction

We begin by discussing feature abstraction, that is, how to compute a first γ_h function and how to refine it when a spurious execution trace is found. Initially, we propose to make every transition executable by all products, that is, we define that $\gamma_h(t) = \top$ for any transition t. This comes down to associating each product of the FTA $fta = (Loc, Act, C, trans, Loc_0, Inv, AP, L, fm, \gamma)$ to the TA $ta = (Loc, Act, C, trans, Loc_0, Inv, AP, L)$. Therefore, efficient model-checking algorithms for TA (such as those implemented in UPPAAL [7]) can be reused and, depending on the results, this might be sufficient to solve the FTA model-checking problem. This gives us confidence that our previous findings regarding the efficiency of this initial abstraction [21] hold in this case as well.

Once an execution trace has been found, we must determine whether all products associated to it are really able to execute it in fta. Let $\sigma = s_0 \xrightarrow{t_1} \dots \xrightarrow{t_k} s_k$ be the execution trace to check, such that $s_i = (l_i, \zeta_i, b_i)$ and t_i is the executed transition that led from s_{i-1} to s_i. The products able to execute it in fta_h are given by $\bigcap_{i=1\dots k} \gamma_h(t_i)$, while those able to execute it in fta are given by $\bigcap_{i=1\dots k} \gamma(t_i)$. Then, σ is spurious for the products

$$\bigcap_{i=1\dots k} \gamma_h(t_i) \wedge \neg \bigcap_{i=1\dots k} \gamma(t_i) \wedge fm$$

and consequently, these products must be verified again. As for the other products, they can be ignored from now on since we know that they can execute σ.

Algorithm 3 generalizes the above reasoning and implements the refinement step of our feature abstraction. In the inputs, the feature expression χ represents the products to verify (initially, all the products in $[\![fm]\!]$). As for the outputs, the algorithm returns true if and only if all the products satisfying χ can execute σ. If that is not the case, it returns two outputs:

1. χ', the feature expression encoding the products that must be verified again, that is, the products satisfying χ that cannot execute σ.
2. a new function $\gamma_{h'}$ that is equivalent to γ_h except for transitions of σ that cannot be executed by the same products in fta_h and fta. For each of these transitions t, $\gamma_{h'}(t)$ equals $\gamma(t)$, that is, we annotate t with its original feature expression as defined in fta.

Initially, the algorithm sets $\gamma_{h'}$ to γ_h (Line 2). It also sets a feature expression σ_b to $\top$. Then it iterates over all transitions of σ (Lines 4 – 10). For each i-th transition t_i, it first checks whether all products satisfying χ can execute t_i (Line 5). If that is not the case (Lines 6 – 8), we first replace $\gamma_{h'}(t_i)$ by $\gamma(t_i)$ so as to make t_i executable only by that can execute it in the concrete FTA. We also conjunct σ_b with $\gamma(t_i)$, that is, σ_b encodes the products that can execute σ until t_i (included). After the loop, if a refinement is needed (Lines 11 – 14), we encode in χ' the products satisfying χ that could not execute σ and return it together with $\gamma_{h'}$.

If a refinement was needed, we create a new abstract FTA $fta_{h'}$ that is equivalent to fta_h except that γ_h has been replaced by $\gamma_{h'}$. Then, we verify $fta_{h'}$ against Φ only for the products satisfying χ' and, if the verification returns an execution trace, we apply Algorithm 3 anew to determine whether some products satisfying χ' cannot execute it. We repeat this CEGAR loop until no (counter-)example is found or until this (counter-)example is not spurious. The termination of this process is established by the following theorem.

Theorem 2 *The CEGAR process for FTA feature abstraction terminates.*

Proof At each refinement step, the returned feature expression χ' encodes an increasingly smaller set of products. Let us assume that the CEGAR loop iterates indefinitely. Then, $[\![\chi']\!]$ converges to the empty set, that is, no product remains to be verified. At this point, the next verification will return no (counter-)example and the CEGAR loop terminates, which contradicts our absurd hypothesis that the loop is infinite.

6.2 Location Abstraction

While the previous section is focused on abstracting from the variability of an FTA, we now study how we can adapt abstraction techniques for TA to apply these to FTA, and use these techniques to reduce the FTA verification effort. In this case, we assume that feature expressions are left untouched.[1] The complexity of TA verification being exponential in the number of clocks [26], a common way to achieve this consists of merging, in an induced TS, all states that correspond to the same location. Thus, every state of this TS (which corresponds to a location and a clock valuation) will be able to execute to execute any discrete transitions. This comes down to replacing all transition guards by

[1] This is without loss of generality, since variability and time in FTA can be considered orthogonally.

Input: fta and fta_h, an execution trace $\sigma = s_0 \xrightarrow{t_1} \dots s_k$ of fta_h and a feature expression χ.
Output: Return $\top$ if $\sigma \in Prefix([\![(fta)_{|p}]\!])$ for all $p \in [\![\chi]\!]$; otherwise, return $(\chi', \gamma_{h'})$ such that $\chi' = \chi \wedge \sigma \notin Prefix([\![(fta)_{|p}]\!])$ and $\gamma_{h'}(t_i) = \gamma(t_i) \wedge \gamma_{h'}(t_i) \neq \gamma_h(t_i)$ for at least one t_i and $\gamma_{h'}(t) = \gamma(t)$ for any transition $t \notin \{t_1 \dots t_k\}$.

$refined \leftarrow \bot$;
$\gamma_{h'} \leftarrow \gamma_h$;
$\sigma_b \leftarrow \top$;
for $i = 1$ *to* k **do**
  if $\chi \wedge \gamma_h(t_i) \not\Rightarrow \gamma(t_i)$ **then**
    $\gamma_{h'}(t_i) \leftarrow \gamma(t_i)$;
    $\sigma_b \leftarrow \sigma_b \wedge \gamma(t_i)$;
    $refined \leftarrow \top$;
  end
end
if $refined$ **then**
  $chi' \leftarrow \chi \wedge \neg\sigma_b$;
  return $(\chi', \gamma_{h'})$;
end
else
  return $\top$;
end

Algorithm 3: IsFeatureSpurious(fta, fta_h, σ, χ)

the clock constraint $\top$, and thus making all transitions *always* executable regardless of the current time zone. It may thus happen that a transition is executable at a time it should not be, which thereby creates additional behaviour. If this behaviour leads to spurious (counter-)examples, one has to refine the abstraction to make it disappear. To achieve this, Nagaoka et al. [34] proposed a refinement technique based on duplicating locations and transitions. The core idea is to separate states from the induced TS that correspond to last valid location of the spurious example in two groups: (i) those that are reached by the last transition of the spurious example and (ii) those that can execute the following, invalid transition. Then the states of each group are merged (abstracted) in a copy of the original location. Since in FTA variability is orthogonal to time, the same approach can be reused for FTA and thus constitutes a second abstraction method.

7 Related Work

7.1 Real-Time VIS Verification

Since our first paper on verification of real-time software product lines [19], recent papers have extended and complemented our work.

Cledou et al. [16] introduce Interface Featured Timed Automata, a formalism that extends FTA with interface variables that allow for synchronized parallel composition. Then they transform an IFTA into a network of timed automata and perform verifications using UPPAAL. Another extension is the Configurable Parametric Timed Automata (CoPTA) [32] which, unlike FTA, support parametric timed constraints where the parameters are feature attributes that impact the behaviour of the VIS [23].

Dimovski and Wasowski [27] design a three-step abstraction method for FTA. First, they define a partition of the valid products. Second, they derive, for each element of the partition (i.e. subset of the products), the FTA encoding only the behaviour of the products in the subset. They achieve this by generalizing the definition of FTA projection (see Definition 9) to a set of products.[2] Then, for each projected FTA, they apply feature abstraction and use UPPAAL to verify the resulting TA. The principle of partitioning the variability of the VIS is interesting and can lead to drastic reductions in verification time if an appropriate partition is found. Combining this approach with the full abstraction-refinement presented above and/or with state abstraction forms a promising direction for future work.

Most recently, Lazreg et al. [31] assess the functional feasibility of multiple configurations of data-flow-oriented systems. Their problem is challenging in that these systems must complete their execution within hard real-time deadlines and at minimal costs. To address it, they designed a tool leaning on ProVeLines [24] and UPPAAL CORA [6].

7.2 Abstraction refinement

Clarke *et al.* [12] were the first to introduce CEGAR. They presented the principles of existential abstraction, designed an algorithm used to detect spurious counterexamples, and proposed a refinement algorithm. Several model checkers make use of predicate abstraction to speed up the verification.

At the TACAS '13 software verification competition [9], UFO [1], LLBMC [29], CPAChecker [10], and ESBMC [33] were nominated the fastest model checkers for product lines. These tools either rely on a product-based approach or verify a model that includes the behaviour of all products (i.e. based on a product simulator [3] or on configuration lifting [35]). In the first case, abstraction can be applied to the models of the individual products but the commonality between these is not exploited. The second case offers interesting perspectives regarding an efficient application of abstraction

[2] This is akin to the restriction operator defined in [19] for FTS.

on all products. However, this approach can determine whether or not *all* products satisfy the property, while we want to identify *which* products are error-prone.

Abstraction refinement has also been studied for real-time. Dierks et al. [25] propose an automated implementation of the CEGAR loop for PLC automata. They show the efficiency of abstraction by applying it to models that are initially too large to be verified. This increases our confidence that studying abstraction for FTA is also worthwhile. Nagaoka et al. [34] propose a state abstraction technique for timed automata. Their refinement step involves the creation of new states and transitions in a way that behavioural equivalence is preserved. None of these work, however, are interested in variability-intensive models.

8 Conclusion

The efficiency of FTA algorithms mostly depends on the data structures used to represent time and variability [22]. To our knowledge, only separate encodings, *viz.* DBMs and BDDs respectively, have been used. As alternatives, we could use a data structure that combines zone and binary variables representations. Clock Restriction Diagrams [36] and Constraint Matrix Diagrams [28] appear as promising candidates.

Also, although CEGAR techniques were effective for FTS [21], this has to be confirmed for FTA empirically. Accordingly, one of our next task is to evaluate our abstraction methods on challenging case studies that involve heavy timed constraints and large variability. This might require us to combine the above-mentioned alternative encodings with additional abstraction heuristics in order to obtain acceptable computation time. We may also lean on Dimovski and Wasowski's work [27] to improve our abstraction-refinement methods.

References

1. Albarghouthi, A., Li, Y., Gurfinkel, A., Chechik, M.: Ufo: A framework for abstraction- and interpolation-based software verification. In: CAV. pp. 672–678 (2012)
2. Alur, R., Courcoubetis, C., Dill, D.: Model-checking in dense real-time. Inf. Comput. **104**, 2–34 (May 1993)
3. Apel, S., Speidel, H., Wendler, P., von Rhein, A., Beyer, D.: Feature-interaction detection using feature-aware verification. In: ASE'11. pp. 372–375. IEEE (2011)
4. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press (2008)
5. ter Beek, M.H., Fantechi, A., Gnesi, S., Mazzanti, F.: Modelling and analysing variability in product families: Model checking of modal transition systems with variability constraints. Journal of Logical and Algebraic Methods in Programming **85**(2), 287 – 315 (2016)
6. Behrmann, G., Fehnker, A., Hune, T., Larsen, K., Pettersson, P., Romijn, J.: Efficient guiding towards cost-optimality in uppaal. In: Margaria, T., Yi, W. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 174–188. Springer Berlin Heidelberg, Berlin, Heidelberg (2001)
7. Bengtsson, J., Larsen, K.G., Larsson, F., Pettersson, P., Yi, W.: UPPAAL in 1995. In: TACAS'96. pp. 431–434. Springer-Verlag (1996)
8. Bengtsson, J., Yi, W.: Timed automata: Semantics, algorithms and tools. In: Lectures on Concurrency and Petri Nets. pp. 87–124 (2003)
9. Beyer, D.: Second competition on software verification - (summary of sv-comp 2013). In: TACAS '13. pp. 594–609 (2013)
10. Beyer, D., Keremoglu, M.E.: Cpachecker: A tool for configurable software verification. In: CAV '11. pp. 184–190 (2011)
11. Chechik, M., Devereux, B., Easterbrook, S.M., Gurfinkel, A.: Multi-valued symbolic model-checking. ACM Trans. Softw. Eng. Methodol. **12**(4), 371–408 (2003)
12. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E., Sistla, A. (eds.) Computer Aided Verification, LNCS, vol. 1855, pp. 154–169. Springer Berlin / Heidelberg (2000)
13. Classen, A., Cordy, M., Schobbens, P.Y., Heymans, P., Legay, A., cois Raskin, J.F.: Featured transition systems: Foundations for verifying variability-intensive systems and their application to LTL model checking. Transactions on Software Engineering pp. 1069–1089 (2013)
14. Classen, A., Heymans, P., Schobbens, P.Y., Legay, A.: Symbolic model checking of software product lines. In: ICSE'11. pp. 321–330. ACM (2011)
15. Classen, A., Heymans, P., Schobbens, P.Y., Legay, A., Raskin, J.F.: Model checking lots of systems: efficient verification of temporal properties in software product lines. In: ICSE'10. pp. 335–344. ACM (2010)
16. Cledou, G., Proença, J., Soares Barbosa, L.: Composing families of timed automata. In: Dastani, M., Sirjani, M. (eds.) Fundamentals of Software Engineering. pp. 51–66. Springer International Publishing, Cham (2017)
17. Clements, P.C., Northrop, L.: Software Product Lines: Practices and Patterns. SEI Series in Software Engineering, Addison-Wesley (August 2001)
18. Comtet, L.: Sperner Systems (§7.2). Springer Berlin Heidelberg (1974)
19. Cordy, M., Classen, A., Heymans, P., Schobbens, P.Y., Legay, A.: Managing evolution in software product lines : A model-checking perspective. In: VaMoS'12. pp. 183–191. ACM (2012)
20. Cordy, M., Classen, A., Perrouin, G., Heymans, P., Schobbens, P.Y., Legay, A.: Simulation-based abstractions for software product-line model checking. In: ICSE'12. pp. 672–682. IEEE (2012)
21. Cordy, M., Heymans, P., Legay, A., Schobbens, P.Y., Dawagne, B., Leucker, M.: Counterexample guided abstraction refinement of product-line behavioural models. In: FSE'14. pp. 190–201. ACM (2014)
22. Cordy, M., Heymans, P., Schobbens, P.Y., Legay, A.: Behavioural modelling and verification of real-time software product lines. In: SPLC'12. ACM (2012)
23. Cordy, M., Schobbens, P.Y., Heymans, P., Legay, A.: Beyond boolean product-line model checking: Dealing with feature attributes and multi-features. In: ICSE'13. pp. 472–481. IEEE (2013)

24. Cordy, M., Schobbens, P.Y., Heymans, P., Legay, A.: Provelines: A product-line of verifiers for software product lines. In: SPLC'13. pp. 141–146. ACM (2013)
25. Dierks, H., Kupferschmid, S., Larsen, K.G.: Automatic abstraction refinement for timed automata. In: Raskin, J.F., Thiagarajan, P.S. (eds.) Formal Modeling and Analysis of Timed Systems. pp. 114–129. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
26. Dill, D.L.: Timing assumptions and verification of finite-state concurrent systems. In: Proceedings of the international workshop on Automatic verification methods for finite state systems. pp. 197–212. Springer-Verlag New York, Inc., New York, NY, USA (1990)
27. Dimovski, A.S., Wasowski, A.: From transition systems to variability models and from lifted model checking back to UPPAAL. In: Models, Algorithms, Logics and Tools - Essays Dedicated to Kim Guldstrand Larsen on the Occasion of His 60th Birthday. pp. 249–268 (2017)
28. Ehlers, R., Fass, D., Gerke, M., Peter, H.J.: Fully symbolic timed model checking using constraint matrix diagrams. In: Proceedings of the 2010 31st IEEE Real-Time Systems Symposium. pp. 360–371. RTSS '10, IEEE Computer Society, Washington, DC, USA (2010)
29. Falke, S., Merz, F., Sinz, C.: The bounded model checker llbmc. In: ASE '13. pp. 706–709 (2013)
30. Kramer, J., Magee, J., Sloman, M., Lister, A.: Conic: an integrated approach to distributed computer control systems. Computers and Digital Techniques, IEE Proceedings E **130**(1), 1–10 (1983)
31. Lazreg, S., Collet, P., Mosser, S.: Assessing the functional feasibility of variability-intensive data flow-oriented systems. In: 33rd Symposium on Applied Computing (2018)
32. Luthmann, L., Stephan, A., Bürdek, J., Lochau, M.: Modeling and testing product lines with unbounded parametric real-time constraints. In: Proceedings of the 21st International Systems and Software Product Line Conference - Volume A. pp. 104–113. SPLC '17, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3106195.3106204, `http://doi.acm.org/10.1145/3106195.3106204`
33. Morse, J., Cordeiro, L., Nicole, D., Fischer, B.: Handling unbounded loops with esbmc 1.20 - (competition contribution). In: TACAS. pp. 619–622 (2013)
34. Nagaoka, T., Okano, K., Kusumoto, S.: An abstraction refinement technique for timed automata based on counterexample-guided abstraction refinement loop **93-D**, 994–1005 (05 2010)
35. Post, H., Sinz, C.: Configuration lifting: Verification meets software configuration. In: ASE'08. pp. 347–350. IEEE CS (2008)
36. Wang, F.: Efficient model-checking of timed automata with clock-restriction diagram. In: APLAS. pp. 207–224 (2001)
37. Yi, W.: Real-time behaviour of asynchronous agents. In: Proceedings of CONCUR '90. pp. 502–520. Springer-Verlag New York, Inc., New York, NY, USA (1990)