From Transition Systems to Variability Models & From Lifted Model Checking Back to **UPPAAL**

Aleksandar S. Dimovski and Andrzej Wąsowski⋆

Computer Science, IT University of Copenhagen, Denmark

Abstract. Variational systems (system families) allow effective building of many custom system variants for various configurations. Lifted (family-based) verification is capable of verifying all variants of the family simultaneously, in a single run, by exploiting the similarities between the variants. These algorithms scale much better than the simple enumerative "brute-force" way. Still, the design of family-based verification algorithms greatly depends on the existence of compact variability models (state representations). Moreover, developing the corresponding family-based tools for each particular analysis is often tedious and labor intensive.

In this work, we make two contributions. First, we survey the history of development of variability models of computation that compactly represent behavior of variational systems. Second, we introduce variability abstractions that simplify variability away to achieve efficient lifted (family-based) model checking for real-time variability models. This reduces the cost of maintaining specialized family-based real-time model checkers. Real-time variability models can be model checked using the standard UPPAAL. We have implemented abstractions as syntactic source-to-source transformations on UPPAAL input files, and we illustrate the practicality of this method on a real-time case study.

1 Introduction

The strong trend for customization in modern economy leads to construction of many highly-configurable systems. Efficient methods to achieve customization, such as *Software Product Line Engineering* (SPLE), use features, or a similar concept, to mark the variable functionality. Family members, called *variants* of a *variational system*, are derived by switching features on and off. The reuse of code common to many variants is maximized. The SPLE method is very popular in the embedded systems domain. Moreover, many of the variational systems, such as device drivers, controllers, and communication protocols are time-critical. A rigorous verification and validation of their timing properties is important. *Model checking* [3] is an automatic technique often used to check for temporal properties of their designs.

Variability and SPLE are major enablers, but also a source of complexity. Analyzing variational systems is challenging. From only a few configuration

---

options, exponentially many variants can be derived. Thus, a simple brute-force application of single-system model checking to each variant is infeasible for realistic systems. In essence, the same behaviour is checked multiple times, whenever it is shared by many variants. To address this problem, we need compact structures exploiting the similarity within the family, on which specialized lifted (family-based) verification algorithms can operate. The quest for obtaining such compact models of computation, underpins a great deal of SPLE research. Many of the efforts are inspired by seminal works of Kim Larsen in concurrency theory, originally conceived with an entirely different goal (abstraction in system modeling and verification). We now survey the history of these efforts.

One of the earliest related models is the *Modal Transition System* (MTS) introduced by Kim Larsen and Bent Thomsen in 1988 [30]. It inspired Larsen, Nyman, and Wąsowski, who proposed to use MTSs as a framework for describing behavioral variational systems 20 years later [27]. In the first part of this work, we survey the history of development of various variability models, largely inspired by the seminal work of Kim and Bent cited above. Ultimately, we arrive at the popular *Featured Transition Systems* (FTSs) introduced by Classen et al. [11,10] and widely accepted as the model essentially sufficient for most purposes of family-based model checking of variational systems.

Then we turn our attention to the corresponding models with a real-time flavor, an area where Kim Larsen was particularly prolific throughout his research career. Here, a similar story of inspiration leading from his early works on *Timed Automata* and UPPAAL [29] to the ultimate *Featured Timed Automata* (FTAs) [13] can be traced—achieving model-checking capability for a wide class of real-time variational systems. Both for FTSs and FTAs specifically designed family-based model checking algorithms exist, which check common execution behaviour only once across variants that are able to produce it. The algorithms are implemented in the ProVeLines family-based model checker [12].

Unfortunately, maintaining specialized family-based model-checkers is expensive, and these tools do not benefit from continuous improvements introduced by research in the classic (non family-based) model checking. Moreover, their performance *still* heavily depends on the size and complexity of the configuration space of the analyzed system. In the second part of this work, we introduce a range of variability abstractions for real-time variational systems. The abstractions are applied at the variability level and aim to reduce the exponential blow-up of the number of variants (configurations) to be more tractable. These new variability abstractions are applied to a FTA, producing an "abstract FTA" which is smaller than the input one, while having at least the same universal Timed CTL properties. We can use the variability abstractions to obtain an abstract FTA (with a low number of variants), which can then be model checked in the brute-force fashion using the (single-system) UPPAAL model-checker. In the extreme case, all variability can be abstracted away, and the classic UPPAAL can be used to show universal properties for the entire system family. We illustrate this method on a simple real-time example, and show that it is still considerably faster than the brute-force enumeration.
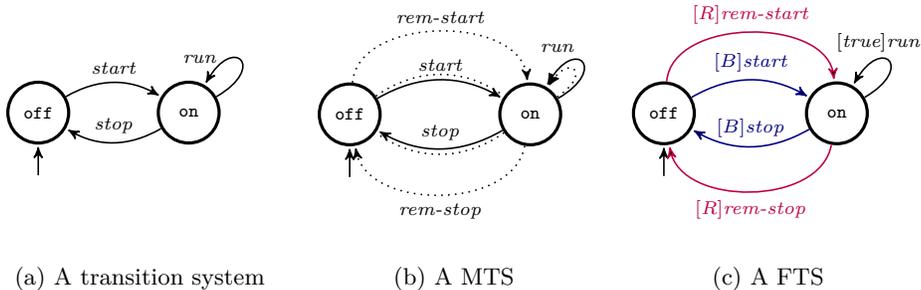
(a) A transition system   (b) A MTS   (c) A FTS

Fig. 1: From single-system models to variability models.

2   Superimposition-based Behavioral Variability Models

We now survey the historical development of several modeling formalisms that have either directly, or indirectly contributed to development of behavioral variability models. We begin with standard (discrete-time) models, and then discuss the parallel line of the related real-time models. The presentation uses a simple example of a mine pump system adapted from models by Cordy et al. [13].

2.1   From Transition Systems to FTSs

Transition systems [32,3] have for long been used to model the behavior of systems. A *transition system* $(S, Act, trans, I, AP, L)$ comprises a set of states S; a set of initial states $I \subseteq S$, a transition relation *trans* relating source and target states from S with action labels from Act; and a labelling function L determining which atomic propositions from a set AP hold at which state. An execution (behavior) of a transition system is a sequence of transitions starting from an initial state. We take the semantics of a transition system to be the set of all its executions.

Figure 1a shows a transition system modeling behavior of a basic mine pump, using the familiar concrete syntax. States are shown as circle nodes. Initial states are pointed to by dangling arrows (a single one in this example, labelled by off). The transition relation is represented by arrows between states: each triple in the relation consists of the source state of the arrow, target state of the arrow and the action label placed on the arc. Finally, the L function is shown by listing the names of propositions that hold in each state. In our example, off holds in the left state, and on holds in the right state. We will use similar notation in further examples, only explaining significant differences from now on. The atomic proposition indicates the pump's current status: on or off. Initially, the pump is in the off state. Transitions *start* and *stop* model switching between states, while the transition *run* indicates that the pump is still working in the state on.

In his PhD dissertation [26], Kim Larsen developed the notion of contextual equivalence between CCS processes known as *relativized bisimulation*. Roughly speaking, two processes S and T (represented by their transition systems) are equivalent in the context process E, written $S \sim_E T$, iff the context process

cannot observe any difference in their behavior. Twenty years later, this notion inspired one of the authors to create an early behavioral variability model, the *colour-blind transition systems* (CBTSs) [28]. In this model, the behavior of many variants in a family is captured in a single transition system, as the union of all behaviors, a kind of *superimposition of all variants* [14]. Variant selection is done by modeling contexts representing different users. Given a family model F and a variant context model V one can obtain a variant product model P by bisimulation minimization against the criterion $P \sim_V F$. CBTSs allowed natural modeling of variability in the input alphabet of a system. For instance, one variant model allows the users less interaction modes with the machine than another, blocking availability of a given feature. It was however cumbersome to model different reactions of variants to the same environment interaction.

Modal Transition Systems (MTSs), also known as *modal specifications*, are a generalization of transition systems that allows describing not just a sum of all behavior of a system but also an over- and under-approximation of the behavior. MTSs were introduced about thirty years ago by Kim Larsen and Bent Thomsen [30]. A MTS is a transition system equipped with two transition relations: *must* and *may*. The former (must) is used to specify the required behavior of a system, while the latter (may) is used to specify the allowed behavior of a system. Now an implementation of a MTS is a standard transition system that realizes all the required (must) behavior, and adds some (possibly none) of the allowed (may) behavior. We take the semantics of a MTS to be the set of all the transition systems that implement the MTS. In this sense, a MTS can be seen as a superimposition of many transition systems, each representing a single system variant. This inspired the idea of using modalities to represent variability in behavior [27].

Figure 1b shows an example of a MTS that models a (minuscule) family of pumps. Must transitions are denoted by solid lines, may transitions by dotted lines. Each must transition is also a may transition. The allowed part of the behavior includes the *rem-start* and *rem-stop* transitions that can be used for remotely changing the state of the pump. The regular *stop*/*start* transitions are modeling the switch placed physically on the device. The transition system of Fig. 1a, discussed above, is one of the possible variants implementing this MTS.

In fact, the example MTS describes infinitely many variants, due to co-inductive semantics, allowing different implementation choices at each visit to a specification state. For instance *rem-start* might become available only from the second power cycle of the pump. This co-inductive variation was an advantage of MTSs when used for abstracting behaviors, but less so in variability modeling, where one would expect a single variant to behave consistently whenever it visits the same specification state (*rem-start* should be always available in off, not only sometimes). Furthermore, there is not easy way in MTSs to represent dependency between variations—for instance, it is not easy to say that if *rem-start* is available in state off then *rem-stop* should be available in state on. These problems have ultimately led to development of feature transition systems and similar models.

4

Featured transition systems (FTSs) [11,10] are a compact representation of the behavior of all instances of a variational system, similar to MTSs but relying on a syntactic notion of implementation (subgraph projection) and allowing to constrain which transitions must, or must not, co-occur in implementation variants. To formally define FTSs, assume a finite set $\mathbb{F} = \{A_1, \ldots, A_n\}$ of Boolean variables representing features. A specific set of features $k \subseteq \mathbb{F}$, known as *configuration*, specifies a variant of a variational system. The *set of all valid configurations* is a subset $\mathbb{K} \subseteq 2^{\mathbb{F}}$ (equivalently represented using a Boolean formula). Each configuration $k \in \mathbb{K}$ can be represented by a term formula: $k(A_1) \wedge \ldots \wedge k(A_n)$, where $k(A_i) = A_i$ if $A_i \in k$, and $k(A_i) = \neg A_i$ if $A_i \notin k$ for $1 \leq i \leq n$. FTSs are an extension of transition systems, where transitions are guarded (labeled) with feature expressions, known as *presence conditions*. Presence conditions are propositional formulae over $\mathbb{F}$: $\psi ::= \mathit{true} \mid A \in \mathbb{F} \mid \neg\psi \mid \psi_1 \wedge \psi_2$. We use *FeatExp*($\mathbb{F}$) to denote the set of all feature expressions. The presence condition ψ labelling a transition indicates for which variants the corresponding transition is enabled. We write $[\![\psi]\!]$ to denote the set of variants $k \in \mathbb{K}$ that satisfy the presence condition ψ, i.e. $k \in [\![\psi]\!]$ iff $k \models \psi$.

Definition 1. *A featured transition system is a tuple $\mathcal{F} = (S, \mathit{Act}, \mathit{trans}, I, \mathit{AP}, L, \mathbb{F}, \mathbb{K}, \delta)$, where $(S, \mathit{Act}, \mathit{trans}, I, \mathit{AP}, L)$ is a transition system; $\mathbb{F}$ is the set of available features, $\mathbb{K}$ is a set of valid configurations, and $\delta : \mathit{trans} \to \mathit{FeatExp}(\mathbb{F})$ is a total function labelling transitions with presence conditions.*

The *projection* of an FTS $\mathcal{F}$ to a variant $k \in \mathbb{K}$, denoted $\pi_k(\mathcal{F})$, is the transition system $(S, \mathit{Act}, \mathit{trans}', I, \mathit{AP}, L)$, where $\mathit{trans}' = \{t \in \mathit{trans} \mid k \models \delta(t)\}$. The *semantics* of a FTS $\mathcal{F}$, denoted $[\![\mathcal{F}]\!]_{FTS}$, is the union of behaviors of the projections on all variants $k \in \mathbb{K}$, i.e. $[\![\mathcal{F}]\!]_{FTS} = \cup_{k \in \mathbb{K}}[\![\pi_k(\mathcal{F})]\!]_{TS}$, where $[\![\mathcal{T}]\!]_{TS}$ denotes the semantics of the transition system $\mathcal{T}$.

Figure 1c presents a FTS describing the behavior of a variational pump. It contains two features: Button (denoted by B) for turning on/off the pump manually using a button; and Remote (denoted by R) for turning on/off the pump using a remote control. The presence condition of a transition is shown next to its action label, placed in square brackets. For ease of reading the transitions enabled by the same feature are colored in the same way. For example, $[B]\mathit{start}$ means that the transition *start* is enabled only for variants satisfying B. Figure 1a shows the basic variant of the pump that can be operated only manually using a button. This variant is selected by configuration $\{B\}$ (or $B \wedge \neg R$). It can be obtained by projecting the FTS of Fig. 1c onto the configuration $\{B\}$. The set of all valid configurations of the variational pump can be obtained by combining the available features $\mathbb{F} = \{B, R\}$. The pump has four variants: $\mathbb{K} = \{\{B, R\}, \{B\}, \{R\}, \emptyset\}$, or written as the formula: $\mathbb{K} = (B \wedge R) \vee (B \wedge \neg R) \vee (\neg B \wedge R) \vee (\neg B \wedge \neg R)$.

Summary. Figure 2 summarizes the history of development of variability models. The left side is concerned with models discussed above. All points in the figure representing variability modeling contributions are typeset with a bold font. The papers introducing foundational models of computation that influenced the later variability models, are typeset with a regular font.

FTS, Classen et al, 2010
PMTS, Benes et al, 2011
CBTS, Larsen et al, 2007

FTA, Cordy et al, 2012

MTS, Larsen et al., 1988

TMS, Cernas et al, 1993
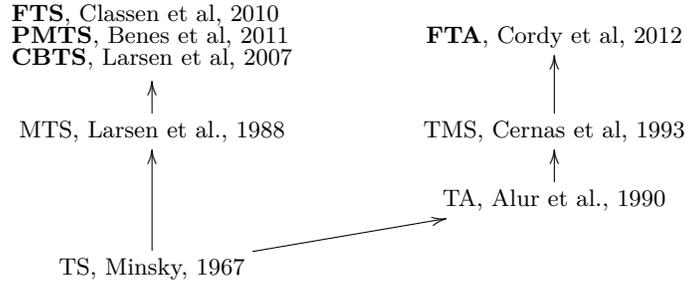
TA, Alur et al., 1990

TS, Minsky, 1967

Fig. 2: The history of development of variability models (left part) and real-time variability models (right part). Bold labels indicate variability models, while the regular font labels denote the basic models of computation that laid the foundation and inspired the variability models.

MTSs generalize transition systems with mandatory (must) transitions and optional (may) transitions, with the main applications originally being under-specification and abstraction. Their semantics superimposes multiple variants. It took however almost 20 years until the first works using MTSs for variability modeling appeared around 2006. Although, MTSs are suitable for representing optional behavior using may transitions, there is no explicit notion of variability in MTSs and so they cannot associate behaviors with the exact set of variants able to execute them. To overcome this limitation, FTSs rely on presence conditions guarding transitions that determine in which variants the transitions appear. Therefore, the presence of a transition may depend on the transitions taken before as well. FTSs are closely related to parametric MTSs (PMTSs) introduced by Benes et al. [5]. PMTSs extend considerably the expressiveness of MTSs, thus overcoming many of their limitations. A PMTS is equipped with a finite set of parameters (which are Boolean variables) that have fixed values for any implementation. Fixing a priori the parameters makes the instantiation of the (may) transitions permanent (uniform) in the whole implementation. However, no model checking tool that works on PMTS have been implemented so far, which was the main limitation for their wider application. So far they were mostly studied from theoretical points of view.

2.2 From Timed Automata to FTA

Alur and Dill have introduced *timed automata* (TA) [2] as a modelling formalism for time-critical systems. Timed automata are an application of transition systems (more precisely, program graphs [3]) in which real-valued clock variables (or clocks for short) are made part of the state and used to measure the elapse of time. The real-time assumptions on system behavior are specified using *clock constraints*, which are conditions that depend on the values of clocks.
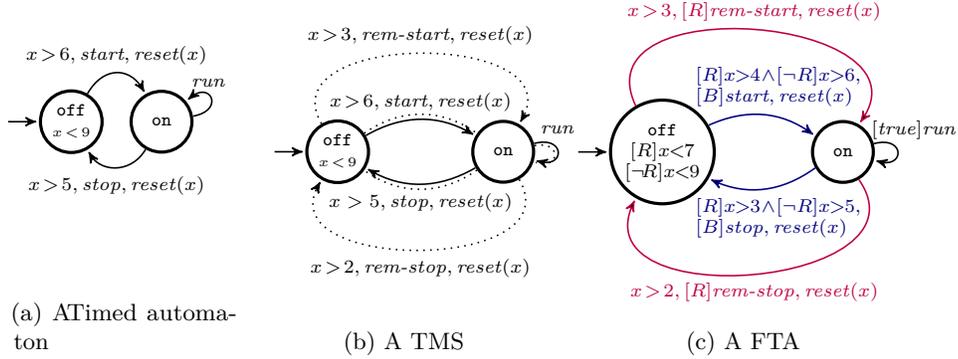
(a) ATimed automaton

(b) A TMS

(c) A FTA

Fig. 3: From timed single-system to timed variability models.

A *timed automaton* $(Loc, Act, C, trans, I, Inv, AP, L)$ consist of a set of locations $l \in Loc$ each equipped with an invariant, $Inv(l)$, which is a constraint over clocks in the set C. The constraint $Inv(l)$ limits the amount of time that may be spent in the location l. A set $I \subseteq Loc$ defines the locations active in the initial state of any execution. A transition relation *trans* comprises guarded transitions between locations. Guards are (again) clock constraints that specify when the transition may be taken. Each transition also has an action label $\lambda \in Act$; and a subset of clocks C which are reset to zero upon the firing of the transition. A labelling function L specifies which atomic propositions from AP hold at what locations. As any program graph, a timed automaton can be unfolded into an (infinite-state) transition system [3]. The semantics of a timed automaton is determined by the semantics of the underlying transition system obtained from unfolding, where only time-divergent executions are considered (infinite executions in which the time progress is unbounded).

Figure 3a shows an example of a timed automaton that models the basic behavior of a pump. Like in the transition system in Figure 1a, the pump has two locations on and off, and transitions *start*, *stop*, and *run* that describe how the locations can evolve. In addition to these constructs, the timed automaton has a clock x to characterize time passing. Initially, the clock x has the value 0. Invariants are shown inside locations and are omitted when they are *true*. Thus, the system can remain in off only when the value of clock x is less than 9. Similarly, it can move from location off to on when the value of x is greater than 6. Overall, this means that the system remains in off between 6 and 9 time units. Upon execution of *start* and *stop* transitions, the value of x is reset to 0. We often omit to write true guards and empty sets of clocks to reset.

Cernas, Godskesen, and Larsen [8] have introduced *timed modal specifications* (TMSs), which represent timed automata equipped with may (allowed) and must (required) transitions. We observe that a family of timed automata can be derived as implementations of a specification given as a TMS. We show one example of a

TMS in Figure 3b, such that the timed automaton in Figure 3a represents an implementation of it.

Like for the MTSs, the timed modal specifications have been generalized to *featured timed automata* (FTA) by Cordy et. al. [12], where different variants (implementations) are derived by feature selection. A *featured clock constraint* over a set $\mathbb{F}$ of features and a set C of clocks is a formula of the form:

$$g ::= true \,|\, [\chi](c < n) \,|\, [\chi](c \leq n) \,|\, [\chi](c > n) \,|\, [\chi](c \geq n) \,|\, g_1 \wedge g_2 \ ,$$

where $c \in C$, $n \in \mathbb{N}$, $\chi \in FeatExp(\mathbb{F})$. We denote the set of featured clock constraints over C and $\mathbb{F}$ by $FCC(C, \mathbb{F})$. We can now define FTA for modelling behavior of real-time variational systems.

Definition 2. *A featured timed automaton is a tuple $\mathcal{FTA} = (Loc, Act, C, trans, I, Inv, AP, L, \mathbb{F}, \mathbb{K}, \delta)$, where $Loc, Act, I, AP, L, \mathbb{F}, \mathbb{K}$, and δ are defined as in FTSs; trans $\subseteq Loc \times FCC(C, \mathbb{F}) \times Act \times C \times Loc$ is a finite set of transitions, $Inv : Loc \rightarrow FCC(C, \mathbb{F})$ is an invariant function that associates featured clock constraints (called invariants) to locations.*

The *projection* of a featured timed automaton $\mathcal{FTA}$ to a variant $k \in \mathbb{K}$, denoted $\pi_k(\mathcal{FTA})$, is the timed automaton $(Loc, Act, C, trans', I, Inv', AP, L)$, with $Inv'(l) = Inv(l)_{|k}$ and $trans' = \{t = (l, g_{|k}, \lambda, R, l') \mid t \in trans \wedge k \models \delta(t)\}$, where the projection of a featured clock constraint g to a variant k is defined inductively:

$$g_{|k} = \begin{cases} (g_1)_{|k} \wedge (g_2)_{|k} & \text{if } g = g_1 \wedge g_2 \\ g' & \text{if } g = [\chi]g' \wedge k \models \chi \\ true & \text{otherwise} \end{cases} \tag{1}$$

The *semantics* of an FTA $\mathcal{FTA}$, denoted $\llbracket \mathcal{FTA} \rrbracket_{FTA}$, is the union of behaviors of the projections on all variants $k \in \mathbb{K}$, that is $\llbracket \mathcal{FTA} \rrbracket_{FTA} = \cup_{k \in \mathbb{K}} \llbracket \pi_k(\mathcal{FTA}) \rrbracket_{TA}$, where $\llbracket \mathcal{TA} \rrbracket_{TA}$ denotes the semantics of the timed automaton $\mathcal{TA}$.

Figure 3c presents an FTA describing the timed behavior of several variants of a pump. Both invariants and time guards depend on variability. For example, $[\neg R](x < 9)$ is a featured clock constraint occurring in the invariant of the location off, which means that the system is forbidden to be in that location when the value of x is greater than 9 for variants that do not have the feature R. On the other hand, the invariant $[R](x < 7)$ specifies that the system is forbidden to be in off when the value of x is greater than 7 for variants that have the feature R. Transitions are also guarded with featured clock constraints in order to model requirement that the pump will need different preheating time before it begins to run. For example, $[R](x > 4) \wedge [\neg R](x > 6)$ means that the transition *start* can be taken after 4 time units for systems that have R, whereas for systems without R this delay is 6 time units. For ease of reading the presence conditions $\delta(t)$ labelling transitions are placed in square brackets next to action labels. The timed automaton shown in Fig. 3a is obtained by projection of the FTA of Fig. 3c to the variant $\{B\}$ (that is, $B \wedge \neg R$).

Summary. The right side of Fig. 2 summarizes the history of development of the real-time variability models. Timed automata were introduced as a concise syntax for a class of infinite transition systems. The timed modal specifications generalized timed automata by adding may and must modality to transitions. Finally, the featured timed automata were developed as generalizations of the timed modal specifications, arriving at an expressive formalism for modeling real-time behavior in variational system models.

3 Variability Abstractions

In this section, first we show how FTA can be transformed into FTA that contain only clock constraints and presence condition (feature expression) labels on transitions. Then, we define variability abstractions [18,19] for decreasing the size of such transformed FTA. Finally, we show that the obtained abstract FTA preserve the universal fragment of Timed CTL properties.

3.1 Transforming FTA

It has been shown [12] that any FTA can be transformed into an equivalent FTA without featured clock constraints. In particular, any featured clock constraint $g \in FCC(C, \mathbb{F})$ can be replaced with a combination of classical clock constraints and presence conditions. For each g, we create a partitionings $\mathbb{K}_1, \ldots, \mathbb{K}_n$ of $\mathbb{K}$ such that any two variants k and k' from the same partitioning $\mathbb{K}_j$ ($1 \leq j \leq n$) have the same projections $g_{|k}$ and $g_{|k'}$. Those projections are classical clock constraints, and we denote them $c_1, \ldots, c_n$, respectively. Let $t = (l, g, \lambda, R, l')$ be a transition and g be a featured clock constraint. We create a copy of t, denoted t_j (that is, (l, c_j, λ, R, l')), for each partitioning $\mathbb{K}_j$ where the guard is c_j and the presence condition is $\delta(t_j) = \delta(t) \wedge \mathbb{K}_j$. We add all transitions t_j in the transformed FTA, but we remove t from it. Let l be a location and $Inv(l)$ be a featured clock constraint. We create a copy of l, denoted l_j, for each partitioning $\mathbb{K}_j$ such that $Inv(l_j)$ is c_j. We copy all outgoing transitions t of l to be outgoing of any l_j, and all incoming transitions t to l to be incoming to any l_j with the corresponding presence condition $\delta(t) \wedge \mathbb{K}_j$. If l is an initial location, we create a new initial location with all clocks set to zero and add transitions to all l_j labelled with presence condition $\mathbb{K}_j$, (silent) action label τ, and time guard *true*. Finally, we remove l and all associated transitions from the transformed FTA. We show in Figure 4 the result of transforming the FTA in Figure 3c. Note that for ease of reading presence conditions are placed in square brackets before the labelling of a transition (which is a triple: time guard, action label, and set of resettable clocks). Both FTA have the same semantics, but the transformed one uses only classical clock constraints and presence conditions. From now on, we only consider such transformed FTA.
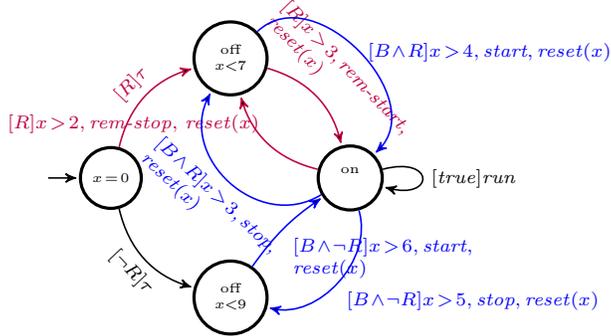
9

Fig. 4: An FTA with classical clock constraints and presence condition labels.

3.2 Abstracting FTA

Sometimes the computational task on a *concrete* complete lattice (domain) may be too costly or even uncomputable and this motivates replacing it with a simpler *abstract* lattice. A *Galois connection* is a pair of total functions, $\alpha : L \to M$ and $\gamma : M \to L$ (respectively known as the *abstraction* and *concretization* functions), connecting two complete lattices, $\langle L, \leqslant_L \rangle$ and $\langle M, \leqslant_M \rangle$ (often called the *concrete* and *abstract* domain, respectively), such that: $\alpha(l) \leqslant_M m \iff l \leqslant_L \gamma(m)$ for all $l \in L, m \in M$, which is often typeset as: $\langle L, \leqslant_L \rangle \xleftrightarrow[\alpha]{\gamma} \langle M, \leqslant_M \rangle$. Here $\leqslant_L$ and $\leqslant_M$ are the partial-order relations for L and M, respectively.

The aim of variability abstractions is to weaken feature expressions, effectively making transitions of an FTS present in more variants. In the following, we define variability abstractions as Galois connections for reducing the Boolean complete lattice of feature expressions (propositional formulae over $\mathbb{F}$): $(FeatExp(\mathbb{F})_{/\equiv}, \models, \vee, \wedge, true, false)$. Elements of $FeatExp(\mathbb{F})_{/\equiv}$ are equivalence classes of propositional formulae $\psi \in FeatExp(\mathbb{F})$ obtained by quotienting by the semantic equivalence $\equiv$. The partial-order relation $\models$ is defined as the satisfaction relation from propositional logic, whereas the least upper bound operator is $\vee$ and the greatest lower bound operator is $\wedge$. Furthermore, the least element is *false*, and the greatest element is *true*. Subsequently, we will lift the definition of variability abstractions to FTA.

The *join abstraction*, $\boldsymbol{\alpha}^{\mathrm{join}}$, merges the control-flow of all variants, obtaining a single variant that includes all executions occurring in any variant. The information about which transitions are associated with which variants is lost. Each feature expression ψ defined over $\mathbb{F}$ is replaced with *true* if there exists at least one configuration from $\mathbb{K}$ that satisfies ψ. The new abstract set of features is empty: $\boldsymbol{\alpha}^{\mathrm{join}}(\mathbb{F}) = \emptyset$, and the abstract set of valid configurations is a singleton: $\boldsymbol{\alpha}^{\mathrm{join}}(\mathbb{K}) = \{true\}$ if $\mathbb{K} \neq \emptyset$. The abstraction $\boldsymbol{\alpha}^{\mathrm{join}} : FeatExp(\mathbb{F}) \to FeatExp(\emptyset)$ and concretization functions $\boldsymbol{\gamma}^{\mathrm{join}} : FeatExp(\emptyset) \to FeatExp(\mathbb{F})$ are:

$$\boldsymbol{\alpha}^{\mathrm{join}}(\psi) = \begin{cases} true & \text{if } \exists k \in \mathbb{K}.k \models \psi \\ false & \text{otherwise} \end{cases} \qquad \begin{aligned} \boldsymbol{\gamma}^{\mathrm{join}}(true) &= true \\ \boldsymbol{\gamma}^{\mathrm{join}}(false) &= \bigvee\nolimits_{k \in 2^{\mathbb{F}} \setminus \mathbb{K}} k \end{aligned}$$

The proposed abstraction-concretization pair is a Galois connection [18,19].

The *feature ignore abstraction*, $\boldsymbol{\alpha}_A^{\text{fignore}}$, ignores a single feature $A \in \mathbb{F}$ by merging the control flow paths that only differ with regard to A, but keeps the precision with respect to control flow paths that do not depend on A. Let ψ be a formula in negation normal form (NNF). We write $\psi[l_A \mapsto true]$ to denote the formula ψ where the literal of A, that is A or $\neg A$, is replaced with *true*. The abstract sets of features and configurations are: $\boldsymbol{\alpha}_A^{\text{fignore}}(\mathbb{F}) = \mathbb{F}\backslash\{A\}$, and $\boldsymbol{\alpha}_A^{\text{fignore}}(\mathbb{K}) = \{k[l_A \mapsto true] \mid k \in \mathbb{K}\}$. The abstraction and concretization functions between $FeatExp(\mathbb{F})$ and $FeatExp(\boldsymbol{\alpha}_A^{\text{fignore}}(\mathbb{F}))$, which form a Galois connection [18,19], are defined as:

$$\boldsymbol{\alpha}_A^{\text{fignore}}(\psi) = \psi[l_A \mapsto true] \qquad \boldsymbol{\gamma}_A^{\text{fignore}}(\psi') = (\psi' \wedge A) \vee (\psi' \wedge \neg A)$$

where ψ and ψ' are in NNF.

The composition $\alpha_2 \circ \alpha_1$ runs two abstractions α_1 and α_2 in sequence (see [18,19] for precise definition). In the following, we will simply write (α, γ) for any Galois connection $\langle FeatExp(\mathbb{F})_{/\equiv}, \models\rangle \xleftrightarrow[\alpha]{\gamma} \langle FeatExp(\alpha(\mathbb{F}))_{/\equiv}, \models\rangle$ constructed using the operators presented in this section.

Given a Galois connection (α, γ) defined on the level of feature expressions, we now induce a notion of abstraction between (transformed) FTA.

Definition 3. *Let $\mathcal{FTA} = (Loc, Act, C, trans, I, Inv, AP, L, \mathbb{F}, \mathbb{K}, \delta)$ be an FTA, and (α, γ) be a Galois connection. We define the abstract FTA $\alpha(\mathcal{FTA})$ as the tuple $(Loc, Act, C, trans, I, Inv, AP, L, \alpha(\mathbb{F}), \alpha(\mathbb{K}), \alpha(\delta))$, where $\alpha(\delta) : trans \rightarrow FeatExp(\alpha(\mathbb{F}))$ is defined as: $\alpha(\delta)(t) = \alpha(\delta(t))$.*

We also define the *projection* of an (transformed) FTA with classical clock constraints $\mathcal{FTA}$ to a set of variants $\mathbb{K}' \subseteq \mathbb{K}$, denoted as $\pi_{\mathbb{K}'}(\mathcal{FTA})$, as the FTA $(Loc, Act, C, trans', I, Inv, AP, L, \mathbb{F}, \mathbb{K}', \delta)$, where $trans' = \{t \in trans \mid \exists k \in \mathbb{K}'.k \models \delta(t)\}$. We observe that we can combine variability abstractions with various projections on FTA, thus obtaining interesting (featured) timed automata that can be used for verification of the concrete FTA.

Example 1. Consider $\mathcal{FTA}$ in Figure 4 with the set of valid configurations $\mathbb{K} = \{\{B\}, \{R\}, \{B, R\}, \emptyset\}$. We show $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![R]\!]}(\mathcal{FTA}))$, $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![\neg R]\!]}(\mathcal{FTA}))$, and $\boldsymbol{\alpha}_R^{\text{fignore}}(\mathcal{FTA})$ in Figure 5. We do not show transitions labelled with the feature expression *false* and unreachable locations. Note that both $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![R]\!]}(\mathcal{FTA}))$ and $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![\neg R]\!]}(\mathcal{FTA}))$ are ordinary timed automata, since all transitions are labelled with the feature expression *true*. For $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![R]\!]}(\mathcal{FTA}))$ in Figure 5a, we have $\mathbb{K} \cap [\![R]\!] = \{\{R\}, \{B, R\}\}$ so transitions annotated with $\neg R$ are removed. For $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![\neg R]\!]}(\mathcal{FTA}))$ in Figure 5b, we have $\mathbb{K} \cap [\![\neg R]\!] = \{\{B\}, \emptyset\}$, so transitions annotated with R are removed. Note that $\boldsymbol{\alpha}_R^{\text{fignore}}(\mathcal{FTA})$ in Figure 5c is an FTA with the singleton set of features $\{B\}$ and two valid configurations $\{B\}$ and $\emptyset$ (that is, B and $\neg B$ respectively). $\qquad \square$

3.3   TCTL properties and their preservation

We consider the universal fragment of the Timed CTL (TCTL) [1]. TCTL is a timed variant of CTL used to express properties of timed automata. An universal
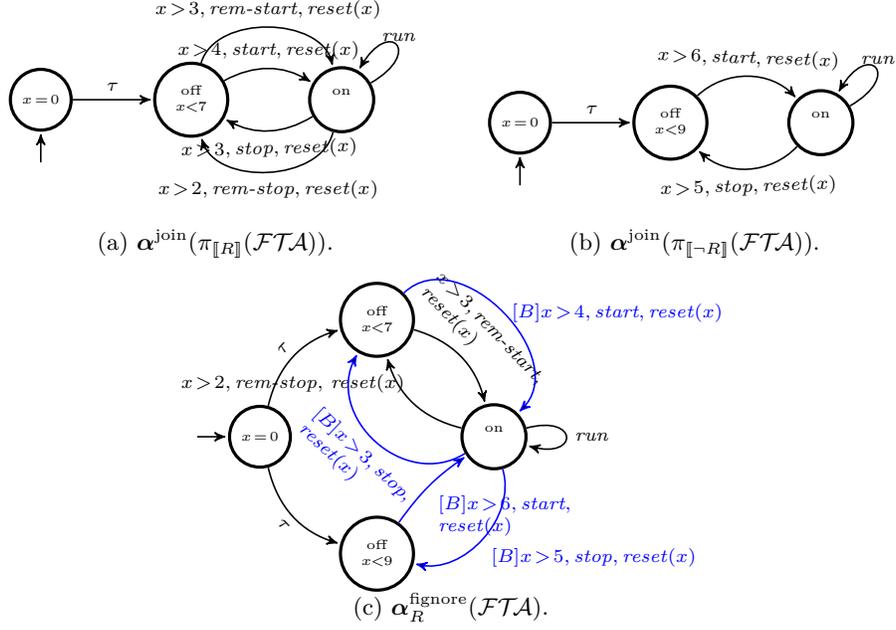
(a) $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![R]\!]}(\mathcal{FTA}))$.

(b) $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![\neg R]\!]}(\mathcal{FTA}))$.

(c) $\boldsymbol{\alpha}_R^{\text{fignore}}(\mathcal{FTA})$.

Fig. 5: Some abstractions of the real-time variability pump.

TCTL formula is defined inductively as:

$$\Phi ::= true \mid a \in AP \mid \neg a \mid g \in CC(C) \mid \Phi_1 \wedge \Phi_2 \mid \forall(\Phi_1 \, U^J \, \Phi_2) \mid \forall \Diamond^J \Phi \mid \forall \Box^J \Phi$$

where the formulae are in negation normal form ($\neg$ is applied only to atomic propositions), $CC(C)$ is a set of classical clock constraints over the set of clocks C, and $J \subseteq \mathbb{R}^+$ is a subinterval of $[0, \infty)$. The quantifier $\forall$ means that all time-divergent executions that start in a state satisfy the following temporal operator. Intuitively, $\forall(\Phi_1 \, U^J \, \Phi_2)$ means that for all (time-divergent) executions whenever at some point in J, a state is reached satisfying Φ_2 then at all previous time instants $\Phi_1 \vee \Phi_2$ holds. $\forall \Diamond^J \Phi = \forall(true \, U^J \, \Phi)$ means that for all (time-divergent) executions a state satisfying Φ can be reached during the interval J; whereas $\forall \Box^J \Phi$ asserts that for all (time-divergent) executions during the interval J the formula Φ always holds.

We show that abstract FTA have some interesting preservation properties. In particular, we show that an universal TCTL formula satisfied by an abstract FTA is also satisfied by the corresponding concrete FTA. First, we use a helping lemma shown in [18,19], which states that for any valid variant $k \in \mathbb{K}$ that can execute a behaviour guarded by feature expressions $\psi_0, \psi_1, \ldots$, there exists an abstract variant $k' \in \alpha(\mathbb{K})$ that can execute the same behaviour.

Lemma 1. *Let $\psi_0, \psi_1, \ldots \in FeatExp(\mathbb{F})$, $\mathbb{K}$ be a set of configurations over $\mathbb{F}$, and (α, γ) be a Galois connection. Let $k \in \mathbb{K}$, such that $k \models \psi_i$ for all $i \geq 0$. Then there exists $k' \in \alpha(\mathbb{K})$, such that $k' \models \alpha(\psi_i)$ for all $i \geq 0$.*

12

By using Lemma 1, we can prove the following result.

Theorem 1 (Soundness). *Let (α, γ) be a Galois connection. We have that $\alpha(\mathcal{FTA}) \models \Phi \implies \mathcal{FTA} \models \Phi$.*

Proof. We proceed by contraposition. Assume $\mathcal{FTA} \not\models \Phi$. Then, there exist a configuration $k \in \mathbb{K}$ and an (time-divergent) execution $\rho = s_0 \lambda_1 s_1 \lambda_2 \ldots \in [\![\pi_k(\mathcal{FTA})]\!]_{TA}$ such that $\rho \not\models \Phi$, i.e. $\rho \models \neg\Phi$. Note that ρ is an execution of the underlying transition system obtained by unfolding $\pi_k(\mathcal{FTA})$. This means that for all transitions in ρ, $t_i = s_i \xrightarrow{\lambda_{i+1}} s_{i+1}$ for $i = 0, 1, \ldots$, we have that $k \models \delta(t_i)$ for all $i \geq 0$. By Lemma 1, we have that there exists $k' \in \alpha(\mathbb{K})$, such that $k' \models \alpha(\delta(t_i))$ for all $i \geq 0$. Hence, the execution ρ is realizable for $\alpha(\mathcal{FTA})$, i.e. $\rho \in [\![\pi_{k'}(\alpha(\mathcal{FTA}))]\!]_{TA}$ and $\rho \models \neg\Phi$. It follows that $\alpha(\mathcal{FTA}) \not\models \Phi$. $\qquad\square$

The family-based model checking problem, $\mathcal{FTA} \models \Phi$, can be reduced to a number of smaller problems by partitioning the set of valid configurations $\mathbb{K}$.

Proposition 1. *Let the subsets $\mathbb{K}_1, \mathbb{K}_2, \ldots, \mathbb{K}_n$ form a* partition *of the set $\mathbb{K}$. Then: $\mathcal{FTA} \models \Phi$, if and only if, $\pi_{\mathbb{K}_1}(\mathcal{FTA}) \models \Phi \wedge \ldots \wedge \pi_{\mathbb{K}_n}(\mathcal{FTA}) \models \Phi$.*

Corollary 1. *Let $\mathbb{K}_1, \mathbb{K}_2, \ldots, \mathbb{K}_n$ form a* partition *of $\mathbb{K}$, and $(\alpha_1, \gamma_1), \ldots, (\alpha_n, \gamma_n)$ be Galois conn. If $\alpha_1(\pi_{\mathbb{K}_1}(\mathcal{FTA})) \models \Phi, \ldots, \alpha_n(\pi_{\mathbb{K}_n}(\mathcal{F})) \models \Phi$, then $\mathcal{FTA} \models \Phi$.*

The soundness results (Theorem 1 and Corollary 1) mean that the correctness of abstract FTA implies correctness of the concrete FTA. Note that verification of the abstract FTA can be drastically (even exponentially) faster. However, if the abstract FTA invalidate a property then the concrete FTA may still satisfy the property, i.e. the found counterexample in the abstract FTA may be *spurious* (introduced due to the abstraction) for some variants.

Example 2. Consider the property: "the pump will move from state `off` to `on` within 7 time unit", which is expressed by the universal TCTL formula $\Phi = \forall\Box(\text{off} \implies \forall\Diamond^7 \text{ on})$. We also consider timed automata $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![R]\!]}(\mathcal{FTA}))$ and $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![\neg R]\!]}(\mathcal{FTA}))$ shown in Figure 5. First, we can successfully verify that $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![R]\!]}(\mathcal{FTA})) \models \Phi$, which implies that all valid variants from $\mathbb{K}$ that contain the feature R satisfy the property Φ. On the other hand, we have $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![\neg R]\!]}(\mathcal{FTA})) \not\models \Phi$ with the counterexample where the system remains in `off` more than 7 time units and afterwards (e.g. at 8.5 time unit) it goes to `on`. This counterexample is genuine for the variants from $\mathbb{K}$ that do not contain the feature R. In this way, the problem of verifying $\mathcal{FTA}$ against Φ can be reduced to verifying whether two timed automata, $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![\neg R]\!]}(\mathcal{FTA}))$ and $\boldsymbol{\alpha}^{\text{join}}(\pi_{[\![R]\!]}(\mathcal{FTA}))$, satisfy Φ. $\qquad\square$

4 A Case Study: The Train-Gate System

The train-gate example comes with the installation of UPPAAL. It represents a railway control system which automatically controls access to a bridge for several

trains, such that the bridge may be accessed only by one train at a time. The system should safely guide trains from several tracks crossing the bridge. First, we describe the basic version of the train-gate system [4,34]. Then, we add variability into it thus creating a variational version of the system. Finally, we evaluate the verification of several interesting universal properties of the variational system using variability abstractions and UPPAAL.

4.1 Basic System

The basic system is modelled as a network of n trains and a controller in parallel. The model of a train, $Train_i$, is shown in Fig. 6a. It has five locations: Safe, Appr, Stop, Start, and Cross. The initial location is Safe, which corresponds to a train not approaching the bridge yet. When a train is approaching the bridge, it sends the signal $appr_i$ to the controller and goes to location Appr. This location has the invariant $x_i \leq 20$ (written next to the location), so it must be left within 20 time units. If the bridge is occupied the controller sends a $stop_i$ signal to prevent the train from entering the bridge by going to the location Stop. Otherwise, if $Train_i$ does not receive a $stop_i$ signal within 10 time units, it will start to cross the bridge by going to location Cross. The crossing train is assumed to leave the bridge within 3 to 5 time units by sending the signal $leave_i$. A stopped train waits for a go_i signal sent from the controller to the first train in the waiting list to restart. A restarted train from Start location reaches the crossing section between 7 and 15 time units non-deterministically.

The model of a gate controller, which synchronizes with trains, is shown in Fig. 6b. It uses a list L to keep record of the trains waiting to cross the bridge and an integer variable len for the length of L. The controller starts in the location Free, where the bridge is free and checks whether the list L is empty. If L is empty and a train is approaching, then this train is added at the back of L with enqueue() operation. If L is not empty, then $Train_i$ at the front of L is restarted with the go_i signal. In the Occ location, the train at the front of L, $Train_i$, is crossing the bridge. When the crossing train leaves the bridge, the controller receives a $leave_i$ signal and removes it from the list L with dequeue() operation. If another $Train_i$ is approaching the bridge in Occ location, that train is added at the back of L and stopped with the $stop_i$ signal. Note that the location C represents a committed location which avoids any time delay in it [4].

4.2 Variational System

We now extend the basic train-gate system given in Fig. 6, to construct a variational system that describes the behaviours of a family of train-gate systems. Fig. 7 shows all additional transitions in the variational system that do not occur in the basic system in Fig. 6. They are labelled with presence conditions, which denote whether a transition is included (present) in a given variant. We assume that all transitions in the basic system in Fig. 6, which are shown in bold in Fig. 7, are enabled in all variants, i.e. their presence condition is *true*. The variational train-gate system has four optional features, which are assigned an identifying
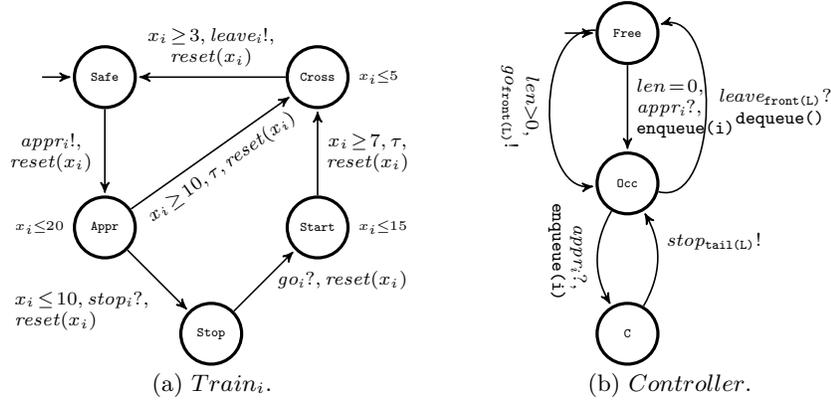
14

$$x_i \geq 3, leave_i!,$$
$$reset(x_i)$$

Safe ⟶ Cross $x_i \leq 5$

$appr_i!,$
$reset(x_i)$

$x_i \geq 10, \tau, reset(x_i)$

$x_i \geq 7, \tau,$
$reset(x_i)$

$x_i \leq 20$ Appr

Start $x_i \leq 15$

$x_i \leq 10, stop_i?,$
$reset(x_i)$

Stop

$go_i?, reset(x_i)$

Free

$go_{front(L)}!$

$len > 0,$

$len = 0,$
$appr_i?,$
enqueue(i)

$leave_{front(L)}?$
dequeue()

Occ

enqueue(i)

$appr_i?,$

$stop_{tail(L)}!$

C

(a) $Train_i$. (b) $Controller$.

Fig. 6: The basic train-gate system.

letter and a color. The feature `Fast` (denoted by F, in red) is used for denoting fast approaching trains, which are placed at the front of the waiting list L thus having higher priority than the others. When a fast approaching $Train_i$ comes to the bridge it sends the $fast_i$ signal to the controller. If the bridge is occupied, the train is stopped and added at the second position of L just after the crossing train using `secqueue(i)` operation. The second feature `Capacity` (denoted by C, in green) is used for a controller that allows $\frac{2}{3}n$ trains to be able to approach the bridge. When the feature `Capacity` is enabled, the controller will ignore any approach signal if the number of approaching and crossing trains is greater or equal than $\frac{2}{3}n$. The feature `GoSecond` (denoted by GS, in brown) is used for controller to allow the second train in the waiting list L to restart instead of the first one, after a crossing train has left the bridge. The transitions enabled by `GoSecond` use the operations: `second()` to retrieve the second element of L, and `desecqueue()` to remove the second element of L. The feature `GoLast` (denoted by GL, in blue) is used for controller to allow the last train in the waiting list L to restart after a crossing train has left the bridge. The operations are: `tail()` to retrieve the last element of L, and `destack()` to remove the last element of L.

4.3 Verification

Implementation. Inputs to UPPAAL represent XML files where all locations and transitions are described in separate tags. To describe variational systems, we use the color attribute of the transition tag to encode the presence condition that labels a transition. The sets of available features and valid configurations are defined using TVL files [9]. We have implemented variability abstractions as source-to-source transformations of XML files that represent variational systems.

Properties. We check several interesting universal properties to check on the variational train-gate system. The property "$\phi_1 = \forall \square$ forall $(i : int[0, n-1])$ forall $(j : int[0, n-1])$ $(Train_i.Cross \wedge Train_j.Cross \implies i == j)$" states that there
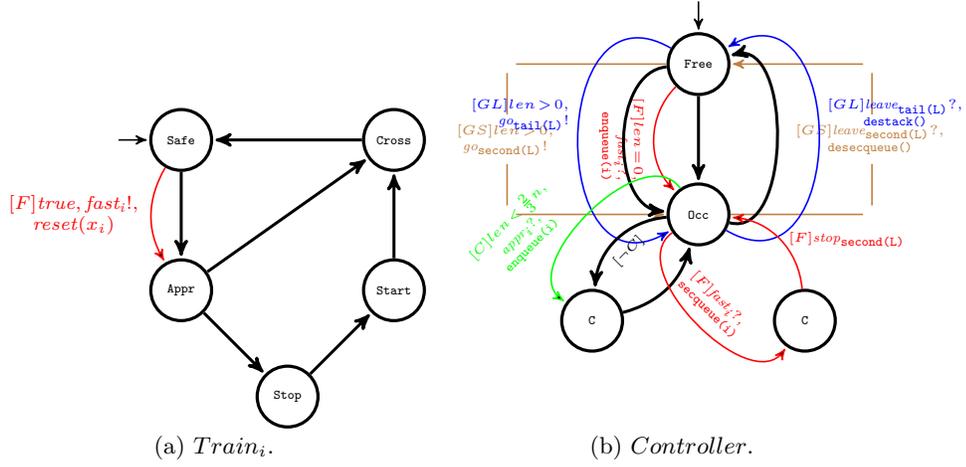
(a) $Train_i$.   (b) $Controller$.

Fig. 7: The variational train-gate system.

is never more than one train crossing the bridge at any time instance. The property "$\phi_2 = \forall \Box \, (Gate.L[n] == 0)$" states that there can never be n elements in the waiting list, thus the list L will not overflow. The property "$\phi_3 = \forall \Box \, (Train_0.Appr \implies \forall \Diamond \, Train_0.Cross)$" states that whenever the train 0 approaches the bridge, it will eventually cross. Similar properties can be written for the other trains from 1 to $n-1$. Finally, the property "$\phi_4 = \forall \Box \, not \, deadlock$" checks that the system is deadlock-free.

The basic system in Fig. 6 satisfies all four properties. All properties also hold for variants where feature `Capacity` is enabled (the others are disabled). For variants with all (or some) of features `Fast`, `GoSecond`, `GoLast` enabled, the properties ϕ_1, ϕ_2 still hold, but ϕ_3 and ϕ_4 are violated. In case of `Fast` enabled and ϕ_3, a counter-example is shown where an approaching train is stopped and added to L but all next trains are fast approaching, so the (no fast) approaching one can never cross the bridge. In case of `GoSecond` enabled and ϕ_3, the reported counter-example shows that when a train leaves the bridge the train that restarts is never the first one, so this train at the head of L is stuck and can never cross the bridge. A similar counter-example is obtained if `GoLast` is enabled. In case of both `Fast` and `GoSecond` enabled and ϕ_4, the system is deadlocked when it chooses the second train in L to restart with *go* signal, but then a fast approaching train is added to L using *fast* signal. Thus, the restarted train is not able to leave the bridge since it is now on the third place in L and the first two trains in L are both stopped.

The variational train-gate system has $2^4 = 16$ variants in total. We use two approaches to check the above four properties. First, the brute-force approach consists of verifying a property by calling UPPAAL to check it for each individual variant (thus we have 16 UPPAAL calls). Second, the approach based on variability abstractions consists of applying an abstraction on the varia-

16

prop.	brute-force			abstraction-based		
	CALLS	TIME	SPACE	CALLS	TIME	SPACE
ϕ_1	16	18.5	347,920	1	1.4	42,145
ϕ_2	16	19.2	347,920	1	1.1	42,145
ϕ_3	16	15.1	16,232	4	3.7	3,244
ϕ_4	16	18.9	275,596	4	4.8	83863

Fig. 8: Performances of the brute-force vs. abstraction-based approaches for the variational train-gate system with $n = 6$. TIME in seconds.

tional system and then verifying the corresponding property on the obtained abstract system. The properties ϕ_1 and ϕ_2 (satisfied by all variants) can be checked by applying $\boldsymbol{\alpha}^{\mathrm{join}}$ on the variational system and then calling UPPAAL once to verify the obtained abstract system. The property ϕ_3 is violated by variants that satisfy Fast $\vee$ GoSecond $\vee$ GoLast (14 in total). We use UPPAAL to verify satisfaction of ϕ_3 against four models obtained by applying $\boldsymbol{\alpha}^{\mathrm{join}}$ on the following projections of the variational train-gate system: $\pi_{[\![\texttt{Fast}]\!]}$, $\pi_{[\![\texttt{GoSecond}]\!]}$, $\pi_{[\![\texttt{GoLast}]\!]}$, and $\pi_{[\![\neg\texttt{Fast}\wedge\neg\texttt{GoSecond}\wedge\neg\texttt{GoLast}]\!]}$. Using four calls to UPPAAL we obtain that ϕ_3 is violated by the first three abstracted projections, and is satisfied by the last abstracted projection. The property ϕ_4 is violated by variants that satisfy (Fast$\wedge$GoSecond)$\vee$GoLast (10 in total). In this case, we verify ϕ_4 against four models obtained by applying $\boldsymbol{\alpha}^{\mathrm{join}}$ on the following projections: $\pi_{[\![\neg\texttt{Fast}\wedge\neg\texttt{GoLast}]\!]}$, $\pi_{[\![\neg\texttt{GoSecond}\wedge\neg\texttt{GoLast}]\!]}$, $\pi_{[\![\texttt{Fast}\wedge\texttt{GoSecond}]\!]}$, and $\pi_{[\![\texttt{GoLast}]\!]}$. The first two abstract models satisfy ϕ_4, but the last two models do not satisfy ϕ_4.

Results. All experiments are executed on a 64-bit Intel®Core^{TM} i5 CPU with 8 GB memory. All times are reported as averages over five runs with the highest and lowest number removed. Fig. 8 compares the performance of our approach based on variability abstractions with the brute-force approach to verify the above four properties for the system with $n = 6$ trains. For each experiment, we report: the number of calls to UPPAAL, the total verification time (TIME) and the total number of explored states (SPACE). TIME (resp., SPACE) is the sum of verification times (resp., the number of explored states) of all individual UPPAAL calls taken in verifying each property. We can see that our abstraction-based approach achieves improvements in both TIME and SPACE for all properties.

5 Related Work

Recently, various family-based techniques have been proposed which lift existing single-program verification techniques to work on the level of program families. This includes family-based syntax checking [25,22], family-based type checking [24], family-based static analysis [7,6,31], family-based verification by rewriting variability [23,33], etc. TYPECHEF [25] and SUPERC [22] are variability-aware parsers, which can parse C language with preprocessor annotations; whereas

family-based type checking for Featherweight Java was presented in [24]. Brabrand et al. [7] show how to lift any single-program dataflow analysis from the monotone framework to work on the level of program families; whereas Midtgaard et al. [31] show the lifting for any static analysis from the abstract interpretation framework. The obtained family-based analyses are much faster than ones based on the naive brute-force approach that generates and analyzes all variants one by one. In order to speed-up such family-based static analyses, variability abstractions have been introduced in [15,20]. They aim to abstract (reduce) the configuration space of the given family. Each abstraction expresses a compromise between precision and speed in the induced abstract family-based analyses [15]. However, the number of possible abstractions is intractably large with most abstractions being too imprecise or too costly to show the analysis's ultimate goal. The work in [20] proposes a technique to efficiently find a suitable variability abstraction for a family-based static analysis to establish a given query. Another efficient implementation of family-based analysis formulated within the IFDS framework for inter-procedural distributive environments has been proposed in SPL$^{\text{LIFT}}$ [6]. The works [23,33] are based on using transformations to generate a single program which simulates the behaviour of all variants in a family. This is achieved by replacing compile-time variability with run-time variability (non-determinism). Then, existing single-system analyzers are used to analyze the generated simulator. An approach for family-based software model checking using game semantics has been introduced in [17]. It verifies safety of `#ifdef`-based second-order program families containing undefined components, which are compactly represented using symbolic game semantics models [16].

6   Conclusion

We have proposed variability abstractions to derive abstract model checking for real-time variational systems. By exploiting the knowledge of a variability model and property, we may carefully devise variability abstractions that are able to verify interesting properties in only a few calls to UPPAAL. As a future work, we want to automate our verification approach by developing an abstraction refinement procedure, similarly to the context of SPIN and FPromela [21]. The abstraction refinement procedure will use spurious counterexample to iteratively refine abstract variational models until either a genuine counter-example is found or the property satisfaction is shown for all variants in the family.

References

1. Alur, R., Courcoubetis, C., Dill, D.L.: Model-checking in dense real-time. Inf. Comput. 104(1), 2–34 (1993), http://dx.doi.org/10.1006/inco.1993.1024
2. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. 126(2), 183–235 (1994), http://dx.doi.org/10.1016/0304-3975(94)90010-8
3. Baier, C., Katoen, J.: Principles of model checking. MIT Press (2008)

4. Behrmann, G., David, A., Larsen, K.G.: A tutorial on uppaal 4.0. In: Formal Methods for the Design of Real-Time Systems, International School on Formal Methods for the Design of Computer, Communication and Software Systems, SFM-RT 2004, Revised Lectures. LNCS, vol. 3185, pp. 200–236. Springer (2004), http://dx.doi.org/10.1007/978-3-540-30080-9_7

5. Benes, N., Kretínský, J., Larsen, K.G., Møller, M.H., Srba, J.: Parametric modal transition systems. In: Automated Technology for Verification and Analysis, 9th International Symposium, ATVA 2011. Proceedings. LNCS, vol. 6996, pp. 275–289. Springer (2011), http://dx.doi.org/10.1007/978-3-642-24372-1_20

6. Bodden, E., Tolêdo, T., Ribeiro, M., Brabrand, C., Borba, P., Mezini, M.: Spl$^{\text{lift}}$: Statically analyzing software product lines in minutes instead of years. In: ACM SIGPLAN Conference on PLDI '13. pp. 355–364 (2013)

7. Brabrand, C., Ribeiro, M., Tolêdo, T., Winther, J., Borba, P.: Intraprocedural dataflow analysis for software product lines. Transactions on Aspect-Oriented Software Development 10, 73–108 (2013)

8. Cerans, K., Godskesen, J.C., Larsen, K.G.: Timed modal specification - theory and tools. In: Computer Aided Verification, 5th International Conference, CAV '93, Proceedings. LNCS, vol. 697, pp. 253–267. Springer (1993), http://dx.doi.org/10.1007/3-540-56922-7_21

9. Classen, A., Boucher, Q., Heymans, P.: A text-based approach to feature modelling: Syntax and semantics of TVL. Sci. Comput. Program. 76(12), 1130–1143 (2011), http://dx.doi.org/10.1016/j.scico.2010.10.005

10. Classen, A., Cordy, M., Heymans, P., Legay, A., Schobbens, P.: Model checking software product lines with SNIP. STTT 14(5), 589–612 (2012), http://dx.doi.org/10.1007/s10009-012-0234-1

11. Classen, A., Cordy, M., Schobbens, P., Heymans, P., Legay, A., Raskin, J.: Featured transition systems: Foundations for verifying variability-intensive systems and their application to LTL model checking. IEEE Trans. Software Eng. 39(8), 1069–1089 (2013), http://doi.ieeecomputersociety.org/10.1109/TSE.2012.86

12. Cordy, M., Classen, A., Heymans, P., Schobbens, P., Legay, A.: Provelines: a product line of verifiers for software product lines. In: 17th International Software Product Line Conference co-located workshops, SPLC 2013 workshops. pp. 141–146. ACM (2013), http://doi.acm.org/10.1145/2499777.2499781

13. Cordy, M., Schobbens, P., Heymans, P., Legay, A.: Behavioural modelling and verification of real-time software product lines. In: 16th International Software Product Line Conference, SPLC '12, Volume 1. pp. 66–75. ACM (2012), http://doi.acm.org/10.1145/2362536.2362549

14. Czarnecki, K., Antkiewicz, M.: Mapping features to models: A template approach based on superimposed variants. In: Generative Programming and Component Engineering, 4th Int. Conf., GPCE 2005. LNCS, vol. 3676, pp. 422–437 (2005), http://dx.doi.org/10.1007/11561347_28

15. Dimovski, A., Brabrand, C., Wąsowski, A.: Variability abstractions: Trading precision for speed in family-based analyses. In: 29th European Conference on Object-Oriented Programming ECOOP 2015. LIPIcs, vol. 37, pp. 247–270. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2015)

16. Dimovski, A.S.: Program verification using symbolic game semantics. Theor. Comput. Sci. 560, 364–379 (2014), http://dx.doi.org/10.1016/j.tcs.2014.01.016

17. Dimovski, A.S.: Symbolic game semantics for model checking program families. In: Model Checking Software - 23nd International Symposium, SPIN 2016, Proceedings. LNCS, vol. 9641, pp. 19–37. Springer (2016)

18. Dimovski, A.S., Al-Sibahi, A.S., Brabrand, C., Wasowski, A.: Family-based model checking without a family-based model checker. In: Model Checking Software - 22nd International Symposium, SPIN 2015, Proceedings. LNCS, vol. 9232, pp. 282–299. Springer (2015), http://dx.doi.org/10.1007/978-3-319-23404-5_18

19. Dimovski, A.S., Al-Sibahi, A.S., Brabrand, C., Wasowski, A.: Efficient family-based model checking via variability abstractions. STTT pp. 1–19 (2016)

20. Dimovski, A.S., Brabrand, C., Wasowski, A.: Finding suitable variability abstractions for family-based analysis. In: FM 2016: Formal Methods - 21st International Symposium, Proceedings. LNCS, vol. 9995, pp. 217–234 (2016), http://dx.doi.org/10.1007/978-3-319-48989-6_14

21. Dimovski, A.S., Wasowski, A.: Variability-specific abstraction refinement for family-based model checking. In: Fundamental Approaches to Software Engineering - 20th International Conference, FASE 2017, Proceedings. LNCS, vol. 10202, pp. 406–423. Springer (2017), http://dx.doi.org/10.1007/978-3-662-54494-5_24

22. Gazzillo, P., Grimm, R.: Superc: parsing all of C by taming the preprocessor. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12. pp. 323–334. ACM (2012), http://doi.acm.org/10.1145/2254064.2254103

23. Iosif-Lazar, A.F., Melo, J., Dimovski, A.S., Brabrand, C., Wasowski, A.: Effective analysis of c programs by rewriting variability. The Art, Science, and Engineering of Programming, Programming'17 1(1), 1–25 (2017)

24. Kästner, C., Apel, S., Thüm, T., Saake, G.: Type checking annotation-based product lines. ACM Trans. Softw. Eng. Methodol. 21(3), 14 (2012)

25. Kästner, C., Giarrusso, P.G., Rendel, T., Erdweg, S., Ostermann, K., Berger, T.: Variability-aware parsing in the presence of lexical macros and conditional compilation. In: OOPSLA'11. pp. 805–824. ACM, Portland, OR, USA (2011)

26. Larsen, K.G.: Context-Dependent Bisimulation Between Processes. Ph.D. thesis, University of Edinburgh, UK (May 1986)

27. Larsen, K.G., Nyman, U., Wasowski, A.: Modal I/O automata for interface and product line theories. In: Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Proceedings. LNCS, vol. 4421, pp. 64–79. Springer (2007), http://dx.doi.org/10.1007/978-3-540-71316-6_6

28. Larsen, K.G., Nyman, U., Wasowski, A.: Modeling software product lines using color-blind transition systems. STTT 9(5-6), 471–487 (2007)

29. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. STTT 1(1-2), 134–152 (1997), http://dx.doi.org/10.1007/s100090050010

30. Larsen, K.G., Thomsen, B.: A modal process logic. In: Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88). pp. 203–210. IEEE Computer Society (1988), http://dx.doi.org/10.1109/LICS.1988.5119

31. Midtgaard, J., Dimovski, A.S., Brabrand, C., Wasowski, A.: Systematic derivation of correct variability-aware program analyses. Sci. Comput. Program. 105, 145–170 (2015), http://dx.doi:10.1016/j.scico.2015.04.005

32. Minsky, M.: Computation : Finite and infinite machines. Prentice Hall, Princeton, N.J., USA (1967)

33. von Rhein, A., Thüm, T., Schaefer, I., Liebig, J., Apel, S.: Variability encoding: From compile-time to load-time variability. J. Log. Algebr. Meth. Program. 85(1), 125–145 (2016)

34. Yi, W., Pettersson, P., Daniels, M.: Automatic verification of real-time communicating systems by constraint-solving. In: Formal Description Techniques VII, Proceedings of the 7th IFIP WG6.1 Int. Conf. on Formal Description Techniques. IFIP Conference Proceedings, vol. 6, pp. 243–258. Chapman & Hall (1994)