# Scheduling and Analysis of Real-Time Software Families

Hamideh Sabouri*, Mohammad Mahdi Jaghoori†, Frank de Boer† and Ramtin Khosravi*‡
*University of Tehran, Tehran, Iran
Email: {rkhosravi,sabouri}@ece.ut.ac.ir
†CWI, Amsterdam, The Netherlands
Email: {frb,jaghoori}@cwi.nl
‡School of Computer Science, Institute for Research in Fundamental Sciences (IPM), Tehran, Iran

***Abstract*—A software product line describes explicitly the commonalities of and differences between different products in a family of (software) systems. A formalization of these commonalities and differences amounts to reduced development, analysis and maintenance costs in the practice of software engineering. An important feature common to next-generation real-time software systems is the need of application-level control over scheduling for optimized utilization of resources provided by for example many-core and cloud infrastructures. In this paper, we introduce a formal model of real-time software product lines which supports variability in scheduling policies and rigorous and efficient techniques for modular schedulability analysis.**

***Keywords*-Application-level Scheduling, Real-Time, Software Product Lines, Formal Methods, Automata Theory**

## I. Introduction

Software Product Line (SPL, for short) engineering enables proactive reuse by developing a family of related products instead of individual products separately. To this end, the commonalities of and differences between the products should be modeled explicitly [1]. Feature models are widely used to model the variability points in software product lines. A feature model is a tree of features containing mandatory and optional features as well as other constraints, e.g., mutual exclusion. A product is then defined by a valid combination of features, and a family is the set of all possible products [2].

Deployment unto many-core and cloud infrastructures poses new challenges for the sofware family abstraction that mostly concern the need of leveraging resources and resource management into SPL engineering. In particular, application-level control over scheduling for optimized utilization of resources requires completely novel formal techniques for the development of software product lines and their analysis with respect to quality-of-service. In this paper, we introduce a formal model of real-time software product lines which supports variability in scheduling policies and rigorous and efficient techniques for modular schedulability analysis.

We propose the actor model [3] as a basic computational model for leveraging application-level scheduling policies into SPL engineering because in this model every actor provides a natural abstraction of a dedicated processor. As such the actor model provides a natural means for modeling parallel and distributed software. Programming languages based on this paradigm, such as Erlang [4] and Scala [5], have recently gained in popularity, in part due to their support for scalable concurrency. However, for optimal use of both hardware and software resources, we cannot avoid leveraging scheduling and performance related issues from the underlying implementation to the application level as argued for example in [6], [7].

Based on automata theory, Jaghoori et al. [8], [9] provide a formal framework for modular schedulability analysis of real-time concurrent objects (i.e., actors extended with synchronization patterns). We define a task to be a message plus a deadline, which defines the schedulability requirement. Each object here exposes a behavioral interface that specifies at a high level of abstraction and in the most general terms how the object may be used, i.e., the timings and deadlines of the messages sent and received. Objects are analyzed individually for schedulability with respect to their behavioral interfaces, thus providing schedulable off-the-shelf modules. As in modular verification [10], a system composed of such modules is guaranteed to be schedulable if it does not put stronger requirements than those specified in the behavioral interfaces. In this paper, we further integrate the priority-based scheme for specifying scheduling strategies introduced in [7] into this framework.

In order to generalize the above framework to actor-based real-time software product lines we introduce variability into both the detailed actor models and their behavioral interfaces by specifying behavior (including scheduling policies) at both levels conditionally on the set of selected features. As a general model of behavior, we introduce the notion of a *timed automata family (TAF)* as an extension of timed automata [11]. Every action and edge in a timed automata family is supplied with an *application condition (AC)*, defined as a boolean formula over the set of features, determining under what conditions the action/edge is applicable. By (de)selecting each feature, some application conditions become unsatisfiable. One can (partially) configure a TAF by explicitly (de)selecting some features, formally resulting in the removal of the edges and actions with an unsatisfiable AC. Removal of an action causes the elimination of all corresponding edges.

Given the notion of a *timed automata family (TAF)* we generalize the modular schedulability analysis of Jaghoori et al. [8] to families of real-time individual actors. Such

This research was in part supported by a grant form IPM. (No. CS1390-4-02).

a modular analysis greatly reduces the complexity of the schedulability analysis of a family of complete actor systems. However, analyzing a family is still much more costly than analyzing a single product (i.e., actor, in our case). Therefore, in order to optimize the sharing of features in a family, we propose in this paper a novel approach in which the decision about including or excluding a feature during analysis is postponed as far as possible. This approach has been implemented using UPPAAL [12].

*The Elevator Case Study:* We consider an elevator system as our case study in this paper. Each floor may send a request to the elevator along with the desired deadline. The elevator moves among different floors to serve their requests in time. A family of elevator systems is formed by considering two optional features named *weight sensor* and *VIP floor*. The elevator systems that support the *weight sensor* feature, do not move after stopping in one floor if the weight of the persons in the lift exceeds its maximum limit. The second feature gives priority to a special VIP floor over the other ones. Consequently, the VIP floor expects a lower service time. Furthermore, we consider a feature named *Floors* to specify the number of floors that the elevator supports.

### A. Related Work

To the best of our knowledge, our work is the first formal attempt to model real-time software families and their schedulability analysis. To this end, we extend timed automata with feature models. As in Feature Petri-nets [13], we use application conditions, i.e., boolean logical formulas over a set of features, to indicate the feature combinations that enable each transition. This is also similar to feature transition systems [14], where annotations are added to transitions to indicate their corresponding features. Additionally, we assign application conditions to actions, allowing for more natural models. We define a refinement notion on Timed Automata Families enabling a modeler to preselect some features, which gives us a part of the family thus simplifying the analysis. This is more powerful than refinement in modal transition systems [15], where a family is modeled by considering *may* and *must* transitions. A may transition can be removed or retained when refining an MTS, thus generating different products. Gruler et al. [16] model a product family using PL-CCS, which extends CCS by the *variant* operator for specifying alternative processes.

In contrast to the fine-grained schemes above, the recent approach of delta modeling [17] provides a compositional paradigm, suitable for object-oriented programs, in which objects or some of their methods can be replaced in different products. Our framework can be used at a fine, as well as coarse, granularity. In fact, we use TAF to model families of concurrent objects, introducing variability in three levels: behavioral interfaces, schedulers and method definitions. In principle, delta models can be translated into our scheme.

There exist also many works on schedulability analysis. Many approaches (e.g., that of operations research) are based on simple task generation patterns like periodic tasks and rate-monotonic analysis [18], [19]. Although useful in many cases, such techniques are too coarse in many distributed systems and produce pessimistic outcomes. Unlike these works, we work with non-uniformly recurring tasks as in task automata [20] which fits better the nature of message passing in concurrent objects. Related work on interfaces extended with user-defined (i.e., application-level) scheduling policies, as for example described in [9], [21], is particularly tailored to basic (component-based or object-oriented) software. The main contribution of this paper is to extend schedulability analysis to software families based on real-time actors.

## II. Background: Software Product Lines

### A. Feature Model

Commonalities and differences among different products are modeled explicitly in software product line engineering. Feature models are widely used for this purpose. A feature model represents all possible products of a software product line in terms of features and relationships among them. A feature is a distinctive aspect, quality, or characteristic of a system. For example, having a weight sensor or supporting a VIP floor are some features of an elevator system.

A basic feature model is a tree of features that allows the *mandatory*, *optional*, *or*, and *xor* relationships among features. It also includes *requires* and *excludes* constraints between features [22]. We also allow features with numeric values. A feature with a numeric value in range $[min, max]$ would correspond to a feature having $max - min + 1$ children with xor relationship, where each child represents one value in the corresponding range.

Figure 1 shows the feature model of the elevator case study. This model includes weight sensor and VIP floor as optional features. It also includes floors feature as a numeric feature. A possible product of this feature model is an elevator with a weight sensor that does not support a VIP floor and serves four floors.

A feature model can be represented by a corresponding propositional logic formula in terms of a set of boolean variables [23]. Each boolean variable corresponds to a feature and its value indicates if the feature is included or excluded. The ultimate formula $\psi_F$ is the conjunction of implications from: Every child feature to its parent feature, every parent to its mandatory child feature, every parent to or/xor of its children that have an or/xor relationship, every feature $f$ to other features that $f$ requires, and every feature $f$ to the negation of other features that $f$ excludes. We may describe a numeric feature $f$ in the propositional logic formula by defining it as an integer instead of a boolean variable and $min \leq f \leq max$ is equivalent to an atomic proposition that represents presence of $f$. The corresponding propositional logic formula of feature model depicted in Figure 1 is:

$$\begin{aligned}\psi_F = [&(WS \rightarrow E) \ \wedge \ (VIP \rightarrow E)\\ &\wedge \ ((3 \leq F \leq 5) \rightarrow E) \ \wedge (E \rightarrow (3 \leq F \leq 5))]\end{aligned}$$

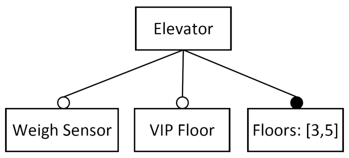


Figure 1. The feature model with numeric values

where $E, WS, VIP$ and $F$ represent elevator, weight sensor, VIP floor, and floors feature respectively.

Using the set of features $\mathcal{F}$ and the propositional formula $\psi_F$ that represents the constraints among features, we define a feature model as follows and use it in the rest of the paper. An advantage of this approach is that we are not limited to a specific notation for describing feature models and we use their semantics instead.

*Definition 1 (Feature Model):* A feature model is a two tuple $FM = (\mathcal{F}, \psi_F)$ where

- $\mathcal{F} = \{f_1, ..., f_n\}$ is the set of features
- $\psi_F$ is a propositional logic formula representing the constraints among features $\square$

## B. Configuration

A configuration keeps track of inclusion, exclusion, or values assigned to features. The root feature does not appear in a configuration as it is included in all products.

*Definition 2 (Configuration):* Having a set of features $\mathcal{F} = \mathcal{F}_\alpha \cup \mathcal{F}_\beta$ with $n$ features, where $\mathcal{F}_\alpha$ and $\mathcal{F}_\beta$ represent sets of boolean features and numeric features respectively, a configuration is defined as $c \in \{true, false, v, ?\}^n$ where

- $c_i = true$ represents inclusion of $f_i \in \mathcal{F}_\alpha$
- $c_i = false$ represents exclusion of $f_i \in \mathcal{F}_\alpha$
- $c_i = v$ represents assigning the value $min \leq v \leq max$ to $f_i \in \mathcal{F}_\beta$ where $[min, max]$ is the range that is associated to $f_i$
- $c_i = \ ?$ shows that no decision is made for $f_i$ $\square$

A configuration is *decided* if it does not contain any '?' values. In other words, a decision is made about inclusion, exclusion, or value of all features in feature set $\mathcal{F}$. Otherwise, we call the configuration *partial*.

We may validate a configuration $c$ with respect to a feature model $FM = (F, \psi_F)$ by applying an SMT-solver [24] on the projection of $\psi_F$ over $c$. We represent an SMT-solver using function $\mathcal{S}$ that is defined as follows.

*Definition 3 (Satisfiability Function):* We represent a SMT-solver using function $\mathcal{S} : \Theta_F \rightarrow \{true, false\}$, where $\Theta_F$ is the set of all possible propositional logic formulas over feature set $\mathcal{F}$. This function returns *true* if there exists a substitution for its variables which makes $\psi_F =$ *true* where $\psi_F \in \Theta_F$. Otherwise, it returns *false*. $\square$

The projection of a propositional logic formula $\psi_F$ over a configuration $c$ is defined as follows.

*Definition 4 (Propositional Logic Formula Projection):* Assume that $v_{f_i}$ is the corresponding variable of feature $f_i$ in propositional logic formula $\psi_F$. The projection of the formula $\psi_F$ over a configuration $c$, denoted by $\psi_F|_c$, results another formula $\psi'_F$ where $v_{f_i}$ is substituted with *true*/*false*/$v \in [min, max]$ if $c_i$ is *true*/*false*/$v$ respectively. $\square$

Ultimately, a configuration $c$ is valid if $\psi_F|_c$ is satisfiable: $\mathcal{S}(\psi_F|_c) =$ *true*.

*The Elevator Case Study: Configurations.* Configuration $c = \langle ?, ?, 3 \rangle$ denotes elevator systems that serve three floors. However we have not decided about having a weight sensor or supporting a VIP floor yet. The projection of $\psi_F$ over this configuration would be:

$$\psi_F|_c = [(WS \rightarrow true) \ \wedge \ (VIP \rightarrow true) \\ \wedge \ ((3 \leq 3 \leq 5) \rightarrow true) \ \wedge (true \rightarrow (3 \leq 3 \leq 5))]$$

Note that the root feature is always substituted with *true*. This configuration is valid according to $\psi_F$ as there exists at least one substitution (e.g. $VIP$, $WS$ = *true*) that makes the entire formula *true*.

## C. Application Conditions

An application condition is a propositional logic formula over features and is used to describe a subset of products.

*Definition 5 (Application Condition):* An application condition $\varphi$ is a propositional logic formula over a set of features $\mathcal{F} = \mathcal{F}_\alpha \cup \mathcal{F}_\beta$, defined by the following grammar:

$$\varphi ::= \textit{true} \mid f_\alpha \mid e \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi \\ e ::= \ f_\beta == d \ \mid \ f_\beta > d \ \mid \ f_\beta < d$$

where $f_\alpha \in \mathcal{F}_\alpha$, $f_\beta \in \mathcal{F}_\beta$, and $d$ is a constant. $\square$

The set of all application conditions over $\mathcal{F}$ is denoted by $\Phi_\mathcal{F}$. The above definition of application condition is an extension of the definition proposed in [13] which was limited to boolean features.

We can use SMT-solvers to investigate if an application condition $\varphi$ is satisfiable according to the feature model $FM = (\mathcal{F}, \psi_F)$. For this purpose, we should check if there exists a substitution for the variables in $\varphi \wedge \psi_F$ that makes the whole formula *true*. Non-satisfiability of $\varphi$ with respect to $\psi_F$ means that $\varphi$ does not hold in any products belonging to the family described by the feature model.

For example, an application condition $\varphi = [WS \wedge VIP \wedge (F > 3)]$ denotes elevators with a weight sensor that supports VIP floor and serves more that three floors. This application condition is satisfiable according to the feature model.

## III. Elevator with Customized Scheduling

In this section, we informally describe how to use the priority-based scheduling (proposed by Nobakht et al [7]) in the automata-theoretic framework for schedulability analysis (proposed by Jaghoori et al. [8]). We first explain the general framework by modeling the elevator case study using timed automata, and then elaborate on how to specify the scheduling policy based on dynamic priorities. Here, we consider a specific product configuration, i.e., with five floors and neither VIP nor weight features.

Formal definitions are given in Section V as applicable to software families.

What is specific to an elevator is its scheduling strategy. When going up, it serves all the requests on its way and it changes direction only if there are no other requests further up; and similarly when going down. Implementing this in usual actor-based languages, which use a FIFO strategy to process the messages, requires a great deal of overhead, for example, in finding the next floor to serve, and also in preserving the deadlines of requests until they are really processed. Instead, we dynamically prioritize the requests such that they are served in the correct order.

## A. *Timed Automata*

In this section, we describe the syntax and semantics of Timed automata [11].

*Definition 6 (Timed Automata):* Suppose $\mathcal{B}(X)$ is the set of all clock constraints on the set of clocks $X$. A timed automaton over actions $\Sigma$ and clocks $X$ is a tuple $\langle L, l_0, \mathcal{T}, I\rangle$ where

- $L$ is a finite set of locations
- $l_0 \in L$ is the initial location
- $\mathcal{T} \subseteq L \times \mathcal{B}(X) \times \Sigma \times 2^X \times L$ is a set of edges
- $I : L \mapsto \mathcal{B}(X)$ is a function that assigns an invariant to each location □

An edge $(l, g, a, r, l')$ implies that action $a$ may change the location $l$ to $l'$ by resetting the clocks in $r$, if the clock constraints in $g$ and the invariant of $l'$ hold.

*Definition 7 (Timed Automata Semantics):* A timed automaton defines an infinite labeled transition system whose states are pairs $(l, u)$ where $l \in L$ and $u : X \to \mathbb{R}_+$ is a clock assignment. We denote by $u_0$ the assignment mapping every clock in $X$ to 0. The initial state is $(l_0, u_0)$. There are two types of transitions:

- action transitions $(l, u) \xrightarrow{a} (l', u')$ where $a \in \Sigma$, if there exists $l \xrightarrow{a,g,r} l'$ such that $u$ satisfies the guard $g$, $u'$ is obtained by resetting all clocks in $r$ and leaving the others unchanged and $u'$ satisfies the invariant of $l'$
- delay transitions $(l, u) \xrightarrow{d} (l, u')$ where $d \in \mathbb{R}_+$, if $u'$ is obtained by delaying every clock for $d$ time units and for each $0 \leq d' \leq d$, $u'$ satisfies the invariant of location $l$. □

## B. *Behavioral Interfaces*

Modeling an object starts with specifying its behavioral interface. A behavioral interface consists of the messages an object may receive and send, thus it provides an abstract overview of the object behavior in a single automaton. A behavioral interface abstracts from specific method implementations, the message buffer in the object and the scheduling strategy.

The behavioral interface of the elevator case study is shown in Figure 2 (left). An instance of this automaton is created for each floor. According to this behavioral interface, each floor may send a request to the elevator and wait to be served. These actions are modeled as `req(floor)(20)` and `serve(floor)` respectively where

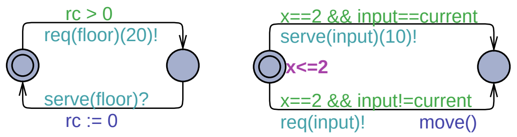


Figure 2. Elevator behavioral interface and its request method.

`floor` indicates which floor is sending the message and 20 denotes the deadline of the message. Note that the initial state is represented by double lined circle.

In our models of behavioral interfaces, input (resp. output) actions are distinguished with ! (resp. ?) marks. With this notation, a behavioral interface can be considered as the object's environment. Deadlines must be assigned to input actions in order to define the schedulability requirements. Formally, the concrete deadline value is part of the action [8], but for simplicity, we may use a global variable to pass the deadline values.

The variable `rc` is a clock variable. We use this clock to ensure that no request is sent for a floor exactly at the same time after it has just been served. Otherwise, the elevator will stay in the same floor. Repeating this situation causes other requests to be postponed arbitrarily long, which makes the elevator un-schedulable.

## C. *Object Definition*

One can define an object as a set of methods implementing a specific behavioral interface. In addition to the `req` method, the elevator object contains two state variables: the current floor is stored in `floor` and the boolean `dir` shows if the elevator currently moves upwards.

Methods are modeled as timed automata and can send messages while computations are abstracted into time delays. Receiving and buffering messages is handled by the scheduler (formally explained in Section V). Figure 2 (right) shows the automata model of the method which handles the `req` message. This method has a parameter `input`, which is set by the scheduler upon starting the method and indicates the number of the floor that sent the request. If the elevator is currently at the requested floor, it serves the floor by sending the `serve` message. Otherwise, the elevator moves by updating the state variables using this function:

```
void move() {
    dir = floor < input;
    floor = (dir)? floor+1 : floor-1;
}
```

In this function, after determining the direction towards the target, the elevator moves only one floor. Thus, it has the possibility to check if new requests have arrived in the meantime which could be served along the way. Additionally, it needs to reinsert the request back in its queue, because the requested floor has not yet been served. In general, when an object chooses to send a message to itself, we call it a *self call*. A self call with no explicit deadline inherits the (remaining) deadline of the task that has triggered it (called *delegation*). In this example, delegation is necessary because the original deadline should remain valid until the request is served.

The variable `x` is a clock variable. We use this clock to model the elapse of time when serving a floor or moving to another floor (which is 2 time units in this example).

### D. Scheduling Policy

This implementation of the `req` method necessitates a strict scheduling policy that favors the requests in the current direction of the elevator. For example, if the elevator is in the $2^{nd}$ floor and there are requests for the $1^{st}$ and $5^{th}$ floors, a first-come-first-served strategy can make the elevator get stuck. Because it will alternate between the processing of the existing two requests, and keeps moving up and down between the $2^{nd}$ and $3^{rd}$ floors.

To avoid the above scenario, we use a priority scheduler for the elevator and dynamically compute a two-level priority for each message. The first-level priority `p1` is zero for the requests in the current direction, thus giving them a higher priority. Other requests set `p1` to 2 (see below). Among the requests with the same `p1` value, those closer to the current floor get a higher second-level priority, i.e., a lower number for `p2`. We combine these two priority levels as `p1 * max(p2) + p2` where `max(p2)` shows the maximum value of `p2`. Therefore, we ensure that the effect of `p1` is higher than `p2`. Given five floors, `max(p2)` is equal to 4. In the function below, `target` shows the target floor associated to a request message. Notice that the first-level priority in the case of a request for the current floor must also be zero.

```
int set_priority(int target) {
    int sign = (target > floor)? 1 : -1;
    int p1 = (dir) ? (1 - sign) : (sign - 1);
    int p2 = sign * (target - floor);
    if (target == floor) p1 = 0;
    return p1 * 4 + p2;
}
```

The scheduler can be formulated as a function that returns the minimum priority value as calculated above.

## IV. Families of Timed Automata

Introducing the notion of variability in timed automata enables us to represent families of real-time objects. In this section, we describe the syntax and the semantics of *timed automata family* based on the notion of timed automata.

A timed automata family (TAF) is syntactically a timed automata in which each action and edge is given an application condition. An application condition of an action/edge indicates the set of products that include that action/edge. An action/edge that is included in all products is annotated with *true*.

*Definition 8 (Timed Automata Family):* A timed automata family over feature set $\mathcal{F}$, actions $\Sigma$, and clocks $X$ is a tuple $\langle L, l_0, \mathcal{T}, I, \Gamma\rangle$ where

- $L$ is a finite set of locations
- $l_0 \in L$ is the initial location
- $\mathcal{T} \subseteq L \times \mathcal{B}(X) \times \Phi_F \times \Sigma \times 2^X \times L$ is a set of edges
- $I : L \mapsto \mathcal{B}(X)$ is a function that assigns an invariant to each location
- $\Gamma : \Sigma \mapsto \Phi_F$ is a function that associates an application condition to each action $\square$

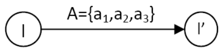


Figure 3. Multiple actions can be associated to an edge if they have mutually exclusive application conditions

As before, $\mathcal{B}(X)$ is the set of all clock constraints and $\Phi_F$ is the set of all application conditions. The function $\Gamma$ associates application conditions to actions. For example, in an elevator with four floors, the action $req(5)$ does not exist. Every edge in a TAF also has an application condition $\varphi \in \Phi_F$. The idea is that such an edge exists in a given configuration $c$ if $\Gamma(a)|_c \wedge \varphi|_c$ is satisfiable, where $a$ is the action on this edge. This is reflected in the semantics of timed automata families, defined below based on the semantics of timed automata.

*Definition 9 (TAF Semantics):* Given an initial constraint $\psi$ over the feature set $\mathcal{F}$, a timed automata family $T$ defines an infinite labeled transition system, denoted $[\![T]\!]_\psi$, whose states are written as $(l, u, p)$ where $l \in L$, $u : X \to \mathbb{R}_+$ is a clock assignment, and $p$ is a propositional logic formula. The initial state is $s_0 = (l_0, u_0, \psi)$, where $u_0$ maps all clocks to zero. There are two types of transitions:

- **action transition** $(l, u, p) \xrightarrow{a} (l', u', p')$ where $a \in \Sigma$ if there exists $l \xrightarrow{a,g,\varphi,r} l'$ such that $p' = p \wedge \Gamma(a) \wedge \varphi$ and $\mathcal{S}(p') = true$, $u$ satisfies the guard $g$, $u'$ is obtained from $u$ by resetting the clocks in $r$, and $u'$ satisfies the invariant of $l'$.
- **delay transition** $(l, u, p) \xrightarrow{d} (l, u', p)$ where $d \in \mathbb{R}_+$, if $u'$ is obtained by delaying every clock for $d$ time units and for each $0 \le d' \le d$, $u'$ satisfies the invariant of location $l$. $\square$

The idea is that the initial constraint $\psi$ represents the feature model. It can incorporate additional constraints in order to preselect some features, for example, as used in TAF projection defined in the next subsection. In this definition, the function $\mathcal{S}$ returns the satisfiability of its input.

*Syntactic Sugar:* To simplify modeling, we may associate an action set $A \subseteq \Sigma$ to an edge instead of one single action, as long as the application conditions of the actions in $A$ are mutually exclusive. This way, every product is still a timed automaton in which this edge has only one action. Figure 3 shows an edge between locations $l$ and $l'$ with an action set $A = \{a_1, a_2, a_3\}$. To avoid ambiguity, the application conditions corresponding to the actions in $A$ should be mutually exclusive: $\Gamma(a_1) \wedge \Gamma(a_2) = false$, $\Gamma(a_1) \wedge \Gamma(a_3) = false$, and $\Gamma(a_2) \wedge \Gamma(a_3) = false$.

### A. Projection and Refinement

Having a timed automata family $T$, we can project it on a (partial) configuration $c$ and obtain another timed automata family $T|_c$. Projecting on a decided configuration results in a specific product. Projection can be done *statically* by checking the satisfiability of the application conditions of actions and edges in $T$ based on configuration $c$. Then, we eliminate those actions and edges whose corresponding application condition is not satisfiable.

*Definition 10 (TAF Projection):* Projection of a timed automata family $T$ on a configuration $c$ leads to another timed automata family with the following action set and transition set:

- $\Sigma' = \{a \mid a \in \Sigma \ \wedge \ \mathcal{S}(\Gamma(a)|_c) = \mathit{true}\}$
- $\mathcal{T}' = \{t : (l, g, \varphi, a, r, l') \mid t \in \mathcal{T} \wedge \mathcal{S}(\varphi(t)|_c \wedge \Gamma(a)|_c) = \mathit{true}\}$ □

As mentioned above, projection is a syntactic process. To preserve the feature selection by $c$ at the semantic level, the semantics of the projection of $T$ on $c$ would be $[\![T|_c]\!]_{\psi|_c}$, where $\psi$ represents the feature model of $T$.

By contrast to projection, we define a refinement process at the semantic level. We will show that these two notions coincide. In other words, a refined TAF corresponds to a smaller set of configurations.

*Definition 11 (TAF Refinement):* Given two timed automata families $T_A$ and $T_B$, we say $T_A$ refines $T_B$, denoted $T_A \sqsubseteq T_B$, if and only if there is a relation $\mathcal{R}$ between the states of their underlying transition systems such that

- $\mathcal{R}(s_{A_0}, s_{B_0})$; and,
- if $\mathcal{R}(s_A, s_B)$ and $s_A \to s'_A$ then there exists $s_B \to s'_B$ and $\mathcal{R}(s'_A, s'_B)$; and,
- for every $\mathcal{R}((l, u, p), (l', u', p'))$ we have $p \implies p'$. □

In this definition, $s \to s'$ represents both action and delay transitions.

*Theorem 1:* Given an initial constraint $\psi$, the projection of a timed automata family $T$ over a configuration $c$ refines $T$, formally written as: $[\![T|_c]\!]_{\psi|_c} \sqsubseteq [\![T]\!]_\psi$ □

### B. More Semantical Properties

To be able to further argue about the semantical properties of timed automata families, we define a similar projection operator on the transition systems. Intuitively, given a configuration $c$, this operator removes those states that are not compatible with $c$.

*Definition 12 (Transition System Projection):* Consider a timed transition system $TS$ with states $St$ and transitions $T$. The projection of $TS$ over a configuration $c$ is another transition system with the set of states $St' = \{(l, u, p) \mid (l, u, p) \in St \wedge \mathcal{S}(p|_c) = \mathit{true}\}$, and the set of transitions $T' = \{(s, s') \mid (s, s') \in T \ \wedge \ s, s' \in St'\}$ □

Intuitively, the set of states are projected on the configuration $c$ and the corresponding transitions are kept. Given a timed automata family $T$, the following theorem shows the difference between applying projection on $T$ itself, and on its semantics.

*Theorem 2:* Given an initial constraint $\psi$, a timed automata family $T$ and a configuration $c$, we have: $[\![T|_c]\!]_{\psi|_c} \sqsubseteq [\![T]\!]_\psi|_c$ □

An instance of the theorems above for decided configurations is the following:

*Corollary 1:* Given an initial constraint $\psi_F$ representing the feature model, the projection of a timed automata family $T$ over a decided configuration $c$ results in a product $p$, such that: $[\![p]\!]_{\psi_F|_c} \sqsubseteq [\![T]\!]_{\psi_F}$ □

This corollary formalizes the common understanding that a product family $T$ semantically covers all products $p$. It also implies that a family $T$ can be analyzed by projecting on all decided configurations first, and then analyzing these projections. With such an early decision making, however, we cannot take advantage of the commonalities of the individual products. To reduce the analysis costs, we introduce in Section VI a late decision making scheme.

## V. Modeling Families of Real-Time Objects

To model a single object (disregarding the variabilities in a family), Jaghoori et al. [8] propose the following. First a synthetic abstract behavior of the object is given in one place, called its behavioral interface. Then we model a class as a set of methods that implement such an interface. Finally, an object is an instance of a class coupled with a given scheduler.

To define a family, one can introduce variability in each of the above mentioned levels. In this section, we formally define the behavior of an object family by allowing variabilities: inside behavioral interfaces and method definitions or in the choice of the scheduling policy.

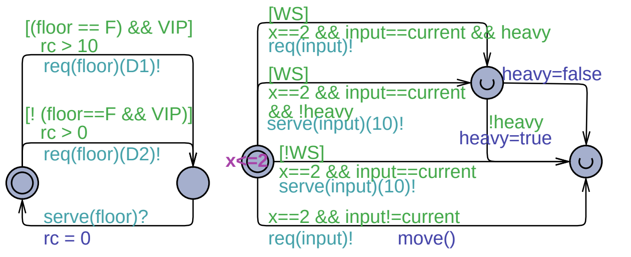
### A. Behavioral Interface Families

Variability in a behavioral interface enables an object to be used in different environments providing/requiring different features. A behavioral interface family describes at a high level, how an object family behaves. It consists of the messages an object family may receive and send in different configurations. To formally define a behavioral interface family, we assume a finite global set $\mathcal{M}$ for method names.

*Definition 13 (Behavioral Interface Family):* A behavioral interface family $B$ providing a set of method names $M_B \subseteq \mathcal{M}$ is a deterministic timed automata family $\langle L_B, B_0, \mathcal{T}_B, I_B, \Gamma_B\rangle$ over alphabet $\mathit{Act}^B$ such that $\mathit{Act}^B$ is partitioned into two sets of actions:

- outputs: $\mathit{Act}^B_O = \{m? \mid m \in \mathcal{M} \wedge m \notin M_B\}$
- inputs: $\mathit{Act}^B_I = \{m(d)! \mid m \in M_B \wedge d \in \mathbb{N}\}$ □

An input action $m(d)!$ corresponds to a message $m$ sent to the object, which must be processed before deadline $d$. We define a *task* to be a message plus a deadline. We consider also variability in the deadlines, as the time to finish a task may depend on the set of selected features. Therefore, we define a (partial) function $\mathcal{D}_{m_i} : \mathcal{C} \mapsto \mathbb{N}$, which returns the deadline of the $i^{th}$ usage of $m$ on an edge, in the given configuration $c$. Applying the syntactic sugar in Section IV, we can then use $m(\mathcal{D}_{m_i})$, representing an action set with all possible deadline values for $m$, on the edge of a behavioral interface family. As discussed in Section IV, this syntactic sugar is valid as long as the application conditions of the resulting concrete actions $m(d)!$ are disjoint. A formal justification is dropped in view of space.



$$\Gamma(\texttt{req(i)}) = \texttt{i} \leq \texttt{F} \quad \forall\, \texttt{i:[1..F]}$$
$$\Gamma(\texttt{serve(i)}) = \texttt{i} \leq \texttt{F} \quad \forall\, \texttt{i:[1..F]}$$

Figure 4. Behavioral interface and request method of the elevator family

*The Elevator Case Study: Behavioral Interface Family.* Figure 4 (left) shows the behavioral interface family of the elevator system. The floors feature affects the behavioral interface as some of the floors may not send requests as they are not served by the elevator in the current configuration. Therefore, every request or serve action from/to the $i^{th}$ floor has the application condition $i \leq F$, where $F$ is the number of supported floors.

The uppermost floor is the VIP floor when this feature is supported. The idea is that this floor has a higher priority, therefore, it gets a smaller deadline. Additionally, we assume that the requests from this floor are less frequent, i.e., at the fastest every 10 time units. In the behavioral interface family, requests from the VIP floor (when this feature is supported) are distinguished by an edge with application condition $\mathit{VIP} \wedge (F == \mathit{floor})$, where `floor` is a constant value denoting the floor that is sending the request and $F$ is the number of supported floors that implicitly indicates the VIP floor as well.

Functions $\mathcal{D}_1$ and $\mathcal{D}_2$ return the appropriate deadlines for request messages from VIP floor and normal floors respectively, according to the configuration. Presence of weight sensor leads to higher deadlines for all floors as the lift may stay longer in each floor. Moreover, supporting VIP floor results in higher deadlines for other floors.

## B. Class Family Definition

To support variability in a method's behavior, we use timed automata family to describe a method instead of timed automata. This way, variability in behavior is described by associating application conditions to the actions and the transitions of the corresponding timed automata family. A class family may implement a behavioral interface family $B$ by providing implementations for the methods in $M_B$ in the corresponding configurations.

*Definition 14 (Class Family):* A class family $R$ implementing the behavioral interface family $B$ is a set $\{(m_1, A_1), \ldots, (m_n, A_n)\}$ where:

- $M_R = \{m_1, \ldots, m_n\} \subseteq \mathcal{M}$ is a set of method names such that $M_B \subseteq M_R$
- for all $i$, $1 \leq i \leq n$, $A_i$ is a timed automata family representing the implementation of the method $m_i$, each having the alphabet $\Sigma_i = \{m! | m \in M_R\} \cup \{m(d)! \mid m \in \mathcal{M} \wedge d \in \mathbb{N}\}$ □

Every message must be associated with a deadline value. In the syntax of a method, we allow the self calls (i.e., $m \in M_R$) to be made with no explicit deadlines. Semantically, this means that the self call should inherit the deadline of its parent. This is called delegation and is explained in the next subsection.

*The Elevator Case Study: Method Automata.* Having defined a family of behavioral interfaces is necessarily reflected in the methods and/or the scheduling policy. Nonetheless, a class may provide a variety of method implementations or scheduling policies even if the behavioral interface is not variable. To exemplify the latter case, assume that the floor and VIP features does not exist in the feature model; in this case, the weight sensor model would only affect the class implementation while the behavioral interface can remain the same (as it depicted in Figure 2 (left)), assuming that the deadlines are large enough.

Figure 4 (right) illustrates the behavior of the request method considering the weight sensor feature. If the weight sensor feature is included (represented by `[WS]` application condition), the behavior of the elevator changes when it reaches a target floor. In this case, if the elevator is heavier than its capacity, it does not serve the floor and does not move. Instead, it puts the message back in the queue until it is not heavy anymore. To abstractly model the change of the weight, we use a `heavy` flag which is nondeterministically set to true or false after processing the request method; the flag is set to true only if it is not currently true. Therefore, the elevator does not get stuck in the same floor because of the weight.

## C. Modeling schedulers

A concurrent object maintains a queue of the messages awaiting to be processed. We formally define a queue as an ordered set of triples $m(d, c)$ representing a method $m$ with deadline $d$, and clock $c$ that is reset to zero upon insertion of $m$ in the queue. This message misses its deadline if $c > d$. We assume that messages are added at the back of the queue with a fresh clock, i.e., not assigned to any message in the queue. In the case of delegation, the deadline and clock assigned to the currently running task are reused. Thus, we can model inheriting the deadline.

As discussed before, we do not assume any predefined order on executing the messages in the queue. This has to be specified in terms of a scheduler function.

*Definition 15 (Scheduler Function (adapted from [8])):* A scheduler function $\mathit{sched} : Q \mapsto (Q, \mathcal{M})$ selects a method from the queue and returns the queue after removing that message. □

For example, a scheduler that returns the first element in the queue produces the first-come, first-served strategy. A more sophisticated scheduler can look into the object state as in Section III. In the elevator family, the scheduling strategy must be adapted in the case of selecting the VIP feature. This can simply be achieved by adding another priority level to favor the VIP floor.

## D. An Object Family

An object family, formally denoted $(R, \Xi)$, is defined as an instance of the class family $R$ coupled with a

| | |
|---|---|
| $(b,Q) \xrightarrow[c_\nu=0]{a?} (b', Q :: a(d,c_\nu))$ | if $b \xrightarrow{a(d)!}_B b'$, $c_\nu$ is a fresh clock |
| $(s) \to (s')$ | if $s \to_M s'$ |
| $(b,s) \xrightarrow{a(d)!} (b',s')$ | if $s \xrightarrow{a(d)!}_M s'$, $b \xrightarrow{a?}_B b'$ |
| $(s,Q) \xrightarrow{a!} (s', Q :: a(d_r,c_r))$ | if $s \xrightarrow{a!}_M s'$, $a \in M_R$ |
| $(s,[\,]) \to (\epsilon,[\,])$ | if $s \not\to_M$ |
| $(\epsilon,Q) \xrightarrow{\varphi(\xi)} (m_0,Q')$ | if $Q \neq [\,]$, $(m,Q')=\xi(Q)$, $\forall \xi \in \Xi$ |

Assumptions:

- The clock and deadline of the currently running task are $c_r$ and $d_r$, respectively, which are updated upon scheduling a new method.
- The initial location of method $m$ is denoted $m_0$.
- $\varphi$ returns the application condition of a scheduler $\xi$.
- $M_R$ is the set of methods defined in class $R$, and $\Xi$ is the set of schedulers in the object.
- Location $s$ implies the currently running method, also identified by subscript $M$ for transitions.

Figure 5. Simplified presentation of the timed automata family encoding the behavior of an object family w.r.t. its behavioral interface $B$.

set of schedulers $\Xi$. Each scheduler has an application condition, such that at each configuration one scheduler can be applied. For example, an elevator family has a normal scheduler plus another one supporting a VIP floor. To define the behavior of an object, we need to know how its methods are going to be called. Therefore, we consider its behavioral interface as an environment. It means that the behavioral interface sends messages to the object, and the object may only send out messages if they are accepted by the behavioral interface.

The behavior of an object family is then defined as a timed automata family. The locations of this TAF are written $(b, s, Q)$ corresponding to the current location of the behavioral interface (shown as $b$), the current location of the currently running method (shown as $s$) and the contents of the queue (shown as $Q$). The initial location is $(b_0, \epsilon, [\,])$ which means that the behavioral interface is in its initial location $b_0$, no method is currently running and the queue is empty. The transition relation of this TAF is to be computed as shown in Figure 5. In this figure, only the relevant components of the state are shown. Furthermore, every guard, clock reset and application condition present on the transitions on the right (i.e., in the behavioral interface family or the method definition) will also be present in the transition on the left. These are dropped in the figure for simplicity in presentation.

We briefly explain each rule in Figure 5 in one sentence: The behavioral interface can send a message $a$ with deadline $d$ which is added to the back to the queue. The currently running method can do one of the following: 1) it can take an invisible action; 2) it may send a message to the environment; 3) it can make a delegation, inheriting the deadline. The $\epsilon$ symbol shows that no method is currently running. The last rule (i.e., starting a new method) is applicable for all schedulers in $\Xi$ resulting in a transition with a corresponding application condition. Note also that the scheduler function needs to look at the queue contents and therefore the generated state depends on the contents of the queue.

*The Elevator Case Study: The Scheduler.* The strategy of the scheduler changes when the elevator supports the VIP floor feature. The two scheduler functions are given the application conditions $\varphi_1 = \neg VIP$ and $\varphi_2 = VIP$, which are mutually exclusive.

## VI. Schedulability Analysis of Real-time Software Families

An object family is said to be *schedulable* if in any of its products, the deadlines of the tasks in the queue (including the currently executing task) never expire, i.e., there is no reachable state where the clock of a task in the queue is greater than its deadline. As described in the previous section, we follow the approach in [8] and use the behavioral interface as the environment of the object.

Checking schedulability with an unrestricted queue length might require us to check an infinite system. For a given object (i.e., a specific product) and its behavioral interface, it is shown that there is an upper bound of $\lceil d_{max}/b_{min} \rceil$ on the queue length of schedulable systems [8], where $d_{max}$ is the biggest largest deadline in methods and the behavioral interface, and all methods take more than or equal to $b_{min}$ to finish. This result can be easily generalized to an object family. While fixing the queue bound for each product individually might save us some memory during analysis, but considering one queue bound for the whole family is easier to implement.

*Corollary 2 (Schedulability):* An object family is schedulable with respect to its behavioral interface family if and only if none of its products exceeds the queue limit of $\lceil d_{max}/b_{min} \rceil$ and no message misses its deadline.

Schedulability analysis of an object family can be done by constructing the timed automata family encoding its behavior up to one more than the above queue length. Then we can add the following two rules to detect queue overflow and deadline miss by moving to an `Err` state:

$$\begin{array}{ll} (Q) \to Err & \text{if } |Q| > \lceil d_{max}/b_{min} \rceil \\ ([\ldots, a(d,c), \ldots]) \xrightarrow[c>d]{} Err & \end{array}$$

Jaghoori et al. [8] have shown how to analyze schedulability of a real-time concurrent object (e.g., obtained as a product from an object family) in a tool like Uppaal. The idea is that all methods, behavioral interface and the scheduler (including the queue) are modeled as a timed automaton. The network of all of these automata will actually produce the behavior automaton of the object. To check the schedulability of a family, a naive approach is to compute all products of the family, and then analyze each of them separately. Alternatively, we explain below how to analyze a TAF in general using the general tools for the analysis of timed automata. With this technique, one

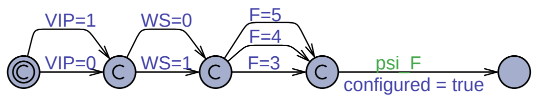


Figure 6. The configurator automaton of the elevator case study

can then apply the modeling technique of [8] to analyze the schedulability of object families directly.

We propose a static analysis technique that can syntactically transform a TAF to a standard timed automaton. The first step in translating timed automata families into standard timed automata is to model features as boolean or integer variables. This enables us to encode the application conditions as guards, which should be conjoined with the original guards of the edges. Recall that the application conditions of the edge as well as the actions on that edge must be considered together. Note that one can preselect some features (i.e., project on a partial configuration) in order to narrow down the state-space if desired. To preconfigure some features, one can turn them into constants.

## A. Early Decision Making

To make a decision about inclusion, exclusion, or the value of features, we can use a *configurator* automaton. The configurator non-deterministically assigns a value to each feature in the beginning of execution of the model. Figure 6 shows the configurator automaton of the elevator case study. The letter `C` in the locations in this figure implies that their outgoing edges are taken without time passing. The configurator automaton can be generated automatically given a feature set $\mathcal{F} = \{f_1, ..., f_n\}$ and a formula $\psi_F$ corresponding to the feature model in a straightforward way. Basically, the automaton has $n + 1$, and at each step the value of one feature is decided non-deterministically. The final step sets the boolean variable *configured*. To avoid invalid configurations, we add the guard $\psi_F$ to this edge. In Figure 7, `psi_F` is equal to the propositional formula presented in Section II. As a result, any invalid assignment to features will deadlock; this is not a problem because the model checker will then only consider the non-deadlocking configurations. Furthermore, the TAF under analysis must wait for the configurator to set the special boolean variable `configured` before it starts.

The above approach is easy to implement, but the number of generated states using this approach is not efficient. Figure 7 shows a TAF on the left with two features `f1` and `f2`. In the middle, we can see the result of applying early decision making. It is clear that considering all possible decided configurations `c1` to `c4` results in unnecessarily duplicating the location `l2`.

## B. Late Decision Making

In order to reuse the states common to different products, we propose to postpone the decision on the value of each feature until it is actually used in an application condition. This can be done by statically processing the

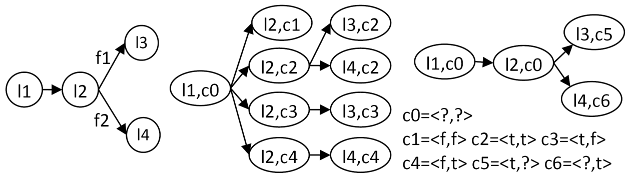


Figure 7. Early vs. late decision making for features

Table I
NUMBER OF STATES IN ANALYZING THE ELEVATOR FAMILY

| Early decision making | Late decision making |
|---|---|
| 4,289,659 | 2,211,534 |

automata and assigning a value to each feature right before they are first used.

First, we define a *select* expression, which is available in UPPAAL as a syntactic sugar. The select expression $v$ `: int[min, max]` on an edge implies replication of that edge for each value between `min` and `max` for $v$. This way we can easily get a new value for each feature. To avoid assigning a new value to a feature that is already decided, we define the function $\gamma$ that receives the current value $f$ and the new value $v$ of a feature and returns $v$ if $f = ?$, and otherwise it returns the old value $f$.

One idea for late decision making is to decide at each location $l$ about the features $F_l$, which affect the applicability of edges from $l$. To this end, every location $l$ is split into $l_i$ and $l_o$ such that the incoming edges go to $l_i$ and the outgoing edges will be from $l_o$. We add one edge from $l_i$ to $l_o$ with a select expression for each feature in $F_l$ supplied with the function $\gamma$ (defined above). For example, in location `l2` in Figure 7, we should decide about the features `f1` and `f2`. The problem here is that the locations `l3` and `l4` will be unnecessarily duplicated. Consider for example `l3` which only depends on the value of `f1`, and is therefore duplicated for both values of `f2`.

To solve this issue, we decide the value of features exactly on the edge in which they are used. Inevitably, we add a select expression $v_k$ `: int[min, max]` for each integer feature $f_k$ (and similarly for boolean features). The new value of the feature can be calculated as $f_k = \gamma(f_k, v_k)$. However, since the guards are evaluated before updating the variables, we need to substitute this value explicitly in the guards as well, or more precisely, where the feature $f_k$ is used in application condition $\varphi_i$:

$$\varphi_i[f_k/\gamma(f_k, v_k)]_{\forall f_k}$$

## C. Empirical Result

We have modeled the elevator family case study in UPPAAL by extending both early and late decision making approaches to the schedulability analysis of [8]. UPPAAL models are available at http://www.cwi.nl/~jaghouri/TAF. The corresponding object family was schedulable with respect to its behavioral interface family as we modeled in this paper. As shown in Table I, use of the late decision making approach amounted to almost halving the number of states explored during model checking.

## VII. Conclusion

In this paper we proposed a formal approach to analyze schedulability of families of real-time objects. Feature models can be defined as a set of features and a propositional logic formula that describes the constraints among them. We used feature models to describe commonalities of and differences between products in a family. In this paper, we introduced timed automata families, defined over a feature set, in which each transition and action is annotated with an application condition. An application condition is a boolean formula in terms of features and describes the products that include the transition or action.

Based on our notion of timed automata families, we introduced the notion of variability in the behavioral interfaces, scheduler, and method implementations of an object. To reuse the common state space during schedulability analysis, we proposed a late decision making scheme for features. Although it is less straightforward than an early decision making scheme, it leads to a smaller state space. We depicted the effectiveness of our approach by modeling and analyzing an elevator system family. The late decision making scheme results in almost halving the state space.

In future work, we first plan to study compatibility of behavioral interfaces of families of individual real-time actors, extending the work described in [9]. Compatibility allows to infer schedulability of a complete system of actors compositionally from the schedulability of its individual actors. Further, we plan to support modeling variability using deltas (as in delta-oriented programming) in our methodology. To this end, each delta gives rise to a new automaton for the methods, scheduler, or behavioral interface.

## References

[1] K. Pohl, G. Böckle, and F. J. v. d. Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

[2] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," Carnegie-Mellon University Software Engineering Institute, Tech. Rep., November 1990.

[3] C. Hewitt, "Procedural embedding of knowledge in planner," in *Proc. the 2nd International Joint Conference on Artificial Intelligence*, 1971, pp. 167–184.

[4] J. Armstrong, "Erlang," *Communications of ACM*, vol. 53, no. 9, pp. 68–75, 2010.

[5] P. Haller and M. Odersky, "Scala actors: Unifying thread-based and event-based programming," *Theoretical Computer Science*, vol. 410, no. 2-3, pp. 202–220, 2009.

[6] F. Berman and R. Wolski, "Scheduling from the perspective of the application," in *Proc. High Performance Distributed Computing (HPDC'96)*. IEEE CS, 1996, pp. 100–111.

[7] B. Nobakht, F. S. de Boer, M. M. Jaghoori, and R. Schlatte, "Programming and deployment of active objects with application-level scheduling," in *Proc. ACM Symposium on Applied Computing (SAC'12)*. ACM, 2012, to appear.

[8] M. M. Jaghoori, F. S. de Boer, T. Chothia, and M. Sirjani, "Schedulability of asynchronous real-time concurrent objects," *J. Logic and Alg. Prog.*, vol. 78, no. 5, pp. 402 – 416, 2009.

[9] M. M. Jaghoori, D. Longuet, F. S. de Boer, and T. Chothia, "Schedulability and compatibility of real time asynchronous objects," in *Proc. Real Time Systems Symposium*. IEEE CS, 2008, pp. 70–79.

[10] O. Kupferman, M. Y. Vardi, and P. Wolper, "Module checking," *Information and Computation*, vol. 164, no. 2, pp. 322–344, 2001.

[11] R. Alur and D. L. Dill, "A theory of timed automata," *Theor. Comput. Sci.*, vol. 126, pp. 183–235, April 1994.

[12] G. Behrmann, A. David, K. G. Larsen, J. Håkansson, P. Pettersson, W. Yi, and M. Hendriks, "Uppaal 4.0," in *QEST*. IEEE Computer Society, 2006, pp. 125–126.

[13] R. Muschevici, D. Clarke, and J. Proenca, "Feature Petri nets," in *Second Proc. Software product lines*, 2010, pp. 99–106.

[14] 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 *Proc. Int'l Conf. on Software Eng. (ICSE'10)*. ACM, 2010, pp. 335–344.

[15] K. G. Larsen, U. Nyman, and A. Wasowski, "Modal I/O automata for interface and product line theories," in *Proc. European Symposium on Programming*, ser. ESOP'07. Springer-Verlag, 2007, pp. 64–79.

[16] A. Gruler, M. Leucker, and K. Scheidemann, "Modeling and model checking software product lines," in *Proc. Formal Methods for Open Object-Based Distributed Systems*, ser. FMOODS '08. Springer-Verlag, 2008, pp. 113–131.

[17] D. Clarke, M. Helvensteijn, and I. Schaefer, "Abstract delta modeling," in *Proc. Generative Prog. And Component Eng. (GPCE'10)*. ACM, 2010, pp. 13–22.

[18] I. Shin and I. Lee, "Compositional real-time scheduling framework with periodic model," *ACM Trans. Embed. Comput. Syst.*, vol. 7, pp. 30:1–30:39, May 2008.

[19] J. J. G. Garcia, J. C. P. Gutierrez, and M. G. Harbour, "Schedulability analysis of distributed hard real-time systems with multiple-event synchronization," in *Proc. Euromicro Conf. on Real-Time Sys.* IEEE, 2000, pp. 15–24.

[20] E. Fersman, P. Krcal, P. Pettersson, and W. Yi, "Task automata: Schedulability, decidability and undecidability," *Inf. Comput.*, vol. 205, no. 8, pp. 1149–1172, 2007.

[21] R. Alur and G. Weiss, "RTComposer: a framework for real-time components with scheduling interfaces," in *Proc. 8th Intl. conf. on Embedded software (EMSOFT'08)*, L. de Alfaro and J. Palsberg, Eds. ACM, 2008, pp. 159–168.

[22] D. Benavides, S. Segura, and A. Ruiz-Cortés, "Automated analysis of feature models 20 years later: A literature review," *Inf. Syst.*, vol. 35, pp. 615–636, September 2010.

[23] D. S. Batory, "Feature models, grammars, and propositional formulas," in *Proc. Software product lines*, ser. SPLC'05, 2005, pp. 7–20.

[24] V. Ganesh and S. University, *Decision procedures for bit-vectors, arrays and integers*. Stanford University, 2007.